Logics on Data Words

Expressivity, Satisfiability, Model Checking

Dissertation

zur Erlangung des Grades eines

DOKTORS DER NATURWISSENSCHAFTEN

der Universität Dortmund
am Fachbereich Informatik
von

Ahmet Kara

Dortmund

2016

Tag der mündlichen Prüfung:	12.9.2016
Dekan:	Prof. Dr.-Ing. Gernot Fink
Gutachter:	Prof. Dr. Thomas Schwentick Dr. Benedikt Bollig

Preface

I am very grateful to my supervisor Prof. Dr. Thomas Schwentick for all his support and advises during my time as a research associate at the computer science department of the Technical University in Dortmund and during the preparation of this work. As an assistant of his lectures, I experienced personal development and learned how to present and illustrate formal contents. Moreover, he teached me what it means to integrate problems from the area of theoretical computer science into a wider scientific context and to work technically precise when solving these problems.

In addition, I acknowledge the financial support of the German Research Foundation (DFG) under grants SCHW 678/4-1 and SCHW 678/4-2. I also thank Meral Sayan very much for proofreading this work. Finally, I thank my family for all their support during my time as a PhD student.

When I started my work at the university, the original plan was that I would work with Volker Weber on a project on non-classical logics on structures with data values. Volker Weber had been, besides Thomas Schwentick, one of the supervisors of my diploma thesis. With him I had my first discussions on data logics. Unfortunately, he died shortly after I had started working at the university. This work is dedicated to his memory.

Dortmund, March 2016 Ahmet Kara

Contents

1 Introduction **1**

2 An Example Scenario: Server and Clients **5**

A Preliminaries **9**

3 Basics **13**
- 3.1 Notational Conventions . . . 13
- 3.2 Some Tools . . . 14
  - 3.2.1 Two-Way Alternating Automata . . . 14
  - 3.2.2 Counter Machines . . . 16
  - 3.2.3 Well-Structured Transition Systems . . . 17
  - 3.2.4 Transducers and the Transduction Problem . . . 18
  - 3.2.5 Post's Correspondence Problem . . . 18

4 Data Words, their Automata and Logics **21**
- 4.1 Data Words . . . 21
- 4.2 Automata for Data Words . . . 23
  - 4.2.1 Register Automata . . . 23
  - 4.2.2 Data Automata . . . 25
  - 4.2.3 Further Automata Models . . . 27
- 4.3 Logics for Data Words . . . 29
  - 4.3.1 First Order Logic . . . 30
  - 4.3.2 Temporal Logic . . . 32
  - 4.3.3 Logics based on Regular Expressions . . . 33
  - 4.3.4 Further Logics . . . 35

B New Insights on Data Logics **39**

5 Motivating Questions on Data Logics **43**

6 Navigation along Data Values **47**
- 6.1 Basic Data Navigation Logic . . . 48
- 6.2 Decidability of Basic Data Navigation Logic . . . 49
- 6.3 Basic Data Navigation Logic on Infinite Data Words . . . 57
- 6.4 Undecidable Extensions of Basic Data Navigation Logic . . . 58
- 6.5 Decidable Extensions of Basic Data Navigation Logic . . . 63

6.6 Expressivity of Data Navigation Logic 71
6.7 Discussion 73

7 The Power of Storing Positions **75**
7.1 Hybrid Temporal Logic on Data Words 76
7.2 Hybrid Temporal Logic vs. Freeze LTL 76
7.2.1 Expressivity 76
7.2.1.1 Multiple Variables 76
7.2.1.2 One Variable 85
7.2.2 Succinctness 88
7.3 Hierarchy Results 90
7.4 Discussion 93

8 Automata for Two-Variable Logic **95**
8.1 Weak Data Automata 96
8.2 Expressivity of Weak Data Automata 97
8.2.1 Comparison with other Automata Models 97
8.2.2 Logical Characterization 99
8.3 Complexity of Weak Data Automata 106
8.4 Weak Data Automata on Infinite Data Words 108
8.5 Discussion 109

C Models and Model Checking **111**

9 From Finite-State towards Infinite-State Model Checking - A Brief Review **115**

10 Motivating Questions on System Models and Model Checking **119**

11 Three Models - Three Views **123**
11.1 Notational Conventions 123
11.2 Dynamic Communicating Automata 124
11.2.1 Non-Emptiness 129
11.2.2 State Reachability 133
11.2.2.1 Dynamic Communicating Automata with Buffers 141
11.3 Process Register Automata 142
11.3.1 Non-Emptiness 146
11.4 Branching High-Level Message Sequence Charts 148
11.4.1 Non-Emptiness 158
11.4.2 Executability 159
11.5 Discussion 167

12 New Results on Model Checking **171**
12.1 Model Checking of Dynamic Communicating Automata 172
12.1.1 Model Checking with Restricted Basic Data LTL 172
12.1.2 Model Checking with Freeze LTL 182
12.2 Model Checking of Process Register Automata 185
12.2.1 Model Checking with Freeze LTL 185
12.2.2 Model Checking with Hybrid Temporal Logic 195
12.3 Model Checking of Branching High-Level MSCs 200
12.3.1 MSC Navigation Logic 200

12.3.2 Model Checking with MSC Navigation Logic 200
12.4 Discussion 207

13 The Journey of Data Logics - A Glance into the Future 211

Acronyms 213

Appendix A Full Synax and Semantics of introduced Logics 231
A.1 First Order Logic on Data Words ($FO^{\sim}$) 231
A.2 Freeze LTL ($LTL^{\Downarrow}$) 232
A.3 Regular Expressions with Memory (REM) 233
A.4 Two-Way Path Logic (PathLog) 234
A.5 Constraint Logic ($CLTL^{XF}$) 234
A.6 Logic of Repeating Values (LRV) 235
A.7 Basic Data Navigation Logic (B-DNL) 235
A.8 Hybrid Temporal Logic on Data Words ($HTL^{\sim}$) 237
A.9 MSC Navigation Logic (MNL) 237

Chapter 1

Introduction

As a sub-area of formal system verification, *model checking* deals with the development of formal models describing software and hardware systems and the design of algorithms testing wether a given model meets its specification [69]. *Finite-state model checking* constitutes in this context a well-established field of research with useful results applied in practice. A successful approach in finite-state model checking is to model a software or hardware system by a transition system with finitely many states and to use logics for the formulation of requirements expected from the system. The states of a transition system are usually equipped with propositions from a finite domain that are representing relevant system properties in corresponding states. Each path in a transition system is associated with a sequential structure called *trace* which is labelled by propositions and describes changing informations about the modeled system during a single execution. A widely accepted logic for the specification of system requirements is *Linear-Time Temporal Logic (LTL)* which allows to express properties on words with labels from finite domains. Given a transition system and a formula specifying a system requirement, a classical task in finite-state model checking is to decide whether the formula is satisfied on all traces of the transition system. Finite-state model checking with LTL-like logics is a well-studied field with good complexity results which have led to successful practical applications, especially in the verification of sequential circuit design and communication protocols [69, 91, 30].

Finite-state models, however, do not always suffice for the adequate description and verification of software and hardware. Systems exceeding the capabilities of this kind of models are, for instance, those in which time aspects play a major role or which operate on variables with an infinite range of possible values. Further example domains are mobile computing and ad-hoc networks in which the number of participating actors is not know in advance, but changes dynamically during system executions. Obviously, infinite-state models are harder to analyze and most of the model checking techniques from the finite-state setting do not extend to the infinite case. Nevertheless, the literature proposes some frameworks like *Regular Model Checking* [201, 133] and *Well-Structured Transition Systems* [5, 98] to deal with infinite-state models. The questions addressed in these frameworks are mainly safety problems which are solved by computing a representation of all reachable system states. There are also attempts to adopt the classical approach of finite-state model checking with logics to the infinite case [61, 20, 99]. However, in contrast to the finite case, there are neither standardized system models, nor standardized specification languages for the infinite case.

In recent years, a lot of effort has been devoted to the design and analysis of logics and automata on so-called *data words* and *data trees*, i.e., words and trees labelled by data values from infinite domains [82, 43, 40, 95]. These works are mainly motivated by the static analysis of semi-structured data in the area of XML and the verification of systems involving unboundedly many concurrent processes. Indeed, XML-trees can be modeled by data trees in which data values represent attribute

values or text nodes. Furthermore, traces of systems where the main source of infinity is the unboundedness of interacting processes can be represented by data words where data values stand for process IDs. Hence, logics on data words, called *data logics* in this work, appear as reasonable formalisms which can be used as specification languages in the model checking of systems with unboundedly many processes. However, besides some exceptions [85, 45, 110], the most considered problem for data logics is the satisfiability problem until now, but not the model checking problem. One possible reason for this might be that, in contrast to formalisms on classical words and trees with propositions, formalisms on data words have bad computational properties. For instance, while the satisfiability problem for First-Order Logic on classical words is decidable, the extension of this logic by an equality relation on data values is undecidable on data words. Undecidability even holds if only three position variables are allowed. Decidability is obtained in case of two position variables [43]. Hence, in the first instance, the aim was to find expressive, but decidable logics and automata. A further reason for the neglect of model checking with data logics could be, as stated above, the lack of a common system model which can be used in the framework of model checking.

Our main objectives in this work are the continuation of the current research on the expressivity and complexity of logics and automata on data words, the analysis of models for concurrent systems with unboundedly many processes and the study of the model checking problem for these models in combination with data logics. We describe the three directions in our work in more detail:

Study of logics and automata on data words. We design and analyze logics and automata on data words. We restrict to formalisms where the only predicate on data values is the equality relation. Results in the literature indicate that even in this case it is quite hard to find expressive formalisms with good computational properties. Besides the expressive power of the designed logics and automata, we are interested in the complexity of their satisfiability and non-emptiness problem, respectively. One of our particular aims in this part is the design of an expressive and decidable data logic suitable for the usage in the framework of model checking for concurrent systems. Furthermore, we study cases in which logics, which are expressively equivalent on classical words, disagree on data words. Finally, we gain new insights into the correspondence between logics and automata on data words.

Study of models for concurrent systems with unboundedly many processes. As stated above, there is a lack of standardized infinite-state models in the literature. After a review of existing models, we focus on three models suited for the design of concurrent systems with unboundedly many processes. Each model provides a different view on the modeled systems. As a first step towards model checking, we analyze the computational properties of these models with respect to common decision problems like non-emptiness and reachability.

Study of model checking with data logics. We investigate the complexity of the model checking of the three system models, mentioned above, with respect to different data logics and compare our results to the complexities of the satisfiability and non-emptiness problem of the corresponding system models and logics. Our results in this part are non-exhaustive and there is no claim to completeness. As a matter of fact, they function as first insights and provide a basis for further research on model checking with data logics.

We already mentioned some basic works related to the investigated topics. Further references are given in corresponding chapters.

Structure of the work

In Chapter 2, we present an exemplary concurrent system to which we refer in subsequent chapters in order to demonstrate the expressivity of introduced logics and system models.

Part A equips the reader with some background informations. Chapter 3 presents, besides basic

notions and notations, some frameworks and decision problems which are used as auxiliary tools in the proofs of this work. Chapter 4 contains the definition of data words and an overview of known automata and logics on these structures along with a summary of basic results on expressivity and complexity.

Part B contains our new results on the expressivity and complexity of logics and automata on data words. It starts with some motivating questions in Chapter 5 which are arising from known results presented in Chapter 4. In the remaining chapters of this part, namely Chapters 6-8, we address these questions and work out our solutions.

Part C is devoted to models for concurrent systems with unboundedly many processes and our results on these models. We first give in Chapter 9 a short survey on existing models and model checking techniques for finite- and infinite-state systems. Then, we concentrate on three models and formulate some open questions on them in Chapter 10. These questions relate, on the one hand, to basic decision problems for these models and, on the other hand, to their computational behaviour with regard to model checking with data logics. The questions of the first kind are tackled in Chapter 11 and those of the second kind in Chapter 12.

Each chapter containing our new results, i.e., Chapters 6-8, 11 and 12, close with a section in which we discuss the outcomes, summarize questions left open and explain in which extent the presented results were already published in our previous papers. A glance into the future of the research on data logics is given in Chapter 13.

In the main parts of this work we usually explain the semantics of introduced logics only at an informal level. Precise definitions of semantics can be found in Appendix A.

Chapter 2

An Example Scenario: Server and Clients

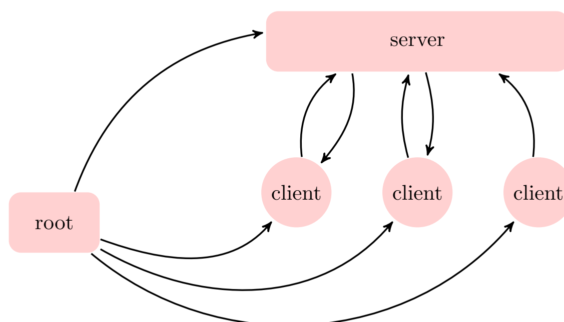


Figure 2.1: Communicating server and clients

In this chapter, we will describe a concurrent system with unboundedly many processes and model its behaviour at an informal level. Our main purpose is to introduce an exemplary system along with some of its basic properties to which we can refer when we demonstrate the expressive power of new logics and system models throughout this work.

We assume that the system we are going to describe realizes some communication protocol between a server process and unboundedly many client processes. We will not give the details of the protocol, nor will we propose a concrete (automata) model implementing the system, we will rather focus on the questions how communication between processes is established and how the network of processes evolves over time. Furthermore, we will discuss how the traces of the system, i.e., sequences of relevant system informations evolving during system executions, can look like and how they can be adequately represented by mathematical structures. Finally, we will state some requirements which are usually expected to hold on traces and translate them into properties on the mathematical structures.

These structures which will be described only very briefly in this chapter, will be specified more formally in subsequent chapters and will be called *data words*. They build the basic structures on which formulas of data logics are evaluated. Whenever a new data logic will be introduced, we will discuss how far the properties listed at the end of this chapter can be formulated with them.

A communication protocol

We assume that the protocol realized by the system requires that there is initially only a unique root process. This process creates a server and unboundedly many client processes. Moreover, it informs the clients how they can reach the server. Having been informed about the server access, the clients can send requests to the server and receive corresponding acknowledgements. The communication between the processes is *rendezvous-based*, i.e., messages are not stored in intermediate buffers, but are delivered without any delay from sender to receiver. This means that in order to carry out a sending, the execution of a send action of some process must be paired with a simultaneous receive action of some other process. This type of communication, also called *handshaking*, is well-known and used in many models of communicating concurrent systems in the literature (see, e.g., [30, 2, 77]).

An implementation model

We assume that the model implementing the protocol provides a unique ID for every process in the system. Such an assumption is very common in models implementing systems with unboundedly many processes (see, e.g., [46, 45]). Each process can create new processes, send informations to other processes and receive informations from them. A process can send a single message from a finite alphabet or a tuple consisting of a message and a process ID known by the sender. The set of IDs known by a process p consists of its own ID, the IDs of the processes created by p and all IDs sent to p. The only processes to which process p can send informations are those whose IDs are known by p. We do not make any assumptions regarding the question how a process stores the received informations internally.

By means of such a model, the protocol described above is implemented as follows: The root process creating the server and the clients sends to each client the ID of the server along with the message `serv`. In this way, the clients get enabled to access the server. Each client which is sending some request to the server is sending its ID, too. Using these IDs the server is able to respond to the corresponding clients. In Figure 2.1, we see a network where the root process has already created three processes besides the server process. An edge from one to another process symbolizes that the first one knows the ID of the latter one. Observe that in the depicted situation the ID of one client is not yet known by the server.

A trace of the system is modeled by a sequence of actions executed by (pairs of) processes. Even though processes act in parallel and may execute concurrent actions, within the trace representation the actions are put into a strict linear order where the order between concurrent actions is chosen non-deterministically. Such an *interleaving* of concurrent actions is a widely-accepted paradigm in models for parallel systems (see e.g., [30]). This approach assumes that there is only one processor on which actions are executed. Within a sequence modeling a trace, a create action is represented by a position which contains the IDs of the creating and created processes and the information that it is a create action. Similarly, a send action is modeled by a position which contains not only the information that it is a sending, but also the IDs of the sender and the receiver and possibly the sent process ID.

A mathematical representation for system traces

We represent a system trace mathematically as a word (of possibly infinite length) where each position stands for a position in the trace. The word is defined over a finite set `Prop` of propositions and a finite set `Att` of attributes. Each position can carry some propositions and, for each attribute, some value from an infinite domain. We use propositions to represent action names and messages. Attribute values serve as process IDs. For the sake of simplicity, let the message set contain a message symbol `serv` notifying that the server ID is sent, a request symbol

`req` and an acknowledgement symbol `ack`. Hence, we set $\texttt{Prop} = \{\texttt{serv}, \texttt{req}, \texttt{ack}\} \cup \{\texttt{crt}, \texttt{snd}\}$ where `crt` and `snd` stand for create and send actions, respectively. Moreover, we define $\texttt{Att} = \{\texttt{creator}, \texttt{created}, \texttt{sender}, \texttt{receiver}, \texttt{sentId}\}$ and use natural numbers as process IDs. Figure 2.2 depicts a word representation of a trace where the root process with ID 1 creates a server process with ID 2 and three client processes with IDs 3, 4 and 5. To each of the clients it sends the ID

	crt	crt	snd serv	crt	snd serv	snd req	crt	snd req	snd serv	snd ack	snd ack	snd req	snd ack
creator	1	1		1			1						
created	2	3		4			5						
sender			1		1	4		3	1	2	2	5	2
receiver			3		4	2		2	5	3	4	2	5
sentId			2		2	4		3	2			5	

Figure 2.2: The word representation of a possible trace

2 of the server along with the message `serv`. Every client sends a request along with its own ID to the server and gets an acknowledgement after some time. Note that we did not set any restriction on the order in which the server handles the requests. Although client 3 sends its request after client 4, client 3 is satisfied before client 4.

Some example properties on system traces

We now state some requirements which a system designer would usually expect from all traces of the modeled system. In addition, we also explain for each property how it is translated to a property on the word representations of traces as described above. We start with a property which does not refer to process IDs at all and proceed with very simple properties.

CS1: *After the first request there is no further process creation.*

On the trace representations, it is necessary to check that there is no `crt`-position after a `req`-position.

CS2: *Every client sending a request must be created before.*

It has to be checked that every ID which occurs as the value of the `sender`-attribute of some `req`-position occurs as the value of the `created`-attribute of some preceding `crt`-position.

CS3: *Every client sending a request gets an acknowledgement after some time.*

Every `req`-position is followed by some subsequent `ack`-position such that the `sender`-value of the first position is equal to the `receiver`-value of the second one.

CS4: *Every client receiving an acknowledgement has previously sent a request.*

Every `ack`-position is preceded by some `req`-position such that the `sender`-value of the latter position is equal to the `receiver`-value of the first one.

We now state three properties which have subtle differences. We will see in Part B that the question whether a logic is decidable or not can depend on the question which of these properties the logic is able to express.

CS5: *Whenever a client sends a request, it does not send any further requests until it receives an answer.*

Every `req`-position is followed by some `ack`-position such that the `sender`-value d of the first one is equal to the `receiver`-value of the latter one and there is no further `req`-position in between whose `sender`-value is equal to d.

CS6: *Between the creation of a client p and the receiving of the server information by p, there is no request to the server.*

It is never the case that there is a `crt`-position i and a following `serv`-position j such that the `created`-value at i is equal to the `receiver`-value at j and there is a `req`-position between i and j.

CS7: *Whenever a client p receives an acknowledgement, the server gets a request after some time and the next such request is from a client different from p.*

Every `ack`-position is followed by some `req`-position such that the `receiver`-value of the first one is different from the `sender`-value of the latter one and there is no `req`-position in between.

Observe that CS5 and CS6 talk about positions between pairs of positions where the same data value d occurs. However, while CS5 considers intermediate positions which also carry value d, CS6 does not set any conditions on the values at intermediate positions. Finally, CS7 talks about inequality conditions between pairs of positions.

Regarding the possibility that the root process may erroneously create two servers, we formulate also some stronger versions of the properties given above.

CS8: *Requests are always sent to the same server.*

There are no two `req`-positions with distinct `receiver`-values.

CS9: *Every client sending a request to a server gets an acknowledgement from the same server after some time.*

Every `req`-position is followed by some subsequent `ack`-position such that the `sender`-value of the first position is equal to the `receiver`-value of the second position and the `sender`-value of the second position is equal to the `receiver`-value of the first one.

Observe that except CS1, CS6 and CS7 all listed requirements are fulfilled by the trace represented in Figure 2.2. The reason why CS7 is not satisfied is that after responding to client 5, the server does not receive any other request.

Part A

Preliminaries

This part aims to equip the reader with some basic notations, tools and background informations on data logics in order to be prepared for the new results in Parts B and C. Besides notational machinery which will be useful throughout this work, Chapter 3 recalls some problems and automata models which will be used in (un-)decidability proofs in further parts of this work. In Chapter 4, we first introduce data words which are the basic structures on which most of the data logics we will deal with are evaluated. The chapter also contains an overview on logics and automata models proposed in the literature for these structures as well as a summary of known results on their expressivity and computational properties. Questions arising from these results will be the starting point of Part B.

Chapter 3

Basics

In this chapter, we will first define some notations which will be used throughout the entire work. Then, we will introduce some automata models, problems and techniques which will be helpful in (un)decidability proofs in later parts.

3.1 Notational Conventions

We denote the set of the natural numbers *without* 0 by $\mathbb{N}$ and for $\mathbb{N} \cup \{0\}$ we use $\mathbb{N}_0$. For two integers $i \leq j$ from $\mathbb{Z}$, the expression $[i, \ldots, j]$ represents the set $\{k \in \mathbb{Z} \mid i \leq k \leq j\}$ of all integers from i to j while $[i, \ldots)$ stands for the infinite set $\{k \in \mathbb{Z} \mid i \leq k\}$ of all integers greater or equal to i. A round bracket at an endpoint excludes the corresponding value from the represented set, for instance, we have $[-3, \ldots, 2) = \{-3, -2, -1, 0, 1\}$ and $(5, \ldots) = \{6, 7, 8, \ldots\}$.

The set of all partial mappings from some set A to some set B is denoted by $[A \rightharpoonup B]$. For a mapping $\mu \in [A \rightharpoonup B]$ and some $a \in A$ for which $\mu(a)$ is undefined, we write $\mu(a) = \bot$. Sometimes, we describe the mapping μ also by the set $\{a \mapsto b \mid \mu(a) = b \in B\}$. We denote the *domain* $\{a \mid \mu(a) \neq \bot\}$ of μ by $\mathtt{dom}(\mu)$ and its *image* $\{b \in B \mid \text{there is some } a \in A \text{ with } \mu(a) = b\}$ by $\mu(A)$. For some subset $A' \subseteq A$ and some $b \in B \cup \{\bot\}$, we let $\mu[A' \mapsto b]$ be the mapping μ' defined by: for every $a \in A'$, $\mu'(a) = b$ and otherwise, $\mu'(x) = \mu(x)$. If A' is some singleton $\{a\}$ we often write $\mu[a \mapsto b]$ and skip the curly brackets. Finally, we use $A_{\mapsto\bot}$ as an abbreviation for the partial mapping from A with empty domain.

For the sake of legibility, we often use the infix notation for binary relations. This means, for some binary relation R and two elements a and b with $(a, b) \in R$, we write aRb.

In this work, we will introduce different automata models which read linearly or partially ordered structures and are equipped with acceptance mechanisms. Likewise, we will deal with logics whose formulas are evaluated on such structures. We call the set of all structures accepted by an automaton $\mathcal{A}$ the *language* of (or *decided* by) $\mathcal{A}$ and denote it by $\mathcal{L}(\mathcal{A})$. Similarly, the set of structures satisfying a formula φ is the *language* of φ and denoted by $\mathcal{L}(\varphi)$. We say that a formula or an automaton is *equivalent* to an other formula or automaton if the corresponding languages are equal. A logic $\mathcal{L}_2$ is called to be *at least as expressive as* a logic $\mathcal{L}_1$ (written as $\mathcal{L}_1 \leq \mathcal{L}_2$) if for every formula from $\mathcal{L}_1$, there is an equivalent formula in $\mathcal{L}_2$. The logic $\mathcal{L}_2$ is *strictly more expressive than* $\mathcal{L}_1$ (written as $\mathcal{L}_1 < \mathcal{L}_2$) if $\mathcal{L}_1 \leq \mathcal{L}_2$ and $\mathcal{L}_2$ contains a formula for which there is no equivalent one in $\mathcal{L}_1$. The logics $\mathcal{L}_1$ and $\mathcal{L}_2$ are called to be *expressively equivalent* (denoted as $\mathcal{L}_1 \equiv \mathcal{L}_2$) if $\mathcal{L}_1 \leq \mathcal{L}_2$ and $\mathcal{L}_2 \leq \mathcal{L}_1$. The equivalence between two classes of automata or between a logic and an automata class is defined analogously.

For logics providing the *negation* operator $\neg$ and the *and* operator $\wedge$, we usually do not insert

the *or* operator $\vee$ into the formal syntax of formulas but use it with the obvious semantics.

3.2 Some Tools

In this section, we will introduce some auxiliary automata models and frameworks which will be utilized in our proofs in Parts B and C. We forgo the definitions of the well-known *Deterministic* and *Non-Deterministic Finite Automata* (in short, DFA and NFA) deciding regular languages.

3.2.1 Two-Way Alternating Automata

In some decidability proofs we will make use of *two-way alternating automata* on finite and infinite strings. While usual NFA read a string from left to right, a two-way automaton can move its reading head into both directions. An alternating automaton is able to split the computation into several sub computations at each step of the run. Informally, it accepts a string if all sub computations do accept. Despite these abilities the expressive power of two-way alternating automata and that of two-way alternating Büchi automata do not go beyond the expressive power of usual NFA and Büchi autmata, respectively. Early references for the definitions of two-way, alternating and two-way alternating automata and their translations to NFA are [185, 177], [56, 62] and [140]. Translations of two-way, alternating and two-way alternating Büchi automata to (one-way) Büchi automata can be found in [198], [163] and [138].

First, we will define alternating automata on finite strings, then, we will describe their extension to two-way alternating automata and finally, we will adapt these models to ω-strings, i.e. strings of infinite length. Following [185, 177, 138], we will not use additional end-markers on the input strings like in [56, 62, 140] when we define two-way automata. Furthermore, our definition of the transition relation of alternating automata is based on positive boolean formulas in the style of [138] and not like in [56, 62] where general boolean formulas are used.

Before giving the definitions of the automata models, we introduce the notions of *positive boolean formulas* and *trees*. A positive boolean formula over a finite set Γ of symbols is a formula using the symbols in $\Gamma \cup \{\top, \bot\}$ as atomic formulas and $\wedge$ and $\vee$ (and no negation operator) as logical operators. More formally, the syntax of positive boolean formulas α over Γ is defined by

$$\alpha := \top \mid \bot \mid \gamma \mid \alpha \wedge \alpha \mid \alpha \vee \alpha$$

where $\gamma \in \Gamma$. A set $\Gamma' \subseteq \Gamma$ satisfies a positive boolean formula α if α delivers the boolean value `true` when $\top$ and all symbols from Γ' occurring in α are set to `true` and all other symbols, inclusively $\bot$ are set to `false`. For instance, the formula $\gamma_1 \vee (\gamma_2 \wedge \gamma_3)$ is satisfied by the sets $\{\gamma_1, \gamma_3\}$ and $\{\gamma_2, \gamma_3\}$ but not by $\{\gamma_3\}$. The set of all boolean formulas over Γ is denoted as $\mathcal{B}^+(\Gamma)$.

A tree $T \subseteq \mathbb{N}^*$ is a (possibly infinite) non-empty subset of $\mathbb{N}^*$ where for all $vc \in T$ with $v \in \mathbb{N}^*$ and $c \in \mathbb{N}$, we have $v \in T$. Each element of T is called a *node* and the empty string ε the *root* of T. For every node v in T, all nodes $vc \in T$ with $c \in \mathbb{N}$ are called the *children* of v. Nodes with no children are called *leaves*. The length of a node determines its *level* in the tree. In particular, the root ε is at level 0. A path π in T is a subset of T such that (i) for every $vc \in \pi$ with $v \in \mathbb{N}^*$ and $c \in \mathbb{N}$, we have $v \in \pi$ and (ii) for every $v \in \pi$ either v has no children or exactly one of its children is contained in π. The length of a path π is defined as $|\pi| - 1$. The *depth* of a tree without infinite paths is defined as the length of the longest path in the tree. A Γ-labelled tree (T, ℓ) for some set Γ consists of a tree T and a labelling function $\ell : T \mapsto \Gamma$ which maps every node in T to a symbol in Γ.

Alternating Automata

An *Alternating Finite Automaton (AFA)* $\mathcal{A} = (\Sigma, S, s_0, \delta, F)$ is a five-tuple where Σ is a finite *input alphabet*, S is a finite set of *states*, s_0 is the *initial state*, $\delta : S \times \Sigma \mapsto \mathcal{B}^+(S)$ is a *transition function* and $F \subseteq S$ is a set of *accepting states*. Before defining formally what it means that $\mathcal{A}$ accept a string, we want to give some intuition about the behaviour of AFA[1]. Let $\mathcal{A}'$ be an NFA with some transition relation δ'. Suppose that both, $\mathcal{A}'$ and $\mathcal{A}$, read an input string $w = \sigma_1 \ldots \sigma_n$ and reach some position i with $1 \leq i \leq n$ in some state s. Let $\delta(s, \sigma_i) = s_1 \vee (s_2 \wedge s_3)$. Intuitively, $\mathcal{A}'$ accepts w if it accepts $\sigma_{i+1} \ldots \sigma_n$ starting in one of the states s' with $(s, \sigma_i, s') \in \delta'$. In comparison, the AFA $\mathcal{A}$ accepts w if it accepts $\sigma_{i+1} \ldots \sigma_n$ (i) starting in state s_1 or (ii) starting in state s_2 and starting in state s_3. More formally, a *run* of $\mathcal{A}$ on a string $w = \sigma_1 \ldots \sigma_n \in \Sigma^*$ of length n is an S-labelled tree (T, ℓ) of depth at most n such that (i) $\ell(\varepsilon) = s_0$, i.e., the root of T is labelled by s_0 and (ii) for every level i with $0 \leq i < n$ and every node $v \in T$ at level i, the set $\{\ell(v \cdot c) \mid v \cdot c \in c \in \mathbb{N} \text{ and } T\}$ satisfies $\delta(\ell(v), \sigma_{i+1})$. A run (T, ℓ) on w is called *accepting* if all leaves at level n are labelled by some accepting state. Note that an accepting run can contain leaves v at levels $i < n$ which are labelled by some non-accepting state s. It follows from the definition that for such nodes it must hold $\delta(\ell(v), \sigma_{i+1}) = \top$. The *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is defined as $\{w \in \Sigma^* \mid \text{there is an accepting run of } \mathcal{A} \text{ on } w\}$.

Two-Way Alternating Automata

A two-way automaton is able to move its head into both directions. Thus, a *Two-Way Alternating Finite Automaton ($AFA^{\leftrightarrow}$)* $\mathcal{A} = (\Sigma, S, s_0, \delta, F)$ differs from a (one-way) AFA only in its transition function $\delta : S \times \Sigma \mapsto \mathcal{B}^+(S \times \{-1, 0, 1\})$ which specifies not only the next states the automaton can enter, but also the direction in which the reading head of the automaton moves: -1 means that the reading head moves one step to the left, 0 means that it stays at the current position and 1 means that it moves one step to the right. For instance, if the automaton reads in some state s a symbol σ at some position i of the input string and $\delta(s, \sigma) = (s_1, -1) \vee (s_1, 1)$, it either moves one step to the left and enters state s_1 or it moves one step to the right and enters state s_2. If $i = 1$ it has to chose the second option. In an accepting run, every computation must either lead to $\top$ or must end up in an accepting state after reading the last symbol of the string and moving one step to the right.

Formally, a *run* of $\mathcal{A}$ on a string $w = \sigma_1 \ldots \sigma_n$ is a (possibly infinite) $S \times \{1, \ldots, n+1\}$-labelled tree (T, ℓ) such that (i) the root of T is labelled by $(s_0, 1)$ and (ii) for every node $v \in T$ labelled by (s, i) with $s \in S$ and $1 \leq i \leq n$, the set $\{(s', k) \mid \text{ there is some } vc \in T \text{ with } c \in \mathbb{N} \text{ and } \ell(vc) = (s', i+k)\}$ satisfies $\delta(\ell(v), \sigma_i)$. The run (T, ℓ) is *accepting* on w if it is finite and for all nodes v with $\ell(v) = (s, n+1)$, it holds $s \in F$. The language of $\mathcal{A}$ is defined as expected.

Two-Way Alternating Büchi Automata

Alternating Finite Büchi Automata (BAFA) and their two-way version $BAFA^{\leftrightarrow}$ read ω-words as inputs. The only difference to AFA and $AFA^{\leftrightarrow}$ is that they are equipped with a Büchi acceptance condition. We briefly explain the semantics of $BAFA^{\leftrightarrow}$, the semantics of the one-way version can be derived straightforwardly. The components of a $BAFA^{\leftrightarrow}$ $\mathcal{A} = (\Sigma, S, s_0, \delta, F)$ are defined exactly in the same way as the components of a $AFA^{\leftrightarrow}$. The definition of runs differs slightly since the automaton can never step out of the string to the right. A *run* of $\mathcal{A}$ on an ω-string $w = \sigma_1\sigma_2 \ldots$ is a (possibly infinite) $S \times \mathbb{N}$-labelled tree (T, ℓ) such that (i) the root of T is labelled by $(s_0, 1)$ and (ii) for every node $v \in T$ labelled by (s, i) for some s and i, the set $\{(s', k) \mid \text{there is some } vc \in T \text{ with } c \in \mathbb{N} \text{ and } \ell(vc) = (s', i+k)\}$ satisfies $\delta(\ell(v), \sigma_i)$. The run (T, ℓ) is *accepting* on w if for every infinite path π in T, the set $\{s \mid s \text{ occurs infinitely often on } \pi\} \cap F$ is non-empty.

[1] The following informal explanation is inspired by [138].

3.2.2 Counter Machines

We introduce two versions of counter machines, one with an undecidable and one with a decidable reachability problem.

A *Minsky Counter Machine (MCM)* [162] is a nondeterministic automaton equipped with counters. Formally, an MCM is a tuple (k, S, s_0, δ) where $k \geq 1$ is the *number of the counters*, S is a finite set of *states* with *initial state* s_0 and δ is a set of *transitions* of the forms $(s, \texttt{inc}_i, s')$, $(s, \texttt{dec}_i, s')$ and $(s, \texttt{ifzero}_i, s')$ with $i \in \{1, \ldots, k\}$ and $s, s' \in S$. Besides changing the state of the automaton, every transition executes an operation or a test on one of the counters. Initially, the value of every counter is 0. Informally, a transition $(s, \texttt{inc}_i, s')$ increments the value of counter i by 1. A transition $(s, \texttt{dec}_i, s')$ decrements its value by 1 and can only be applied if the value is greater than 0. Finally, a transition $(s, \texttt{ifzero}_i, s')$ performs a zero-test on counter i, i.e. it does not change any counter value and can only be applied if the value of counter i equals 0. An MCM with k counters is also called a *k-MCM*.

In order to give the formal semantics of a k-MCM $\mathcal{M} = (k, S, s_0, \delta)$, we first define the set of *configurations* of $\mathcal{M}$. A configuration $(s, v_1, \ldots, v_k) \in S \times \mathbb{N}_0^k$ of $\mathcal{M}$ consists of the current state s of $\mathcal{M}$ and the values of all k counters. A configuration $c' = (s', v'_1, \ldots, v'_n)$ results from a configuration $c = (s, v_1, \ldots, v_n)$ (written as $c \rightarrow_{\mathcal{M}} c'$) if for some i with $1 \leq i \leq k$, one of the following conditions holds:

- There is a transition $(s, \texttt{inc}_i, s') \in \delta$, $v'_i = v_i + 1$ and $v'_j = v_j$ for all $j \neq i$.
- There is a transition $(s, \texttt{dec}_i, s') \in \delta$, $v_i \neq 0$, $v'_i = v_i - 1$ and $v'_j = v_j$ for all $j \neq i$.
- There is a transition $(s, \texttt{ifzero}_i, s') \in \delta$, $v_i = 0$ and $v'_j = v_j$ for all j.

A configuration $(s, v_1, \ldots, v_k)$ is called *initial* if $s = s_0$ and $v_1 = \ldots = v_k = 0$. A sequence $c_0 \rightarrow_{\mathcal{M}} \ldots \rightarrow_{\mathcal{M}} c_n$ of configurations of $\mathcal{M}$ is called a *run* of $\mathcal{M}$ if c_0 is initial. We say that a configuration c is *reachable* by $\mathcal{M}$ if there is a run $c_0 \rightarrow_{\mathcal{M}} \ldots \rightarrow_{\mathcal{M}} c_n$ of $\mathcal{M}$ with $c_n = c$. Given a state $s \in S$, the *reachability problem* for $\mathcal{M}$ asks whether $(s, 0, \ldots, 0)$ is reachable by $\mathcal{M}$.

It is well-known that the reachability problem for 2-MCMs, i.e. MCMs with only two counters, is not decidable [162]. When proving the undecidability of a computational problem for a formalism by reduction from the reachability problem for 2-MCMs, it is a common approach to show that the formalism allows to encode sequences of 2-MCM-transitions and to check whether an encoded sequence induces a run reaching a designated configuration. Since several undecidability proofs in this work will be based on such reductions, we list here sufficient conditions implying that a transition sequence corresponds to a run reaching a configuration in some particular state with all counter values 0. To this end, let $\mathcal{M} = (2, S, s_0, \delta)$ be a 2-MCM and $s \in S$. It is easy to see that $\mathcal{M}$ reaches $(s, 0, 0)$ if and only if $s = s_0$ or there is a sequence $\tau = (s_1, \texttt{act}_1, s'_1), \ldots, (s_n, \texttt{act}_n, s'_n)$ of transitions from δ, such that the following conditions are fulfilled.

- *Consistency with respect to states*: The first state is initial, the last one is s and states of consecutive transitions are compatible with each other, that is, $s_1 = s_0$, $s_n = s$ and for all i with $1 \leq i < n$, we have $s'_i = s_{i+1}$.
- *Consistency with respect to counters*: For each counter $k \in \{1, 2\}$, there is a one-to-one mapping between increment and decrement actions for counter k such that each increment is followed by its corresponding decrement. To put it in formal terms, there is a bijection m from the set $\texttt{DECS}_\tau = \{i \mid 1 \leq i \leq n$ and $\texttt{act}_i$ is a decrement action$\}$ to the set $\texttt{INCS}_\tau = \{i \mid 1 \leq i \leq n$ and $\texttt{act}_i$ is an increment action$\}$ such that for each counter k and index $i \in \texttt{DECS}_\tau$ with $\texttt{act}_i = \texttt{dec}_k$, it holds $m(i) < i$ and $\texttt{act}_{m(i)} = \texttt{inc}_k$.

- *Consistency with respect to zero-tests*: Between an increment and the corresponding decrement of a counter, there is no zero-test for this counter, i.e., for every counter k and index $i \in \mathtt{DECS}_\tau$ with $\mathtt{act}_i = \mathtt{dec}_k$, there is no ℓ with $m(i) < \ell < i$ and $\mathtt{act}_\ell = \mathtt{ifzero}_k$.

In our undecidability proofs for satisfiability and model checking questions working with 2-MCMs, we will encode sequences of 2-MCM-transitions by data words and show that our formalisms are strong enough to express the conditions above.

We conclude this section by mentioning *Multicounter Automata (MCA)* which are restrictions of MCMs not containing any transition performing a zero-test. For every number of counters, the reachability problem is decidable for MCA [160, 134].

3.2.3 Well-Structured Transition Systems

Decidability results on infinite-state models like Timed Automata [21], Lossy Channel Systems [7], Vector Addition Systems [132] and Petri Nets [120] motivated the search for common structures in these models which explain these results. *Well-Structured Transition Systems (WSTS)* [5, 98, 135] were proposed as a general framework which incorporates these structures and provides sufficient conditions for new decidability results on infinite-state models. In this section, we present a result within the framework of WSTS which was used in the literature in many decidability proofs concerning safety properties of infinite state models.

First, we introduce some notions regarding well-quasi ordered sets. Let S be a possibly infinite set and $\preceq \subseteq S \times S$ a binary relation on S. The relation $\preceq$ is called *decidable* if there is an algorithm which for every two elements s_1 and s_2 from S decides whether $s_1 \preceq s_2$. The relation $\preceq$ is called a *well-quasi ordering* on S if it is *reflexive*, *transitive* and for every infinite sequence $s_1, s_2, \ldots$ of elements from S, there are $i < j$ with $s_i \preceq s_j$. For a subset $U \subseteq S$, we call $\uparrow U := \{s \in S \mid$ there is some $s' \in U$ with $s' \preceq s\}$ the *upward closure* of U. The set U is called *upward closed* if $U = \uparrow U$. Higman [117] proved that for every upward closed set U, there is a finite *basis* $B \subseteq U$ such that (i) for every $s \in U$, there is some $s' \in B$ with $s' \preceq s$ and (ii) the elements in B are incomparable, i.e., for every two elements $s_1, s_2 \in B$, we have that from $s_1 \preceq s_2$ it follows $s_1 = s_2$. Note that an upward closed set can have (infinitely) many different bases.

We consider *transition systems* $\mathcal{A} = (S, S_0, \longrightarrow)$ where S is a possibly infinite set of states, $S_0 \subseteq S$ is a set of *initial states* and $\longrightarrow \subseteq S \times S$ is a *transition relation* on S. By $\longrightarrow^*$ we denote the reflexive and transitive closure of $\longrightarrow$, i.e., $s_1 \longrightarrow^* s_2$ if $s_1 = s_2$ or there are states $s'_1, \ldots, s'_n$ with $n \geq 2$ such that $s_1 = s'_1 \longrightarrow s'_2 \longrightarrow \ldots \longrightarrow s'_n = s_2$. For a state $s \in S$, the set $\mathtt{Pre}(s)$ of *predecessors* of s is defined as $\{s' \in S \mid s' \longrightarrow s\}$. We say that a state $s \in S$ is *reachable* in $\mathcal{A}$ if there is some state $s_0 \in S_0$ with $s_0 \longrightarrow^* s$. The transition relation $\longrightarrow$ is called *monotonic* with respect to some well-quasi ordering $\preceq$ on S if for every three states s_1, s_2, s'_1 with $s_1 \longrightarrow s_2$ and $s_1 \preceq s'_1$, there is some s'_2 such that $s'_1 \longrightarrow^* s'_2$ and $s_2 \preceq s'_2$.

A transition system $\mathcal{A} = (S, S_0, \longrightarrow)$ is called a WSTS if

(1) there is a well-quasi ordering $\preceq$ on S and

(2) the transition relation $\longrightarrow$ is monotonic with respect to $\preceq$.

We are interested in the *coverability problem* for WSTS which, given a WSTS $\mathcal{A} = (S, S_0, \longrightarrow)$ equipped with a well-quasi ordering $\preceq$ on S and a state $s \in S$, asks whether there is some state $s' \succeq s$ reachable in $\mathcal{A}$. Before giving sufficient conditions for the decidability of this problem, we introduce the notion of *computable predecessor bases*. A WSTS $\mathcal{A} = (S, S_0 \rightarrow)$ has computable predecessor bases if there is an algorithm which computes for every $s \in S$ a basis for $\mathtt{Pre}(\uparrow\{s\}) \cup \uparrow\{s\}$.

The following theorem states two sufficient conditions for the decidability of the coverability problem.

Theorem 1 ([5, 98]). *Let $\mathcal{A} = (S, S_0, \longrightarrow)$ be a WSTS with a well-quasi ordering $\preceq$ on S. If*

(1) $\preceq$ is decidable,

(2) $\mathcal{A}$ has computable predecessor bases, and

(3) for every $s \in S$, it is decidable whether $\uparrow\{s\} \cap S_0$ is non-empty,

then the coverability problem for $\mathcal{A}$ is decidable.

Observe that item (3) of the theorem above does not follow from item (1), because S_0 can be infinite. The algorithm solving the coverability problem is based on a backward reachability analysis. It makes use of the fact that every sequence $U_0 \subseteq U_1 \subseteq \ldots$ of upward closed sets reaches a fix-point. Starting from $\uparrow\{s\}$, where s is the state for which coverability has to be checked, the algorithm computes for every $i \geq 0$, a basis for the set from which a state in $\uparrow\{s\}$ is reachable in i or less steps. Due to the property mentioned above, this procedure has to terminate at some finite basis B. Finally, thanks to item (3) of Theorem 1, the algorithm decides whether $\uparrow B$ contains an initial state by checking non-emptiness of $\uparrow\{s'\} \cap S_0$ for every $s' \in B$.

3.2.4 Transducers and the Transduction Problem

Letter-To-Letter Transducers occur as an integral part of an important automata model called data automata which will be introduced in Section 4.2.2 and will play a significant role for our new results in Part B. Moreover, in Part C we will carry out several undecidability proofs through a reduction from the *Transduction Problem*.

We first define Letter-To-Letter Transducers. A *Letter-To-Letter Transducer (LLT)* $\mathcal{T}$ is a tuple $(\Sigma, \Gamma, S, s_0, \delta, F)$ where Σ is a finite *input alphabet*, Γ is a finite *output alphabet*, S is a finite set of *states* with *initial state* s_0, $\delta \subseteq S \times \Sigma \times \Gamma \times S$ is a *transition relation* and $F \subseteq S$ is a set of *accepting states*. A transition $(s, \sigma, \gamma, s') \in \delta$ informally means that whenever the transducer $\mathcal{T}$ is in state s and reads symbol σ, it can output γ and move to state s'. Given a string $v = \sigma_1 \ldots \sigma_n \in \Sigma^*$, a *run* of $\mathcal{T}$ on v is a sequence $(s_0, \sigma_1, \gamma_1, s_1)(s_1, \sigma_2, \gamma_2, s_2) \ldots (s_{n-1}, \sigma_n, \gamma_n, s_n)$ of transitions. The run is called *accepting* if $s_n \in F$. In this case, we say that v is accepted by $\mathcal{T}$ and the string $w = \gamma_1 \ldots \gamma_m \in \Gamma^*$ is a *transduction* of $\mathcal{T}$ on v. Thus, $\mathcal{T}$ induces a transduction relation $\texttt{Rel}^{\mathcal{T}} \subseteq \Sigma^* \times \Gamma^*$ such that for every two strings v and w, we have $(v, w) \in \texttt{Rel}^{\mathcal{T}}$ if w is a transduction of $\mathcal{T}$ on v. Given a string $v \in \Sigma^*$, let $\mathcal{T}(v) = \{w \in \Gamma^* | \, (v, w) \in \texttt{Rel}^{\mathcal{T}}\}$ denote the set of all possible transductions of $\mathcal{T}$ on v. We extend the notion of transduction to languages $\mathcal{L} \subseteq \Sigma^*$ by defining $\mathcal{T}(\mathcal{L}) = \bigcup_{v \in \mathcal{L}} \mathcal{T}(v)$. Provided that $\Sigma = \Gamma$, we define in an iterative way for every $i \in \mathbb{N}_0$, the i-th transduction of $\mathcal{T}$ on $\mathcal{L}$ by $\mathcal{T}^0(\mathcal{L}) := \mathcal{L}$ and $\mathcal{T}^{i+1}(\mathcal{L}) := \mathcal{T}(\mathcal{T}^i(\mathcal{L}))$.

Büchi Letter-To-Letter Transducers (BLLT) are a straightforward adaption of LLT to ω-strings. Syntactically, they are defined in exactly the same way as LLT and differ only in their acceptance condition. A BLLT accepts an ω-string if it has a run on the string where at least one accepting state occurs infinitely often. The notions of transduction and transduction relation carry over to BLLT straightforwardly.

An instance of the *Transduction Problem TransProb* consists of a LLT $\mathcal{T}$ with input and output alphabet Σ and two NFA $\mathcal{A}$ and $\mathcal{B}$ with input alphabet Σ. In TransProb it is checked whether there is a number $i \in \mathbb{N}_0$ such that $\mathcal{T}^i(\mathcal{L}(\mathcal{A})) \cap \mathcal{L}(\mathcal{B}) \neq \emptyset$. The problem TransProb is known to be undecidable [2].

3.2.5 Post's Correspondence Problem

Emil L. Post showed that the following problem, known as *Post's Correspondence Problem (PCP)*, is not decidable [175]. An instance $(u_1, v_1), \ldots, (u_k, v_k) \in \Sigma^* \times \Sigma^*$ of PCP is a finite list of pairs of

nonempty strings over some finite alphabet Σ. Given such an instance I, PCP asks whether there is a finite sequence $i_1, \dots, i_n \in \{1, \dots, k\}$ of indices such that $u_{i_1} \dots u_{i_n} = v_{i_1} \dots v_{i_n}$. If the answer is yes, the sequence $i_1, \dots, i_n$ constitutes a *solution* and $u_{i_1} \dots u_{i_n}$ the corresponding *solution string* for I.

Chapter 4

Data Words, their Automata and Logics

First, we will first present in Section 4.1 data words which constitute the main kind of mathematical structures in the core of this thesis. Although the literature mainly considers data words where each position carries exactly one symbol from a finite alphabet and one data value from an infinite domain, following [50, 79, 81, 59, 131], we will take generalized data words with multiple symbols and data values at each position as a basis. In several works, such generalized data words are suggested as a convenient representation for traces of concurrent systems [59, 44, 45].

In Sections 4.2 and 4.3, we will introduce known automata models and logics on data words from the literature and will give an overview of the results with regard to their expressivity and computational properties. As mentioned in the introduction, we focus on formalisms where the only predicate on data values is the equality relation. Automata models and logics which subsume most of the formalisms given in the literature or which will be subject to our own analyses in the following parts of this work, will be explained and illustrated in more detail. Some of the questions arising from the results presented here will be addressed and tackled in Part B.

4.1 Data Words

Let $\mathcal{D}$ be an infinite set of data values, $\texttt{Prop}$ a finite set propositions and $\texttt{Att}$ a finite set of attributes. Throughout this work we mostly will use the set $\mathbb{N}$ of natural numbers as a representative for $\mathcal{D}$. A *data word* over $\texttt{Prop}$ and $\texttt{Att}$ is formally defined as a finite sequence $(P_1, v_1) \ldots (P_n, v_n)$ of pairs (P_i, v_i) where for every i with $1 \leq i \leq n$, the component $P_i \subseteq \texttt{Prop}$ is a set of propositions and $v_i \in [\texttt{Att} \rightharpoonup \mathcal{D}]$ is a partial attribute-value mapping. We introduce a graphical representation for data words. For instance, the data word

$$v = (\{p, q\}, \{\texttt{b} \mapsto 3\})(\{p\}, \{\texttt{a} \mapsto 2, \texttt{b} \mapsto 5\})(\{r, p\}, \{\texttt{a} \mapsto 5\})$$

of length 3 defined over the proposition set {p,q,r} and attribute set $\{\texttt{a}, \texttt{b}\}$, is represented as given in Figure 4.1. As it can be observed, the value of attribute $\texttt{a}$ at the first position and the value of attribute $\texttt{b}$ at the last position are not defined.

Given a data word $w = (P_1, v_1) \ldots (P_n, v_n)$, the set $\{1, \ldots, n\}$ of positions of w is denoted by $\texttt{pos}(w)$. For every i with $1 \leq i \leq n$, we call P_i the *set of propositions* at position i of w and denote it by $\texttt{props}(w, i)$. If $v_i(\texttt{a})$ is defined for some attribute $\texttt{a}$, it is called the *value of attribute* $\texttt{a}$ *at position* i and is denoted by $\texttt{val}(w, i, @\texttt{a})$. If for some position i and proposition p we have

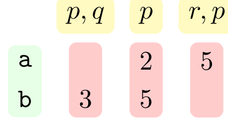


Figure 4.1: The graphical representation of v

$p \in \texttt{props}(w, i)$, we say that position i of w is *labelled* by p. For two positions $i \leq j$ of w, we denote the *subword* $(P_i, v_i) \ldots (P_j, v_j)$ of w by $w[i, \ldots, j]$. The expression $w[i, \ldots]$ represents the suffix $(P_i, v_i) \ldots (P_n, v_n)$ of w starting at position i. The *word projection* $\texttt{wrdproj}(w)$ of w is $P_1 \ldots P_n$. For some $d \in \mathcal{D}$, the maximal set of positions in w where at least one attribute has value d is called a *class* or the *d-class* of w and is denoted by $\texttt{clpos}(w, d)$. If $\texttt{clpos}(w, d) = \{i_1 < \ldots < i_k\}$ for some k, we call $P_{i_1} \ldots P_{i_k}$ a *class word* or the *d-class word* of w and denote it by $\texttt{clwrd}(w, d)$. For two positions $i \leq j$ of w, the *d-class subword* $w_d[i, \ldots, j]$ of w is the restriction of $w[i, \ldots, j]$ to the d-class of w. In the following, we illustrate on v the introduced notions and notations:

- $\texttt{pos}(v) = \{1, 2, 3\}$ constitutes the set of positions of v
- the set of propositions at position 3 of v is $\texttt{props}(w, i) = \{r, p\}$
- the value of attribute b at position 2 is $\texttt{val}(v, 2, \texttt{@b}) = 5$
- the first position of v is labelled by p and q
- $v[2, \ldots, 3] = (\{p\}, \{\texttt{a} \mapsto 2, \texttt{b} \mapsto 5\})(\{r, p\}, \{\texttt{a} \mapsto 5\})$
- the word projection of v is $\texttt{wrdproj}(v) = \{p, q\}\{p\}\{r, p\}$
- the 5-class of v is $\texttt{clpos}(v, 5) = \{2, 3\}$
- the 5-class word of v is $\texttt{clwrd}(v, 5) = \{p\}\{r, p\}$, and
- the 5-class subword of v is $v_5[1, \ldots, 2] = (\{p\}, \{\texttt{a} \mapsto 2, \texttt{b} \mapsto 5\})$

A data word w is called *propositionless* if for every position i, the set $\texttt{props}(w, i)$ is empty. If $|\texttt{Att}| = m$ and at every position i in w, the values of all attributes in Att are defined, w is called *complete*, *Att-complete* or *m-complete*. In the graphical representation of 1-complete data words we often skip the attribute name; for instance, the 1-complete data word

$$(\{r, q\}, \{\texttt{a} \mapsto 3\})(\{q\}, \{\texttt{a} \mapsto 2\})(\{r, p\}, \{\texttt{a} \mapsto 5\})$$

will be visualized as given in Figure 4.2. If the attribute value at some position i of a 1-complete

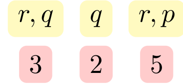


Figure 4.2: The graphical representation of a 1-complete data word

data word is d we often say that position i *carries* value d. The literature mainly considers 1-complete data words where each position is labelled by exactly one proposition. In this work, we call such structures *simple data words*. Sequences of proposition sets without any data values are

just called *words*. To have a clear distinction, we reserve the term *string* for sequences of symbols from some finite alphabet.

A *data ω-word* is a data word of infinite length. All notions and notations introduced above can be adapted to data ω-words straightforwardly.

4.2 Automata for Data Words

This section is devoted to known automata models on data words and their properties with respect to expressivity and complexity. Since Register and Data Automata will play a significant role in our studies in later parts of this work, their syntax and semantics will be explained in more detail. For convenience, we will define the automata models over a finite input alphabet Σ (and possibly some output alphabet Γ) of symbols. An input symbol is usually used to represent a finite set of propositions at a data word position. In the literature, these automata often appear as auxiliary tools into which logical formulas are converted when the computational properties of data logics are analyzed. Although they are defined on data words carrying only a single data value at each position, it will become clear in Part B how they can be used to solve the satisfiability of logical formulas on general data words.

4.2.1 Register Automata

Register Automata were first introduced in [124] and several variants were studied intensively in [180, 169, 35, 39, 196]. While they were originally designed for propositionless 1-complete data words, i.e., sequences of data values, following [39], we present here a straightforward generalization for data words with propositions.

A Register Automaton (RA) is a finite automaton equipped with finitely many registers in which data values of input data words can be stored. In each step, the automaton can compare register contents with the data value at the current position of the input data word. Based on this comparison, the current state of the automaton and the propositions at the current position, the automaton can store the current data value in one of its registers and change its state. Formally, an RA $\mathcal{A} = (\Sigma, R, S, s_0, \delta, F)$ consists of a finite *input alphabet* Σ, a finite set R of *registers*, a finite set S of *states* with *initial state* $s_0 \in S$, a set $F \subseteq S$ of *accepting* states and a set δ of *transitions* of the forms $(r, s, \sigma) \to s'$ and $(s, \sigma) \to (r, s')$ where $r \in R$, $s, s' \in S$ and $\sigma \in \Sigma$. Starting in the initial state with empty registers, an RA reads the input word from left to right. The automaton can execute a transition $(r, s, \sigma) \to s'$ at some position i if its current state is s, the data value at the i-th position is equal to the current data value in register r, and the set of propositions at position i is represented by σ. As a result of such a transition, the automaton changes its state to s' and goes one position further. It can perform a transition $(s, \sigma) \to (r, s')$ at position i if its current state is s, the data value at the i-th position is not contained in any register and the proposition set of position i is represented by σ. If these conditions hold and the automaton executes such a transition, it changes its state to s', puts the value of position i into register r and steps one position further.

For the formal definition of the semantics of RA, let $\mathcal{A} = (\Sigma, R, S, s_0, \delta, F)$ be an RA with $\Sigma = 2^{\texttt{Prop}}$ for some proposition set $\texttt{Prop}$. A *configuration* (s, λ) of $\mathcal{A}$ is a pair consisting of a state s and a partial *register assignment* $\lambda \in [R \rightharpoonup \mathcal{D}]$. A configuration (s', λ') results from (s, λ) by reading a data word position $\begin{smallmatrix} P \\ d \end{smallmatrix}$ (written as $(s, \lambda) \xrightarrow{\begin{smallmatrix} P \\ d \end{smallmatrix}} (s', \lambda')$) with $P \subseteq \texttt{Prop}$ and $d \in \mathcal{D}$ if (i) there is a transition $(r, s, P) \to s' \in \delta$, $\lambda(r) = d$ and $\lambda' = \lambda$, or (ii) there is a transition $(s, P) \to (r, s') \in \delta$, $\lambda(r') \neq d$ for all $r' \in R$ and $\lambda' = \lambda[r \mapsto d]$. A *run* of $\mathcal{A}$ on a 1-complete data word $w = \begin{smallmatrix} P_1 \\ d_1 \end{smallmatrix} \cdots \begin{smallmatrix} P_n \\ d_n \end{smallmatrix}$ is a sequence $\tau = (s_0, \lambda_0) \dots (s_n, \lambda_n)$ of configurations such that (i) $\lambda_0 = R_{\mapsto\perp}$, i.e., initially, all

register values are undefined, and (ii) for every i with $0 \leq i < n$, we have $(s_i, \lambda_i) \xrightarrow[d_{i+1}]{P_{i+1}} (s_{i+1}, \lambda_{i+1})$. The run τ is *accepting* if $s_n \in F$. The data word w is *accepted* by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. Observe that for each run $(s_0, \lambda_0) \dots (s_n, \lambda_n)$ of $\mathcal{A}$ on w and each position i of w, the value d_i is contained in the image of λ_i, that is, the data value of the recent position is always contained in some register of the current configuration. Moreover, all register assignments in runs are injective which means that two registers can never hold the same data value at the same time.

It is not hard to see that due to the finiteness of its register set, the expressive power of Register Automata is quite restricted. An RA cannot check, for instance, that all data values of the input word are pairwise distinct (this insight follows from Proposition 4 in [124]). Nevertheless, there are many interesting properties decidable by RA. We give two examples:

Example 1. The following property on 1-complete data words can be checked by an RA with only one register.

Every two consecutive positions have distinct data values.

At each position, the RA checks via a transition of the form $(s, \sigma) \to (r, s')$ that the current data value d of the input word is not contained in its register and puts d into its register. □

Example 2. To check the following property, an RA needs only two registers.

There are three positions $i \leq j \leq k$ such that i is labelled by proposition p, j is labelled by proposition p', k is labelled by proposition p'', and positions i and k carry the same data value.

Let r_1 and r_2 be the two registers of the RA. The second register serves as an auxiliary register storing irrelevant data values. The automaton "guesses" the two positions $i \leq k$ which are supposed to be the p and p''-position, respectively. At each position less or equal to i, it uses r_1 as a storage for the current data value, that is, if a current data value d is not contained in any register, d is put into r_1, and if the current value is in r_1, the registers are left unchanged. If the automaton reaches position i, it assures that i is labelled by p. At each position between i and k, it stores the current input value into r_2 if the value is not equal to the content of r_1. Additionally, it checks that proposition p' occurs at some position j with $i \leq j \leq k$. At position k, it assures that the current data value is stored in register r_1. □

The reader may have recognized the fact that the current input data value has always to be stored in some register what makes it a bit uncomfortable to describe algorithms on RA. For instance, the automaton in Example 2 needs register r_2 basically only for technical reasons. Therefore, in subsequent chapters where we explain at an informal level the construction or behaviour of a Register Automaton, we will take a relaxed, but expressively equivalent RA-version as a basis. This version allows the simultaneous containment of the same data value in different registers and has transitions of the forms $(E, s, \sigma) \to s'$ and $(U, s, \sigma) \to (r, s')$ where E and U are sets of registers, r is a single register or of the form $\perp$, s and s' are states and σ is an input symbol. Informally, a transition of the first type checks that the current data value is equal to the contents of all registers in E. A transition of the second form assures that the current value d is unequal to the value of each register in U and stores d into register r, unless $r = \perp$. Obviously, transitions of the forms $(r, s, \sigma) \to s'$ and $(s, \sigma) \to (r, s')$ of an usual RA with register set R can be simulated by transitions of the forms $(\{r\}, s, \sigma) \to s'$ and $(R, s, \sigma) \to (r, s')$, respectively, of a relaxed RA. There is also an easy translation from a relaxed RA $\mathcal{A}$ with k registers to an usual RA $\mathcal{A}'$ with $k+1$ registers and an exponentially bigger state space compared to the state set of $\mathcal{A}$. The automaton $\mathcal{A}'$ maintains in its state an equivalence relation on the register set so that all registers in an equivalence class simulate registers which are carrying the same data value. The additional register of $\mathcal{A}'$ is needed to store incoming data values which are not stored by $\mathcal{A}$.

The Büchi, two-way and alternating versions of Register Automata are defined in the obvious way. We only emphasize on the main differences. Büchi Register Automata read data ω-words and differ from their non-Büchi versions only in the acceptance condition. A data ω-word is accepted by a Büchi Register Automaton if the automaton has a run on the word which infinitely often visits an accepting state. An alternating Register Automaton can split runs into sub runs such that all of them have to accept so that the input word can be accepted. The transitions of two-way Register Automata contain additional information on the direction of the next step of the reading head (for the formal definition of alternation and two-wayness for classical automata, see Section 3.2.1). We add to the acronym RA a preceding "A" to denote the alternating version of Register Automata and we add a "B" in case of Büchi Register Automata. Two-wayness is symbolized by the superscript $^{\leftrightarrow}$. Given a Register Automata version C and a $k \geq 1$, we denote by k-C the restriction of C to k registers. For instance, by 2-BARA$^{\leftrightarrow}$ we mean the class of Two-Way Alternating Büchi Register Automata with two registers.

In [180], [82, 83] and [72], complexity analyses on different versions of Register Automata are carried out. The version considered in [180] is the original model introduced in [124] which works on propositionless 1-complete data words, i.e., sequences of data values. The model in [82, 83] and [72] is an extension on simple data words, that is, 1-complete data words carrying a single proposition at each position. For the original model, it is shown that non-emptiness is NP-complete [180] on finite data words. Compared to this, the problem is PSpace-complete for the model in [82, 83] on finite and infinite data words. Two-way Register Automata are strictly more expressive than their one-way version, since they can test that all data values occurring in a data word are pairwise distinct, a property which is not expressible with one-way (non-alternating) Register Automata [124]. However, it is shown that for the two-way version of the model in [82, 83], non-emptiness is undecidable already on finite data words and in case of one register [72]. In spite of this, non-emptiness for the (one-way) alternating version with one register on finite data words is decidable with non-primitive recursive complexity [82, 83]. The problem becomes undecidable if a further register is added or if data words of infinite length are considered [82, 83].

4.2.2 Data Automata

Just as Register Automata, also *Data Automata* [41] are defined on 1-complete data words. Before presenting their definition, we introduce the notion of *marked word projections* of 1-complete data words. Remember that in Section 4.1 we have defined the word projection of a data word as the sequence of proposition sets which we get after discarding all data values (along with attributes). A marked word projection contains for every position the additional information whether the position carries the same data value as the next position or not. Formally, given a 1-complete data word $w = \genfrac{}{}{0pt}{}{P_1}{d_1} \cdots \genfrac{}{}{0pt}{}{P_n}{d_n}$ over some proposition set Prop, the *marked word projection* $\texttt{mwrdproj}(w)$ of w is defined as $(P_1, b_1) \dots (P_n, b_n) \in 2^{\texttt{Prop}} \times \{\perp, \top\}$ where for every i with $1 \leq i \leq n$, $b_i = \top$ if and only if position $i+1$ exists and $d_i = d_{i+1}$.

A *Data Automaton (DA)* $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ is a pair consisting of a *base automaton* $\mathcal{B}$ and a *class automaton* $\mathcal{C}$. The base automaton is a non-deterministic Letter-To-Letter Transducer (LLT) as defined in Section 3.2.4 with input alphabet $\Sigma \times \{\perp, \top\}$ for some finite Σ and some output alphabet Γ. The class automaton is a classical NFA over Γ. The set Σ is regarded as the input alphabet of $\mathcal{A}$. To put it briefly, a 1-complete data word w is accepted by a $\mathcal{A}$ if $\mathcal{B}$ accepts $\texttt{mwrdproj}(w)$ after transforming the propositional part of w and $\mathcal{C}$ accepts all class words (for the definition of classes, see Section 4.1) of the resulting data word. To be more precise, let $w = \genfrac{}{}{0pt}{}{P_1}{d_1} \cdots \genfrac{}{}{0pt}{}{P_n}{d_n}$ be a 1-complete data word over some proposition set Prop and $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ a DA with input alphabet $\Sigma = 2^{\texttt{Prop}}$. The data word w is accepted by $\mathcal{A}$ if

- there is some $v = \gamma_1 \ldots \gamma_n$ with $\mathtt{mwrdproj}(w)\mathrm{R}^{\mathcal{B}}v$, i.e., v is a transduction of $\mathcal{B}$ on the marked word projection of w, and
- $\mathcal{C}$ accepts all class words of $\frac{\gamma_1}{d_1} \cdots \frac{\gamma_n}{d_n}$.

We demonstrate the expressive power of DA by some examples. Checking the property which says that all consecutive positions of the input data word carry different values and which is captured by Register Automata as shown in Example 1, is obviously an easy job for DA, because base automata get the information about the equality of consecutive data values in their input. The following example shows that DA can even check properties which are not captured by Register Automata:

Example 3. Remember from Section 4.2.1 that Register Automata are not able to check the following property:

All data values of the input word are pairwise distinct.

This property can easily be checked by a DA. Note that propositions do not play any role for this property. The base automaton accepts all input words and outputs at all positions an arbitrary symbol. The class automaton only has to check that all input words have length 1. □

Since class automata navigate through class words, it is easy to construct a DA checking regular constraints within class words:

Example 4. Let us consider the following property:

Every position labelled by proposition p is followed by some position labelled by proposition q and carrying the same data value.

The base automaton of a DA checking this property accepts all input words and outputs at each position a symbol representing the set of propositions of the position. The class automaton has to check the regular property that every p-position is followed by some q-position. □

At first glance, it might not seem obvious that DA are able to check some involved relationships between different classes. The next example demonstrates that DA can "look beyond" single classes up to a certain degree. In Section 6.5, we will see that DA can capture even more complicated properties.

Example 5. We take the following property:

There is a position which is labelled by p and followed by a position labelled by q such that both positions carry distinct data values.

The class automaton checks that the input word has at least one p- and a following q-position. If this is the case, it outputs at exactly one such p-position a designated symbol, let us say #, and at exactly one following q-position another designated symbol, let us say \$. The outputs at other positions are irrelevant, except that the designated symbols must not be used. The class automaton just checks that # and \$ do not occur within the same input word. □

In [41], the authors give the definition of *Büchi Data Automaton (BDA)*, that is, an adaption of DA for data ω-words. A BDA $\mathcal{A} = (\mathcal{B}, \mathcal{C}, \mathcal{C}_\omega)$ consist of

- a base automaton $\mathcal{B}$ which is a non-deterministic Büchi Letter-To-Letter Transducer (BLLT) with some input alphabet $\Sigma \times \{\bot, \top\}$ and some output alphabet Γ,

- a class automaton $\mathcal{C}$ over Γ for classes of finite length and
- a class automaton $\mathcal{C}_\omega$ over Γ for classes of infinite length.

The class automaton $\mathcal{C}$ is a classical NFA and the class automaton $\mathcal{C}_\omega$ is a classical Büchi automaton. A 1-complete data ω-word $w = \frac{P_1}{d_1}\frac{P_2}{d_2}\cdots$ over some proposition set $\mathtt{Prop}$ is accepted by a BDA $\mathcal{A} = (\mathcal{B}, \mathcal{C}, \mathcal{C}_\omega)$ with input alphabet $\Sigma = 2^{\mathtt{Prop}}$ if

- there is some transduction $v = \gamma_1\gamma_2\ldots$ of $\mathcal{B}$ on $\mathtt{mwrdproj}(w)$,
- $\mathcal{C}$ accepts all class words of $\frac{\gamma_1}{d_1}\frac{\gamma_2}{d_2}\cdots$ which have *finite* length and
- $\mathcal{C}_\omega$ accepts all class words of $\frac{\gamma_1}{d_1}\frac{\gamma_2}{d_2}\ldots$ which have *infinite* length.

In [41], it is shown that the non-emptiness problem for both, DA and BDA, is decidable, however, an elementary upper is not known. Remember that in the definition of DA borrowed from [41], the base automaton reads the marked word projection of the input word, i.e., at each position it "sees" whether the data value of the current position is different from the data value of the next one or not. In [39], it is shown that DA in which the base automaton reads only the (unmarked) word projection of the input word, are expressively equivalent to usual DA. Moreover, the authors prove that every RA can be converted into an equivalent DA[1].

4.2.3 Further Automata Models

Recall that during a run, a Register Automaton cannot store any data value which does not appear in the input word. Moreover, due to the finiteness of its register set, a one-way Register Automaton can "remember" only a finite amount of the data values appeared in the "history" of a run. To overcome these limitations several extensions are introduced in [127, 196, 197]. In [127], Register Automata are allowed to store in each step non-deterministically an arbitrary data value. We call this form of Register Automata *Guessing Register Automata (GRA)*. In [196], *Fresh-Register Automata (FRA)* are considered, which can check that an incoming data value is different not only from the current content of the registers, but from all data values seen so far in the current run. Since both automata models can, for instance, test that the last data value of an input word differs from all previous ones, they are strictly more expressive than usual RA. An extension of FRA, called *History-Register Automata (HRA)*, is presented in [197]. Besides usual registers, an HRA is equipped with finitely many unbounded history sets. In each step an HRA can ask whether an incoming data value is contained in some register or history set and can update and reset registers and history sets. The non-emptiness problem of all three extensions of Register Automata is decidable. In addition, it is shown that in terms of expressivity FRA are subsumed by DA and HRA and that the latter are incomparable with DA. Extensions of Register Automata by pushdown stacks or on trees are studied in [65, 126, 123].

In [39], a model called *Class Memory Automata (CMA)* is introduced which has the same expressive power as DA. While acceptance of an input word by a DA depends on several runs on the word (a single run of the base automaton and multiple runs of the class automaton), a CMA simulates all runs of a DA within a single run. Intuitively, a state of a CMA is a composition of a state of the base automaton and several states of the class automaton belonging to runs on classes. In the context of designing learning algorithms for data languages, a restriction of DA called *Transparent Data Automata (TDA)* is studied in [74]. Just like a DA, a TDA consist of a

[1]The authors of [39] have recognized that there is a bug in the proof of this result. During the completion of this work they were fixing the bug and were convinced that the result holds.

base automaton and a class automaton. The difference is that the base automaton is a usual NFA which only has to accept the word projection of an input word. Furthermore, the language of the base automaton has to be included in that of the class automaton. As checking non-emptiness of TDA reduces to checking non-emptiness of the class automaton, non-emptiness for TDA is NL-complete. It is not hard to prove that the class of languages decided by TDA is strictly included in the class of languages of DA. The models TDA and RA are not comparable with respect to expressivity. Motivated by the design of an automata model capturing XPath, the class of *Extended Data Automata (EDA)* is introduced in [42][2]. They differ from DA only with regard to the class automaton part. While the class automaton of a DA reads class words, the one of an EDA can also see positions outside a class. To be more precise, the class automaton reads for every class C, the entire word where all positions belonging to C are marked by a special symbol. While usual DA capture RA, the extension EDA subsume even 1-ARA. It is not surprising that the non-emptiness problem for EDA is not decidable [42].

In [158], the authors introduce *Data Walking Automata (DWA)*, a two-way automata model. At each position of the data word, a DWA can not only "step" to the direct predecessor or successor position, but also to the predecessor or successor in the class of the current data value. The non-emptiness problem for DWA is as hard as the same problem for DA. Expressivity-wise, DWA are strictly included in DA and are subsumed by 1-RA$^{\leftrightarrow}$ but not comparable with RA [158].

Recall that a Register Automaton memorizes data values and not word positions. With other words, it "forgets" the positions the data values in its registers originate from. In [169], *Pebble Automata (PA)* are introduced which can place *pebbles* on word positions and refer to these positions (and the corresponding data values) during their runs. The pebbles are placed according to a stack discipline. Each new pebble is placed at the initial word position and serves as the current head of the automaton. The authors also consider *Weak Pebble Automata (WPA)* where new pebbles are placed at the current position. In terms of expressivity, PA are strictly more expressive than their weak versions [169] and DWA [158], but are incomparable with RA [191]. While non-emptiness for both PA-versions is undecidable [169], the problem is shown to be decidable for a restriction version called *Top-View Weak Pebble Automata (TWPA)* [192]. In the latter model, equality tests can only be performed between the data values at the positions of the two most recently placed pebbles. Decidability is shown by reduction to 1-ARA.

Variable Finite Automata (VFA) [109] constitute a simple extension of classical NFA to words with data values. In this model, transitions can be labelled by data values and by variables which serve as placeholders for arbitrary data values. It is distinguished between *bounded* and *free* variables. Once assigned to a data value, bounded variables cannot change their value whereas free variables can always be assigned to fresh values. The non-emptiness problem for VFA is NL-complete, thus, its complexity does not go beyond that of classical NFA. However, its expressive power is quite limited. For instance, it cannot decide the language of data words where the data value of every odd position is equal to the data value of the consecutive position. This language can easily be decided by an RA with two registers. On the other hand, VFA can check that the value of the last position is different from all other values in the data word. As this property cannot be checked by RA, the expressive power of VFA is not comparable to that of RA. However, VFA are strictly less expressive than GRA. Due to the fact that DA cannot handle constants, VFA and DA are not comparable with respect to expressivity. If constants are skipped, VFA-languages are strictly included in the class of DA-languages.

In Figure 4.3, we give an overview of the relative expressivity of the automata models mentioned in this chapter. A dashed line from a lower to a higher logic indicates that expressivity-wise the latter model captures the first one. If the line is solid it indicates that the inclusion is strict. A

[2] Although the model introduced in [42] is called *Class Automata*, we use here the term *Extended Data Automata* to avoid naming conflicts with class automata which constitute a sub component of Data Automata.

dotted line between two models signalizes incomparability. The labels at the edges are references to the literature the results stem from. Note that some works consider Register Automata starting with an *initial* register assignment containing data values which serve as constants. Similarly, as mentioned above, VFA can deal with constant data values. The other introduced models, however, are not equipped with mechanisms dealing (directly) with constants. To make the comparison between the models easy, in Figure 4.3, we assume that the depicted Register Automata versions do not have initial register assignments and VFA do not contain constants.

In Figures 4.4 and 4.5, we list results on the complexity of the non-emptiness problem of the automata models. A "c" after a complexity class signalizes that the problem is complete for the class. It has to be mentioned that for questions which are left open in the figures, we did not find any results in the literature.

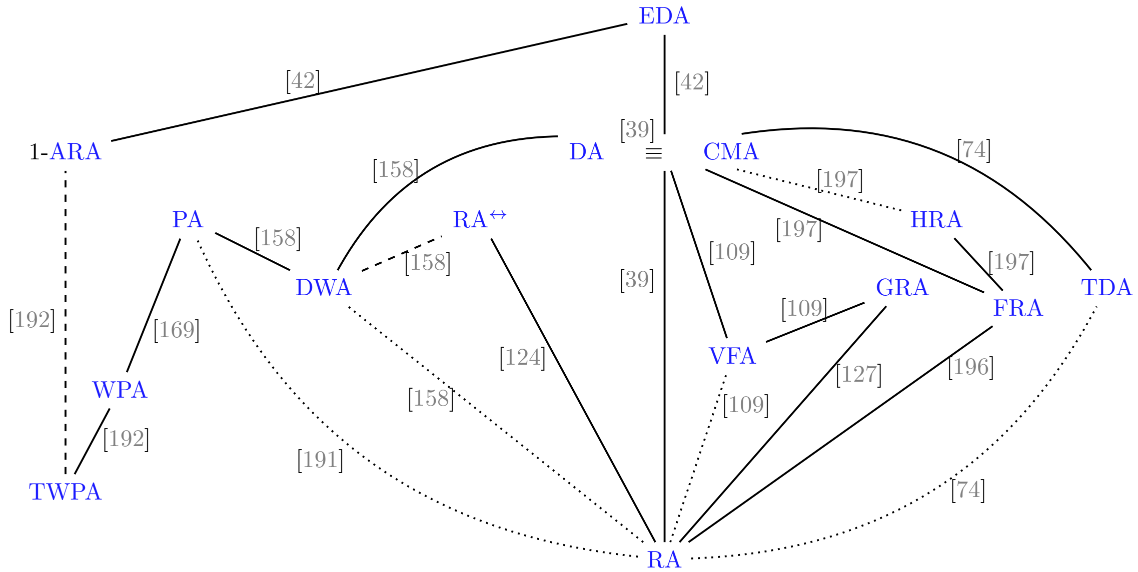


Figure 4.3: Comparison of automata models with respect to expressivity (a dashed line from a lower to a higher logic indicates that expressivity-wise the latter model captures the first one; if the line is solid it indicates that the inclusion is strict; a dotted line between two models signalizes incomparability; the considered Register Automata versions do not have initial register assignments and the considered VFA version does not contain constants)

4.3 Logics for Data Words

In this section, we will give an overview of so-called data logics, i.e., logics on data words, and summarize known results on their expressivity and complexity. Due to our investigations in Parts B and C, particular emphasis will be set on logics based on first order concepts, temporal navigation and regular expressions. While most of the data logics given in the literature are defined on simple data words, we will present their generalizations on data words with multiple values at each position.

In this section as well as in the entire main part of this work, the semantics of logics will be explained at a more informal level, but the precise definitions can be found in the Appendix (Appendix A). We assume that the reader is familiar with classical First Order Logic (FO), Linear-

	on finite data words	on infinite data words
RA	PSPACE-c [83]	PSPACE-c [83]
1-RA$^{\leftrightarrow}$	undecidable [72]	undecidable [72]
1-ARA	decidable, non-prim. rec. [83]	undecidable [83]
2-ARA	undecidable [83]	undecidable [83]
DA	decidable [41]	decidable [41]
CMA	decidable [39]	
TDA	NL-c [74]	
EDA	undecidable [42]	undecidable [42]
DWA	decidable [158]	
PA	undecidable [169]	undecidable [169]
WPA	undecidable [169]	undecidable [169]
TWPA	decidable, non-prim. rec. [192]	

Figure 4.4: Complexity of the non-emptiness problem for automata on simple data words

	on finite data words	on infinite data words
RA	NP-c [180]	in PSPACE [83]
GRA	decidable [127]	
FRA	decidable [196]	
HRA	decidable, non-prim. rec. [197]	
VFA	NL-c [109]	NL-c [109]

Figure 4.5: Complexity of the non-emptiness problem for automata on sequences of data values

Time Temporal Logic (LTL) and its extension PLTL by past operators. Therefore, we leave out their formal definitions on usual string.

4.3.1 First Order Logic

In [169] and [41], *First Order Logic* on 1-complete data values is investigated. While we comply with the notation used in these works, we present an extension to general data words and notate the logic by *FO*$^{\sim}$. Besides existential and universal quantifiers over position variables, the classical unary relations for propositions and the binary relations $=$, $\texttt{Suc}$ and $<$ on word positions, the logic contains the binary relations $\sim$ and $\texttt{Suc}_{\sim}$. The interpretation of the relations mentioned first is as usual: the atomic formula $p(x)$ for a proposition p and a position variable x holds on a data word if the position assigned to x is labelled by p; the formula $x = y$ holds if the x-postion is equal to the y-position; $\texttt{Suc}(x, y)$ is true if the y-position is the immediate successor of the x-position; $x < y$ holds if the y-position is strictly greater than the x-position. The relations $\sim$ and $\texttt{Suc}_{\sim}$ are used in the forms $x.\texttt{@a}{\sim}y.\texttt{@b}$ and $\texttt{Suc}_{\sim}(x.\texttt{@a}, y.\texttt{@b})$, respectively, for position variables x and y and attributes $\texttt{a}$ and $\texttt{b}$. The first formula holds on a data word if the data value of attribute $\texttt{a}$ at the x-position is equal to the data value of attribute $\texttt{b}$ at the y-position. The second formula logically implies the first one and additionally requires that there is no position z between x and y such that attribute $\texttt{b}$ at the z-position has the same value as attribute $\texttt{a}$ at the x-position.

We illustrate the semantics of the logic by expressing some properties from our introductory client-server example from Chapter 2. The full formal semantics is given in the Appendix (Section A.1).

Example 6. We first take Property CS8:

Requests are always sent to the same server.

The property can be expressed by the following $\mathrm{FO}^{\sim}$-formula:

$$\neg\exists x\exists y(\mathtt{req}(x) \wedge \mathtt{req}(y) \wedge x.\mathtt{@receiver} \not\sim y.\mathtt{@receiver})^{3}$$

We now consider property CS9:

Every client sending a request to a server gets an acknowledgement from the same server after some time.

The property can be expressed as follows:

$$\forall x[\mathtt{req}(x) \rightarrow \exists y(y > x \wedge \mathtt{ack}(y) \wedge x.\mathtt{@sender} \sim y.\mathtt{@receiver} \wedge x.\mathtt{@receiver} \sim y.\mathtt{@sender}])$$

□

If $\mathrm{FO}^{\sim}$ is evaluated on data words with a single attribute, for convenience, we usually skip the attribute name in formulas: for instance, we write $\exists x\exists y\ x \sim y$ instead of $\exists x\exists y\ x.\mathtt{@a} \sim y.\mathtt{@a}$.

The extensions *$MSO^{\sim}$* and *$EMSO^{\sim}$* are defined analogously to MSO and EMSO on strings. The logic $\mathrm{MSO}^{\sim}$ allows universal and existential set quantifications of the forms $\forall X$ and $\exists X$ and atomic formulas $X(x)$ where X is a set and x a position variable. The atomic formula $X(x)$ means that the position assigned to x is contained in the set assigned to X. By $\mathrm{EMSO}^{\sim}$ we denote the fragment of $\mathrm{MSO}^{\sim}$ consisting of formulas of the form $\exists X_1 \ldots \exists X_n \varphi$ for $n \geq 0$ where each X_i is a set variable and φ does not contain any set quantification.

Given a $k \geq 1$, the restriction of a logic $\mathcal{L} \in \{\mathrm{MSO}^{\sim}, \mathrm{EMSO}^{\sim}, \mathrm{FO}^{\sim}\}$ to formulas which contain at most k different position variables is denoted by $\mathcal{L}_k$. For $\mathcal{L} \in \{\mathrm{MSO}^{\sim}, \mathrm{EMSO}^{\sim}, \mathrm{FO}^{\sim}\} \cup \{\mathrm{MSO}^{\sim}_k, \mathrm{EMSO}^{\sim}_k, \mathrm{FO}^{\sim}_k \mid k \geq 1\}$ and $\mathcal{O} \subseteq \{\mathtt{Suc}, <, \mathtt{Suc}_{\sim}\}$, the logic $\mathcal{L}(\mathcal{O})$ stands for the restriction of $\mathcal{L}$ to formulas in which the relations in $\{\mathtt{Suc}, <, \mathtt{Suc}_{\sim}\} \backslash \mathcal{O}$ are not allowed. For instance, by $\mathrm{EMSO}^{\sim}_2(\mathtt{Suc})$ we mean the restriction of $\mathrm{EMSO}^{\sim}$ where at most 2 position variables are allowed and the relations $<$ and $\mathtt{Suc}_{\sim}$ are skipped.

The literature mainly considers $\mathrm{FO}^{\sim}$ on data words with a single proposition at each position. Unless stated otherwise, the following results belong to such structures. The satisfiability problem for $\mathrm{EMSO}^{\sim}$ is not decidable [43]. Undecidability holds even for the first-order fragment $\mathrm{FO}^{\sim}_3(\mathtt{Suc})$ with three position variables on finite 1-complete data words. Therefore, the authors in [43] focus on the fragment of $\mathrm{EMSO}^{\sim}$ with only two position variables. It turns out that while on 2-complete data words, satisfiability for $\mathrm{FO}^{\sim}_2(\mathtt{Suc}, <)$ remains undecidable, the problem becomes decidable for $\mathrm{EMSO}^{\sim}_2(\mathtt{Suc}, <)$ in the 1-complete case. Decidability is obtained by showing that every formula can be converted into an equivalent DA for which non-emptiness is decidable. The complexity of the problem is as hard as non-emptiness for Multicounter Automata (MCA) for which no elementary upper bound is known [106, 134, 160]. The decidability result for $\mathrm{EMSO}^{\sim}_2(\mathtt{Suc}, <)$ carries over to infinite data words by using BDA as the target automata model [43]. Satisfiability for the fragment $\mathrm{FO}^{\sim}_2(<)$ on finite 1-complete data words is NExpTime-complete [43]. On the same kind of structures the problem is in 2NExpTime for $\mathrm{FO}^{\sim}_2(\mathtt{Suc})$, even in the case of multiple propositions at each position [170]. However, it is unclear whether this upper bound is optimal. The best known lower bound is NExpTime [94].

We conclude by mentioning some relationships between $\mathrm{FO}^{\sim}$-like logics and automata. We already mentioned that on finite and infinite 1-complete data words $\mathrm{EMSO}^{\sim}_2(\mathtt{Suc}, <)$-formulas can be converted into equivalent DA and BDA, respectively. In [43], it is moreover shown that DA are logically characterized by $\mathrm{EMSO}^{\sim}_2(\mathtt{Suc}, \mathtt{Suc}_{\sim})$ on finite 1-complete data words. From the last results it is obtained that the latter logic is decidable on these structures. Finally, it follows from [169] that on the same kind of structures full $\mathrm{FO}^{\sim}$-formulas can be converted into equivalent PA.

[3] Note that we use $x.\mathtt{@a} \not\sim y.\mathtt{@b}$ as an abbreviation for $\neg(x.\mathtt{@a} \sim y.\mathtt{@b})$.

4.3.2 Temporal Logic

The logic *Freeze LTL ($LTL^{\Downarrow}$)* [82] basically extends classical LTL [174] by the ability to store data values in freeze registers and compare them with other data values occurring in the data word. That is, in contrast to $FO^{\sim}$-formulas which memorize positions by variables, $LTL^{\Downarrow}$-formulas memorize data values by freeze registers. Given a set Prop of propositions, a set Att of attributes and an infinite supply R of freeze registers, the syntax of $LTL^{\Downarrow}$-formulas is defined as follows:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \Downarrow^r_{@\mathtt{a}}.\varphi \mid \Uparrow^r_{@\mathtt{a}} \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi \mid \mathbf{X}^{\leftarrow}\varphi \mid \varphi\mathbf{U}^{\leftarrow}\varphi$$

with $r \in R$, $p \in \mathtt{Prop}$ and $\mathtt{a} \in \mathtt{Att}$.

Formulas of $LTL^{\Downarrow}$ are evaluated with respect to a current position of a data word and a register assignment defining the contents of the registers. The operators $\mathbf{X}$ (*next*), $\mathbf{U}$ (*until*), $\mathbf{X}^{\leftarrow}$ (*previous*) and $\mathbf{U}^{\leftarrow}$ (*since*) are called *temporal operators*. The formula $\mathbf{X}\varphi$ expresses that the current position is not the last one and that φ holds at the immediate successor position. The formula $\varphi_1\mathbf{U}\varphi_2$ is true at a current position i if there is a position $k \geq i$ such that φ_2 holds at position k and φ_1 holds on all positions in $[j, \dots, k-1]$. The temporal operators $\mathbf{X}^{\leftarrow}$ and $\mathbf{U}^{\leftarrow}$ are the past counterparts of $\mathbf{X}$ and $\mathbf{U}$, respectively. Additionally, we use the abbreviations $\mathbf{F}$ (*future*), $\mathbf{G}$ (*globally in the future*), $\mathbf{F}^{\leftarrow}$ (*past*) and $\mathbf{G}^{\leftarrow}$ (*globally in the past*) defined by $\mathbf{F}\varphi = \top\mathbf{U}\varphi$, $\mathbf{G}\varphi = \neg\mathbf{F}\neg\varphi$, $\mathbf{F}^{\leftarrow}\varphi = \top\mathbf{U}^{\leftarrow}\varphi$ and $\mathbf{G}^{\leftarrow}\varphi = \neg\mathbf{F}^{\leftarrow}\neg\varphi$. The formula $\Downarrow^r_{@\mathtt{a}}.\varphi$ demands that the data value of attribute a is defined at the current position, stores this value in register r and evaluates φ. The atomic formula $\Uparrow^r_{@\mathtt{a}}$ is true at the current position if the value of attribute a is defined and equal to the value in register r.

We give some example formulas:

Example 7. We start with Property CS4 from Chapter 2. Due to the benefits of past operators, we can express it in a very simple way:

Every client receiving an acknowledgement has previously sent a request.

$$\mathbf{G}[\mathtt{ack} \rightarrow \Downarrow^r_{@\mathtt{receiver}}.\mathbf{F}^{\leftarrow}(\mathtt{req} \wedge \Uparrow^r_{@\mathtt{sender}})].$$

We now express the three discussed Properties CS5, CS6 and CS7 which have subtle differences with regard to the access of data values:

Whenever a client sends a request, it does not send any further request until it receives an answer.

$$\mathbf{G}\Big[\mathtt{req} \rightarrow \Downarrow^r_{@\mathtt{sender}}.\mathbf{X}\big((\mathtt{req} \rightarrow \neg \Uparrow^r_{@\mathtt{sender}})\mathbf{U}(\mathtt{ack} \wedge \Uparrow^r_{@\mathtt{receiver}})\big)\Big]$$

Between the creation of a client p and the receiving of the server information by p, there is no request to the server.

$$\neg\mathbf{F}\Big(\mathtt{crt} \wedge \Downarrow^r_{@\mathtt{created}}.\mathbf{X}\mathbf{F}\big(\mathtt{req} \wedge \mathbf{X}\mathbf{F}(\mathtt{serv} \wedge \Uparrow^r_{@\mathtt{receiver}})\big)\Big)$$

Whenever a client p receives an acknowledgement, the server gets a request after some time and the next such request is from a client different from p.

$$\mathbf{G}\Big[\mathtt{ack} \rightarrow \Downarrow^r_{@\mathtt{receiver}}.\mathbf{X}\big((\neg\mathtt{req})\mathbf{U}(\mathtt{req} \wedge \neg \Uparrow^r_{@\mathtt{sender}})\big)\Big]$$

Finally, we formulate CS9 which we was also used in Example 6:

Every client sending a request to a server gets an acknowledgement from the same server after some time.

Remember that position variables of FO$^{\sim}$ are assigned to positions and freeze registers of LTL$^{\Downarrow}$ are assigned to data values. Therefore, while in FO$^{\sim}$ we managed this property with a single variable, here we need two registers r_1 and r_2:

$$\mathbf{G}[\mathtt{req} \rightarrow \Downarrow^{r_1}_{\mathtt{@sender}} . \Downarrow^{r_2}_{\mathtt{@receiver}} . \mathbf{F}(\mathtt{ack} \wedge \Uparrow^{r_1}_{\mathtt{@receiver}} \wedge \Uparrow^{r_2}_{\mathtt{@sender}})]$$

□

The full semantics of LTL$^{\Downarrow}$ is given in the Appendix (Section A.2).

The fragment of LTL$^{\Downarrow}$ where at most $k \geq 1$ freeze registers are allowed in formulas is denoted by LTL$^{\Downarrow}_k$. For a logic $\mathcal{L} \in \{\text{LTL}^{\Downarrow}\} \cup \{\text{LTL}^{\Downarrow}_k \mid k \geq 1\}$ and a set $\mathcal{O}$ of temporal operators, we denote by $\mathcal{L}(\mathcal{O})$ the fragment of $\mathcal{L}$ in which only temporal operators from $\mathcal{O}$ can be used.

In LTL$^{\Downarrow}_1$-formulas we often skip the register name, i.e., we write, for instance, $\Downarrow_{\mathtt{@a}}$ and $\Uparrow_{\mathtt{@b}}$ instead of $\Downarrow^{r}_{\mathtt{@a}}$ and $\Uparrow^{r}_{\mathtt{@b}}$. Moreover, if it is clear from the context that an LTL$^{\Downarrow}$-formula is evaluated on data words with a single attribute we skip the the attribute name in formulas; for instance, we write $\Downarrow^{r}$ and $\Uparrow^{r}$ instead of $\Downarrow^{r}_{\mathtt{@a}}$ and $\Uparrow^{r}_{\mathtt{@b}}$.

The results mentioned below refer to 1-complete data words with a single proposition at each position. In [82], it is shown that the satisfiability problem for LTL$^{\Downarrow}_1(\mathbf{X}, \mathbf{U})$, i.e., the future fragment of LTL$^{\Downarrow}$ with only one freeze variable, is decidable on finite data words. Decidability is proven by a reduction to the non-emptiness problem for Alternating Register Automata with a single register (1-ARA). The problem has non-primitive recursive complexity. This lower bound holds even for the fragment LTL$^{\Downarrow}_1(\mathbf{F})$ [97]. In contrast to EMSO$^{\sim}_2(\mathtt{Suc}, <)$, decidability of LTL$^{\Downarrow}_1(\mathbf{X}, \mathbf{U})$ does not carry over to ω-words. Moreover, as soon as a further freeze variable or the past operator $\mathbf{F}^{\leftarrow}$ is added, the problem becomes undecidable [82, 97]. In [82], it is moreover shown that for every $k \geq 1$, every LTL$^{\Downarrow}_k$-formula on finite data words can be converted into an equivalent k-ARA$^{\leftrightarrow}$. In case of infinite words, the automaton is a Büchi ARA$^{\leftrightarrow}$ and if past operators are skipped it suffices that the automaton is one-way. Formulas of LTL$^{\Downarrow}_1(\mathbf{X}, \mathbf{U})$ can also be converted into equivalent TWPA [192].

4.3.3 Logics based on Regular Expressions

We introduce two logics from the literature which are closely related to Regular Expressions, namely *Regular Expressions with Memory* and *Two-Way Path Logic.*

Searching for suitable query languages for graph databases, the authors in [147] invented *Regular Expressions with Memory (REM).* They can be seen as usual regular expressions with additional registers whose usage is similar to those in LTL$^{\Downarrow}$. While classical regular expressions only specify the order of occurrences of symbols, an REM-expression can additionally impose conditions on data values. Roughly speaking, the basic components of REM-expressions are of the form $p[c]\downarrow^{R}_{\mathtt{@a}}$ where p is a proposition, c consists of a set of *register conditions*, R is a set of registers and $\mathtt{a}$ is an attribute. It expresses that the current position is labelled by p and its data values satisfy the conditions in c. Moreover, it requires that the value of attribute $\mathtt{a}$ is assigned to all all registers in R. A register condition is a boolean formula over atomic components of the form $\uparrow^{r}_{\mathtt{@a}}$ for registers r and attributes $\mathtt{a}$ asserting that the $\mathtt{a}$-value is equal to the input of register r. For instance, the expression $p[\uparrow^{r_1}_{\mathtt{@a}} \wedge \neg \uparrow^{r_2}_{\mathtt{@b}}]\downarrow^{\{r_3\}}_{\mathtt{@c}}$ means that (i) the current position is labelled by p, (ii) the values of $\mathtt{a}$ and r_1 are currently defined and equal, (iii) either one of the values of $\mathtt{b}$ and r_2 are not defined or they differ and (iv) the value of $\mathtt{c}$ is assigned to register r_3. A REM-expression is composed of concatenation, disjunction and iteration over $\emptyset$, ε and such basic components. Formally, the syntax of a REM-expression over a proposition set $\mathtt{Prop}$, an attribute set $\mathtt{Att}$ and a register set R is defined by the following grammar:

$$\alpha := \emptyset \mid \varepsilon \mid p[c]\downarrow^{R'}_{\mathtt{@a}} \mid \alpha \cdot \alpha \mid \alpha + \alpha \mid \alpha^*$$

where $p \in \texttt{Prop}$, $\texttt{a} \in \texttt{Att}$, c is a register condition and $R' \subseteq R$.

Example 8. We first conisder property CS1 from Chapter 2 which does not refer to data values:

After the first request there is no further process creation.

The following expression describing this property says that either (i) there is no req-position in the entire trace, or (ii) after the first occurrence of such a position there is no crt-position. Due to the syntax of REM, the expression makes needless accesses to attribute-values which are not stored in any register:

$$\big(\texttt{crt}[\top]\downarrow^{\emptyset}_{\texttt{@creator}} + \texttt{serv}[\top]\downarrow^{\emptyset}_{\texttt{@sender}} + \texttt{ack}[\top]\downarrow^{\emptyset}_{\texttt{@sender}}\big)^*$$
$$+$$
$$\Big[\big(\texttt{crt}[\top]\downarrow^{\emptyset}_{\texttt{@creator}} + \texttt{serv}[\top]\downarrow^{\emptyset}_{\texttt{@sender}} + \texttt{ack}[\top]\downarrow^{\emptyset}_{\texttt{@sender}}\big)^* \cdot \texttt{req}[\top]\downarrow^{\emptyset}_{\texttt{@sender}} \cdot$$
$$\big(\texttt{req}[\top]\downarrow^{\emptyset}_{\texttt{@sender}} + \texttt{serv}[\top]\downarrow^{\emptyset}_{\texttt{@sender}} + \texttt{ack}[\top]\downarrow^{\emptyset}_{\texttt{@sender}}\big)^*\Big]$$

We conclude with property CS8 which says:

Requests are always sent to the same server.

In the following formulation we use one register r in which the receiver-value of the first req-position is stored and compared with the receiver-values of all subsequent req-positions:

$$\big(\texttt{crt}[\top]\downarrow^{\emptyset}_{\texttt{@creator}} + \texttt{serv}[\top]\downarrow^{\emptyset}_{\texttt{@sender}} + \texttt{ack}[\top]\downarrow^{\emptyset}_{\texttt{@sender}}\big)^*$$
$$+$$
$$\Big[\big(\texttt{crt}[\top]\downarrow^{\emptyset}_{\texttt{@creator}} + \texttt{serv}[\top]\downarrow^{\emptyset}_{\texttt{@sender}} + \texttt{ack}[\top]\downarrow^{\emptyset}_{\texttt{@sender}}\big)^* \cdot \texttt{req}[\top]\downarrow^{\{r\}}_{\texttt{@receiver}} \cdot$$
$$\big(\texttt{req}[\uparrow^{r}_{\texttt{@receiver}}]\downarrow^{\emptyset}_{\texttt{@sender}} + \texttt{serv}[\top]\downarrow^{\emptyset}_{\texttt{@sender}} + \texttt{ack}[\top]\downarrow^{\emptyset}_{\texttt{@sender}}\texttt{crt}[\top]\downarrow^{\emptyset}_{\texttt{@creator}}\big)^*\Big]$$

□

The full syntax and semantics of REM is given in Section A.3 of the Appendix.

The language *Two-Way Path Logic (PathLog)* [96] was invented as a fragment of XPath [66] on simple data words. It can be seen as a sort of temporal logic where temporal operators are expressed by some kind of regular expressions. Given a proposition set Prop and an attribute set Att, formulas are composed of *path expressions* α and *position formulas* φ:

$$\varphi := p \mid \texttt{@a}\langle \overleftarrow{\alpha} \sim \overrightarrow{\alpha} \rangle\texttt{@b} \mid \texttt{@a}\langle \overleftarrow{\alpha} \not\sim \overrightarrow{\alpha} \rangle\texttt{@b} \mid \neg\varphi \mid \varphi \wedge \varphi$$
$$\alpha := \varepsilon \mid [\varphi] \cdot \alpha$$

with $p \in \texttt{Prop}$ and $\texttt{a}, \texttt{b} \in \texttt{Att}$.

We say that a position j is *reachable* from a position i by a path expression $\alpha = [\varphi_1] \cdots [\varphi_n]$ composed of position formulas $\varphi_1 \ldots \varphi_n$ if there are positions $i \leq i_1 \leq \ldots \leq i_n = j$ such that for every k with $1 \leq k \leq n$, the formula φ_k is satisfied at position i_k. Analogously, j is said to be *backward reachable* from i by α if there are positions $i \geq i_1 \geq \ldots \geq i_n = j$ such that for every k, formula φ_k holds at position i_k. The position formula $p \in \texttt{Prop}$ is satisfied at a position i if i is labelled by p. A formula $\texttt{@a}\langle \overleftarrow{\alpha} \sim \overrightarrow{\beta} \rangle\texttt{@b}$ holds at i if there are positions j and k such that k is reachable from i by β, j is backward reachable from i by α and the a-value at j and the b-value at k are both defined and equal. Conversely, the formula $\texttt{@a}\langle \overleftarrow{\alpha} \not\sim \overrightarrow{\beta} \rangle\texttt{@b}$ demands that the corresponding values of positions j and k are distinct from each other. Note that the order of the formulas composing a path expression is not strict. Therefore, $\texttt{@a}\langle \overleftarrow{[p]} \sim \overrightarrow{[q] \cdot [q]} \rangle\texttt{@a}$ is equivalent to $\texttt{@a}\langle \overleftarrow{p} \sim \overrightarrow{q} \rangle\texttt{@a}$. The logic consists of all position formulas.

Example 9. We express Property CS8 from Chapter 2 by a PathLog-formula. The property says:

Requests are always sent to the same server.

Our formula checks that there is no req-position followed by another req-position such that the receiver-values are distinct. Since formulas of PathLog turn out to become quite cumbersome, here we use the abbreviation $\langle\alpha\rangle$ to express that a position can be reached by α. Observe that this can be formulated by a disjunction over all possible equality conditions between attribute values at the current and the target position. Using the abberaviation, Property CS8 can be described as follows:

$$\neg\langle\overrightarrow{[\texttt{req} \wedge \texttt{@receiver}\langle\overleftarrow{\varepsilon} \not\sim \overrightarrow{[\texttt{req}]}\rangle\texttt{@receiver}]}\rangle$$

□

The full definition of the semantics of PathLog is given in the Appendix (Section A.4).

As usual, we skip the attribute name in REM- and PathLog-formulas when only a single attribute is used.

As far as we know, the literature provides complexity results for PathLog and REM only with regard to finite simple data words, that is, 1-complete data words with a single proposition at each position. The satisfiability problem for PathLog is ExpSpace-complete [96]. In comparison, satisfiability for REM is only PSpace-complete [147]. It is further shown in [147] that REM and RA are expressively equivalent. In the same work, the authors introduce a restriction of REM called *Regular Expressions with Equality (REME)* where equality tests between data values are only allowed at the beginning and at the end of subwords matching subexpressions. This formalism is strictly weaker than REM and its satisfiability problem is in PTime.

4.3.4 Further Logics

Due to the results on $\text{LTL}^{\Downarrow}$ mentioned in Section 4.3.2, several works arose that were searching for decidable fragments of $\text{LTL}^{\Downarrow}$ which contain past operators or preserve decidability on ω-words. One example is the consideration of the *safety fragment* of $\text{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$ in [142, 143]. A formula of $\text{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$ is considered to be in the safety fragment if the operator $\mathbf{U}$ does not occur in the scope of an even number of negations. While safety $\text{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$ is expressively equivalent to full $\text{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$ on finite words it is strictly less expressive than $\text{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$ on ω-words. The satisfiability problem for safety $\text{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$ is shown to be ExpSpace-complete on infinite data words with a single proposition and a single data value at each position. Adding the $\mathbf{F}^{\leftarrow}$-operator or a further freeze register leads immediately to undecidability. As an intermediate step in the proof of the upper bound of the satisfiability of safety $\text{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$ on infinite words, the authors show that each formula of this logic can be converted into an equivalent so-called *safety* 1-BARA. Languages of these automata are safety properties, i.e., every word not satisfying such a property must have a finite prefix such that every possible extension of the prefix does not satisfy the property either. Further attempts to design decidable logics on ω-words are made in [80, 81]. In [80], a two-way fragment of $\text{LTL}^{\Downarrow}_1$, called *Constraint Logic* ($\text{CLTL}^{\texttt{XF}}$), on propositionless data words with multiple data values at each position is considered. In $\text{CLTL}^{\texttt{XF}}$, the freeze operator $\Downarrow$ is not used explicitly, but the access to data values is realized through atomic formulas of the forms $\texttt{@a} \sim \mathbf{X}^{\ell}\texttt{@b}$ and $\texttt{@a} \sim \langle\rangle\texttt{@b}$ where a and b are attributes. Additionally, it contains the past counterpart $\texttt{@a} \sim \langle\rangle^{\leftarrow}\texttt{@b}$ of the latter kind of formulas. The formula $\texttt{@a} \sim \mathbf{X}^{\ell}\texttt{b}$ is equivalent to the $\text{LTL}^{\Downarrow}$-formula $\Downarrow^{r}_{\texttt{@a}}.\mathbf{X}^{\ell} \Uparrow^{r}_{\texttt{@b}}$, and $\texttt{@a} \sim \langle\rangle\texttt{@b}$ is equivalent to $\Downarrow^{r}_{\texttt{@a}}.\mathbf{XF} \Uparrow^{r}_{\texttt{@b}}$. The full syntax and semantics of $\text{CLTL}^{\texttt{XF}}$ can be found in the Appendix (Section A.5). It is shown that the satisfiability problem for $\text{CLTL}^{\texttt{XF}}$ on finite and infinite data words is decidable and, in case of one data value at each position, PSpace-complete [80]. The work on $\text{CLTL}^{\texttt{XF}}$ is continued in [81]. The authors consider an extension of $\text{CLTL}^{\texttt{XF}}$ called

Logic of Repeating Values (LRV). Instead of $\mathtt{@a} \sim \langle\rangle\mathtt{@b}$, LRV uses formulas of the forms $\mathtt{@a} \sim \langle\varphi\rangle\mathtt{@b}$ and $\mathtt{@a} \not\sim \langle\varphi\rangle\mathtt{@b}$. The first formula can be expressed by $\Downarrow^r_{\mathtt{@a}}.(\mathbf{XF} \Uparrow^r_{\mathtt{@b}} \wedge \varphi)$ while the second one is equivalent to $\Downarrow^r_{\mathtt{@a}}.(\mathbf{XF}\neg \Uparrow^r_{\mathtt{@b}} \wedge \varphi)$. The extension of LRV by the past counterparts of these formulas is denoted by PLRV. Full syntax and semantics of LRV and PLRV are given in Section A.6 of the Appendix. The decidability of the satisfiability problem for $\mathrm{CLTL}^{\mathtt{XF}}$ on finite and infinite words carries over to PLRV. For LRV, the problem is even 2ExpSpace-complete.

In [125], some kind of regular expressions on infinite alphabets is introduced. However, as argued in [147], these expressions are not very intuitive and do not even allow inequality tests between data values. Therefore, they are not able to define, for instance, the simple language of data words where the first two positions have different data values.

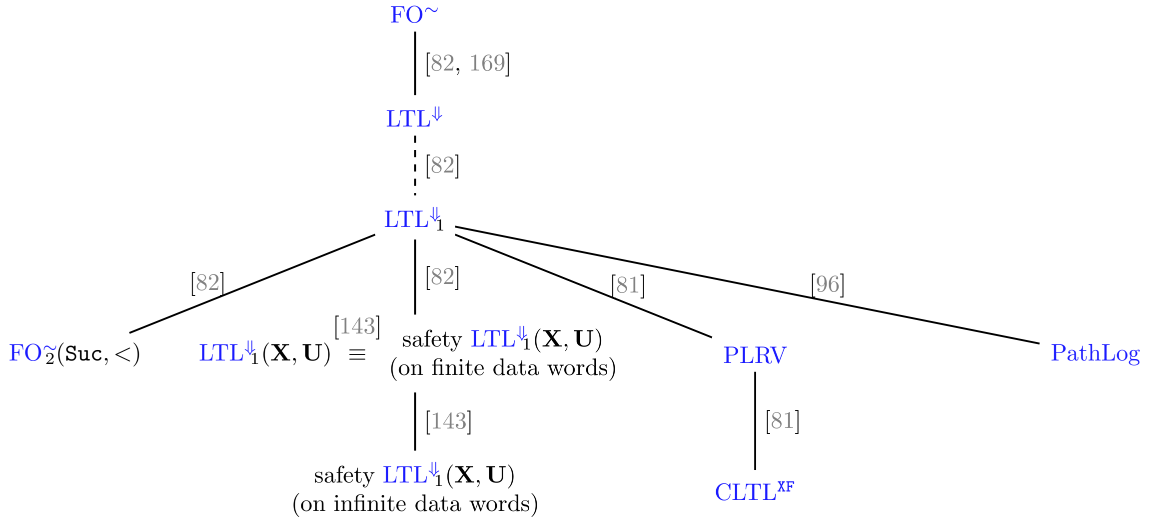


Figure 4.6: Comparison of data logics with respect to expressivity (a dashed line from a higher to a lower logic indicates that the first one is at least as expressive as the latter one; a solid line means that the inclusion is strict)

In Figure 4.6, we give an overview on the expressive power of some basic logics introduced in this section. Just like in Figure 4.3, a dashed line from a higher to a lower logic indicates that the first one is at least as expressive as the latter one. A solid line means that the inclusion is strict. The depicted relationship between $\mathrm{FO}^{\sim}$ and $\mathrm{LTL}^{\Downarrow}$ follows from the results that $\mathrm{FO}^{\sim}$ can express a property which cannot be tested by any $\mathrm{ARA}^{\leftrightarrow}$ [169] and each $\mathrm{LTL}^{\Downarrow}$-formula can be converted into an equivalent $\mathrm{ARA}^{\leftrightarrow}$ [82]. The languages REM and REME are not comparable to any of the depicted logics besides PathLog. On the one hand, REM and REME capture conventional regular expressions which is not the case for any of the logics in the figure. On the other hand, due to the equivalence between REM and RA, the languages REM and REME cannot even test simple properties like the requirement that all data values of a data word are pairwise distinct. The relationship between REM and REME on the one side and PathLog on the other side is unclear. Figures 4.7 and 4.8 give an overview about known results on the satisfiability problem for data logics. Figure 4.9 contains some known results on the relationship between data logics and automata on data words.

	on finite data words	on infinite data words
$\mathrm{FO}^{\sim}_{3}$	undecidable [41]	undecidable [41]
$\mathrm{EMSO}^{\sim}_{2}(\mathtt{Suc}, <)$ $\mathrm{FO}^{\sim}_{2}(\mathtt{Suc}, <)$	decidable [41]	decidable [41]
$\mathrm{EMSO}^{\sim}_{2}(\mathtt{Suc}, \mathtt{Suc}_{\sim})$ $\mathrm{FO}^{\sim}_{2}(\mathtt{Suc}, \mathtt{Suc}_{\sim})$	decidable [41]	
$\mathrm{FO}^{\sim}_{2}(\mathtt{Suc})$	in 2NExpTime [170]	decidable [41]
$\mathrm{FO}^{\sim}_{2}(<)$	NExpTime-c [41]	decidable [41]
$\mathrm{LTL}^{\Downarrow}_{2}(\mathbf{F})$	undecidable [97]	undecidable [97]
$\mathrm{LTL}^{\Downarrow}_{1}(\mathbf{F}^{\leftarrow}, \mathbf{F})$	undecidable [97]	undecidable [97]
$\mathrm{LTL}^{\Downarrow}_{1}(\mathbf{X}, \mathbf{U})$ $\mathrm{LTL}^{\Downarrow}_{1}(\mathbf{F})$	decidable, non-prim. rec. [82, 97]	undecidable [82, 97]
safety $\mathrm{LTL}^{\Downarrow}_{1}(\mathbf{X}, \mathbf{U})$	decidable, non-prim. rec. [142, 143]	ExpSpace-c [142, 143]
PLRV	decidable, non-prim. rec. [81]	decidable, non-prim. rec.[81]
LRV	2ExpSpace-c [81]	2ExpSpace-c [81]
$\mathrm{CLTL}^{\mathtt{XF}}$	PSpace-c [80]	PSpace-c [80]
PathLog	ExpSpace-c [96]	
REM	PSpace-c [147]	
REME	in PTime [147]	

Figure 4.7: Complexity of the satisfiability problem for data logics on 1-complete data words (the 2NExpTime upper bound for $\mathrm{FO}^{\sim}_{2}(\mathtt{Suc})$ holds for data words with multiple propositions at each position; all results for PLRV, LRV and $\mathrm{CLTL}^{\mathtt{XF}}$ hold for propositionless data words; all other results hold for data words with a single proposition at each position)

	on finite data words	on infinite data words
$\mathrm{FO}^{\sim}_{3}$	undecidable [41]	undecidable [41]
$\mathrm{EMSO}^{\sim}_{2}(\mathtt{Suc}, <)$ $\mathrm{FO}^{\sim}_{2}(\mathtt{Suc}, <)$	undecidable [41]	undecidable [41]
$\mathrm{LTL}^{\Downarrow}_{2}(\mathbf{F})$	undecidable [97]	undecidable [97]
$\mathrm{LTL}^{\Downarrow}_{1}(\mathbf{F}^{\leftarrow}, \mathbf{F})$	undecidable [97]	undecidable [97]
$\mathrm{LTL}^{\Downarrow}_{1}(\mathbf{X}, \mathbf{U})$ $\mathrm{LTL}^{\Downarrow}_{1}(\mathbf{F})$		undecidable [82, 97]
PLRV	decidable, non-prim. rec. [81]	decidable, non-prim. rec.[81]
LRV	2ExpSpace-c [81]	2ExpSpace-c [81]
$\mathrm{CLTL}^{\mathtt{XF}}$	decidable [80]	decidable [80]

Figure 4.8: Complexity of the satisfiability problem for data logics on m-complete data words for $m \geq 2$ (all results for PLRV, LRV and $\mathrm{CLTL}^{\mathtt{XF}}$ hold for propositionless data words; all other results hold for data words with a single proposition at each position)

on finite data words	on infinite data words
$FO^{\sim} \leq PA$ [169]	
$EMSO_2^{\sim}(\mathtt{Suc}, <) \leq DA$ [41]	$EMSO_2^{\sim}(\mathtt{Suc}, <) \leq BDA$ [41]
$EMSO_2^{\sim}(\mathtt{Suc}, \mathtt{Suc}_{\sim}) \equiv DA$ [41]	
for every $k \geq 1$, $LTL^{\Downarrow}_k \leq k\text{-}ARA^{\leftrightarrow}$ [82]	for every $k \geq 1$, $LTL^{\Downarrow}_k \leq k\text{-}BARA^{\leftrightarrow}$ [82]
for every $k \geq 1$, $LTL^{\Downarrow}_k(\mathbf{X}, \mathbf{U}) \leq k\text{-}ARA$ [82]	for every $k \geq 1$, $LTL^{\Downarrow}_k(\mathbf{X}, \mathbf{U}) \leq k\text{-}BARA$ [82]
$LTL^{\Downarrow}_1(\mathbf{X}, \mathbf{U}) \leq TWPA$ [192]	
	safety $LTL^{\Downarrow}_1(\mathbf{X}, \mathbf{U}) \leq$ safety 1-ARA [143]
$REM \leq RA$ [147]	

Figure 4.9: Correspondence between data logics and automata on 1-complete data words

Part B

New Insights on Data Logics

In this part, we first formulate some questions on the expressivity and complexity of logics and automata on data words which arise from the results summarized in Sections 4.2 and 4.3 of the last part. In Chapters 6, 7 and 8, we try to answer these questions.

Chapter 5

Motivating Questions on Data Logics

Designing a logic for system verification

As mentioned in the introduction of this work, although data logics were studied mainly with regard to expressivity and satisfiability until now, in many works it is stated that this kind of logics can be appropriate to be applied in the field of model checking of concurrent systems with unboundedly many processes (see, e.g., in [41, 45, 44, 47]). As illustrated in Chapter 2, traces of such systems can be represented by data words where data values stand for process IDs. Thus, data logics can be used to specify desired properties on traces. Then, by means of suitable model checking procedures, it can be tested whether all traces of a system satisfy the requirements. One of our aims in this part of the work is the design of an expressive, but decidable data logic suitable for the usage in the framework of model checking concurrent systems with unboundedly many processes.

In order to get an idea about the features such a logic should have, we take a closer look at the area of finite-state model checking with LTL where systems are usually described by Kripke-structures [30]. Firstly, we observe that system traces are modeled by words where each position can carry multiple propositions which stand for properties at specific states. Secondly, since systems are ideally expected to run ad infinitum, traces are usually modeled as infinite words. Thirdly, though it is known that past operators do not have any effect (besides succinctness) on the expressive power of LTL, it is argued (see, e.g., in [141]) that past operators make it more comfortable to express system properties. If we turn our gaze towards systems with unboundedly many processes and consider our initial example in Chapter 2 as well as the model in [45] where model checking with data logics is studied, it seems that it is convenient to describe system traces by words with multiple data values at each position so that all IDs of processes participating in a common action at a certain time can be grouped at a single position. From this train of thought we can conclude that it would be beneficial if the data logic we are going to design (i) contains operators enabling reference to the past and (ii) is decidable on data ω-words where (iii) every position can have multiple propositions and data values.

Now, we review the logics introduced in Section 4.3 with respect to these features. We first recognize that, although we have presented their extensions on general data words, almost all of them were originally introduced and studied either on propositionless data words as, for instance, CLTL^{XF} and PLRV, or on words with at most one proposition at each position as, for instance, $\text{LTL}^{\Downarrow}$ and PathLog. Nonetheless, at least in terms of decidability, extensions to multiple propositions per position do not seem to be a cause for major harms. Yet, the logics have further peculiarities which are more crucial with regard to the features mentioned above. As we have stated in Section 4.3.2,

the quite convenient and expressive logic $\mathrm{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$ looses its decidability when past operators are added or ω-words are taken under consideration. These results led to several attempts to find a fragment of $\mathrm{LTL}^{\Downarrow}_1$ which gives the ability to explicitly access past positions and which preserves its decidability on ω-words. One outcome of this search was the study of safety $\mathrm{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$ which does not contain past operators, but is decidable on 1-complete ω-words. However, the strong syntactical restriction that the $\mathbf{U}$-operator must not occur under an even number of negations can lead to very long and inconvenient safety $\mathrm{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$-formulas. For instance, the authors in [142] express the simple $\mathbf{U}$-formula $\varphi\mathbf{U}\psi$ by $\neg\Big[(\neg\psi)\mathbf{U}\big(\neg\psi\wedge(\neg\varphi\vee\neg\mathbf{X}\top)\big)\Big]$ and it does not seem that there is a shorter equivalent formula in safety $\mathrm{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$. A further logic which is expressivity-wise subsumed by $\mathrm{LTL}^{\Downarrow}_1$ is $\mathrm{FO}^{\sim}_2(\mathtt{Suc},<)$. Due to its order relation, it allows to distinguish between the future and the past of positions and is, moreover, decidable on 1-complete data ω-words. However, it looses its decidability when more than one data value per position is allowed. Another fragment of $\mathrm{LTL}^{\Downarrow}_1$ (and an extension of $\mathrm{CLTL}^{\mathtt{XF}}$) is PLRV. This logic contains past operators and is decidable on infinite data words with multiple data values at each position. However, one shortcoming in PLRV is that the access to data values is limited. Besides formulas of the form $\mathtt{@a}\sim\mathbf{X}^{\ell}\mathtt{@b}$ which allow to compare data values of positions of some bounded distance ℓ, the access to data values is realized by formulas of the forms $\mathtt{@a}\sim\langle\varphi\rangle\mathtt{@b}$ and $\mathtt{@a}\not\sim\langle\varphi\rangle\mathtt{@b}$. They allow to fix some data value d of the current position, to "jump" to *some* target position where some constraint with respect to d holds and to evaluate some formula φ at the target position. The shortcoming of this kind of formulas is that the distance between the current and the target position within the class of d is arbitrary. This makes it difficult to express properties talking about all consecutive positions within a single class like Property CS5 which we formulated in $\mathrm{LTL}^{\Downarrow}$ by $\mathbf{G}\big[\mathtt{req}\rightarrow\Downarrow^r_{\mathtt{@sender}}.\mathbf{X}\big((\mathtt{req}\rightarrow\neg\Uparrow^r_{\mathtt{@sender}})\mathbf{U}(\mathtt{ack}\wedge\Uparrow^r_{\mathtt{@receiver}})\big)\big]$.

The logic PLRV constitutes the starting point for the design of our new logic in Chapter 6. We propose a logic called *Data Navigation Logic* which (i) contains past operators, (ii) is strictly more expressive than PLRV (and the decidable fragment of $\mathrm{FO}^{\sim}_1(\mathtt{Suc},<)$), (iii) allows navigation via regular expressions in the spirit of REM and PathLog and (iv) remains decidable on finite and infinite data words with multiple data values and propositions at each position. Our logic is also inspired by [115, 116, 146, 181, 48] which propose different temporal logics containing regular expressions to describe paths on finitely labelled structures. Naturally, in order to preserve decidability, we have to be careful with regard to the access of data values. We call our concept of data value access *navigation along data values*. This means that besides the classical navigation along consecutive word positions (like, for instance, in LTL), our logic additionally allows navigation along consecutive class positions. For example, the $\mathrm{LTL}^{\Downarrow}$-formula above can be expressed in Data Navigation Logic by $\mathbf{G}\big[\mathtt{req}\rightarrow\mathcal{C}_{\mathtt{@sender}}(\mathbf{X}_{=}((\neg\mathtt{req})\mathbf{U}_{=}\mathtt{ack}))\big]$. Here, the class operator $\mathcal{C}$ allows to "dive" into the class of the `sender`-value at the current position whereupon temporal operators are evaluated within this class. Since Data Navigation Logic allows to simulate all classical temporal LTL-operators, we used them as abbreviations in the latter formula. Our decidability proof is quite simple and relies on a reduction to the non-emptiness problem for Data Automata. We also show that limited extensions of this kind of navigation result in undecidability. Finally, we discuss how far our logic can be extended by some kind of navigation along unequal data values while preserving its decidability.

Storage of positions vs. storage of data values

In Section 4.3.2, we introduced $\mathrm{LTL}^{\Downarrow}$ [82] and summarized known results on this logic. To put it in a nutshell, the logic is an extension of usual LTL which allows to store data values in registers ($\Downarrow^r_{\mathtt{@a}}$) and to ask whether the value of some attribute is equal to the input of some register ($\Uparrow^r_{\mathtt{@b}}$). In [82, 84, 200], the authors noted that $\mathrm{LTL}^{\Downarrow}$ is essentially a *hybrid temporal logic*. The term hybrid logics appears in the literature as a generic term for modal and temporal logics extended by first-order concepts of binding variables to positions. Hybrid logics were first considered in [176] and

intensively studied on linear structures in, e.g., [28, 100, 184]. Besides the operator $\downarrow x$ which binds a variable x to the current position, hybrid logics usually contain expressions like $\mathtt{on}(x).\varphi$[1] and x. The formula $\mathtt{on}(x).\varphi$ demands that φ holds at the position bound to x and the atomic formula x evaluates to true only at the x-position. It is known that such a "hybrid machinery" does not give additional expressive power to LTL on classical words [102, 100].

In Chapter 7, we introduce *Hybrid Temporal Logic on data words ($HTL^{\sim}$)* and compare it to $LTL^{\Downarrow}$ in terms of expressivity. Instead of the $LTL^{\Downarrow}$-operators $\Downarrow^r_{\mathtt{@a}}$ and $\Uparrow^r_{\mathtt{@b}}$, the logic $HTL^{\sim}$ contains the operators $\downarrow^x$ and $\mathtt{@a} \sim x.\mathtt{@b}$ for variables x and attributes $\mathtt{a}$ and $\mathtt{b}$. Additionally, it contains atomic formulas of the forms x and $\mathtt{on}(x).\varphi$. We highlight the differences between the *binding* mechanism in $HTL^{\sim}$ and the *freezing* mechanism in $LTL^{\Downarrow}$: While the $LTL^{\Downarrow}$-operator $\Downarrow^r_{\mathtt{@a}}$ stores in register r only the value of attribute $\mathtt{a}$ at a current position i, the $HTL^{\sim}$-operator $\downarrow^x$ memorizes the whole position i by assigning x to it. After having left position i, $LTL^{\Downarrow}$-operations of the form $\Uparrow^r_{\mathtt{@b}}$ allow to compare current data values only against values stored in registers, while $HTL^{\sim}$-operations of the form $\mathtt{@b} \sim x.\mathtt{@a}$ allow equality tests between all attributes $\mathtt{a}$ at the x-position and all attributes $\mathtt{b}$ at the current position.

We show in Chapter 7 that $HTL^{\sim}$ is strictly more expressive than $LTL^{\Downarrow}$ and there is actually an $HTL^{\sim}$-formula using only two variables and one attribute for which no equivalent $LTL^{\Downarrow}$-formula exists. Moreover, $HTL^{\sim}$-formulas can be non-elementarily more succinct than $LTL^{\Downarrow}$-formulas. Surprisingly, the additional expressive power vanishes when the consideration is restricted to a single variable. We show that every $HTL^{\sim}$-formula using at most one variable can be translated into an equivalent $LTL^{\Downarrow}$-formula. If we restrict the number of attributes to one, it can be even shown that $HTL^{\sim}$ with a single variable and $LTL^{\Downarrow}$ with a single register are expressively equivalent. Yet, $HTL^{\sim}$-formulas with only one variable can be exponentially more succinct than $LTL^{\Downarrow}$. Finally, we show that the variable hierarchy of $HTL^{\sim}$ and the register hierarchy for $LTL^{\Downarrow}$ are infinite, i.e., for every natural number k there is always a $k' > k$ such that $HTL^{\sim}$ with k' variables is strictly more expressive than with k variables and $LTL^{\Downarrow}$ with k' registers is strictly more expressive than with k registers.

Designing an automata model for two-variable logic

In Section 4.3.1, we mentioned that in [41] the decidability of $EMSO^{\sim}_2(\mathtt{Suc}, <)$ on 1-complete data words was shown by a reduction to the non-emptiness problem for Data Automata. Moreover, the authors proved that this automata model can be characterized logically by $EMSO^{\sim}_2(\mathtt{Suc}, \mathtt{Suc}_{\sim})$. The complexity of $FO^{\sim}_2(\mathtt{Suc}, <)$ (and that of $EMSO^{\sim}_2(\mathtt{Suc}, <)$) was shown to be equivalent to reachability in Petri Nets [106] which is a hard problem whose precise complexity is not known yet. This result motivated the search for fragments of $FO^{\sim}_2(\mathtt{Suc}, <)$ with moderate complexities. If the successor relation $\mathtt{Suc}$ is skipped in $FO^{\sim}_2(\mathtt{Suc}, <)$, complexity drops down to NExpTime [41]; if, instead, the linear order $<$ is skipped, complexity drops down to 2NExpTime [170]. Here, we focus on the latter fragment, namely $FO^{\sim}_2(\mathtt{Suc})$, and its extension $EMSO^{\sim}_2(\mathtt{Suc})$. To the best of our knowledge, there is no automata model which, in analogy to the relationship between $EMSO^{\sim}_2(\mathtt{Suc}, \mathtt{Suc}_{\sim})$ and DA, corresponds to $EMSO^{\sim}_2(\mathtt{Suc})$. Thus, one arising question is whether there is a natural restriction of Data Automata which complexity-wise behaves well and is logically characterized by $EMSO^{\sim}_2(\mathtt{Suc})$. Such a model can be useful in the study of expressivity questions and the design of model checking procedures for $FO^{\sim}_2(\mathtt{Suc})$.

In Chapter 8, we introduce a restriction of Data Automata called *Weak Data Automata (WDA)* for which we prove that it is expressively equivalent to $EMSO^{\sim}_2(\mathtt{Suc})$ on finite data words. Weak Data Automata differ from Data Automata in that they do not contain any class automaton, but can check simpler conditions on sets of data values. We show that WDA are strictly less expressive

[1] In hybrid logics, the operator $\mathtt{on}(x)$ is usually notated as $\mathtt{@}x$. To avoid confusion with the operator $\mathtt{@a}$ which accesses the data value of attribute $\mathtt{a}$ in data logics, we use a different notation here.

than Data Automata and incomparable with Register Automata. With the help of this model we can prove that $\mathrm{EMSO}_2^{\sim}(\mathtt{Suc})$ is strictly less expressive than $\mathrm{EMSO}_2^{\sim}(\mathtt{Suc}, <)$, a contrast to the equivalence of these logics on classical strings. Furthermore, we derive from existing results that the non-emptiness problem for WDA can be solved in non-deterministic doubly exponential time.

As one of the motivations for the design of WDA is model checking and system traces are usually modeled as infinite words, we are also interested in the question whether the complexity and expressivity results for WDA carry over to data ω-words. Following the approach in Büchi Data Automata, we define Weak Büchi Data Automata (WBDA) which result from WDA by equipping the base automata by a Büchi acceptance condition. We show that WBDA can be characterized by $\mathrm{EMSO}_2^{\sim}(\mathtt{Suc})$ extended by existential quantification over infinite sets. Furthermore, all expressivity results for WDA carry over to WBDA, that is, WBDA are strictly less expressive than Büchi Data Automata and incomparable with Büchi Register Automata. Finally, there is a polynomial reduction from the non-emptiness problem for WBDA to the same problem for WDA. The last result is not presented in this work, but can be found in our work [130].

Chapter 6

Navigation along Data Values

	p,q	p ($\mathcal{C}_{\mathtt{@b}}$)	r,p	r	q	r,q	p,r
a	2	8	3	5	3		5
b		3	5	8	2	3	2
c	3	4		2	3	2	8

As mentioned in Chapter 5, in this chapter we will design and study a logic called Data Navigation Logic which allows navigation by regular expressions and enables reference to the past. The logic is strictly more expressive than PLRV on general data words and than $\mathrm{FO}^2_\sim(\mathtt{Suc}, <)$ on 1-complete data words. Furthermore, it preserves decidability on finite and infinite data words. First, we will introduce in Section 6.1 Basic Data Navigation Logic, the core of Data Navigation Logic, which, besides navigation along consecutive positions, allows navigation along consecutive class positions. To give an example from this fragment, the formula $\langle \psi_1^* \rangle \psi_2$ simulates the LTL-formula $\psi_1 \mathbf{U} \psi_2$. Similarly, the formula $\mathcal{C}_{\mathtt{@a}} \langle \psi_1^* \rangle_= \psi_2$ demands that $\psi_1 \mathbf{U} \psi_2$ holds within the class of the value of attribute $\mathtt{a}$ at the current position. Within brackets of the forms $\langle\rangle$ and $\langle\rangle_=$, the logic also allows arbitrary regular expressions over formulas so that its navigational capabilities exceed classical LTL. The decidability of the satisfiability problem of this logic is obtained by reduction to the non-emptiness problem for Data Automata. In Section 6.2, we will stress the main techniques in this reduction in the case of finite data words. In Section 6.3, we will show that these techniques can be adapted to the case of data ω-words. Afterwards, we will demonstrate in 6.4 that subtle extensions in the style of navigation lead to undecidability. In brief, these extensions include the ability to access positions between consecutive class positions, the restriction of the scope of past navigation and the simultaneous navigation along two data values. In Section 6.5, we will turn towards features allowing inequality tests on data values and discuss how far Basic Data Navigation Logic can be enriched by them such that decidability is preserved. Our discussion will be conducted around a powerful $\mathbf{U}$-like operator allowing inequality tests at some target position as well as all positions between the current and the target position. As two outcomes of these discussions we will define Data Navigation Logic and Extended Data Navigation Logic. While for the first logic we will show that it remains decidable on finite and infinite data words, for the latter one we will only be able to prove decidability for the finite case. In Section 6.6, we will compare the expressive power of Data Navigation Logic to other logics introduced in Section 4.3. Open questions and further works which build on our results, but were published during the preparation of this thesis will be discussed in the concluding Chapter 6.7.

6.1 Basic Data Navigation Logic

In Data Navigation Logic we distinguish between *global formulas*, *class formulas*, *global path expressions* and *class path expressions*. Class formulas and class path expressions only occur in the scope of the class quantifier $\mathcal{C}$. Path expressions describe navigation on the data word and can be seen as generalizations of the temporal operators of LTL. In this section, we introduce *Basic Data Navigation Logic (B-DNL)*, the core of Data Navigation Logic. Given a set Prop of propositions and a set Att of attributes, the syntax of global formulas φ and class formulas ψ in B-DNL is defined as follows:

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\rho\rangle\varphi \mid \langle\rho\rangle^{\leftarrow}\varphi \mid \mathcal{C}^{\ell}_{\texttt{@a}}\psi$$

$$\psi := \varphi \mid \neg\psi \mid \psi \wedge \psi \mid \langle\theta\rangle_{=}\psi \mid \langle\theta\rangle^{\leftarrow}_{=}\psi \mid \sim\texttt{@a}$$

with $p \in \texttt{Prop}$, $\texttt{a} \in \texttt{Att}$ and $\ell \in \mathbb{Z}$. Next, we give the syntax of global path expressions ρ and class path expressions θ:

$$\rho := \epsilon \mid \varphi \mid \rho \cdot \rho \mid \rho + \rho \mid \rho^*$$

$$\theta := \epsilon \mid \psi \mid \theta \cdot \theta \mid \theta + \theta \mid \theta^*$$

where φ is a global and ψ a local formula. The logic B-DNL consists of all global formulas.

Intuitively, global path expressions are used for navigation along consecutive positions while class path expressions enable navigation in classes. Observe that the elementary components of these expressions are not only propositions, but entire formulas. The global formula $\langle\rho\rangle\varphi$ holds at a position i of a data word w if there is some position $j \geq i$ in w such that φ holds at j and the expression ρ holds on $w[i, \ldots, j-1]$, i.e., the subword of w from i to $j-1$. The application of the class operator $\mathcal{C}^{\ell}_{\texttt{@a}}$ at some position i (i) restricts navigation to the class word of the value of attribute a at position i and (ii) starts the evaluation of the formula in its scope at position $i+\ell$. Let $\langle\theta\rangle_{=}\psi$ and $\sim\texttt{@b}$ be class formulas occurring within the scope of a class operator which restricts navigation to the class of some data value d. Then, the formula $\langle\theta\rangle_{=}\psi$ holds at some position i of w if there is a position $j \geq i$ carrying value d for some attribute, ψ holds at position j and θ holds on $w_d[i, \ldots, j-1]$, i.e., the subword of w from i to $j-1$ restricted to the class of d. The formula $\sim\texttt{@b}$ is true at position i if the value of attribute b is defined at i and equal to d. The path expressions $\langle\rho\rangle^{\leftarrow}$ and $\langle\theta\rangle^{\leftarrow}_{=}$ are the counterparts of $\langle\rho\rangle$ and $\langle\theta\rangle_{=}$ and describe navigation to the past. The full formal definition of the semantics can be found in the Appendix (Section A.7).

We call formulas of the forms $\langle\rho\rangle\varphi$ and $\langle\theta\rangle_{=}\psi$ *future formulas* and those of the forms $\langle\rho\rangle^{\leftarrow}\varphi$ and $\langle\theta\rangle^{\leftarrow}_{=}\psi$ *past formulas*. The superscript ℓ in $\mathcal{C}^{\ell}_{\texttt{@a}}$ is called the *shift value* of the class operator. We often omit the shift value if it is equal to 0. Furthermore, in formulas over a single attribute a we usually skip the attribute reference @a and write $\mathcal{C}^{\ell}$ and $\sim$ instead of $\mathcal{C}^{\ell}_{\texttt{@a}}$ and $\sim\texttt{@a}$, respectively.

Classical LTL-navigation can be easily expressed in B-DNL. We define $\mathbf{X}\varphi = \langle\top\rangle\varphi$, $\varphi_1\mathbf{U}\varphi_2 = \langle\varphi_1^*\rangle\varphi_2$, $\mathbf{X}_{=}\psi = \langle\top\rangle_{=}\psi$ and $\psi_1\mathbf{U}_{=}\psi_2 = \langle\psi_1^*\rangle_{=}\psi_2$ for global formulas $\varphi, \varphi_1, \varphi_2$ and class formulas ψ, ψ_1, ψ_2. The operators $\mathbf{F}$, $\mathbf{F}_{=}$, $\mathbf{G}$ and $\mathbf{G}_{=}$ and the past-versions of all of these LTL-operators are defined analogously. For convenience, we will often use LTL-operators within B-DNL-formulas. The fragment of B-DNL where in place of path expressions only these restricted operators are allowed, is called *Basic Data LTL (B-DLTL)*.

We proceed with some example formulas.

Example 10. We first consider Property CS3 from our initial example in Chapter 2:

Every client sending a request gets an acknowledgement after some time.

The property is expressed by the following B-DNL-formula:

$$\mathbf{G}\Big[\texttt{req} \rightarrow \mathcal{C}_{\texttt{@sender}}\big(\mathbf{F}_{=}(\texttt{ack} \wedge \sim\texttt{@receiver})\big)\Big]$$

We now turn to the three Properties CS5, CS6 and CS7 to which we drew particular attention in Chapter 2. Property CS5 says:

Whenever a client sends a request, it does not send any further request until it receives an answer.

Since this property can be checked by exploring the classes of all process IDs separately and B-DNL is tailored for this purpose, the formulation of this property is an easy task:

$$\mathbf{G}\Big[\mathtt{req} \to \mathcal{C}^1_{\mathtt{@sender}}\big((\neg\mathtt{req})\mathbf{U}_=\mathtt{ack}\big)\Big].$$

However, it is not clear how to express the following Property CS6:

Between the creation of a client p and the receiving of the server information by p, there is no request to the server.

The reason is that the property talks about *all* positions between two positions in the same class. Indeed, we will see in Section 6.4 that extending B-DNL by some ability to capture this kind of properties leads to undecidability. Now, we look at Property 7:

Whenever a client p receives an acknowledgement, the server gets a request after some time and the next such request is from a client different from p.

The property talks about two distinct data values at two distinct positions. While by formulas of the form $\neg\mathcal{C}^\ell_{\mathtt{@a}}\sim\mathtt{@b}$ we can express that attribute values of positions of some bounded distance ℓ are not equal, it is not obvious how to express the same for positions of arbitrary distance. However, in contrast to the case with CS6, we will show in Section 6.5 that there is a decidability preserving extension of B-DNL capturing properties like CS7. □

Before we continue, we recall some notions and notations useful in the following sections. For $\ell \in \mathbb{Z}$, the shortcut $\mathbf{X}^\ell$ stands for (i) ℓ consecutive repetitions of $\mathbf{X}$ if $\ell \geq 0$ and (ii) $|\ell|$ consecutive repetitions of $\mathbf{X}^\leftarrow$, otherwise. Likewise, for $\ell \in \mathbb{N}_0$ and path expressions τ, we use τ^ℓ as a shortcut for the ℓ-times concatenation of τ. For example, $p^3 = p \cdot p \cdot p$. For a finite word $w = w_1 \dots w_n$ of length n, we denote by $w^R = w_n \dots w_1$ the *reverse* of w. For a langage $\mathcal{L}$ consisting of finite words, the reverse $\mathcal{L}^R$ of $\mathcal{L}$ is defined as $\{w^R \mid w \in \mathcal{L}\}$. Accordingly, by the *reverse language* of an automaton $\mathcal{A}$ deciding words of finite length, we mean the language $\{w^R \mid w \text{ is accepted by } \mathcal{A}\}$.

6.2 Decidability of Basic Data Navigation Logic

This section is devoted to the decidability proof of the satisfiability problem for B-DNL. First we will show that the satisfiability problem for this logic is decidable on 1-complete data words. To this end, we will define a normal form for B-DNL-formulas and proof that every B-DNL-formula can be translated into an equivalent formula in normal form. Then, we will explain how some parts of B-DNL-formulas in normal form can be expressed by NFA and Register Automata (RA). Remember that RA (and of course NFA) are captured by Data Automata (DA) [39]. With the help of these auxiliary automata we will reduce the satisfiability problem of B-DNL on 1-complete data words to the non-emptiness problem for DA which is decidable [41]. Finally, we will proof that the satisfiability problem for B-DNL on general data words is reducible to the the same problem on 1-complete data words.

We start with some new notions. We call formulas of the forms $\langle\rho\rangle\varphi$ and $\langle\rho\rangle^\leftarrow\varphi$ *basic global formulas* and those of the forms $\langle\rho\rangle_=\psi$ and $\langle\rho\rangle^\leftarrow_=$ *basic class formulas*. By *basic formula* we mean all formulas of these kinds. We say that a B-DNL-formula is in *normal form* if in all of its sub-formulas of the form $\mathcal{C}^\ell_{\mathtt{@a}}\psi$ with $\ell \neq 0$ either we have $\psi = \sim\mathtt{@b}$ or $\psi = \neg\sim\mathtt{@b}$ for some attribute $\mathtt{b}$.

Next, we define an auxiliary formula which will be useful in the translations of B-DNL-formulas into formulas in normal form. Given an $\ell \geq 1$, a $k \in [0, \ldots, \ell]$, an attribute set $\mathtt{Att}$ and an attribute $\mathtt{a} \in \mathtt{Att}$, we define the formula $\mathtt{count}^{k,\ell}_{\mathtt{Att},\mathtt{@a}}$ in normal form which is true at a position i of a data word w over $\mathtt{Att}$ if and only if position $i+\ell$ exists and $(i, \ldots, i+\ell] \cap \mathtt{clpos}(w, \mathtt{val}(w, i, \mathtt{@a})) = k$, i.e., there are exactly k positions in $(i, \ldots, i+\ell]$ where the value of attribute $\mathtt{a}$ from position i occurs:

$$\mathtt{count}^{k,\ell}_{\mathtt{Att},\mathtt{@a}} = \mathbf{X}^{\ell}\top \wedge \bigvee_{I \subseteq [1,\ldots,\ell], |I|=k} \Big[\big(\bigwedge_{j \in I} \bigvee_{\mathtt{b} \in \mathtt{Att}} \mathcal{C}^{j}_{\mathtt{@a}} \sim\mathtt{@b} \big) \wedge \big(\bigwedge_{j \in [1,\ldots,\ell] \setminus I} \bigwedge_{\mathtt{b} \in \mathtt{Att}} \neg \mathcal{C}^{j}_{\mathtt{@a}} \sim\mathtt{@b} \big) \Big]$$

For $\ell \leq -1$ and $k \in [0, \ldots, |\ell|]$ we can similarly define a formula $\mathtt{count}^{k,\ell}_{\mathtt{Att},\mathtt{@a}}$ which expresses that position $i+\ell$ exists and there are exactly k positions in $[i+\ell, \ldots, i)$ where the $\mathtt{a}$-value of position i appears.

Proposition 1. Every B-DNL-formula can be translated into an equivalent formula in normal form.

Proof. Let φ be a B-DNL-formula over some attribute set $\mathtt{Att}$. We give a procedure which translates φ into an equivalent formula in normal form. Starting from the innermost sub-formulas, the procedure converts every sub-formula into an equivalent one in normal form. Clearly, it suffices to consider formulas of the form $\mathcal{C}^{\ell}_{\mathtt{@a}}\psi$.

Let $\mathcal{C}^{\ell}_{\mathtt{@a}}\psi$ be a sub-formula of φ such that $\ell \neq 0$ and ψ is already in normal form, but neither of the form $\sim\mathtt{@b}$ nor of the form $\neg \sim\mathtt{@b}$ for any attribute $\mathtt{b}$. By performing the following steps, $\mathcal{C}^{\ell}_{\mathtt{@a}}\psi$ is converted into an equivalent formula in the desired form.

1. Using the rules of De Morgan, the formula ψ is translated into an equivalent formula where the negation operator occurs at most immediately in front of global formulas, basic class formulas and the $\sim\mathtt{@}$-operator.

2. With the help of the equivalences

   - $\mathcal{C}^{\ell}_{\mathtt{@a}}(\chi_1 \wedge \chi_2) \equiv \mathcal{C}^{\ell}_{\mathtt{@a}}\chi_1 \wedge \mathcal{C}^{\ell}_{\mathtt{@a}}\chi_2$,
   - $\mathcal{C}^{\ell}_{\mathtt{@a}}(\chi_1 \vee \chi_2) \equiv \mathcal{C}^{\ell}_{\mathtt{@a}}\chi_1 \vee \mathcal{C}^{\ell}_{\mathtt{@a}}\chi_2$,
   - $\mathcal{C}^{\ell}_{\mathtt{@a}}\neg\chi \equiv \mathcal{C}_{\mathtt{@a}} \sim\mathtt{@a} \wedge \mathbf{X}^{\ell}\top \wedge \neg\mathcal{C}^{\ell}_{\mathtt{@a}}\chi$ (observe that $\mathcal{C}_{\mathtt{@a}} \sim\mathtt{@a}$ just guarantees that the value of attribute $\mathtt{a}$ is defined at the current position), and
   - $\mathcal{C}^{\ell}_{\mathtt{@a}}\chi \equiv \mathbf{X}^{\ell}\chi$ for global formulas χ

   it is enforced that in all sub-formulas of the form $\mathcal{C}^{\ell}_{\mathtt{@a}}\chi$, the sub-formula χ is either a basic class formula or of the form $\sim\mathtt{@a}$. Thus, it remains to deal with formulas $\mathcal{C}^{\ell}_{\mathtt{@a}}\chi$ where χ is a basic class formula.

3. In this step, every formula of the form $\mathcal{C}^{\ell}_{\mathtt{@a}}\chi$ where χ is a basic class formula, is replaced by an equivalent formula $\chi^{\ell}_{\mathtt{@a}}$ in normal form which uses the auxiliary formula $\mathtt{count}^{k,\ell}_{\mathtt{Att},\mathtt{@a}}$. The definition of $\chi^{\ell}_{\mathtt{@a}}$ depends on whether ℓ is positiv or negativ and χ is a future or a past formula. For instance, in case that $\ell > 0$ and χ is a future formula, the formula $\chi^{\ell}_{\mathtt{@a}}$ evaluates χ at the smallest position which is in the class of the current data value and greater by at least ℓ. In the case that $\ell > 0$ and χ is a past formula, χ is evaluated at the greatest class position of distance at most ℓ. Observe that this is in accordance with the formal semantics of B-DNL. We give the full definition of $\chi^{\ell}_{\mathtt{@a}}$:

- If χ is a future formula and $\ell > 0$, then

$$\chi^{\ell}_{@a} = \bigvee_{k=0}^{\ell} \left[\mathtt{count}^{k,\ell}_{\mathtt{Att},@a} \wedge \left[\left((\bigvee_{b \in \mathtt{Att}} \mathcal{C}^{\ell}_{@a} \sim @b) \wedge \mathcal{C}^{0}_{@a} \mathbf{X}^{k}_{=} \chi \right) \vee \left((\neg \bigvee_{b \in \mathtt{Att}} \mathcal{C}^{\ell}_{@a} \sim @b) \wedge \mathcal{C}^{0}_{@a} \mathbf{X}^{k+1}_{=} \chi \right) \right] \right].$$

- If χ is a past formula and $\ell < 0$, then

$$\chi^{\ell}_{@a} = \bigvee_{k=0}^{|\ell|} \left[\mathtt{count}^{k,\ell}_{\mathtt{Att},@a} \wedge \left[\left((\bigvee_{b \in \mathtt{Att}} \mathcal{C}^{\ell}_{@a} \sim @b) \wedge \mathcal{C}^{0}_{@a} \mathbf{X}^{-k}_{=} \chi \right) \vee \right. \right.$$
$$\left. \left. \left((\neg \bigvee_{b \in \mathtt{Att}} \mathcal{C}^{\ell}_{@a} \sim @b) \wedge \mathcal{C}^{0}_{@a} \mathbf{X}^{-(k+1)}_{=} \chi \right) \right] \right].$$

- If χ is a future formula and $\ell < 0$, then

$$\chi^{\ell}_{@a} = \bigvee_{k=0}^{|\ell|} (\mathtt{count}^{k,\ell}_{\mathtt{Att},@a} \wedge \mathcal{C}^{0}_{@a} \mathbf{X}^{-k}_{=} \chi).$$

- If χ is a past formula and $\ell > 0$, then

$$\chi^{\ell}_{@a} = \bigvee_{k=0}^{\ell} (\mathtt{count}^{k,\ell}_{\mathtt{Att},@a} \wedge \mathcal{C}^{0}_{@a} \mathbf{X}^{k}_{=} \chi).$$

□

Next, we highlight the relationship between path expressions and DFA. To this end, we first define some notions. Let ρ be a global path expression. The *component set* $\mathtt{Comp}(\rho)$ of ρ consists of all maximal formulas the expression ρ is composed of. More formally,

- $\mathtt{Comp}(\rho) = \emptyset$ if $\rho = \varepsilon$,
- $\mathtt{Comp}(\rho) = \{\varphi\}$ if ρ is a global formula φ,
- $\mathtt{Comp}(\rho) = \mathtt{Comp}(\rho_1) \cup \mathtt{Comp}(\rho_2)$ if ρ is of the form $\rho_1 \cdot \rho_2$ or $\rho_1 + \rho_2$, and
- $\mathtt{Comp}(\rho) = \mathtt{Comp}(\rho')$ if ρ is of the form ρ'^*.

We define $\mathtt{Prop}^{\mathtt{C}}_{\rho}$ as the set $\{p_\varphi | \varphi \in \mathtt{Comp}(\rho)\}$ of propositions. For instance, for $\rho = (\langle (p \cdot p) + q \rangle r \cdot \langle r^* \rangle p)^*$ we have $\mathtt{Prop}^{\mathtt{C}}_{\rho} = \{p_{\langle (p \cdot p)+q \rangle r}, p_{\langle r^* \rangle p}\}$. Obviously, if ρ' results from ρ by replacing every maximal formula $\varphi \in \mathtt{Comp}(\rho)$ by p_φ, then, ρ' constitutes a usual regular expression over $\mathtt{Prop}^{\mathtt{C}}_{\rho}$. For a basic global formula $\varphi = \langle \rho \rangle \chi$, we call the regular language described by the regular expression $\rho' \cdot p_\chi$ the *regular language induced* by φ. The induced language for basic class formulas is defined analogously. The following observation is straightforward.

Observation 1. For every basic formula φ, one can construct a DFA $\mathcal{A}_\varphi$ which decides the regular language induced by φ.

The class of RA constructed in the proof of the next proposition will help to deal with the shift values of class operators in the decidability proof for B-DNL. We say that a 1-complete data word w is *valid* with respect to a proposition $=_\ell$ with $\ell \in \mathbb{Z}$ if for every position i of w it holds that i is labelled by $=_\ell$ if and only if position $i + \ell$ exists and has the same data value as position i. For $\ell \in \mathbb{Z}$, let $\mathcal{L}_{=_\ell}$ be the language of 1-complete data words over the single proposition $=_\ell$ which are valid with respect to $=_\ell$.

Proposition 2. For every $\ell \in \mathbb{Z}$, the language $\mathcal{L}_{=_\ell}$ can be decided by an RA.

Proof. We show that for every $\ell \in \mathbb{Z}$, one can construct an RA which decides the language $\mathcal{L}_{=_\ell}$. For the sake of simplicity, we describe a relaxed register automaton as introduced in Section 4.2.1. The automaton needs only ℓ registers. We define its input alphabet by $\Sigma = \{\emptyset, \{=_\ell\}\}$. In case of $\ell = 0$ the construction of the automaton is obvious. We consider the case for $\ell > 0$. The strategy of the automaton is described as follows. In each step, the automaton holds the data values of the last ℓ positions in its registers. Furthermore, it keeps in its state the equality conditions for all registers, i.e., the answer of the question after how many steps the input of which register has to be equal or unequal to the data value at that position. If an equality test succeeds at some position i, the corresponding register r is rewritten by the data value of position i and, depending on the proposition at i, the equality conditions are updated. In particular, if position i is marked by $=_\ell$, the automaton updates its state by the information that the data value of position $i + \ell$ has to be equal to the data value of register r. Analogously, if it is not marked by $=_\ell$, the automaton stores the information that position $i + \ell$ must not exist or must have a different data value than that of r. Note that an equality test with some register r can be carried out by a (relaxed) transition of the form $(r, s, \sigma) \to s'$ and an inequality test by one of the form $(\{r\}, s, \sigma) \to (r', s')$. In case of $\ell < 0$ the only difference is that for each position, the automaton guesses in advance whether the ℓ-next position is marked by $=_\ell$ or not. □

After these preparations, we can give the proof of the following theorem.

Theorem 2. *Satisfiability for B-DNL on finite 1-complete data words is decidable.*

Proof. Let φ be a B-DNL-formula over some proposition set $\mathtt{Prop}$ and a single attribute. We will construct a DA $\mathcal{D}_\varphi = (\mathcal{B}_\varphi, \mathcal{C}_\varphi)$ which is non-empty if and only if φ is satisfiable. Having this, the desired result follows from the decidability of the non-emptiness problem for DA [40].

Due to Proposition 1, w.l.o.g. we can assume that φ is in normal form. We introduce the proposition set $\mathtt{Prop}_\varphi^{\mathtt{sub}} = \{p_\chi \mid \chi \text{ is a (global or class) sub-formula of } \varphi\}$. In order to deal with the class operator $\mathcal{C}^\ell$, we additionally define the proposition set $\mathtt{Prop}_\varphi^{=} = \{=_r, =_{r+1}, \dots, =_{k-1}, =_k\}$ where r and k are, respectively, the smallest and greatest shift values occurring in φ. The DA $\mathcal{D}_\varphi$ reads 1-complete data words over the proposition set $\mathtt{Prop} \cup \mathtt{Prop}_\varphi^{\mathtt{sub}} \cup \mathtt{Prop}_\varphi^{=}$. We say that a 1-complete data word w is *valid* with respect to a sub-formula χ of φ if for every position i of w, it holds that i is labelled by p_χ if and only if

- $w, i \models \chi$ in the case that χ is a global formula, and
- $w, i \models \mathcal{C}\chi$ in the case that χ is a class formula.

It is easy to see that φ is satisfiable if and only if there is a data word w such that w is valid with respect to all sub-formulas of φ, it is valid with respect to all propositions in $\mathtt{Prop}_\varphi^{=}$ and the first position of w is labelled by p_φ. Keeping in mind that DA are closed under intersection [40], it is easy to see that $\mathcal{D}_\varphi$ is obtained by the intersection of

- some DA $\mathcal{D}_{\mathtt{init}}$ which checks that the first position of the input word is labelled by p_φ,
- some DA $\mathcal{D}_{=_\ell}$ for every $=_\ell \in \mathtt{Prop}_\varphi^{=}$, which checks that the input word is valid with respect to $=_\ell$, and
- some DA $\mathcal{D}_\chi$ for every sub-formula χ of φ, which checks that the input word is valid with respect to χ, under the assumption that it is valid with respect to all strict sub-formulas of χ.

The construction of $\mathcal{D}_{\texttt{init}}$ is easy. By Proposition 2, for every $=_\ell \in \texttt{Prop}^{=}_{\varphi}$, one can construct an RA which tests that the input word is valid with respect to $=_\ell$. As RA are captured by DA [39], the DA $\mathcal{D}_{=_\ell}$ is constructible for every $=_\ell \in \texttt{Prop}^{=}_{\varphi}$. It remains to show how for every sub-formula χ of φ, the automaton $\mathcal{D}_\chi$ can be constructed. We make a case distinction on the structure of χ and remind the reader that φ is in normal form. This means that no basic class formula is in the scope of any class operator with another shift value than 0. Note also that for every sub-formula χ of φ, we can assume that $\mathcal{D}_\chi$ reads data words which are already valid with respect to all strict sub-formulas of χ. We omit the straightforward cases.

- $\chi = \langle\rho\rangle\psi$: The main work is done by the base automaton of $\mathcal{D}_\chi$. The class automaton accepts all input words. The base automaton is constructed as follows. By Observation 1, we can construct a DFA $\mathcal{A}_\chi$ which decides the regular language induced by χ. Let $\mathcal{A}'_\chi$ be a simple extension of $\mathcal{A}_\chi$ which checks that the input word has a prefix matching a word from the language of $\mathcal{A}_\chi$. Moreover, let $\overline{\mathcal{A}'_\chi}$ be the automaton deciding the complement of the language of $\mathcal{A}'_\chi$. Using $\mathcal{A}'_\chi$ and $\overline{\mathcal{A}'_\chi}$ we can construct a Finite Alternating Automaton (AFA, for definition, see Section 3.2) $\mathcal{A}''_\chi$ which starts $\mathcal{A}'_\chi$ at every position marked by p_χ and starts $\overline{\mathcal{A}'_\chi}$ at all other positions. By [56, 62] $\mathcal{A}''_\chi$ can be converted into a DFA $\mathcal{A}'''_\chi$. From this DFA we easily construct a Letter-To-Letter Transducer (LLT) which simulates $\mathcal{A}'''_\chi$ on the input part and outputs arbitrary symbols. This LLT constitutes the base automaton of $\mathcal{D}_\chi$.

- $\chi = \langle\rho\rangle^{\leftarrow}\psi$: The construction is similar to the last case. The main difference is that the base automaton results from a Two-Way Alternating Finite Automaton ($\text{AFA}^{\leftrightarrow}$) which at every position marked by p_χ starts a sub automaton $\mathcal{A}'_\chi$ which moves backwards and ensures that the so far read word has a suffix matching a word from the reverse language of that induced by $\langle\rho\rangle\psi$. At positions which are not marked by p_χ it is checked by a complementary sub automaton that there is no such suffix.

- $\chi = \langle\theta\rangle_{=}\psi$: Note that due to the definition of the normal form, χ cannot occur in the scope of a class formula whose shift value is different from 0. Thus, we can assume that χ is implicitly preceded by $\mathcal{C}^0$. Moreover, due to the semantics of B-DNL, it can be assumed that all formulas in the component set of θ and the formula ψ are also implicitly preceded by $\mathcal{C}^0$. Hence, by the assumption that the input word w is valid with respect to all strict sub-formulas of χ, it suffices to check that every position i with some data value d the label p_χ is a starting point of a sequence in the d-class in w which matches a word from the language induced by χ. Consequently, this case is analogue to the case $\chi = \langle\rho\rangle\psi$ with the difference that the roles of the base and class automaton are interchanged. Here, the class automaton fulfills the main work and the base automaton just relays the input symbols to the output. The construction of the class automaton is analogue to the construction of the base automaton in case $\chi = \langle\rho\rangle\psi$. To be precise, we first construct an automaton $\mathcal{A}_\chi$ which decides the language induced by $\langle\theta\rangle_{=}\psi$. Then, the class automaton is the DFA resulting from the AFA which (i) for every position marked by p_χ, starts a sub automaton assuring that the remaining part of the word has a prefix matching a word in the language of $\mathcal{A}_\chi$, and (ii) for every other position, spawns an automaton testing that there is no such prefix.

- $\chi = \mathcal{C}\psi$: Due to the validity of the input word with respect to strict sub-formulas of χ, it suffices that the base automaton checks that a position is labelled by p_χ if and only if it is labelled by p_ψ also. There is no task to do for the class automaton.

- $\chi = \mathcal{C}^\ell\!\sim$ with $\ell \neq 0$: As it can be assumed that the input word is valid with respect to $=_\ell$, the base automaton just ensures that a position is labelled by p_χ if and only if it is labelled by $=_\ell$.

- $\chi = \mathcal{C}^{\ell}\neg\sim$ with $\ell \neq 0$: In analogy to the latter case, the base automaton makes sure that a position is labelled by p_χ if and only if it is not labelled by $=_\ell$.
- $\chi = \sim$: Since formulas of the form $\mathcal{C}^{\ell}\sim$ with $\ell \neq 0$ are handled separately, we can assume here that χ is implicitly preceded by $\mathcal{C}$. Then, due to the reason that we deal with 1-complete data words, such a formula holds at all positions. Hence, the base automaton guarantees that all positions are labelled by p_χ.
- $\chi = \neg\psi$: Observe that, due to our construction in the other cases, it is sufficient that the base automaton assures that a position is labelled by p_χ if and only if it is not labelled by p_ψ.

□

Now, we can state the main theorem of this section.

Theorem 3. *Satisfiability for B-DNL on finite data words is decidable.*

Proof. We will show that the general satisfiability problem for B-DNL can be reduced to the satisfiability problem for B-DNL on 1-complete data words. Then, the result follows by Theorem 2.

Let φ be a B-DNL formula. Due to Proposition 1, we can assume w.l.o.g. that φ is in normal form. We will translate φ into a formula φ' such that φ is satisfiable if and only if φ' is satisfiable on 1-complete words. The formula φ' will simulate φ on an encoding of general data words which is similar to the encoding used in [79].

Let $\texttt{Prop}$ and $\texttt{Att} = \{\texttt{a}_1, \dots, \texttt{a}_m\}$ be, respectively, the set of propositions and attributes occurring in φ. We first explain how we encode general data words over $\texttt{Prop}$ and $\texttt{Att}$ by 1-complete ones over a single attribute and the proposition set $\texttt{Prop} \cup \{\texttt{a}_1, \dots \texttt{a}_m, D\}$ with some fresh proposition $D \notin \texttt{Prop} \cup \{\texttt{a}_1, \dots, \texttt{a}_m\}$. Figure 6.1 presents the encoding of an exemplary word.

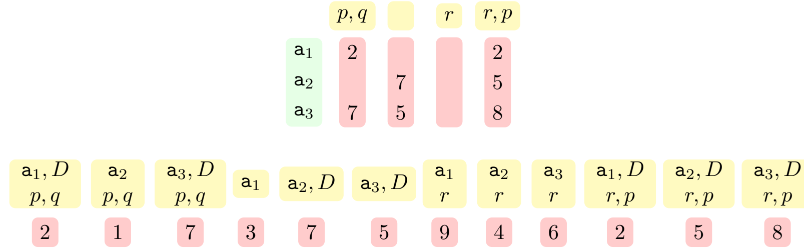


Figure 6.1: The encoding of a general data word by a 1-complete one

Every position i of a general data word w is represented in an encoding w' by a sequence of m positions called *the block for position* i. Each position of such a block carries the proposition set $\texttt{props}(w, i)$. Moreover, for every $j \in \{1, \dots, m\}$, the j-th position of the block carries proposition $\texttt{a}_j$ and no other $\texttt{a}_k$ with $k \neq j$. Furthermore, if attribute $\texttt{a}_j$ is defined at position i of w, then, the j-th position of the block additionally carries proposition D and the data value $\texttt{val}(w, i, @\texttt{a}_j)$ of attribute $\texttt{a}_j$ at position i of w. Otherwise, the j-th position of the block does not have proposition D and it carries an arbitrary data value not appearing anywhere else in the encoding.

The formula φ' is evaluated on 1-complete data words over the proposition set $\texttt{Prop} \cup \{\texttt{a}_1, \dots, \texttt{a}_m, D\}$ and a single attribute, say $\texttt{a}$. It enforces that every satisfying word represents an encoding of a data word over $\texttt{Prop}$ and $\texttt{Att}$. The formula consists of a conjunction of the formulas φ_e and $t(\varphi)$ where φ_e expresses that the data word is indeed an encoding as described above and $t(\varphi)$ simulates φ on the encoding.

The formula φ_{e} is the conjunction of φ_{e1}, φ_{e2} and φ_{e3}. The first conjunct expresses that every position carries exactly one proposition from $\{\mathtt{a}_1, \ldots, \mathtt{a}_m\}$ and the word consists of blocks of length m:

$$\varphi_{e1} = \mathtt{a}_1 \wedge \mathbf{G}\Big[\bigvee_{i=1}^{m} \mathtt{a}_i \wedge \bigwedge_{i=1}^{m} \big(\mathtt{a}_i \rightarrow \bigwedge_{j\in\{1,\ldots,m\}\setminus\{i\}} \neg \mathtt{a}_j\big) \wedge \bigwedge_{i=1}^{m-1} \big(\mathtt{a}_i \rightarrow \mathbf{X}\mathtt{a}_{i+1}\big) \wedge \big(\mathtt{a}_m \rightarrow (\mathbf{X}\mathtt{a}_1 \vee \neg\mathbf{X}\top)\big)\Big].$$

The second conjunct φ_{e2} states that all positions within the same block carry the same propositions from $\mathtt{Prop}$:

$$\varphi_{e2} = \mathbf{G}\Big[\bigwedge_{i=1}^{m-1} \big(\mathtt{a}_i \rightarrow \bigwedge_{p\in\mathtt{Prop}} (p \leftrightarrow \mathbf{X}p)\big)\Big].$$

The last conjunct guarantees that each data value appearing at a position without proposition D occurs only once:

$$\varphi_{e3} = \mathbf{G}\big[\neg D \rightarrow \mathcal{C}(\neg\mathbf{X}^{\leftarrow}\top \wedge \neg\mathbf{X}\top)\big].$$

Before giving the full definition of the translation t, we illustrate by an example the underlying idea. Suppose that m equals 5. We will define t in such a way that for a formula

$$\xi = \langle p \cdot q\rangle(\mathcal{C}_{@\mathtt{a}_3}\langle r^*\rangle_{=}q)$$

the formula $t(\xi)$ is equivalent to

$$\langle p \cdot \top^4 \cdot q \cdot \top^4\rangle\mathbf{X}^2\Big(D \wedge \mathcal{C}\big\langle\big[\Sigma_{i=1}^{5}\Sigma_{k=0}^{5-i}((r \wedge \mathtt{a}_i \wedge \mathtt{count}^{k,5-i}_{\{\mathtt{a}\},@\mathtt{a}}) \cdot \top^k)\big]^*\big\rangle_{=}q\Big).$$

Due to our assumption that every block has length 5, after the evaluation of each of the formulas p and q, the control moves (by $\top^4$) to the last position of the current block so that the following formula is evaluated in the following block. As the class operator in ξ fetches the data value of $\mathtt{a}_3$, the control in the resulting formula first navigates (by $\mathbf{X}^2$) to the position representing $\mathtt{a}_3$ in the current block, assures (by D) that the value of this attribute is defined and applies the class operator. In order to avoid the failure that two concatenated formulas within the following class path expression are evaluated in the same block, the control is shifted to the last class position within the current block after each evaluation of r. The distance of these position is determined with the help of the attribute represented by the current position and the auxiliary formula $\mathtt{count}$.

We now describe the definition of translation t in detail. Given a formula ψ, we obtain the formula $t(\psi)$ by inductive construction. The construction makes use of the auxiliary translations t_i, $t_{\mathtt{gex}}$, $t_{\mathtt{cex}}$, $t^{-}_{\mathtt{gex}}$ and $t^{-}_{\mathtt{cex}}$. For a global formula χ, the formula $t_i(\chi)$ enforces that χ is evaluated on the i-th position of the current block:

$$t_i(\chi) = \bigwedge_{j=1}^{m} (\mathtt{a}_j \rightarrow \mathbf{X}^{i-j}\chi)$$

The other auxiliary translations are applied on path expressions and will be explained later. In the following, please keep in mind that t gets formulas in normal form. We start with the translation of global formulas. Recall that ℓ steps into one direction on the original word corresponds to $m\ell$ steps into the same direction on the encoding:

- $t(p) = p$ for propositions p
- $t(\neg\chi) = \neg t(\chi)$

- $t(\chi_1 \wedge \chi_2) = t(\chi_1) \wedge t(\chi_2)$
- $t(\langle\rho\rangle\chi) = \langle t_{\mathtt{gex}}(\rho)\rangle t(\chi)$
- $t(\langle\rho\rangle^{\leftarrow}\chi) = \langle t^{-}_{\mathtt{gex}}(\rho)\rangle^{\leftarrow} t(\chi)$
- $t(\mathcal{C}^{\ell}_{@\mathtt{a}_i} \sim @\mathtt{a}_j) = t_i(D \wedge \mathcal{C}^{m\ell+j-i}(D\wedge \sim))$ for $i, j \in \{1, \dots, m\}$ and $\ell \in \mathbb{Z}$
- $t(\mathcal{C}^{\ell}_{@\mathtt{a}_i}\neg \sim @\mathtt{a}_j) = t_i(D \wedge \mathcal{C}^{m\ell+j-i}(\neg D \vee \neg \sim))$ for $i, j \in \{1, \dots, m\}$ and $\ell \in \mathbb{Z}$
- $t(\mathcal{C}_{@\mathtt{a}_i}\chi) = t_i(D \wedge \mathcal{C}t(\chi))$ for $i \in \{1, \dots, m\}$.

We proceed with the translation for class formulas. Remember that the data values of a single position in the original word are distributed over an entire block in the encoding:

- $t(\sim @\mathtt{a}_i) = \bigwedge_{j=1}^{m} \big(\mathtt{a}_j \to \mathcal{C}^{i-j}(D\wedge \sim)\big)$ where $i \in \{1, \dots, m\}$
- $t(\langle\theta\rangle_{=}\chi) = \langle t_{\mathtt{cex}}(\theta)\rangle_{=} t(\chi)$
- $t(\langle\theta\rangle^{\leftarrow}_{=}\chi) = \langle t^{-}_{\mathtt{cex}}(\theta)\rangle^{\leftarrow}_{=} t(\chi)$.

Now, we present with the translation of global path expressions. As demonstrated in the example translation of ξ, for global formulas χ that are occurring within future expressions, the formula $t_{\mathtt{gex}}(\chi)$ first evaluates χ on the current position and then navigates to the last position of the current block. The result is that a potentially concatenated following global formula will be evaluated on the consecutive block. For global formulas that are occurring in past expressions, the first position of the current block has to be found. The distance of the last or first position of a block is determined with the help of the propositions $\{\mathtt{a}_1, \dots, \mathtt{a}_m\}$:

- $t_{\mathtt{gex}}(\chi) = \Sigma_{i=1}^{m}((t(\chi) \wedge \mathtt{a}_i) \cdot \top^{m-i})$
- $t^{-}_{\mathtt{gex}}(\chi) = \Sigma_{i=1}^{m}((t(\chi) \wedge \mathtt{a}_i) \cdot \top^{i-1})$
- $t_{\mathtt{gex}}(\varepsilon) = t^{-}_{\mathtt{gex}}(\varepsilon) = \varepsilon$
- $t_{\mathtt{gex}}(\rho_1 \cdot \rho_2) = t_{\mathtt{gex}}(\rho_1) \cdot t_{\mathtt{gex}}(\rho_2)$ and $t^{-}_{\mathtt{gex}}(\rho_1 \cdot \rho_2) = t^{-}_{\mathtt{gex}}(\rho_1) \cdot t^{-}_{\mathtt{gex}}(\rho_2)$
- $t_{\mathtt{gex}}(\rho_1 + \rho_2) = t_{\mathtt{gex}}(\rho_1) + t_{\mathtt{gex}}(\rho_2)$ and $t^{-}_{\mathtt{gex}}(\rho_1 + \rho_2) = t^{-}_{\mathtt{gex}}(\rho_1) + t^{-}_{\mathtt{gex}}(\rho_2)$
- $t_{\mathtt{gex}}(\rho^*) = (t_{\mathtt{gex}}(\rho))^*$ and $t^{-}_{\mathtt{gex}}(\rho^*) = (t^{-}_{\mathtt{gex}}(\rho))^*$.

Finally, we give the translation for class path expressions. The evaluation of formulas within class path expressions is similar to that in global path expressions. If a formula χ occurs within a future class path expression, the evaluation of χ is followed by a navigation to the last position of the current class within the current block. Analogously, if χ occurs in a past expression, the control moves to the first position of the current class within the current block. In order to find these position we use the auxiliary formula `count`:

- $t_{\mathtt{cex}}(\psi) = \Sigma_{i=1}^{m}\Sigma_{k=0}^{m-i}((t(\psi) \wedge \mathtt{a}_i \wedge \mathtt{count}^{k,m-i}_{\{\mathtt{a}\},@\mathtt{a}}) \cdot \top^{k})$
- $t^{-}_{\mathtt{cex}}(\psi) = \Sigma_{i=1}^{m}\Sigma_{k=0}^{i-1}(t(\psi) \wedge \mathtt{a}_i \wedge \mathtt{count}^{k,-(i-1)}_{\{\mathtt{a}\},@\mathtt{a}} \cdot \top^{k})$.

We omit the translations for more complex class path expressions as their translations are along the lines of the translations for global path expressions. □

6.3 Basic Data Navigation Logic on Infinite Data Words

In this section, we will work out how the decidability proof for the satisfiability of B-DNL on finite data words which has been given in the previous section can be extended to data ω-words. We will highlight the main differences between the finite and infinite cases.

First of all, note that the translation given in Proposition 1 carries over to B-DNL-formulas on ω-words which means that every B-DNL-formula on data ω-words can be converted into an equivalent one in normal form. Moreover, we observe that the encoding of general data words by 1-complete ones developed in Theorem 3 as well as the simulation of B-DNL-formulas (in normal form) on this encoding works smoothly for infinite words. Thus, satisfiability for B-DNL on data ω-words can be reduced to the satisfiability for B-DNL on 1-complete data ω-words. Therefore, it suffices to transfer the result in Theorem 2 to ω-words, i.e., to prove that satisfiability for B-DNL on 1-complete data ω-words is decidable.

To this end, we first notice that Observation 1 and Proposition 2 used in the the proof of Theorem 2 can be easily extended to data ω-words and Büchi automata. We just have to adapt some notions. Let ρ' be the regular expression resulting from a global path expression ρ by replacing every formula φ from the component set of ρ by p_φ. For a basic global formula $\varphi = \langle\rho\rangle\chi$, we call the ω-regular language consisting of all ω-words which contain a prefix matching $\rho' \cdot p_\chi$ the *ω-regular language induced* by φ. The induced ω-regular language for basic class formulas is defined analogously. Using these notions, the adaption of Observation 1 is straightforward:

Observation 2. For every basic formula φ one can construct a Büchi automaton $\mathcal{A}_\varphi$ which decides the ω-regular language induced by φ.

Now, we turn to the adaption of Proposition 2. For $\ell \in \mathbb{Z}$, let ω-$\mathcal{L}_{=_\ell}$ be the set of all 1-complete data ω-words over the single proposition $=_\ell$ which are valid with respect to $=_\ell$. Basically, the same technique described in the proof of Proposition 2 can be used to decide these languages by a Büchi Register Automata (BRA). Thus we get:

Proposition 3. For every $\ell \in \mathbb{Z}$, the language ω-$\mathcal{L}_{=_\ell}$ can be decided by a BRA.

After this preparation we can state the ω-counterpart of Theorem 2. While there, decidability was established by reduction to non-emptiness of Data Automata, here, we reduce to non-emptiness of Büchi Data Automata (BDA).

Theorem 4. *Satisfiability for B-DNL on 1-complete data ω-words is decidable.*

Proof. Let φ be a B-DNL-formula over some proposition set $\texttt{Prop}$ and a single attribute. We construct a BDA $\mathcal{D}_\varphi = (\mathcal{B}, \mathcal{C}, \mathcal{C}_\omega)$ which is non-empty if and only if φ is satisfiable. Then, the result follows from the decidability of the non-emptiness problem for BDA [40].

The proof proceeds along the same lines as that of Theorem 4. We put particular emphasis on the differences. We assume that φ is in normal form. In analogy to the finite case, the automaton $\mathcal{D}_\varphi$ reads 1-complete data ω-words over the proposition set $\texttt{Prop} \cup \texttt{Prop}_\varphi^{sub} \cup \texttt{Prop}_\varphi^{=}$ and is the intersection of several auxiliary data automata which check that (i) the first position of the input word is labelled by p_φ, (ii) the word is valid with respect to every $=_\ell \in \texttt{Prop}_\varphi^{=}$, and (iii) it is valid with respect to every sub-formula χ of φ, under the assumption that the it is valid with respect to all strict sub-formulas of χ. If we can show that these auxiliary automata are constructible, the constructibility of $\mathcal{D}_\varphi$ will follow easily from the closure of BDA under intersection [40].

In the rest of this proof, we deal with the construction of the mentioned auxiliary BDA. The question how it can be checked that the initial position of the input word is marked by p_φ does not need any explanation. Moreover, using Proposition 3 we can construct for every $=_\ell \in \texttt{Prop}_\varphi^{=}$, a BRA which checks the validity of the input word with respect to $=_\ell$. By [39, 41] BRA are captured by BDA.

In the following, we explain how to construct an automaton $\mathcal{D}_\chi$ for every sub-formula χ of φ which checks the validity of the input word with respect to χ under the assumption that the word is valid with respect to all strict sub-formulas of χ. We omit the cases where the construction is almost the same as in the case of finite words:

- $\chi = \langle\rho\rangle\psi$: Such formulas are checked by the base automaton. The difference to the proof of Theorem 2 is that we use, instead of Alternating Finite Automata, Alternating Finite Büchi Automata (BAFA, for definition, see Section 3.2) as an intermediate tool for the construction of the base automaton. By Observation 2, we can construct a Büchi automaton $\mathcal{A}_\chi$ which decides the ω-regular language induced by χ. It is known that Büchi automata are closed under complementation [193]. Let $\overline{\mathcal{A}_\chi}$ be the Büchi automaton deciding the complement language of $\mathcal{A}_\chi$. Using these automata we construct a BAFA which starts $\mathcal{A}_\chi$ at every position marked by p_χ and starts $\overline{\mathcal{A}_\chi}$ at all other positions. By [163], this automaton can be converted into a Büchi automaton $\mathcal{A}'_\chi$. From this we obtain easily a Büchi Letter-To-Letter Transducer (BLLT) simulating $\mathcal{A}'_\chi$. The latter BLLT constitutes the base automaton of $\mathcal{D}_\chi$.

- $\chi = \langle\rho\rangle^{\leftharpoonup}\psi$: In comparison to the proof of Theorem 2, in this case we work with Two-Way Alternating Büchi Automata (BAFA$^{\leftrightarrow}$). At every position marked by p_χ the automaton starts an automaton which checks that the so far read word has a suffix matching a word from the reverse language of the language induced by $\langle\rho\rangle\psi$. At positions not marked by p_χ it is checked that there is no such suffix. By [138], this BAFA$^{\leftrightarrow}$ can be converted into a Büchi automaton from which we get the base automaton of $\mathcal{D}_\chi$.

- $\chi = \langle\rho\rangle_{=}\psi$: This kind of formulas are checked by the class automaton of $\mathcal{D}_\chi$. The main difference to the proof of Theorem 2 is that we have to take into account that the data ω-word can have finite as well as infinite classes. The finite classes are handled by $\mathcal{C}$, the infinite ones by $\mathcal{C}_\omega$.

□

Based on the reduction in Theorem 3 we conclude:

Theorem 5. *Satisfiability for B-DNL on data ω-words is decidable.*

6.4 Undecidable Extensions of Basic Data Navigation Logic

In this section, we show that small extensions of B-DNL lead to undecidability of the satisfiability problem. In all proofs of this section, we reduce some undecidable problem to the satisfiability problem of the considered extended logic, i.e., we construct a formula on finite data words encoding the problem. Nevertheless, the same problems can easily be encoded on data ω-words by requiring that a finite prefix of the words are labelled by some special proposition and restricting formula evaluation to these prefixes. Therefore, although our proofs are carried out on finite words, the results hold also for ω-words.

Breaking up class navigation

Basically, the very essence of B-DNL is constituted by navigation along consecutive word positions and consecutive class positions. Therefore, its abilities to express properties *between* two class positions are restricted. If the corresponding class positions are of bounded distance, specifying positions in between is easy. For instance, the property saying that

there is a p-position i and a following r-position j such that i and j have distance 5, they agree on the value of attribute a *and there is a q-position in between*

is expressed by the formula $\mathbf{F}(p \wedge \mathcal{C}^5_{\texttt{@a}}(\sim\texttt{@a} \wedge r) \wedge \bigvee_{i=1}^{4} \mathbf{X}^i q)$. However, it is not clear how the following Property CS6 discussed in Chapter 2 and Example 10 can be expressed in B-DNL, because there is no bound on the distance between the creation of the client and the receiving of the server information:

Between the creation of a client p and the receiving of the server information by p, there is no request to the server.

In the sequel, we will introduce a new operator which makes it possible to express CS6 but whose availability leads B-DNL to undecidability. We consider an operator **E** with five arguments. The evaluation of the operator does not depend on a current position. For a data word w, two attributes $\texttt{a}, \texttt{b}$ and three propositions p_1, p_2, p_3, we define:

- $w \models \mathbf{E}_{\texttt{@a},\texttt{@b}}(p_1, p_2, p_3)$ if there are positions $i \leq j \leq k$ in w such that $(w, i) \models p_1$, $(w, j) \models p_2$, $(w, k) \models p_3$ and $\texttt{val}(w, i, \texttt{@a})$ and $\texttt{val}(w, k, \texttt{@b})$ are (defined and) equal.

Using this operator, we can express Property CS6 easily by $\neg\mathbf{E}_{\texttt{@created},\texttt{@receiver}}(\texttt{crt}, \texttt{req}, \texttt{serv})$. It is interesting that even though the property expressed by **E** (on 1-complete data words) can be checked by Register Automata (by Example 2 in Section 4.2.1) and these automata as well as B-DNL can be converted into equivalent Data Automata which are decidable, the addition of **E** to B-DNL leads to undecidability.

Theorem 6. *Satisfiability for B-DNL extended by* **E** *is undecidable on finite (and infinite) data words.*

Proof. We will show that the logic is undecidable even in the case with one attribute. The proof is by reduction from the reachability problem for Minsky Machines with two counters (2-MCM, for definition, see Section 3.2.2). As this problem is not decidable [162], the result follows.

Given a 2-MCM $\mathcal{M} = (2, S, s_0, \delta)$ and a state $s \in S$, we will construct a B-DNL-formula $\varphi_{\mathcal{M}}$ involving the **E**-operator such that the configuration $(s, 0, 0)$ is reachable in $\mathcal{M}$ if and only if $\varphi_{\mathcal{M}}$ is satisfiable. Without loss of generality, we assume that $s \neq s_0$ (otherwise it is obvious that $(s, 0, 0)$ is reachable and the construction of $\varphi_{\mathcal{M}}$ is trivial). The formula φ_M is defined over a single attribute and the proposition set $S \cup \texttt{Actions}_{\mathcal{M}}$ with $\texttt{Actions}_{\mathcal{M}} = \{\texttt{inc}_k, \texttt{dec}_k, \texttt{ifzero}_k \mid 1 \leq k \leq 2\}$. It encodes a sequence of transitions of $\mathcal{M}$ as a 1-complete data word and ensures that this sequence induces a run of $\mathcal{M}$ reaching $(s, 0, 0)$.

Before giving the definition of $\varphi_{\mathcal{M}}$, we first reiterate briefly the three consistency conditions listed in Section 3.2.2 which ensure that a sequence of transitions corresponds to a run reaching $(s, 0, 0)$. Then, we describe how a sequence of transitions can be encoded as a 1-complete data word. Finally, we show how such an encoding and the consistency conditions can be expressed in B-DNL by means of the **E**-operator.

According to Section 3.2.2, a sequence $\tau = (s_1, \texttt{act}_1, s'_1), \ldots, (s_n, \texttt{act}_n, s'_n)$ of transitions from δ induces an $\mathcal{M}$-run reaching $(s, 0, 0)$ if and only if the following conditions are fulfilled:

- *Consistency with respect to states*: It holds that $s_1 = s_0$, $s_n = s$ and for all i with $1 \leq i < n$, $s'_i = s_{i+1}$.
- *Consistency with respect to counters*: There is a bijection m from the set $\texttt{DECS}_\tau = \{i \mid 1 \leq i \leq n \text{ and } \texttt{act}_i \text{ is a decrement action}\}$ to the set $\texttt{INCS}_\tau = \{i \mid 1 \leq i \leq n \text{ and } \texttt{act}_i \text{ is an increment action}\}$ such that for each counter k and index $i \in \texttt{DECS}_\tau$ with $\texttt{act}_i = \texttt{dec}_k$, it holds $m(i) < i$ and $\texttt{act}_{m(i)} = \texttt{inc}_k$.

- *Consistency with respect to zero-tests*: For every counter k and index $i \in \texttt{DECS}_\tau$ with $\texttt{act}_i = \texttt{dec}_k$, there is no ℓ with $m(i) < \ell < i$ and $\texttt{act}_\ell = \texttt{ifzero}_k$.

Now, we explain how a sequence of n transitions can be encoded as a 1-complete data word w of length n over the proposition set $S \cup \texttt{Actions}_\mathcal{M}$. Every position i of w carries exactly one proposition $s_i \in S$ and exactly one proposition $\texttt{act}_i \in \texttt{Actions}_\mathcal{M}$. The propositions s_1 and $\texttt{act}_1$ represent the transition $(s_0, \texttt{act}_1, s_1)$ and for every i with $1 < i \leq n$, the propositions s_i and $\texttt{act}_i$ represent a transition $(s_{i-1}, \texttt{act}_i, s_i)$.

The fact that the value of the single attribute is defined at each position can be expressed by $\mathbf{G}\mathcal{C}{\sim}$. The property that a word encodes a sequence of transitions which is consistent with respect to states is also an easy task and can be done without referring to data values. In order to ensure that an encoding is consistent with respect to counters, we simulate the bijection m by making use of data values. We assure that the word respects the following conditions:

(1) All data values at inc-positions are pairwise distinct; the same holds for all data values at dec-positions.

(2) For each counter k with $1 \leq k \leq 2$ and for every $\texttt{inc}_k$-position, there is a greater $\texttt{dec}_k$-position with the same data value.

(3) Likewise, for each counter k with $1 \leq k \leq 2$ and for every $\texttt{dec}_k$-position, there is a smaller $\texttt{inc}_k$-position with the same data value.

Consistency with respect to zero-tests is assured by the following property:

(4) There is no counter k with $1 \leq k \leq 2$ such that there is an $\texttt{ifzero}_k$-position between an $\texttt{inc}_k$-position and its corresponding $\texttt{dec}_k$-position with the same data value.

Property (1) is expressed by

$$\bigwedge_{1\leq k\leq 2} \mathbf{G}\Big[\texttt{inc}_k \rightarrow \mathcal{C}\big((\neg\mathbf{X}_=\mathbf{F}_= \bigvee_{1\leq \ell\leq 2} \texttt{inc}_\ell) \wedge (\neg\mathbf{X}^{\leftarrow}_=\mathbf{F}^{\leftarrow}_= \bigvee_{1\leq \ell\leq 2} \texttt{inc}_\ell)\big)\Big]$$

$$\wedge$$

$$\bigwedge_{1\leq k\leq 2} \mathbf{G}\Big[\texttt{dec}_k \rightarrow \mathcal{C}\big((\neg\mathbf{X}_=\mathbf{F}_= \bigvee_{1\leq \ell\leq 2} \texttt{dec}_\ell) \wedge (\neg\mathbf{X}^{\leftarrow}_=\mathbf{F}^{\leftarrow}_= \bigvee_{1\leq \ell\leq 2} \texttt{dec}_\ell)\big)\Big].$$

The following formula expresses Properties (2) and (3):

$$\bigwedge_{1\leq k\leq 2} \mathbf{G}\Big[\big(\texttt{inc}_k \rightarrow \mathcal{C}\mathbf{F}_=\texttt{dec}_k\big) \wedge \big(\texttt{dec}_k \rightarrow \mathcal{C}\mathbf{F}^{\leftarrow}_=\texttt{inc}_k\big)\Big].$$

Property (4) is expressed by means of the **E**-operator. Since we have only one attribute, we forgo the attribute names attached to the operator:

$$\bigwedge_{1\leq k\leq 2} \neg\mathbf{E}(\texttt{inc}_k, \texttt{ifzero}_k, \texttt{dec}_k).$$

□

Setting limits to the past

The *From-Now-On-operator* $\mathbf{N}$ was introduced in [141] for temporal logics with access to the past and is used to restrict the range of past operators. If the operator $\mathbf{N}$ is applied at a position i, the range of all temporal operators within the scope of $\mathbf{N}$ is restricted to positions $j \geq i$. After defining the formal semantics of $\mathbf{N}$, we will show that B-DNL extended by $\mathbf{N}$ is not decidable.

We use the operator $\mathbf{N}$ only in front of global formulas. Given a data word w, a position i in w and a global B-DNL-formula φ, we define:

- $(w, i) \models \mathbf{N}\varphi$ if $(w[i, \ldots], 0) \models \varphi$.

Theorem 7. *Satisfiability for B-DNL extended by* $\mathbf{N}$ *is undecidable on finite (and infinite) data words.*

Proof. The proof is again by reduction from the reachability problem for 2-MCMs and along the same lines as the proof of Theorem 6. We highlight the differences. Given a 2-MCM $\mathcal{M} = (2, S, s_0, \delta)$, we encode sequences of transitions in the same way as in that proof. Observe that the formulas expressing Properties (1)-(3) use usual B-DNL-operators. Thus, we only have to show how Property (4) ensuring consistency with respect to zero-tests can be expressed in B-DNL+$\mathbf{N}$. Here, we make use of the $\mathbf{N}$-operator. In order to determine the sequence between an `inc`- and its corresponding `dec`-position, we apply the $\mathbf{N}$-operator at the `inc`-position, "jump" to the `dec`-position and from then on we can be sure that all past positions are in between the two positions:

$$\bigwedge_{1 \leq k \leq 2} \mathbf{G}\Big[\mathtt{inc}_k \rightarrow \mathbf{N}\mathcal{C}\mathbf{F}_{=}(\mathtt{dec}_k \wedge \mathbf{G}^{\leftarrow}\neg\mathtt{ifzero}_k)\Big]$$

□

Navigation along tuples of data values

In B-DNL, class navigation is performed with respect to a single data value: the class operator $\mathcal{C}^{\ell}_{\mathtt{@a}}$ fixes the data value of attribute `a` and restricts navigation to the class of this data value. As demonstrated in Example 10 of Section 6.1, this form of navigation suffices to express properties like CS3. For this property it is enough to keep track of the ID of the sending process. However, Property CS9 (see formulations in $\mathrm{FO}^{\sim}$ and $\mathrm{LTL}^{\Downarrow}$ in Examples 6 and 7) talks repeatedly about interactions of two processes, namely a selected server and a selected client. In order to express properties like CS9, it would be desirable to equip the class operator by the ability to fix the data values of two attributes so that navigation is restricted to all positions where both values occur. However, in the following we will show that even a restricted version of such a tuple navigation leads to undecidability.

We introduce the *Tuple-Next-operator* $\mathbf{TX}$ with three arguments and the following semantics: Given two attributes `a`, `b` and a global formula φ, the formula $\mathbf{TX}_{\mathtt{@a@b}}\varphi$ holds at some position i of a data word w if

- $\mathtt{val}(w, i, \mathtt{@a}) = d_{\mathtt{a}}$ and $\mathtt{val}(w, i, \mathtt{@b}) = d_{\mathtt{b}}$ for some data values $d_{\mathtt{a}}$ and $d_{\mathtt{b}}$,
- there is a position $j \in \mathtt{pos}(w)$ with $j > i$ where $\mathtt{val}(w, j, \mathtt{@a}) = d_{\mathtt{a}}$ and $\mathtt{val}(w, j, \mathtt{@b}) = d_{\mathtt{b}}$, and
- at the smallest such j, formula ψ holds.

We additionally define the *Tuple-Previous-operator* $\mathbf{TX}^{\leftarrow}$ as the past counterpart of $\mathbf{TX}$.

Theorem 8. *Satisfiability for B-DNL extended by* $\mathbf{TX}$ *and* $\mathbf{TX}^{\leftarrow}$ *is undecidable on finite (and infinite) data words.*

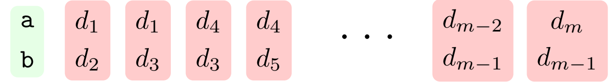
Proof. The proof is along the lines of [41]. We reduce from Post's Correspondence Problem (PCP, for definition, see Section 3.2.5).

Before giving the reduction, we note that every PCP instance I over an alphabet Σ can be extended to an instance I' such that I has a solution if an only if I' has a solution with a solution word of odd length. The instance I' results from I by adding for each pair (u_i, v_i) in I, a pair $(\#u_i, \#v_i)$ where $\#$ is a new symbol not occurring in Σ. Observe that a word u is a solution word for I if and only if $\#u$ is a solution word for I'.

Now, let $I = (u_1, v_1), \ldots, (u_k, v_k)$ be a PCP instance over Σ. We assume w.l.o.g. that if I has a solution, it has one with a solution word of odd length. We will construct a B-DNL-formula φ_I which includes the operators **TX** and $\mathbf{TX}^{\leftarrow}$ and is satisfiable if and only if there is a solution for I. The formula φ_I is defined over the proposition set $\Sigma \cup \overline{\Sigma}$ and the attribute set $\{\mathtt{a}, \mathtt{b}\}$ where $\overline{\Sigma} := \{\overline{\sigma} \mid \sigma \in \Sigma\}$. It encodes a possible solution word $u_{i_1} \ldots u_{i_n}$ for I as an $\{\mathtt{a}, \mathtt{b}\}$-complete data word w with word projection $u_{i_1}\overline{v_{i_1}} \ldots u_{i_n}\overline{v_{i_n}}$ where for every j with $1 \leq j \leq n$, $\overline{v_{i_j}}$ results from v_{i_j} by replacing every symbol σ by $\overline{\sigma}$. Additionally, the formula checks by access to data values that w indeed induces a solution word for I. More precisely, the formula φ_I expresses the following properties:

(1) At every position, both attribute values are defined and exactly one proposition holds. Moreover, the word projection of the data words is of the form $u_{i_1}\overline{v_{i_1}} \ldots u_{i_n}\overline{v_{i_n}}$ with $i_1, \ldots, i_n \in \{1, \ldots, k\}$.

(2) The data values in the sub-word corresponding to $u = u_{i_1} \ldots u_{i_n}$ are subject to the following pattern.



The data values for the sub-word corresponding to $\overline{v} = \overline{v_{i_1}} \ldots \overline{v_{i_n}}$ have the same pattern.

(3) Every data value occurring in the sub-word corresponding to u does occur exactly twice in this sub-word, except for the data value of attribute $\mathtt{b}$ at the first position and the data value of attribute $\mathtt{a}$ at the last position. The same holds for the data values occurring in the sub-word corresponding to $\overline{v}$. Note that from this it follows that the $\mathtt{b}$-value at the first position and the $\mathtt{a}$-value at the last position must be unique.

(4) Every pair (d_a, d_b) of data values occurring at some position does occur exactly twice, once in the u-part and once in the $\overline{v}$-part. Moreover, the corresponding position in the u-part carries a proposition σ if and only if the corresponding position in the $\overline{v}$-part carries $\overline{\sigma}$.

Note that Conditions (2)-(4) guarantee that $u = v'$ where v' results from $\overline{v}$ by replacing every $\overline{\sigma} \in \overline{\Sigma}$ by σ.

Condition (1) can easily be expressed in B-DNL. Next, we explain how the pattern mentioned in Condition (2) can be expressed for the u-part. The $\overline{v}$-part can be handled analogously. Let m be the length of the longest word occurring in I. For every ℓ with $1 \leq \ell \leq m$, we define a formula $\overrightarrow{\chi_\ell}$ which states that the closest position to the right carrying a proposition from Σ is ℓ positions far away, i.e., $\overrightarrow{\chi_\ell}$ holds at a position i if and only if position $i + \ell$ exists, it is labelled by some proposition from Σ and all positions j with $i < j < i + \ell$, are labelled by propositions from $\overline{\Sigma}$:

$$\overrightarrow{\chi_\ell} = (\mathbf{X}^\ell \bigvee_{\sigma \in \Sigma} \sigma) \wedge \bigwedge_{i=1}^{\ell-1} \mathbf{X}^i \bigvee_{\overline{\sigma} \in \overline{\Sigma}} \overline{\sigma}.$$

Likewise, let $\overleftarrow{\chi_\ell}$ be the past counterpart of $\overrightarrow{\chi_\ell}$, i.e., $\overleftarrow{\chi_\ell}$ holds at a position i if and only if position $i-\ell$ exists, it is labelled by some proposition from Σ and all positions j with $i-\ell > j > i$ are labelled by propositions from $\overline{\Sigma}$. Using these formulas, the u-part of Condition (2) can be expressed by the following formula. Note that for every position, there are at most one ℓ_1 and one ℓ_2 such that $\overrightarrow{\chi_{\ell_1}}$ and $\overleftarrow{\chi_{\ell_2}}$ hold.

$$\mathbf{G}\Bigg[\Big(\bigvee_{\sigma\in\Sigma}\sigma \wedge (\mathbf{XF}\bigvee_{\sigma\in\Sigma}\sigma) \wedge (\mathbf{X}^{\leftarrow}\mathbf{F}^{\leftarrow}\bigvee_{\sigma\in\Sigma}\sigma)\Big) \rightarrow \\ \Big(\Big[\bigvee_{\ell=1}^{m}(\overrightarrow{\chi_\ell} \wedge \mathcal{C}^{\ell}_{\texttt{@a}} \sim\!\texttt{@a}) \wedge \bigvee_{\ell=1}^{m}(\overleftarrow{\chi_\ell} \wedge \mathcal{C}^{-\ell}_{\texttt{@b}} \sim\!\texttt{@b})\Big] \vee \Big[\bigvee_{\ell=1}^{m}(\overrightarrow{\chi_\ell} \wedge \mathcal{C}^{\ell}_{\texttt{@b}} \sim\!\texttt{@b}) \wedge \bigvee_{\ell=1}^{m}(\overleftarrow{\chi_\ell} \wedge \mathcal{C}^{-\ell}_{\texttt{@a}} \sim\!\texttt{@a})\Big]\Big)\Bigg]$$

The property referring to the u-part in Condition (3) is expressed by the following formula:

$$\mathbf{G}\Bigg[\Big(\bigvee_{\sigma\in\Sigma}\sigma \wedge \mathbf{XF}\bigvee_{\sigma\in\Sigma}\sigma\Big) \rightarrow \\ \Big(\mathcal{C}_{\texttt{@a}}\Big([\neg\mathbf{X}^{\leftarrow}_{=}\top \wedge \mathbf{X}_{=}(\sim\!\texttt{@a} \wedge \neg\mathbf{X}_{=}\top)] \vee [\neg\mathbf{X}_{=}\top \wedge \mathbf{X}^{\leftarrow}_{=}(\sim\!\texttt{@a} \wedge \neg\mathbf{X}^{\leftarrow}_{=}\top)]\Big)\Big)\Bigg] \\ \wedge \\ \mathbf{G}\Bigg[\Big(\bigvee_{\sigma\in\Sigma}\sigma \wedge \mathbf{X}^{\leftarrow}\mathbf{F}^{\leftarrow}\bigvee_{\sigma\in\Sigma}\sigma\Big) \rightarrow \\ \Big(\mathcal{C}_{\texttt{@b}}\Big([\neg\mathbf{X}^{\leftarrow}_{=}\top \wedge \mathbf{X}_{=}(\sim\!\texttt{@b} \wedge \neg\mathbf{X}_{=}\top)] \vee [\neg\mathbf{X}_{=}\top \wedge \mathbf{X}^{\leftarrow}_{=}(\sim\!\texttt{@b} \wedge \neg\mathbf{X}^{\leftarrow}_{=}\top)]\Big)\Big)\Bigg]$$

The property belonging to the $\overline{v}$-part in Condition (3) can be expressed analogously.

Finally, by means of $\mathbf{TX}_{\texttt{@a@b}}$ and $\mathbf{TX}^{\leftarrow}_{\texttt{@a@b}}$ we express Condition (4):

$$\mathbf{G}\Bigg[\bigwedge_{\sigma\in\Sigma}\Big(\sigma \rightarrow \Big([\mathbf{TX}_{\texttt{@a@b}}\overline{\sigma} \wedge \neg\mathbf{TX}^{\leftarrow}_{\texttt{@a@b}}\top \wedge \neg\mathbf{TX}_{\texttt{@a@b}}\mathbf{TX}_{\texttt{@a@b}}\top] \vee \\ [\mathbf{TX}^{\leftarrow}_{\texttt{@a@b}}\overline{\sigma} \wedge \neg\mathbf{TX}_{\texttt{@a@b}}\top \wedge \neg\mathbf{TX}^{\leftarrow}_{\texttt{@a@b}}\mathbf{TX}^{\leftarrow}_{\texttt{@a@b}}\top]\Big)\Big)\Bigg] \wedge \\ \mathbf{G}\Bigg[\bigwedge_{\overline{\sigma}\in\overline{\Sigma}}\Big(\overline{\sigma} \rightarrow \Big([\mathbf{TX}_{\texttt{@a@b}}\sigma \wedge \neg\mathbf{TX}^{\leftarrow}_{\texttt{@a@b}}\top \wedge \neg\mathbf{TX}_{\texttt{@a@b}}\mathbf{TX}_{\texttt{@a@b}}\top] \vee \\ [\mathbf{TX}^{\leftarrow}_{\texttt{@a@b}}\sigma \wedge \neg\mathbf{TX}_{\texttt{@a@b}}\top \wedge \neg\mathbf{TX}^{\leftarrow}_{\texttt{@a@b}}\mathbf{TX}^{\leftarrow}_{\texttt{@a@b}}\top]\Big)\Big)\Bigg]$$

□

6.5 Decidable Extensions of Basic Data Navigation Logic

In the previous section, we discussed the restricted ability of B-DNL to refer to positions between positions in the same class. A further weakness of B-DNL is that it does not provide operators which allow to move to some position of unbounded distance which has a data value which is different from

some value at the current position. For instance, it is not obvious how the following Property CS7, which we presented in our introductory client-server example and formulated in $\mathrm{LTL}^{\Downarrow}$ (Example 7), can be expressed in B-DNL:

Whenever a client p receives an acknowledgement, the server gets a request after some time and the next such request is from a client different from p.

One difficult aspect in this property is to identify pairs i, j of positions of unbounded distance which belong to distinct client IDs. The second challenge is to set conditions on all positions between i and j. In this section we will discuss in what extent B-DNL can be expanded so that properties like CS7 can be expressed and decidability is preserved.

We will consider a strong **U**-like operator $\underline{\mathbf{U}}$ and its past counterpart $\underline{\mathbf{U}}^{\leftarrow}$. Roughly speaking, they allow to fix some data value from a current position i, move to some position j where some constraints with respect to d hold and set further constraints on positions between i and j. The operators are used in the forms $\varphi_{\mathtt{int}}\underline{\mathbf{U}}^{\ell}_{\mathtt{@a}}\varphi_{\mathtt{tar}}$ and $\varphi_{\mathtt{int}}\underline{\mathbf{U}}^{\leftarrow\ell}_{\mathtt{@a}}\varphi_{\mathtt{tar}}$ where $\mathtt{a}$ is an attribute, $\ell \in \mathbb{Z}$ and $\varphi_{\mathtt{int}}$ and $\varphi_{\mathtt{tar}}$ are called the *intermediate* and the *target* formula, respectively. The syntax of intermediate and target formulas χ is defined as follows:

$$\chi := \varphi \mid \chi \vee \chi \mid \chi \wedge \chi \mid \sim\!\mathtt{@b} \mid \overline{\sim\!\mathtt{@b}}$$

where φ is a global formula and $\mathtt{b}$ is an attribute. The formula $\varphi_{\mathtt{int}}\underline{\mathbf{U}}^{\ell}_{\mathtt{@a}}\varphi_{\mathtt{tar}}$ and its past counterpart are treated as global formulas and evaluated with respect to a data word w and a position i. We give the semantics for the future version of the operator. The semantics of the past version is defined analogously:

- $(w, i) \models \varphi_{\mathtt{int}}\underline{\mathbf{U}}^{\ell}_{\mathtt{@a}}\varphi_{\mathtt{tar}}$ if $\mathtt{val}(w, i, \mathtt{@a}) = d$ is defined, there is some position $j \geq i + \ell$ on w such that $(w, j, d) \models \varphi_{\mathtt{tar}}$ and $(w, k, d) \models \varphi_{\mathtt{int}}$ for all k with $i + \ell \leq k < j$

Satisfaction of intermediate and target formulas is defined with respect to a data word w, a position i and a data value d:

- $(w, i, d) \models \varphi$ if $(w, i) \models \varphi$ for global formulas φ
- $(w, i, d) \models \sim\!\mathtt{@a}$ if $\mathtt{val}(w, i, \mathtt{@a}) = d$
- $(w, i, d) \models \overline{\sim\!\mathtt{@a}}$ if $\mathtt{val}(w, i, \mathtt{@a})$ *is defined* and $\mathtt{val}(w, i, \mathtt{@a}) \neq d$

As usual, if the shift-value ℓ equals 0, we skip it in the notation and if formulas are defined over a single attribute $\mathtt{a}$, we skip the attribute reference in formulas. That is, we write $\overline{\sim}$ and $\underline{\mathbf{U}}$ instead of $\overline{\sim\!\mathtt{@a}}$ and $\underline{\mathbf{U}}^{\ell}_{\mathtt{@a}}$, respectively.

We first observe that Condition (4) assuring consistency with respect to zero-tests on transition sequences of 2-MCMs in the proof of Theorem 6 can be expressed by the formula

$$\bigwedge_{1 \leq k \leq 2} \mathbf{G}\Big[\mathtt{inc}_k \rightarrow (\neg\mathtt{ifzero}_k\underline{\mathbf{U}}(\sim \wedge \mathtt{dec}_k))\Big].$$

This means that if we allow positive attribute tests $\sim\!\mathtt{@b}$ in target formulas, the logic immediately becomes undecidable. Therefore, we focus on target formulas without positive attribute tests. As $\varphi\underline{\mathbf{U}}^{\ell}_{\mathtt{@a}}(\varphi_1 \vee \varphi_2)$ is equivalent to $(\varphi\underline{\mathbf{U}}^{\ell}_{\mathtt{@a}}\varphi_1) \vee (\varphi\underline{\mathbf{U}}^{\ell}_{\mathtt{@a}}\varphi_2)$ and global formulas are closed under conjunction, we can go a step further and concentrate on target formulas of the form $\chi \wedge \overline{\sim\!\mathtt{@b}_1} \wedge \ldots \wedge \overline{\sim\!\mathtt{@b}_n}$. Our decidability proof which we are going to present works for very limited forms of intermediate and target formulas. However, if we restrict ourselves to finite data data words, we can allow quite sophisticated types of intermediate formulas. Nevertheless, we cannot determine

the precise decidability borders, because there are more general forms of intermediate and target formulas for which we do not know whether their usage leads to undecidability or not. We will discuss open questions in Section 6.7.

The logic for which we will show decidability on finite and infinite data words is called *Data Navigation Logic (DNL)* and results from B-DNL by extending the grammar of global formulas by the formation rules $\varphi_{\mathtt{int}}\underline{\mathbf{U}}^{\ell}_{\mathtt{@a}}\varphi_{\mathtt{tar}} \mid \varphi_{\mathtt{int}}\underline{\mathbf{U}}^{\leftarrow\ell}_{\mathtt{@a}}\varphi_{\mathtt{tar}}$ where $\varphi_{\mathtt{tar}}$ is restricted to formulas of the type $\overline{\sim\mathtt{@b}} \wedge \xi$ with global formula ξ and $\varphi_{\mathtt{int}} = \top$. We also introduce *Extended Data Navigation Logic (X-DNL)* for which we are able to show decidability only on finite data words. The latter logic allows formulas $\varphi_{\mathtt{int}}\underline{\mathbf{U}}^{\ell}_{\mathtt{@a}}\varphi_{\mathtt{tar}}$ and $\varphi_{\mathtt{int}}\underline{\mathbf{U}}^{\leftarrow\ell}_{\mathtt{@a}}\varphi_{\mathtt{tar}}$ where

- $\varphi_{\mathtt{tar}}$ is of the type $\overline{\sim\mathtt{@b}} \wedge \xi$ where ξ is a global formula and
- $\varphi_{\mathtt{int}}$ is of the form $\chi \vee (\sim\mathtt{@b} \wedge \chi^{=}) \vee (\overline{\sim\mathtt{@b}} \wedge \chi^{\neq})$ where $\mathtt{b}$ is the same attribute used in the target formula and χ, $\chi^{=}$ and $\chi^{\neq}$ are global formulas such that $\chi^{\neq}$ logically implies $\chi^{=}$, i.e., for every data word w and position i, we have $(w,i) \models \chi^{\neq} \Rightarrow (w,i) \models \chi^{=}$.

Example 11. Observe that Property CS7 mentioned above can be expressed in X-DNL by the formula

$$\mathbf{G}\Big[\mathtt{ack} \rightarrow \big((\neg\mathtt{req})\underline{\mathbf{U}}^{1}_{\mathtt{@receiver}}(\mathtt{req} \wedge \overline{\sim\mathtt{@sender}})\big)\Big]$$

where we abbreviated the intermediate formula. □

For convenience, we use the abbreviation $\underline{\mathbf{F}}^{\ell}_{\mathtt{@a}}(\overline{\sim\mathtt{@b}} \wedge \xi)$ (and $\underline{\mathbf{F}}^{\leftarrow\ell}_{\mathtt{@a}}(\overline{\sim\mathtt{@b}} \wedge \xi)$) expressing that there is some future (past) position where ξ holds and the $\mathtt{b}$-value differs from the value of attribute $\mathtt{a}$ at the current position. Observe that the operators $\underline{\mathbf{U}}$ and $\underline{\mathbf{U}}^{\leftarrow}$ in DNL can be expressed via these abbreviations. In the rest of this section we first show the decidability of X-DNL on finite data words and conclude with the decidability of DNL on finite as well as infinite data words.

Decidability of X-DNL on finite data words

The proof strategy for the decidability of X-DNL follows the same steps as in the corresponding proof for B-DNL (Section 6.2). We first show that X-DNL is decidable on 1-complete data words and then reduce the general case to the 1-complete case.

Before diving into the proof of the first part, we introduce the notion of *Backward Register Automata* (RA$^{\leftarrow}$) which are RA reading words from right to left. In more detail, the components of a RA$^{\leftarrow}$ $\mathcal{A} = (\Sigma, R, S, s_0, \delta, F)$ and its configurations are defined exactly in the same way as for usual RA. Given a proposition set $\mathtt{Prop}$, a run of a RA$^{\leftarrow}$ $\mathcal{A}$ with input alphabet $\Sigma = 2^{\mathtt{Prop}}$ on a 1-complete data word $w = \begin{smallmatrix} P_1 \\ d_1 \end{smallmatrix} \cdots \begin{smallmatrix} P_n \\ d_n \end{smallmatrix}$ over $\mathtt{Prop}$ is a sequence $(s_0,\lambda_0)\ldots(s_n,\lambda_n)$ of configurations such that (i) s_0 is the initial state of $\mathcal{A}$, (ii) $\lambda_0 = R_{\mapsto\perp}$ is the empty register assignment, and (iii) for every i with $0 \leq i < n$, we have $(s_i,\lambda_i) \overset{P_{n-1}}{\underset{d_{n-i}}{\longrightarrow}} (s_{i+1},\lambda_{i+1})$. The run is accepting if $s_n \in F$. In Section 4.2.2, we stated the result that every RA can be converted into an equivalent DA which can in turn be converted into an equivalent DA where the base automaton reads the (unmarked) word projection of the input word [39]. From this we can easily derive that RA$^{\leftarrow}$ can also be converted into equivalent DA. To see this, let $\mathcal{A}$ be a RA$^{\leftarrow}$ and $\mathcal{A}'$ a usual RA whose components are defined exactly in the same way as for $\mathcal{A}$. Obviously, the language of $\mathcal{A}'$ is the reverse language of $\mathcal{A}$. Thus, by the results above, we can construct a DA $\mathcal{D}$ which decides $\mathcal{L}(\mathcal{A})^R$ and whose base automaton reads word projections. Then, we can easily derive from $\mathcal{D}$ a DA $\mathcal{D}'$ deciding $\mathcal{L}(\mathcal{A})$. In order to obtain $\mathcal{D}'$ we basically "reverse" the base and class automaton of $\mathcal{D}$ by using standard techniques for finite automata. Hence, we remark:

Observation 3. Every RA$^{\leftarrow}$ can be converted into an equivalent DA.

Now, we can turn towards the proof that X-DNL is decidable on finite 1-complete data words.

Theorem 9. *Satisfiability for X-DNL on finite 1-complete data words is decidable.*

Proof. The proof is an extension of the decidability proof for B-DNL on 1-complete data words (Theorem 2). That proof basically relied on the idea that for every B-DNL-formula φ one can construct a DA $\mathcal{D}_\varphi$ which checks that a data word is valid with respect to φ, given that it is valid with respect to all strict sub-formulas of φ. Remember that validity of some data word w with respect to some global formula φ means that a position i of w is labelled by proposition p_φ if and only if φ holds at i. Accordingly, in this proof we will show that, given a formula $\varphi = \varphi_{\mathtt{int}}\underline{\mathbf{U}}^{\ell}\varphi_{\mathtt{tar}}$ or $\varphi = \varphi_{\mathtt{int}}\underline{\mathbf{U}}^{\leftarrow\ell}\varphi_{\mathtt{tar}}$, we can construct a DA $\mathcal{D}_\varphi$ which checks that an input word is valid with respect to φ, assumed that it is valid with respect to all strict sub-formulas of φ.

Before we explain the construction of $\mathcal{D}_\varphi$, we observe that, w.l.o.g., we can restrict consideration to syntactically simplified forms of φ. The first simplification is that we can assume that $\underline{\mathbf{U}}^{\ell}$ only occurs with positive shift-value ℓ and $\underline{\mathbf{U}}^{\leftarrow\ell}$ only with negative ℓ. We explain the idea underlying the elimination of $\underline{\mathbf{U}}^{\ell}$ in case of $\ell \leq 0$. The operator $\underline{\mathbf{U}}^{\leftarrow\ell}$ with $\ell \geq 0$ is handled analogously. If $\ell = 0$, we can replace ℓ by 1, because we deal with 1-complete words. Otherwise, assume that a formula $\varphi_{\mathtt{int}}\underline{\mathbf{U}}^{\ell}\varphi_{\mathtt{tar}}$ with some $\ell < 0$ holds at some position i. Then, the target position j satisfying $\varphi_{\mathtt{tar}}$ can either be smaller or grater than i. In the first case, the intermediate formula holds at positions $[i+\ell,\ldots,j)$ and the target formula holds at j. As the cardinality of $[i+\ell,\ldots,j]$ is bounded by $|\ell|$, we can make a disjunction over all positions at which $\varphi_{\mathtt{int}}$ or $\varphi_{\mathtt{tar}}$ can hold without using the operator $\underline{\mathbf{U}}^{\ell}$. In the other case, the intermediate formula holds at the positions $[i+\ell,\ldots,i)$ and the formula $\varphi_{\mathtt{int}}\underline{\mathbf{U}}^{1}\varphi_{\mathtt{tar}}$ holds at position i. Thus, $\varphi_{\mathtt{int}}\underline{\mathbf{U}}^{\ell}\varphi_{\mathtt{tar}}$ with $\ell < 0$ can be replaced by

$$\bigvee_{j=\ell}^{-1}\Big(\varphi^{j}_{\mathtt{tar}} \wedge \bigwedge_{k=\ell}^{j-1}\varphi^{k}_{\mathtt{int}}\Big) \vee \Big(\bigwedge_{k=\ell}^{-1}\varphi^{k}_{\mathtt{int}} \wedge \varphi_{\mathtt{int}}\underline{\mathbf{U}}^{1}\varphi_{\mathtt{tar}}\Big)$$

where for $k \in [\ell,\ldots,-1]$, the formulas $\varphi^{k}_{\mathtt{int}}$ and $\varphi^{k}_{\mathtt{tar}}$ result from $\varphi_{\mathtt{int}}$ and $\varphi_{\mathtt{tar}}$, respectively, by replacing

- χ by $\mathbf{X}^k\chi$,
- $\sim \wedge\chi^{=}$ by $\mathbf{X}^k\chi^{=} \wedge \mathcal{C}^k \sim$,
- $\not\sim \wedge\, \chi^{\neq}$ by $\mathbf{X}^k\chi^{\neq} \wedge \mathcal{C}^k\neg \sim$, and
- $\not\sim \wedge\, \xi$ by $\mathbf{X}^k\xi \wedge \mathcal{C}^k\neg \sim$.

The second simplification relies on the observation that the sub-formula χ in intermediate formulas can be "pushed" into $\chi^{=}$ and $\chi^{\neq}$, i.e., we can replace $\chi \vee (\sim \wedge\chi^{=}) \vee (\not\sim \wedge\, \chi^{\neq})$ equivalently by $(\sim \wedge(\chi^{=} \vee \chi)) \vee (\not\sim \wedge\, (\chi^{\neq} \vee \chi))$. Therefore, it suffices to consider intermediate formulas of the form $(\sim \wedge\chi^{=}) \vee (\not\sim \wedge\, \chi^{\neq})$.

In this proof, we will concentrate on the future operator $\underline{\mathbf{U}}^{\ell}$ and give at the end some notes how our construction can be adapted to the case for $\underline{\mathbf{U}}^{\leftarrow\ell}$. Thus, let $\varphi = ((\sim \wedge\chi^{=}) \vee (\not\sim\wedge\chi^{\neq}))\underline{\mathbf{U}}^{\ell}(\not\sim\wedge\xi)$ with $\ell > 0$. We introduce some notions and explain the main idea of the construction of $\mathcal{D}_\varphi$. We call a position *special* if $\chi^{=}$ holds at that position but not $\chi^{\neq}$. The restriction that $\chi^{\neq}$ implies $\chi^{=}$ leads to the observation that φ holds at a position i of a 1-complete data word if and only if there is a ξ-Position $j \geq i+\ell$ such that

- j has a different data value than i,

- at all positions in $[i+\ell,\dots,j)$, it holds $\chi^{=}$ or $\chi^{\neq}$, and
- all special positions in $[i+\ell,\dots,j)$ have the same data value as i.

The construction is primarily based on the insight that the data value of a potential φ-position i is determined by the special positions in the sequence from $i+\ell$ to the next ξ-position. Let us formulate it in a comprehensible way for the case $\ell = 1$: Assume we pass the word backwards and find a special position j with data value d which is not a ξ-position. Until we meet a ξ-position, a position i can be considered as a potential φ-position only if i and all special positions in $[i,\dots,j)$ have data value d.

The validity of a 1-complete data word with respect to φ can be tested by $\text{RA}^{\leftarrow}$. As these automata can be converted into DA (Observation 3), the result follows. In order to split the validity test into sub tasks which can be fulfilled by rather simple $\text{RA}^{\leftarrow}$, we consider data words enriched with additional propositions u_ξ, u_ξ^ℓ, u_s and u_s^ℓ. In the following, we will first define what it means that a data word is *valid* with respect to these propositions. Then, we will show how this kind of validity can be checked with $\text{RA}^{\leftarrow}$. Finally, we will explain how, based on these propositions, it can be ensured that a 1-complete data word is valid with respect to φ.

We say that a data word is *valid* with respect to u_ξ (or u_s, respectively) if for all positions i it holds that i is labelled by u_ξ (or u_s, respectively) if and only if there is a p_ξ- position (or special position, respectively) $j > i$ and the smallest such position carries a different data value than i. Validity with respect to u_ξ^ℓ (or u_s^ℓ) is defined similarly with the difference that the corresponding j-position has to be at least ℓ positions far away. That is, a data word is valid with respect to u_ξ^ℓ (or u_s^ℓ, respectively) if for all positions i it holds that i is labelled by u_ξ^ℓ (or u_s^ℓ, respectively) if and only if there is a p_ξ-position (or special position, respectively) $j \geq i+\ell$ and the smallest such position carries a different data value than i.

Now, we describe how validity with respect to u_ξ^ℓ can be checked by an $\text{RA}^{\leftarrow}$ assumed that validity with respect to ξ is given. The cases for the other propositions can be solved analogously. For the validity check with respect to u_ξ^ℓ we use an $\text{RA}^{\leftarrow}$ with ℓ registers. Informally, at each position i of the data word and for every k with $1 \leq k < \ell$, the automaton keeps track of the data value at position $i+k$ if this position is labelled by p_ξ. Additionally, it keeps track of the data value of the smallest p_ξ-position $j \geq i+\ell$ if such a position exists. Furthermore, it assures that the current position is labelled by u_ξ^ℓ if such a position j exists and its data value differs from the current one. After this informal description, we work out some technical details of the behaviour of the $\text{RA}^{\leftarrow}$. Assume that the registers are numbered from 1 to ℓ. The automaton preserves in its state a partial *history mapping* $m \in [\{1,\dots,\ell\} \rightharpoonup \{1,\dots,\ell\}]$ such that when reading a position i,

(1) for every k with $1 \leq k < \ell$, we have $m(k) = k'$ for some register k' if position $i+k$ carries p_ξ as well as the data value of register k', and

(2) $m(\ell) = \ell'$ for some register ℓ' if there is a p_ξ-position $j \geq i+\ell$ and the smallest such j carries the data value of register ℓ'.

The maintenance of the history mapping and the validity check are realized by the following strategy. At the beginning, the mapping is undefined on the entire domain. When after some step i with current history mapping m_i the automaton reads position $i-1$ (remember that the automaton moves backwards), it assures that position $i-1$ is labelled by u_ξ^ℓ if and only if $m_i(\ell)$ is defined and register $m_i(\ell)$ contains a different data value than that of position $i-1$. Moreover, m_{i-1} results from m_i as follows:

- For every k with $1 < k < \ell$, we have $m_{i-1}(k) = m_i(k-1)$.
- If $m_i(\ell-1)$ is defined, then $m_{i-1}(\ell) = m_i(\ell-1)$, otherwise $m_{i-1}(\ell) = m_i(\ell)$.

- If position $i-1$ is labelled by p_ξ, then the value of position $i-1$ is stored in some register and $m_{i-1}(1)$ is mapped to this register, otherwise $m_{i-1}(1)$ is undefined (observe that there is always at least one register whose input can be overwritten).

Before we turn to the question how with the help of the propositions u_ξ, u_ξ^ℓ, u_s and u_s^ℓ the validity of a 1-complete data word with respect to φ can be checked, we introduce the notion of *consistency* between two positions. We call two positions i and j with $j \geq i+\ell$ *consistent* if (i) all positions in $[i+\ell,\ldots,j)$ are labelled by $p_{\chi^=}$ or $p_{\chi^{\neq}}$, (ii) and all special positions in $[i+\ell,\ldots,j)$ have the same data value as i. Now, given that a data word is valid with respect to $\chi^=$, $\chi^{\neq}$, ξ and the propositions u_ξ, u_ξ^ℓ, u_s, u_s^ℓ, validity with respect to φ can be reformulated by the following *φ-Validity Condition*: For all positions i, it holds that it is labelled by p_φ if and only if one of the following conditions holds:

(1) Position i is labelled by u_ξ^ℓ and the smallest p_ξ-Position $j \geq i+\ell$ is consistent with i.

(2) There is a p_ξ-position $j \geq i+\ell$ consistent with i such that there is a p_ξ-position in $[i+\ell,\ldots,j)$ labelled by u_ξ.

To be convinced of the correctness of this condition, let w be a 1-complete data word valid with respect to φ and let i be some position. Position i is labelled by p_φ if and only if there is a p_ξ-position $j \geq i+\ell$ carrying a different data value than i and the smallest such position is consistent with i. This in turn is equivalent to saying that i is labelled by p_φ if and only if either (i) the smallest p_ξ-position $j \geq i+\ell$ is consistent with i and carries a different data value than i, or (ii) there is a p_ξ-position $j \geq i+\ell$ consistent with i such that all p_ξ-positions in $[i+\ell,\ldots,j)$ have the same data value as i and last p_ξ-position in $[i+\ell,\ldots,j)$ has a different data value than j. The latter is equivalent to the φ-Validity Condition.

Now, we turn to the construction of the data automaton $\mathcal{D}_\varphi$. First, we formulate the φ-Validity Condition in first order logic. Observe that on words which are valid with respect to u_s and u_s^ℓ, the property that all special positions in some $[i+\ell,\ldots,j)$ have the same data value as i is equivalent to the conjunction of the following two properties:

- If there is special position in $[i+\ell,\ldots,j)$, then i is not labelled by u_s^ℓ.
- There are no two special positions $k_1 < k_2$ in $[i+\ell,\ldots,j)$, such that k_1 is labelled by u_s.

Thus, the consistency between two positions i and j can be expressed by the following first order formula $\varphi_{cons}(x,y)$ with two free variables x and y representing positions i and j:

$$\begin{aligned}\varphi_{cons}(x,y) = & y \geq x+\ell \wedge \forall z\Big[\big(z \geq x+\ell \wedge z < y\big) \rightarrow \big(p_{\chi^=}(z) \vee p_{\chi^{\neq}}(z)\big)\Big] \wedge \\ & \Big[\exists z\big(z \geq x+\ell \wedge z < y \wedge p_{\chi^=}(z) \wedge \neg p_{\chi_{\neq}}(z)\big) \rightarrow \neg u_s^\ell(x)\Big] \wedge \\ & \neg\exists z_1 \exists z_2\Big[z_1 \geq x+\ell \wedge z_1 < y \wedge z_2 \geq x+\ell \wedge z_2 < y \wedge z_1 < z_2 \wedge u_s(z_1)\Big].\end{aligned}$$

Now, part (1) of the φ-Validity Condition can be expressed by the following formula with a free variable x representing position i:

$$\varphi_1(x) = u_\xi^\ell(x) \wedge \exists y\Big[y \geq x+\ell \wedge p_\xi(y) \wedge \varphi_{cons}(x,y) \wedge \neg\exists z\big(z \geq x+\ell \wedge z < y \wedge p_\xi(z)\big)\Big].$$

Part (2) is expressed by

$$\varphi_2(x) = \exists y\Big[y \geq x+\ell \wedge p_\xi(y) \wedge \varphi_{cons}(x,y) \wedge \exists z\big(z \geq x+\ell \wedge z < y \wedge u_\xi(z)\big)\Big].$$

Combining both formulas, we describe the φ-Validity Condition by

$$\forall x\Big[\big(\varphi_1(x) \vee \varphi_2(x)\big) \leftrightarrow p_\varphi(x)\Big].$$

Let $\mathcal{A}$ be an NFA equivalent to the above formula. The desired DA $\mathcal{D}_\varphi$ is the intersection of the DA equivalent to the $\text{RA}^{\leftarrow}$ checking validity with respect to u_ξ, u_ξ^ℓ, u_s and u_s^ℓ and the automaton $\mathcal{A}$ which can be seen as a data automaton not using its class automaton.

The construction for $\underline{\mathbf{U}}^{\leftarrow\ell}$ (with $\ell < 0$) proceeds along the same lines. In this case, for each notion introduced in this proof, we define its dual past counterpart and use usual (forward) RA instead of backward ones. For instance, we introduce the past counterpart $u_s^{\leftarrow}$ of u_s and define that a data word is valid with respect to $u_s^{\leftarrow}$ if a position i is labelled by u_s if and only if there is a special position $j < i$ and the greatest such position carries a different data value than i. Such a property can easily be tested by usual RA. ☐

It is worth noting that the technique in the last proof does not extend to ω-words as we use $\text{RA}^{\leftarrow}$ which are not defined for infinite words.

Now, we generalize the last result to data words with multiple attributes.

Theorem 10. *Satisfiability for X-DNL on finite data words is decidable.*

Proof. The proof is an adaption of the proof of Theorem 3 to X-DNL. We reduce the satisfiability problem for the general case to the 1-complete case. Then, the result follows from Theorem 9. We use the same encoding of general data words by 1-complete ones and the same translation of formulas as in the proof of Theorem 3. We just have to give the translation $t(\varphi)$ for formulas $\varphi = (\chi \vee (\sim\!\mathtt{@a}_i \wedge \chi^{=}) \vee (\overline{\sim\!\mathtt{@a}_i} \wedge \chi^{\neq}))\underline{\mathbf{U}}^{\ell}_{\mathtt{@a}_j}(\overline{\sim\!\mathtt{@a}_i} \wedge \xi)$:

$$t(\varphi) = t_j\Big[\big[(t(\chi) \vee \neg\mathtt{a}_i) \vee (\sim \wedge\mathtt{a}_i \wedge D \wedge t(\chi^{=})) \vee (\overline{\sim} \wedge \mathtt{a}_i \wedge D \wedge t(\chi^{\neq}))\big]\underline{\mathbf{U}}^{m\ell-j+i}\big[\overline{\sim} \wedge \mathtt{a}_i \wedge D \wedge t(\xi)\big]\Big].$$

Note that the formula first navigates to the position representing attribute $\mathtt{a}_j$ in the current block, *fixes* its data value and evaluates φ at the i-th position of the block encoding the ℓ-next position in the original word.

The translation for the past operator $\underline{\mathbf{U}}^{\leftarrow\ell}_{\mathtt{@a}_j}$ is defined analogously. ☐

DNL on infinte data words

Remember that DNL is the extension of B-DNL by $\underline{\mathbf{F}}$ and $\underline{\mathbf{F}}^{\leftarrow}$ which are restrictions of $\underline{\mathbf{U}}$ and $\underline{\mathbf{U}}^{\leftarrow}$, respectively. As the translation in Theorem 10 works on finite as well as infinite words, the reduction in that theorem smoothly carries over to DNL on data ω-words. Hence, it remains to show that satisfiability for DNL on 1-complete data ω-words is decidable.

Theorem 11. *Satisfiability for DNL on 1-complete data ω-words is decidable.*

Proof. Like in Theorem 4, the proof is by reduction to the non-emptiness problem for Büchi Data Automata (BDA). We outline the main ideas for the construction of BDA checking that input data words are valid with respect to formulas $\chi = \underline{\mathbf{F}}^{\ell}(\overline{\sim} \wedge \xi)$ and $\chi = \underline{\mathbf{F}}^{\leftarrow\ell}(\overline{\sim} \wedge \xi)$. According to the argumentation on the shift-values of $\underline{\mathbf{U}}^{\ell}$ and $\underline{\mathbf{U}}^{\leftarrow\ell}$ in the case for X-DNL, we can assume, w.l.o.g., that $\underline{\mathbf{F}}^{\ell}$ occurs only with positive ℓ and $\underline{\mathbf{F}}^{\leftarrow\ell}$ only with negative ℓ. In this proof, we focus on $|\ell| = 1$. The generalization is straightforward.

- $\chi = \underline{\mathbf{F}}^{\leftarrow 1}(\overline{\sim} \wedge \xi)$: This case is handled in analogy to the sketched case for $\underline{\mathbf{U}}^{\leftarrow 1}$ in the proof of Theorem 9. We construct a Büchi Register Automaton (BRA) with a single register checking that input data words are valid with respect to χ. Then, we refer to the result that for every

BRA one can construct an equivalent BDA [39, 41]. The BRA works as follows: As long as it does not read any p_ξ-position, it assures that no position is labelled by p_χ. When it reads the first p_ξ-position, it stores its data value d in its register. Then, as long as it does not meet any further p_ξ-position with a different data value than d, it checks that exactly those positions are labelled with p_χ which do not carry value d. When it passes a p_ξ-position whose value is not equal to d, it assures that all positions are labelled by p_χ, because they all have in their past at least one p_ξ-position with a different data value.

- $\chi = \underline{\mathbf{F}}^1(\not\sim \wedge \xi)$: Here, we adapt an idea from the decidability proof for $\mathrm{FO}^{\sim}_2(\mathtt{Suc}, <)$ in [41]. Remember that BDA contain two class automata, one for finite classes, another one for infinite classes. In the following, whenever we say that the BDA *marks* some position i by some symbol x, we mean that it outputs x at i. For the sake of systematization, we partition the behaviour of the BDA in sub tasks, but it should be clear that all of them can be accomplished in parallel by a single BDA.

  The base automaton first guesses whether

  (1) the word does not contain any p_ξ-position,

  (2) there is at least one but there are only finitely many p_ξ-positions,

  (3) there are only finitely many classes with p_ξ-positions and there is exactly one class c with infinitely many p_ξ-positions,

  (4) there are at least two classes with infinitely many p_ξ-positions, or

  (5) there are infinitely many classes with p_ξ-positions[1].

  Next, we explain how the BDA assures that its guess is correct. In the first two cases, the base automaton can check by itself that its guess is correct. In the other cases, it needs the help of the class automata. In Case (3), it first ensures that p_ξ occurs infinitely often. Then, it marks some p_ξ-position for which it assumes that this and all following p_ξ-positions are in c, by some special symbol x. Additionally, it marks all p_ξ-positions after x by x'. The class automaton for finite classes checks that none of x and x' occurs. The class automaton for infinite classes assures that either none or both of x and x' occur. In Case (4), the base automaton checks that there are infinitely many p_ξ-positions and outputs x and y at two different p_ξ-positions. The class automaton for infinite classes ensures that x and y do not occur in the same input word and if one of them occur, infinitely many p_ξ-positions follow. In Case (5), the base automaton outputs at infinitely many p_ξ-positions the symbol x. The class automaton for infinite classes checks that in every word there are only finitely many x.

  Now, we describe for each case, how the BDA decides that an input data word is valid with respect to χ.

  In Case (1), the base automaton just checks that there are no p_χ-positions.

  In Case (2), the base automaton first checks that no p_χ- occurs at the last p_ξ-position or later. Then, it guesses whether all p_ξ are in the same class (Case (2.a)) or at least in two different classes (Case (2.b)). In order to assure that its guess is correct, in Case (2.a) it marks the first p_ξ-position by x, the last one by y and all intermediate p_ξ-postions by y'. The class automata assure that they either do not see any of x, y' and y or all of them. For the sake of validity with respect to χ they further check that exactly those positions are labelled by p_χ which are not followed by y. For the assurance that the guess is correct in Case (2.b), the base automaton marks the last p_ξ-position by y and marks one position for which it assumes that it is the largest p_ξ-position before y which is in a different class class than y, by x. Moreover,

[1] We believe that this case is missing in [41].

it marks all positions between x and y by y'. The class automata check that x and y do not appear in the same word and that every p_ξ-position which is marked by y' is followed by y. For validity with respect to χ, the base automaton assures that all positions until x are labelled by p_χ. Additionally, the class automata check that a y'-position is labelled by p_χ if and only if it is not followed by y.

In Case (3), the base automaton first guesses whether all p_ξ are in c (Case (3.a)) or there is at least one class besides c containing a p_ξ-position (Case (3.a)). In Case (3.a), it marks exactly one p_ξ-position by some y. The class automata check that input words contain a p_ξ-position if and only if they also contain the y-position. They additionally assure: if no y occurs, then, all positions are labelled by p_χ, otherwise, no position is labelled by p_χ. We now turn to Case (3.b). Let i be the first p_ξ-position in c whereupon all p_ξ-positions are in c. The base automaton guesses this position, marks it by y, marks all following p_ξ-positions by y' and marks the last p_ξ-position before y (which has to be in a different class than i) by z. Additionally, it checks that all positions before z are labelled by p_χ. The class automata ensure that (i) z and y do not occur in the same word, (ii) a word either contains none of y and y' or both of them, and (iii) all positions of all words which do not contain any y are labelled by p_χ.

In Cases (4) and (5), the base automaton just guarantees that all positions carry p_χ.

□

Together with the reduction in the proof of Theorem 10, we conclude:

Theorem 12. *Satisfiability for DNL on data ω-words is decidable*

6.6 Expressivity of Data Navigation Logic

In this section, we compare the expressive power of DNL with the expressivity of some logics introduced in Section 4.3. First, we formulate some observations. We consider the following property Even.

Even: *The word is of even length.*

This property can be easily expressed in DNL by the formula $\langle \top \cdot (\top \cdot \top)^* \rangle \neg \langle \top \rangle \top$. As it does not refer to data values, the corresponding language must contain data words where all attribute values are undefined. Obviously, on such words, $\mathrm{FO}^\sim$ is expressively equivalent to FO and $\mathrm{LTL}^\Downarrow$ is expressively equivalent to PLTL. As neither FO nor PLTL is able to express that a word is of even length [194], it follows that Even cannot be expressed in $\mathrm{FO}^\sim$ or $\mathrm{LTL}^\Downarrow$. By taking into account that $\mathrm{LTL}^\Downarrow$ captures PLRV [81], we observe:

Observation 4. (1) Even is expressible in DNL.

(2) Even is not expressible in $\mathrm{LTL}^\Downarrow$, $\mathrm{FO}^\sim$ or PLRV.

Let us consider now the following property parametrized by three propositions p, q and r on data words with a single attribute:

Triple(p,q,r): *There are positions $i \le j \le k$ such that i is labelled by p, j is labelled by q, k is labelled by r and positions i and k carry the same data value.*

This property can be expressed with the operator $\mathbf{E}$ introduced in Section 6.4: $\mathbf{E}(p, q, r)$. For the sake of contradiction, assume that there is a DNL-formula $\varphi_{\mathbf{E}}$ equivalent to this formula. It follows from Theorem 6 in Section 6.4 that the satisfiability problem for B-DNL extended by $\varphi_{\mathbf{E}}$ is not decidable. As this contradicts the decidability of DNL (Theorem 12), property $\textsc{Triple}(p, q, r)$ cannot be expressible in DNL. On the other hand, by Example 2 in Section 4.2.1 and the result that REM are expressively equivalent to Register Automata (RA) [147], $\textsc{Triple}(p, q, r)$ must be expressible in REM. Moreover, it can also be formulated by the following two logics:

- $\text{LTL}^{\Downarrow}_1$: $\mathbf{F}\big[p \wedge \Downarrow.\mathbf{F}(q \wedge \mathbf{F}(r \wedge \Uparrow))\big]$
- PathLog: $\langle \overleftarrow{\varepsilon} \rangle \sim \overrightarrow{\langle q \wedge \langle \overleftarrow{[p]} \rangle \sim \langle \overrightarrow{[r]} \rangle \rangle} \vee \langle \overleftarrow{\varepsilon} \rangle \nsim \overrightarrow{\langle q \wedge \langle \overleftarrow{[p]} \rangle \sim \langle \overrightarrow{[r]} \rangle \rangle}$.

We conclude:

Observation 5. (1) $\textsc{Triple}$ is expressible in $\text{LTL}^{\Downarrow}_1$, PathLog and REM.

(2) $\textsc{Triple}$ is not expressible in DNL.

From Observations 4 and 5, it directly follows:

Proposition 4. In terms of expressivity, $\text{LTL}^{\Downarrow}$ and DNL are not comparable.

As mentioned in [96], PathLog cannot express that the underlying word contains at least two positions, because it contains only reflexive and transitive modalities. This property is expressed by the DNL-formula $\langle\top\rangle\top$. Together with Observation 5 we obtain:

Proposition 5. In terms of expressivity, PathLog and DNL are not comparable.

The DNL-formula $\mathbf{G}(\mathcal{C}(\neg\mathbf{X}\top \wedge \neg\mathbf{X}^{\leftarrow}\top))$ expresses that all data values of the underlying 1-complete data word are pairwise distinct. This property is not expressible in RA [124] nor in REM as RA and REM are equivalent [147]. Together with Observation 5 we conclude:

Proposition 6. In terms of expressivity, REM is not comparable with DNL.

As stated in Section 4.3.1, $\text{FO}^{\sim}_2(\texttt{Suc}, <)$ is decidable on 1-complete data words and looses its decidability when a further position variable is included or two data values at each position are allowed. We can prove that on 1-complete structures, DNL strictly subsumes this decidable fragment of $\text{FO}^{\sim}$:

Proposition 7. On 1-complete data words, DNL is strictly more expressive than $\text{FO}^{\sim}_2(\texttt{Suc}, <)$.

Proof. Due to Observation 4, it suffices to show that every formula in $\text{FO}^{\sim}_2(\texttt{Suc}, <)$ using at most one attribute can be expressed in DNL. In [82] it is shown that this fragment of $\text{FO}^{\sim}_2(\texttt{Suc}, <)$ is equivalent to the *simple fragment* of $\text{LTL}^{\Downarrow}_1$. As temporal operators, this fragment only allows $\mathbf{X}$, $\mathbf{X}^{\leftarrow}$ and combinations of the forms $\mathbf{XXF}$ and $\mathbf{X}^{\leftarrow}\mathbf{X}^{\leftarrow}\mathbf{F}^{\leftarrow}$. Each of the latter combinations is considered as a single temporal operator. Furthermore, each occurrence of a temporal operator must be immediately preceded by $\Downarrow$ (and $\Downarrow$ must not occur anywhere else). We will describe a translation t which converts formulas of simple $\text{LTL}^{\Downarrow}_1$ into equivalent formulas in DNL.

We call a formula *elementary* if (i) it is a proposition, (ii) it is of one of the forms $\Uparrow$ or $\Downarrow.\chi'$, or (iii) it is the negation of one of these formulas. We omit the straightforward cases in the definition of t:

- $t(\Uparrow) = \sim$
- $t(\Downarrow.\mathbf{X}\psi) = \mathcal{C}^1 t(\psi)$

- $t(\Downarrow.\mathbf{XXF}\psi)$ is obtained as follows:

  1. Using classical rules in propositional logic, $\Downarrow.\mathbf{XXF}\psi$ is converted into an equivalent formula $\varphi = \Downarrow.\mathbf{XXF}\psi'$ where ψ' is a disjunction of conjunctions of elementary formulas.
  2. Using the equivalence $\Downarrow.\mathbf{XXF}(\psi_1 \vee \psi_2) \equiv \Downarrow.\mathbf{XXF}\psi_1 \vee \Downarrow.\mathbf{XXF}\psi_2$, we get from φ a disjunction φ' of formulas $\Downarrow.\mathbf{XXF}\psi''$ where ψ'' is a conjunction of elementary formulas.
  3. Every disjunct $\Downarrow.\mathbf{XXF}\psi''$ in φ' is replaced as follows:
     - If ψ'' contains $\Uparrow$ as well as $\neg\Uparrow$ as conjuncts, $\Downarrow.\mathbf{XXF}\psi''$ is replaced by $\bot$.
     - If ψ'' contains at least one $\Uparrow$ and no $\neg\Uparrow$ as conjuncts, $\Downarrow.\mathbf{XXF}\psi''$ is replaced by $\mathcal{C}^2\mathbf{F}_=\psi'''$ where ψ''' results from ψ'' by replacing every conjunct $\Uparrow$ by $\sim$ and all other conjuncts χ by $t(\chi)$.
     - If ψ'' contains at least one $\neg\Uparrow$ and no $\Uparrow$ as conjuncts, $\Downarrow.\mathbf{XXF}\psi''$ is replaced by $\underline{\mathbf{F}}^2\psi'''$ where ψ''' results from ψ'' by replacing every conjunct $\neg\Uparrow$ by $\overline{\sim}$ and all other conjuncts χ by $t(\chi)$.
     - If ψ'' contains neither $\Uparrow$ nor $\neg\Uparrow$ as conjuncts, $\Downarrow.\mathbf{XXF}\psi''$ is replaced by $\mathbf{XXF}\psi'''$ where ψ''' results from ψ'' by replacing every conjunct χ by $t(\chi)$.

□

Among the logics introduced in Part A, the logic CLTL^{XF} and its extensions PLRV were the only ones for which decidability on data words with multiple data values at each position was shown. It is not hard to prove that PLRV is entirely captured by DNL.

Proposition 8. DNL is strictly more expressive than PLRV.

Proof. Again, we give a translation t which converts every PLRV-formula φ into an equivalent formula $t(\varphi)$ in DNL. The strictness follows from Observation 4. We omit the straightforward cases:

- $t(\mathtt{@a} \sim \mathbf{X}^\ell \mathtt{@b}) = \mathcal{C}^\ell_{\mathtt{@a}} \sim\!\mathtt{@b}$
- $t(\mathtt{@a} \sim \langle\varphi\rangle\mathtt{@b}) = \mathcal{C}^1_{\mathtt{@a}}\mathbf{F}_=(\sim\!\mathtt{@b} \wedge t(\varphi))$
- $t(\mathtt{@a} \not\sim \langle\varphi\rangle\mathtt{@b}) = \underline{\mathbf{F}}^1_{\mathtt{@a}}(\overline{\sim\!\mathtt{@b}} \wedge t(\varphi))$
- $t(\mathtt{@a} \sim \langle\varphi\rangle^{\leftarrow}\mathtt{@b}) = \mathcal{C}^{-1}_{\mathtt{@a}}\overleftarrow{\mathbf{F}}_=(\sim\!\mathtt{@b} \wedge t(\varphi))$
- $t(\mathtt{@a} \not\sim \langle\varphi\rangle^{\leftarrow}\mathtt{@b}) = \underline{\overleftarrow{\mathbf{F}}}^{-1}_{\mathtt{@a}}(\overline{\sim\!\mathtt{@b}} \wedge t(\varphi))$

□

6.7 Discussion

We introduced and analyzed Data Navigation Logic (DNL), a logic for which we argued in Chapter 5 that it can be suitable for the usage in the framework of model checking of concurrent systems with unboundedly many processes. We first proved that the fragment B-DNL of DNL is decidable on finite and infinite data words and showed that this decidability carries over to full DNL. We moreover showed that the latter logic is strictly more expressive than $\text{FO}^{\sim}_2(\mathtt{Suc},<)$ on 1-complete and than PLRV on general data words. Even though some extensions of B-DNL lead to undecidability, we were able to show that X-DNL, the extension of DNL containing the powerful $\underline{\mathbf{U}}$-operator (and its past version), is decidable on finite data words. As explained in the corresponding section, our proof technique used for the decidability of X-DNL on finite words does not extend to infinite words. Thus, one open question is whether X-DNL remains decidable on the latter kind of structures. Furthermore, recall that it turned out that the permission of positive attribute tests in target

formulas of the operator $\underline{\mathbf{U}}$ leads to undecidability. However, whether X-DNL remains decidable on finite words if inequality tests on more than one attribute are permitted, is a further open question. Likewise, the intermediate formulas of the $\underline{\mathbf{U}}$-operator in X-DNL are still quite restricted. It is an interesting challenge to pinpoint how far one can allow more general boolean combinations of global formulas and attribute tests in intermediate formulas while preserving decidability.

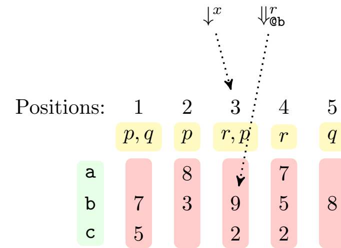
We conclude by mentioning some works built upon our results presented in this chapter. First, recall that we defined Basic Data LTL (B-DLTL) as the restriction of B-DNL which uses temporal operators instead of path expressions. In [75], the authors consider two fragments of B-DLTL, namely *Class Future Basic Data LTL (B-DLTL$^+$)* and *Class Past Basic Data LTL (B-DLTL$^-$)*. In B-DLTL$^+$ (or, respectively, B-DLTL$^-$), the class past operators $\mathbf{X}^{\leftarrow}_{=}$ and $\mathbf{U}^{\leftarrow}_{=}$ (or, respectively, the class future operators $\mathbf{X}_{=}$and $\mathbf{U}_{=}$) are not allowed. The authors show that each formula of B-DLTL$^+$ can be converted to *Locally Prefix-Closed Data Automata (PDA)* such that the formula is satisfiable if and only if the automaton is non-empty. There is also an analogous translation from B-DLTL$^-$-formulas to *Locally Suffix-Closed Data Automata (SDA)*. A PDA (or, respectively, SDA) is a restriction of a usual DA where every state of the class automaton is accepting (or, respectively, initial). Non-emptiness for these automata is in ExpSpace. With the help of these automata, the authors show that satisfiability for B-DLTL$^+$ as well as for B-DLTL$^-$ is 2ExpSpace-complete on finite and infinite data words. The authors also consider *Nested Data LTL (N-DLTL)*, an extension of B-DLTL with a restricted form of tuple navigation. Recall from Theorem 8 that extending B-DLTL by the ability to choose two arbitrary attributes a and b at some position i and to navigate to the next or previous position where these attributes carry the same data values as at i, leads to undecidability. The logic N-DLTL allows tuple navigation only with respect to some tree order defined on the set of attributes. The fragments *N-DLTL$^+$* and *N-DLTL$^-$* result from N-DLTL by imposing the same restrictions used to obtain B-DLTL$^+$ and B-DLTL$^-$ from B-DLTL. While full N-DLTL on finite and infinite and N-DLTL$^-$ on infinite data words are undecidable, it is shown that N-DLTL$^-$ on finite and N-DLTL$^+$ on finite and infinite data words are decidable and Ackermann-hard. To simplify the decidability proofs, the authors introduce *Nested Data Automata (NDA)* which contain multiple linearly ordered class automata, one for each attribute. While formulas of N-DLTL$^+$ are converted into *Locally Prefix-Closed Nested Data Automata (PNDA)*, those of N-DLTL$^-$ are translated into *Locally Suffix-Closed Nested Data Automata (SNDA)*.

Recall from Section 4.2.3 that Class Memory Automata, simulating runs of the base and class automaton of a Data Automaton within a single run, are expressively equivalent to Data Automata. Natural restrictions and extension of Class Automata corresponding to PDA and PNDA are considered in [71]. In [70], PNDA are used to decide observational equivalence of call-by-value functional languages.

The results presented in this chapter are extensions of results published in [131] which was a joint work with Thomas Schwentick and Thomas Zeume. In [131], the logic Basic Data LTL and some extensions were considered. While the navigational abilities in Basic Data LTL are based on LTL-operators, the logic presented here is an extension whose navigation is based on regular expressions. Hence, the decidability proofs in [131] had to be adapted to this extension. The proofs for the decidability of B-DNL on finite and infinite data words presented in Sections 6.2 and 6.3 are adaptions of the decidability proofs in [131] for Basic Data LTL on such structures. Moreover, I recognized that the decidability proof in [131] for Extended Data LTL including the $\underline{\mathbf{U}}$-operator does not work. In Section 6.5, I presented for X-DNL which also contains this operator a shorter proof which works at least on finite data words. On infinite data words I gave a proof for DNL which contains only a restricted version of this operator. Furthermore, I compared the expressivity of DNL with more logics than in [131]. Finally, I added a further undecidability result (Theorem 6) which helped to find new expressivity results in Section 6.6.

Chapter 7

The Power of Storing Positions



As we mentioned in Chapter 5 of motivating questions, in this chapter we will introduce *Hybrid Temporal Logic (HTL$^\sim$)* on data words and compare its expressivity to LTL$^\Downarrow$ (for the definition of LTL$^\Downarrow$, see Section 4.3.2). The logic HTL$^\sim$ is an extension of LTL where formulas allow to assign some variable x to the "current" position ($\downarrow^x$), to compare some data value of the current position to a data value at the x-position ($\texttt{@a}\sim x.\texttt{@b}$), to shift evaluation to the x-position ($\texttt{on}(x).\psi$) and to ask whether the current position is the x-position (x).

After having defined the syntax and semantics of the logic in Section 7.1, we will describe in Section 7.2.1.1 how it can be derived from existing results in the literature that HTL$^\sim$ is in general strictly more expressive than LTL$^\Downarrow$. Then, we will strengthen this result by proving that even HTL$^\sim$ with only two variables can express properties which are not expressible in full LTL$^\Downarrow$. Afterwards, we will try to figure out by which operators this additional expressive power of HTL$^\sim$ is caused. It will turn out that the ability of HTL$^\sim$-formulas to shift evaluation to positions bound to variables is an important factor. Such shifts can be realized by formulas of the form $\texttt{on}(x).\psi$ or the permission of atomic formulas x in the presence of past operators. Indeed, we will show that every HTL$^\sim$-formula where these (combinations of) operators are prohibited can be converted into an equivalent LTL$^\Downarrow$-formula. A further case where the expressive power of HTL$^\sim$ is tamed is the case where the number of variables is restricted to one. This fragment of HTL$^\sim$ will be considered in Section 7.2.1.2. We will prove that HTL$^\sim$-formulas which use at most one variable can also be converted into equivalent LTL$^\Downarrow$-formulas. In addition, it will be shown that, in the case where the number of attributes is restricted to one, HTL$^\sim$ with one variable is expressively equivalent to LTL$^\Downarrow$ with one freeze register. The question whether this equality carries over to the case with multiple attributes remains open and will be discussed at appropriate points.

Compared to LTL$^\Downarrow$, the logic HTL$^\sim$ is not only more expressive, but it also provides the opportunity to express properties with shorter formulas. In Section 7.2.2 we will show that HTL$^\sim$-formulas can be non-elementarily more succinct than LTL$^\Downarrow$-formulas. Even in the case of a single variable

the succinctness is at least exponentially.

Finally, we will show in Section 7.3 that the variable and register hierarchies for $\mathrm{HTL}^{\sim}$ and $\mathrm{LTL}^{\Downarrow}$, respectively, are infinite, that is, for every k there is $k' > k$ such that $\mathrm{HTL}^{\sim}$ with k' variables is strictly more expressive than with k variables and $\mathrm{LTL}^{\Downarrow}$ with k' registers is strictly more expressive than with k registers. We will derive these results from the strictness of the variable hierarchy of first-order logic on finite undirected ordered graphs [178].

In the proofs of this chapter we will deal with finite data words, but our main results carry over easily to data ω-words. We will provide some notes on this issue in the discussion section.

7.1 Hybrid Temporal Logic on Data Words

We give the formal syntax of $\mathrm{HTL}^{\sim}$ and describe informally its semantics. As usual, the full formal semantics can be found in the Appendix (Section A.8). Let $\mathtt{Prop}$ be a finite set of propositions, $\mathtt{Att}$ a finite set of attributes and $\mathtt{PV}$ an infinite supply of position variables. Formulas of $\mathrm{HTL}^{\sim}$ over $\mathtt{Prop}$, $\mathtt{Att}$ and $\mathtt{PV}$ are constructed according to the following grammar:

$$\varphi ::= p \mid x \mid \varphi \wedge \varphi \mid \neg\varphi \mid {\downarrow^x}.\varphi \mid \mathtt{@a}{\sim}x.\mathtt{@b} \mid \mathtt{on}(x).\varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi \mid \mathbf{X}^{\leftarrow}\varphi \mid \varphi\mathbf{U}^{\leftarrow}\varphi$$

where $p \in \mathtt{Prop}$, $\mathtt{a}, \mathtt{b} \in \mathtt{Att}$ and $x \in \mathtt{PV}$.

An $\mathrm{HTL}^{\sim}$-formula is evaluated with respect to a data word w, a position i on w and a *variable assignment* μ, i.e., a partial mapping assigning variables in $\mathtt{PV}$ to positions of w. Loosely speaking, the formula ${\downarrow^x}.\varphi$ places variable x on the current position and evaluates φ. The operator $\mathtt{on}(x)$ is used to "jump" to the x-position. Hence, the formula $\mathtt{on}(x).\varphi$ evaluates φ at the position where x refers to. The only position at which the atomic formula x evaluates to true is the position where the variable is currently placed. Finally, $\mathtt{@a}{\sim}x.\mathtt{@b}$ holds at some position if the value of attribute $\mathtt{a}$ at the current position and the value of attribute $\mathtt{b}$ at the x-position are both defined and equal. The temporal operators $\mathbf{X}$, $\mathbf{U}$, $\mathbf{X}^{\leftarrow}$ and $\mathbf{U}^{\leftarrow}$ are defined as in LTL. If a formula φ is satisfied by a data word w, a position i and an assignment μ, we write $(w, i, \mu) \models \varphi$. If φ uses only a single variable x, we also write $(w, i, \mu(x)) \models \varphi$. We say that a data word w *satisfies* a formula φ (written as $w \models \varphi$) if $(w, 1, \mathtt{PV}_{\mapsto\perp}) \models \varphi$, i.e., φ evaluates to true at the initial position with empty variable assignment.

The abbreviations $\mathbf{F}$, $\mathbf{G}$, $\mathbf{F}^{\leftarrow}$ and $\mathbf{G}^{\leftarrow}$ are defined as usual. If a formula is set up over a single attribute $\mathtt{a}$, we skip the reference $\mathtt{@a}$ in formulas. The notions of *bounded* and *free* variables are defined as in FO. A formula is called *closed* if no free variable occurs. The fragment of $\mathrm{HTL}^{\sim}$ where at most $k \geq 1$ variables are allowed is denoted as $\mathrm{HTL}^{\sim}_k$. For a set $\mathcal{O}$ of temporal operators and a logic $\mathcal{L} \in \{\mathrm{HTL}^{\sim}\} \cup \{\mathrm{HTL}^{\sim}_k \mid k \geq 1\}$, we denote by $\mathcal{L}(\mathcal{O})$ the fragment of $\mathcal{L}$ where at most temporal operators from $\mathcal{O}$ are used.

7.2 Hybrid Temporal Logic vs. Freeze LTL

7.2.1 Expressivity

7.2.1.1 Multiple Variables

In this section, we will compare $\mathrm{HTL}^{\sim}$ and $\mathrm{LTL}^{\Downarrow}$ with respect to expressivity in the case that there is no bound on the number of $\mathrm{HTL}^{\sim}$-variables. First, we will show that $\mathrm{HTL}^{\sim}$ is strictly more expressive than $\mathrm{LTL}^{\Downarrow}$ and actually, it only needs two variables to express a property that is not expressible in $\mathrm{LTL}^{\Downarrow}$. Then we will identify fragments of $\mathrm{HTL}^{\sim}$ in which the expressive power of the logic does not go beyond that of $\mathrm{LTL}^{\Downarrow}$.

Let us try two filter out the two main differences between $\mathrm{HTL}^{\sim}$ and $\mathrm{LTL}^{\Downarrow}$ at an informal level:

(1) *Moving to fixed positions*: The logic $\mathrm{HTL}^{\sim}$ provides operators to "fix" positions and "shift" evaluation to them. By applying $\downarrow^x$, a $\mathrm{HTL}^{\sim}$-formula "memorizes" a current position i: as long as x is not shifted it can always "move back" to position i and call a sub-formula ψ at that position (by formulas of the forms $\mathtt{on}(x).\psi$ and $\mathbf{FF}^{\leftarrow}(x \wedge \psi)$). Compared to this, with the application of $\Downarrow^r_{\mathtt{@a}}$ at i, an $\mathrm{LTL}^{\Downarrow}$- formula, roughly speaking, records only a data value of position i, but not the position itself. After having left position i, the abilities of the formula to "find" position i is restricted, at least, the logic does not provide explicit operators which allow to "move back" to position i and evaluate some sub-formula at that position.

(2) *Accessing all attributes of fixed positions*: The logics $\mathrm{HTL}^{\sim}$ and $\mathrm{LTL}^{\Downarrow}$ differ in their way they access attributes. By applying $\Downarrow^r_{\mathtt{@a}}$ at a position i, an $\mathrm{LTL}^{\Downarrow}$-formula "decides" which attribute (in this case $\mathtt{a}$) is going to be compared to attributes of other positions. More precisely, in the scope of $\Downarrow^r_{\mathtt{@a}}$, as long as the $\Downarrow$-operator is not reapplied, only attribute $\mathtt{a}$ of position i can be used for comparisons with attributes of positions different from i. On the other side, the $\mathrm{HTL}^{\sim}$-operation $\downarrow^x$ does not restrict to any attribute of the current position. Thus, in the scope of $\downarrow^x$ *all* attributes of the x-position can be used for comparisons against other attributes.

We will show that, in spite of the difference in the access of attributes, every $\mathrm{LTL}^{\Downarrow}$-formula can be translated into an equivalent $\mathrm{HTL}^{\sim}$-formula. However, translations into the other direction are in general not possible. This insight will be formulated in Corollary 1. Then, we will try to understand from which features $\mathrm{HTL}^{\sim}$ gains its additional expressive power. It will turn out that the ability of $\mathrm{HTL}^{\sim}$-formulas to move to positions fixed by variables, mentioned in (1), is a critical factor. Indeed, in cases where we allow the usage of the $\mathtt{on}$-operator or the combined usage of atomic formulas x and past operators, $\mathrm{HTL}^{\sim}$ needs only two variables to express a property which is not expressible in full $\mathrm{LTL}^{\Downarrow}$ (Corollary 3). Otherwise, there is always a translation from $\mathrm{HTL}^{\sim}$-formulas into equivalent $\mathrm{LTL}^{\Downarrow}$-formulas (Propositions 11 and 12).

We start with the result that $\mathrm{HTL}^{\sim}$ is at least as expressive as $\mathrm{LTL}^{\Downarrow}$.

Lemma 1. *For every $k \geq 1$, every closed $\mathrm{LTL}^{\Downarrow}_k$-formula can be translated into an equivalent $\mathrm{HTL}^{\sim}_k$-formula.*

Proof. The main idea of the translation is that every sub-formula of the form $\Downarrow^r_{\mathtt{@a}}.\psi$ can be replaced by $\downarrow^x.\psi'$ for some variable x where ψ' results from ψ by taking into account that x has to be used in relation to attribute $\mathtt{a}$, i.e., comparisons of the form $\Uparrow^r_{\mathtt{@b}}$ have to be translated into $\mathtt{@b} \sim x.\mathtt{@a}$. We additionally have to keep the subtle peculiarity in mind that operations $\Downarrow^r_{\mathtt{@a}}$ implicitly demand that the value of $\mathtt{a}$ is defined at the current position.

Now, we give the details of the translation. Let φ be an $\mathrm{LTL}^{\Downarrow}_k$-formula for some $k \geq 1$. Without loss of generality, we assume that φ uses registers from $R = \{r_1, \ldots, r_k\}$ and attributes from $A = \{\mathtt{a}_1, \ldots, \mathtt{a}_m\}$ for some $m \geq 1$. Furthermore, let $V = \{x_1, \ldots, x_k\}$ be a set of $\mathrm{HTL}^{\sim}$-variables. Our translation simulates every register r_j with $1 \leq j \leq k$ by a variable x_j and uses x_j in relation with the attributes whose data values are stored in r_j. We establish this correspondence between variables and registers by a mapping κ from variables to attributes. Figure 7.1 illustrates the relationship between a register assignment λ, a corresponding variable assignment μ and the mapping κ linking the two assignments. Now, we formalize the correspondence between register and variable assignments. Given a data word w over an attribute set $A' \supseteq A$, we say that a register assignment $\lambda \in [R \rightharpoonup \mathcal{D}]$ is *compatible* with a variable assignment $\mu \in [V \rightharpoonup \mathtt{pos}(w)]$ on w if there is some partial mapping $\kappa \in [V \rightharpoonup A]$ such that for every j with $1 \leq j \leq k$, we have $\lambda(r_j) = d$ for some data value d if and only if $\mu(x_j)$ and $\kappa(x_j)$ are defined and $\mathtt{val}(w, \mu(x_j), \mathtt{@}\kappa(x_j)) = d$.

We define for every mapping $\kappa \in [V \rightharpoonup A]$, a translation t_κ which converts each sub-formula ψ of φ into an $\mathrm{HTL}^{\sim}_k$-formula $t_\kappa(\psi)$ such that for every data word w, every position i and every pair of a register assignment λ and a variable assignment μ which are compatible on w due to κ, it holds

$$(w, i, \lambda) \models \psi \Leftrightarrow (w, i, \mu) \models t_\kappa(\psi).$$

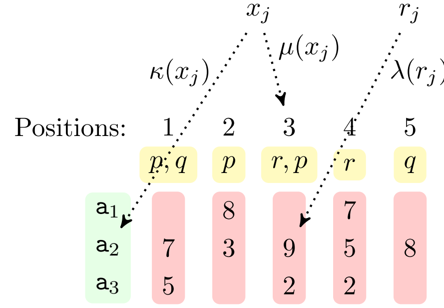


Figure 7.1: Correspondence of μ and λ established by κ

Then, it follows for every data word w:

$$w \models \varphi \Leftrightarrow w \models t_{\kappa_{\mapsto\perp}}(\varphi).$$

We omit the straightforward cases in the definition of t_κ:

- $t_\kappa(p) = p$
- $t_\kappa(\Downarrow^{r_j}_{@a_\ell}.\psi) = \downarrow^{x_j}.(@a_\ell \sim x_j.@a_\ell \wedge t_{\kappa[x \mapsto a_\ell]}(\psi))$
- $t_\kappa(\Uparrow^{r_j}_{@a_\ell}) = @a_\ell \sim x_j.@\kappa(x_j)$
- $t_\kappa(\mathbf{X}\psi) = \mathbf{X}t_\kappa(\psi)$
- $t_\kappa(\psi_1 \mathbf{U} \psi_2) = t_\kappa(\psi_1)\mathbf{U}t_\kappa(\psi_2)$
- $t_\kappa(\mathbf{X}^\leftarrow\psi) = \mathbf{X}^\leftarrow t_\kappa(\psi)$
- $t_\kappa(\psi_1 \mathbf{U}^\leftarrow \psi_2) = t_\kappa(\psi_1)\mathbf{U}^\leftarrow t_\kappa(\psi_2)$

for propositions p, $j \in \{1, \ldots, k\}$ and $\ell \in \{1, \ldots, m\}$. Note that $\downarrow^{x_j}.@a_\ell \sim x_j.@a_\ell$ just ensures that the value of attribute $\mathtt{a}$ at the current position is defined. □

In order to prove that HTL$^\sim$ is strictly more expressive than LTL$^\Downarrow$, we first show that HTL$^\sim$ and FO$^\sim$ have the same expressive power.

Proposition 9. The logics HTL$^\sim$ and FO$^\sim$ are expressively equivalent.

Proof. By a standard translation (similar to that in [100]), for every $k \geq 1$, each closed HTL$^\sim_k$-formula can be translated into an equivalent FO$^\sim$-formula with at most $k + 3$ variables. The translation of closed first-order formulas into HTL$^\sim$-formulas is also along standard lines. One important issue in the translation is the simulation of FO$^\sim$-quantifications of the form $\exists x$ which choose an arbitrary position and assign it to variable x. As we permit past operators in HTL$^\sim$, such quantifications can always be simulated by $\mathbf{F}\mathbf{F}^\leftarrow \downarrow^x$. In order to emulate FO$^\sim$-comparisons of the form $x.@a \sim y.@b$, the corresponding HTL$^\sim$-formula first navigates to one of the positions assigned to x or y and then performs the comparison.

Now, we describe the details of the translation. Given a $k \geq 1$, we will define a translation t from FO$^\sim_k$-formulas φ into HTL$^\sim_k$-formulas $t(\varphi)$ such that for every data word w, every position i in w and every variable assignment μ, we have

$$(w, \mu) \models \varphi \Leftrightarrow (w, i, \mu) \models t(\varphi).$$

Obviously, this results in $w \models \varphi \Leftrightarrow w \models t(\varphi)$. We omit the trivial cases:

- $t(p(x)) = \texttt{on}(x).p$
- $t(\exists x\psi) = \mathbf{FF}^{\leftarrow} \downarrow^{x}.t(\psi)$
- $t(x = y) = \texttt{on}(x).y$
- $t(\texttt{Suc}(x,y)) = \texttt{on}(x).\mathbf{X}y$
- $t(x < y) = \texttt{on}(x).\mathbf{XF}y$
- $t(x.\texttt{@a}{\sim}y.\texttt{@b}) = \texttt{on}(x).\texttt{@a}{\sim}y.\texttt{@b}$

for propositions p, variables x, y and attributes $\texttt{a}, \texttt{b}$. □

In the sequel, we will explain how it can be inferred from the last proposition and results in [169] and [82] that there is a property expressible in HTL$^{\sim}$ which cannot be expressed in LTL$^{\Downarrow}$. Then, by Lemma 1, it will follow that HTL$^{\sim}$ is *strictly* more expressive than LTL$^{\Downarrow}$.

First, we introduce the notion of *hypersets*. A 1-hyperset over the set $\mathcal{D}$ of data values is a finite subset of $\mathcal{D}$. For every natural number $m > 1$, an m-hyperset is a finite set of $(m-1)$-hypersets. In [169], the authors develop an encoding for hypersets by sequences of data values. For $m \in \mathbb{N}$, we call a sequence $u\#v$ of values from $\mathcal{D} \cup \{\#\}$, with some fresh data value $\#$ not contained in $\mathcal{D}$, *proper with respect to* m, if u and v encode the same m-hyperset. The authors in [169] consider for every $m \in \mathbb{N}$, the language $\mathcal{L}_m$ consisting of sequences of data values proper with respect to m. Furthermore, they introduce two-way alternating register automata which are similar to ARA$^{\leftrightarrow}$ as defined in Section 4.2.1 and read sequences of data values as inputs. It is proven that

(1) for every $m \geq 1$, the language $\mathcal{L}_m$ can be expressed in FO$^{\sim}$, and

(2) for $m \geq 4$, there is no two-way alternating register automaton deciding $\mathcal{L}_m$.

The proof of (2) is based on a communication complexity argument (see, e.g., [118]). The main idea is that on a sequence $w = u\#v \in \mathcal{L}_m$ for some $m \geq 4$, two-way alternating register automata are not able to transfer enough information from the u-part to the v-part of w and vice-versa, in order to check that both parts encode the same m-hyperset. Even though the definition of two-way alternating register automata given in [169] does not coincide completely with ARA$^{\leftrightarrow}$, the argument in [169] carries over to ARA$^{\leftrightarrow}$ easily. Since we know from [82] that every property expressible in LTL$^{\Downarrow}$ can be decided by ARA$^{\leftrightarrow}$, it immediately follows that for $m \geq 4$, there cannot be any LTL$^{\Downarrow}$-formula describing the language $\mathcal{L}_m$. Along with (1), Proposition 9 and Lemma 1, we get:

Corollary 1. *The logic HTL$^{\sim}$ is strictly more expressive than LTL$^{\Downarrow}$.*

Next, we will investigate how much hybrid machinery is indeed needed in HTL$^{\sim}$ in order to express a property which is not expressible in LTL$^{\Downarrow}$. For a fragment $\mathcal{L}$ of HTL$^{\sim}$ and $\mathcal{O} \subseteq \{\texttt{on}, x\}$, we write $\mathcal{L}^{-\mathcal{O}}$ for the restriction of $\mathcal{L}$ for which it holds:

- If $\texttt{on} \in \mathcal{O}$, then sub-formulas of the form $\texttt{on}(x).\psi$ for any variable x are not contained in $\mathcal{L}^{-\mathcal{O}}$.
- If $x \in \mathcal{O}$, then atomic formulas of the form x for any variable x are not contained in $\mathcal{L}^{-\mathcal{O}}$.

We will prove that even in HTL$^{\sim}_{2}(\mathbf{X},\mathbf{U})^{-\{x\}}$ it is possible to express a property for which there is no equivalent formula in entire LTL$^{\Downarrow}$. Before doing that, we need some preparations. We proceed by defining an encoding for hypersets which is very similar to the one given in [169]. We encode m-hypersets as simple data words over the proposition set $\{z, b_1, e_1, \ldots, b_m, e_m\}$. A 1-hyperset

$H = \{d_1, \ldots, d_j\} \subseteq \mathcal{D}$ is represented by the data word $w = \substack{b_1 \\ n}\ \substack{z \\ d_1} \cdots \substack{z \\ d_j}\ \substack{e_1 \\ n'}$ where n and n' are arbitrary data values. If for some $m \geq 2$ and $\ell \geq 1$, the data words $w_1, \ldots, w_\ell$ represent $(m-1)$-hypersets $H_{m-1}(w_1), \ldots, H_{m-1}(w_\ell)$, then, $w = \substack{b_m \\ n_m}\ w_1 \cdots w_\ell\ \substack{e_m \\ n'_m}$ with arbitrary data values n_m and n'_m represents the m-hyperset $H_m(w) = \{H_{m-1}(w_1), \ldots, H_{m-1}(w_\ell)\}$. For instance, the data word w in Figure 7.2 represents the 2-hyperset

$$H_2(w) = \{\{1, 2\}, \{7, 8, 9\}, \{2, 5\}\}.$$

Note that the order of data values within a sequence encoding a 1-hyperset and the order of subse-

$$w = \substack{b_2 \\ 4}\ \substack{b_1 \\ 5}\ \substack{z \\ 1}\ \substack{z \\ 2}\ \substack{e_1 \\ 2}\ \substack{b_1 \\ 3}\ \substack{z \\ 7}\ \substack{z \\ 9}\ \substack{z \\ 8}\ \substack{e_1 \\ 2}\ \substack{b_1 \\ 1}\ \substack{z \\ 2}\ \substack{z \\ 5}\ \substack{e_1 \\ 9}\ \substack{e_2 \\ 3}$$

Figure 7.2: Data word w representing $H_2(w)$

quences encoding $(m-1)$-hypersets within a data word representing an m-hyperset are irrelevant. Thus, the data word $w' = \substack{b_2 \\ 7}\ \substack{b_1 \\ 6}\ \substack{z \\ 5}\ \substack{z \\ 2}\ \substack{e_1 \\ 4}\ \substack{b_1 \\ 8}\ \substack{z \\ 2}\ \substack{z \\ 1}\ \substack{e_1 \\ 3}\ \substack{b_1 \\ 3}\ \substack{z \\ 7}\ \substack{z \\ 8}\ \substack{z \\ 9}\ \substack{e_1 \\ 5}\ \substack{e_2 \\ 9}$ also encodes the 2-hyperset represented by w. If a data word w does not represent any m-hyperset, we set $H_m(w) = \bot$. For every $m \geq 1$, we define the language $\mathcal{L}_m^{\sim}$ of data words over $\{s, z, b_1, e_1, \ldots, b_m, e_m\}$ as

$$\mathcal{L}_m^{\sim} = \{w_1 \substack{s \\ d} w_2 \mid H_m(w_1) = H_m(w_2) \neq \bot, d \in \mathcal{D}\}.$$

The argumentation in [169] leading to the result that the languages $\mathcal{L}_m$ with $m \geq 4$ cannot be decided by two-way alternating register automata carries over to the sets $\mathcal{L}_m^{\sim}$ and $\mathrm{ARA}^{\leftrightarrow}$ easily. Thus, the proof of the following proposition is an easy adaption of the proof of the corresponding result in [169].

Proposition 10. ([169]) For $m \geq 4$, there is no $\mathrm{ARA}^{\leftrightarrow}$ deciding $\mathcal{L}_m^{\sim}$.

However, these languages can be expressed in $\mathrm{HTL}_2^{\sim}(\mathbf{X}, \mathbf{U})$, even without using atomic formulas of the form x.

Theorem 13. *For every $m \geq 1$, there is a formula in $\mathrm{HTL}_2^{\sim}(\mathbf{X}, \mathbf{U})^{-\{x\}}$ expressing $\mathcal{L}_m^{\sim}$.*

Proof. For every $m \geq 1$, we define a formula φ_m in $\mathrm{HTL}_2^{\sim}(\mathbf{X}, \mathbf{U}) - \{x\}$ over the proposition set $\{s, z, b_1, e_1, \ldots, b_m, e_m\}$ and some single attribute such that for every data word w, it holds $w \in \mathcal{L}_m^{\sim}$ if and only if $w \models \varphi_m$. The formula φ_m is a conjunction of several sub-formulas which we describe separately. The following three sub-formulas χ_{one}, χ_{main} and χ_{hyp} express that w is a 1-complete data word of the form $w_1 \substack{s \\ d} w_2$ with $H_m(w_1) \neq \bot$ and $H_m(w_2) \neq \bot$.

- The formula χ_{one} is a straightforward formula expressing that every position carries a data value and exactly one proposition from $\{z, s, b_1, \ldots, b_m, e_1, \ldots, e_m\}$.

- The formula χ_{main} expresses that w is of the form $w_1 \substack{s \\ d} w_2$, w_1 and w_2 start with a b_m-position and end with an e_m-position and there are no other positions carrying b_m, e_m or s.

$$\chi_{main} = b_m \wedge \mathbf{X}\Big[\neg\Big(b_m \vee s \vee e_m\Big)\mathbf{U}\Big(e_m \wedge \mathbf{X}\Big(s \wedge \mathbf{X}\big(b_m \wedge (\neg(b_m \vee s \vee e_m)\mathbf{U}(e_m \wedge \neg\mathbf{X}\top))\big)\Big)\Big)\Big]$$

- The formula χ_{hyp} expresses that both sides of w are encodings of hypersets. Note that we have to take into account that hypersets may be empty. In more detail, the formula expresses that
  - every b_1-position is immediately followed by a z- or an e_1-position,
  - for i with $2 \leq i \leq m$, every b_i-position is immediately followed by a b_{i-1}- or an e_i-position,
  - every z-position is immediately followed by a z- or an e_1-position, and
  - for $i < m$, every e_i-position is immediately followed by a b_i- or an e_{i+1}-position.

$$\chi_{hyp} = \mathbf{G}\Big[\big(b_1 \to \mathbf{X}(z \vee e_1)\big) \wedge \bigwedge_{i=2}^{m} \big(b_i \to \mathbf{X}(b_{i-1} \vee e_i)\big) \wedge z \to \mathbf{X}(z \vee e_1) \wedge \bigwedge_{i=1}^{m-1} \big(e_i \to \mathbf{X}(b_i \vee e_{i+1})\big)\Big]$$

Next, we construct a formula ψ_m that expresses $H_m(w_1) = H_m(w_2)$, i.e., w_1 and w_2 encode the same hyperset. The formula is defined inductively.

- Given that the variables x and y are bound to b_1-positions, the formula ψ_1 checks that the two 1-hypersets whose encodings start at x and y, respectively, are equal. To describe it like a procedure, the formula first "jumps" to the x-position, navigates towards the next e_1-position and checks during this navigation that every data value found between the x- and the corresponding e_1-position is also available in the 1-hyperset encoding preceded by y. The same procedure with reversed roles for x and y is performed for the sequence between the y- and its next e_1-position.

$$\begin{aligned}\psi_1 = &\,\mathtt{on}(x).\mathbf{X}\Big[\big(\neg e_1 \wedge \downarrow^x\!.\mathtt{on}(y).\mathbf{X}(\neg e_1 \mathbf{U}(\neg e_1 \wedge \sim x))\big)\mathbf{U} e_1\Big] \wedge \\ &\mathtt{on}(y).\mathbf{X}\Big[\big(\neg e_1 \wedge \downarrow^y\!.\mathtt{on}(x).\mathbf{X}(\neg e_1 \mathbf{U}(\neg e_1 \wedge \sim y))\big)\mathbf{U} e_1\Big]\end{aligned}$$

- Likewise, for $2 \leq i \leq m$ the formula ψ_i expresses that, if x and y are bound to b_i-positions, the i-hypersets starting at x and y, respectively, are equal. To this end, on every b_{i-1}-position located between the x- and its corresponding e_i-position, the formula "places" the variable x (thus, x is reused), guesses a corresponding b_{i-1}-position in the sequence starting at y, places the variable y at that position and "calls" ψ_{i-1} which by induction checks that the sequences starting at the (new) x- and y-positions encode the same b_{i-1}-hyperset. Observe that x and y are reused in the scope of $\downarrow^x$ and $\downarrow^y$. An analogous procedure is conducted for all b_{i-1}-positions in the sequence starting at the y-position.

$$\begin{aligned}\psi_i = &\,\mathtt{on}(x).\Big(\big(b_{i-1} \to \downarrow^x\!.\mathtt{on}(y).(\neg e_i \mathbf{U}(b_{i-1} \wedge \downarrow^y\!.\psi_{i-1}))\big)\mathbf{U} e_i\Big) \wedge \\ &\mathtt{on}(y).\Big(\big(b_{i-1} \to \downarrow^y\!.\mathtt{on}(x).(\neg e_i \mathbf{U}(b_{i-1} \wedge \downarrow^x\!.\psi_{i-1}))\big)\mathbf{U} e_i\Big)\end{aligned}$$

Finally, the desired formula is $\varphi_m = \chi_{one} \wedge \chi_{main} \wedge \chi_{hyp} \wedge \downarrow^x\!.\mathbf{F}(s \wedge \mathbf{X} \downarrow^y\!.\psi_m)$.

Every word $w = w_1 \,{}^{s}_{d}\, w_2 \in \mathcal{L}_m^{\sim}$ satisfies χ_{one}, χ_{main} and χ_{hyp} by construction. As w_1 and w_2 represent the same hypersets, both parts of ψ_m are satisfied, too.

If a data word w satisfies φ_m, the formulas χ_{one}, χ_{main} and χ_{hyp} ensure that w is of the form $w_1 \,{}^{s}_{d}\, w_2$ and that w_1 and w_2 encode m-hypersets. The two parts of ψ_m make sure that every $(m-1)$-hyperset encoded in w_1 also occurs in w_2 and vice-versa. Thus, the completeness and correctness of φ_m follow. □

From Proposition 10 and Theorem 13, we get:

Corollary 2. *The logic $LTL^{\Downarrow}$ cannot express all properties expressible in $HTL_2^{\sim}(\mathbf{X},\mathbf{U})^{-\{x\}}$.*

As already observed in [184], in hybrid logics on linear structures, sub-formulas of the form $\mathtt{on}(x).\psi$ can be replaced by $\mathbf{F}\mathbf{F}^{\leftarrow}(x \wedge \psi)$. From this and Theorem 13 it follows:

Corollary 3. *The logic $LTL^{\Downarrow}$ cannot express all properties expressible in $HTL_2^{\sim -\{on\}}$.*

To forge a link to our initial discussion in this section on the differences between $HTL^{\sim}$ and $LTL^{\Downarrow}$ ((1) moving to fixed positions and (2) accessing all attributes of fixed positions), we observe that formulas in $HTL_2^{\sim}(\mathbf{X},\mathbf{U})^{-\{x\}}$ or $HTL_2^{\sim -\{\mathtt{on}\}}$, roughly speaking, have the ability to "move back" to positions "fixed" by variables. In $HTL_2^{\sim}(\mathbf{X},\mathbf{U})^{-\{x\}}$ this is realized by the on-operator and in $HTL_2^{\sim -\{\mathtt{on}\}}$ by $\mathbf{F}\mathbf{F}^{\leftarrow}(x \wedge \psi)$. Next, we will examine two fragments, namely $HTL^{\sim -\{\mathtt{on},x\}}$ and $HTL^{\sim}(\mathbf{X},\mathbf{U})^{-\{\mathtt{on}\}}$, in which the mentioned ability is not supported explicitly by operators. It will turn out that all formulas in both fragments can be translated into equivalent $LTL^{\Downarrow}$-formulas. From this we can conclude that the ability of $HTL^{\sim}$ to "move" to positions referenced by variables, mentioned in item (1), is an important factor for the additional expressive power of $HTL^{\sim}$ compared to $LTL^{\Downarrow}$.

Before we continue, we outline a technique which will be used in the translations in the following two proofs. The technique helps to deal with the difference stated in item (2). Remember that in the scope of $\downarrow^x$, an $HTL^{\sim}$-formula is able to access all attributes of the x-position. The $LTL^{\Downarrow}$-operator $\Downarrow^r_{@\mathtt{a}}$, however, makes only a single attribute accessible for sub-formulas in its scope. Therefore, when translating from $HTL^{\sim}$ to $LTL^{\Downarrow}$, we simulate the operation $\downarrow^x$ at some position i by a sequence of $\Downarrow$-operations which store all available data values at position i. To do this, we first have to check which attributes are defined at position i. Whether an attribute $\mathtt{a}$ is defined can be tested by $\Downarrow^r_{@\mathtt{a}}.\Uparrow^r_{@\mathtt{a}}$. If an attribute $\mathtt{a}$ is not defined, equality tests $@\mathtt{a}{\sim}x.@\mathtt{b}$ in the scope of $\downarrow^x$ can be replaced by $\bot$. For instance, one can translate $\downarrow^x.\mathbf{F}(@\mathtt{a}{\sim}x.@\mathtt{a} \wedge \neg @\mathtt{b}{\sim}x.@\mathtt{b})$ to the following $LTL^{\Downarrow}$-formula with two registers, one for attribute $\mathtt{a}$ and the other for $\mathtt{b}$.

$$\Big[(\Downarrow^{r_a}_{@\mathtt{a}}.\Uparrow^{r_a}_{@\mathtt{a}} \wedge \Downarrow^{r_b}_{@\mathtt{b}}\Uparrow^{r_b}_{@\mathtt{b}}) \rightarrow \Downarrow^{r_a}_{@\mathtt{a}}.\Downarrow^{r_b}_{@\mathtt{b}}.\mathbf{F}(\Uparrow^{r_a}_{@\mathtt{a}} \wedge \neg \Uparrow^{r_b}_{@\mathtt{b}})\Big] \wedge \Big[(\Downarrow^{r_a}_{@\mathtt{a}}.\Uparrow^{r_a}_{@\mathtt{a}} \wedge \neg \Downarrow^{r_b}_{@\mathtt{b}}.\Uparrow^{r_b}_{@\mathtt{b}}) \rightarrow \Downarrow^{r_a}_{@\mathtt{a}}.\mathbf{F}\Uparrow^{r_a}_{@\mathtt{a}}\Big] \wedge$$

$$\Big[(\neg \Downarrow^{r_a}_{@\mathtt{a}}.\Uparrow^{r_a}_{@\mathtt{a}} \wedge \Downarrow^{r_b}_{@\mathtt{b}}.\Uparrow^{r_b}_{@\mathtt{b}}) \rightarrow \bot\Big] \wedge \Big[(\neg \Downarrow^{r_a}_{@\mathtt{a}}.\Uparrow^{r_a}_{@\mathtt{a}} \wedge \neg \Downarrow^{r_b}_{@\mathtt{b}}.\Uparrow^{r_b}_{@\mathtt{b}}) \rightarrow \bot\Big]$$

As it can be seen, by an inductive application of this technique during a translation, the length of a formula can grow exponentially.

In the following proofs, we use for $j \in \mathbb{N}$ and $L = \{\ell_1, \ldots, \ell_h\} \subseteq \mathbb{N}$, the expression $\Downarrow^j_L$ as an abbreviation for $\Downarrow^{r_{(j,\ell_1)}}_{@\mathtt{a}_{\ell_1}} \ldots \Downarrow^{r_{(j,\ell_h)}}_{@\mathtt{a}_{\ell_h}}$.

Proposition 11. For every $k \geq 1$ and $m \geq 0$, each closed formula in $HTL_k^{\sim -\{\mathtt{on},x\}}$ using m attributes can be translated into an equivalent $LTL^{\Downarrow}_{mk}$-formula.

Proof. In case $m = 0$, we can simply delete expressions of the form $\downarrow^x$ and get an equivalent PLTL-formula. We consider the case for $m \geq 1$. For some $k \geq 1$, let φ be a closed formula of $HTL_k^{\sim -\{\mathtt{on},x\}}$ using m attributes. Without loss of generality, we assume that φ uses variables from $V = \{x_1, \ldots, x_k\}$ and attributes from $A = \{\mathtt{a}_1, \ldots, \mathtt{a}_m\}$. We will translate φ into a formula using a register $r_{(j,\ell)}$ for every pair of a variable x_j and an attribute $\mathtt{a}_\ell$. Thus, we use the register set $R = \{r_{(j,\ell)} \mid 1 \leq j \leq k \text{ and } 1 \leq \ell \leq m\}$. We will define a translation which relates every variable x_j with the registers $r_{(j,1)}, \ldots, r_{(j,m)}$ and takes for every operation $\downarrow^{x_j}$ into account which attributes at the current position are defined.

Given a data word w, we call a variable assignment $\mu \in [V \rightharpoonup \texttt{pos}(w)]$ and a register assignment $\lambda \in [R \rightharpoonup \mathcal{D}]$ *consistent* on w if $\lambda = \{r_{(j,\ell)} \mapsto \texttt{val}(w, i, \texttt{@a}_\ell) \mid 1 \leq j \leq k, 1 \leq \ell \leq m$ and the mappings $\mu(x_j) = i$ and $\texttt{val}(w, i, \texttt{@a}_\ell)$ are defined$\}$. Figure 7.3 illustrates the relationship between consistent variable and register assignments. We define for every mapping $\kappa \in [\{1, \ldots, k\} \rightharpoonup 2^{\{1,\ldots,m\}}]$,

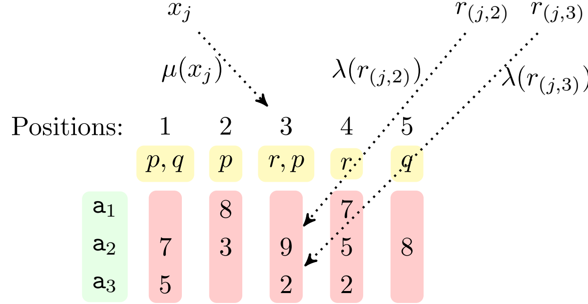


Figure 7.3: Consistent assignments μ and λ

a translation t_κ such that for every sub-formula ψ of φ, every data word w, every position i and every pair consisting of a consistent variable assignment μ and a register assignment λ with $\kappa = \{j \mapsto L \mid 1 \leq j \leq k, L \subseteq \{1, \ldots, \ell\}, \mu(x_j) = i$ is defined and L is the set of all ℓ such that $\texttt{val}(w, i, \texttt{@a}_\ell)$ is defined$\}$, it holds

$$(w, i, \mu) \models \psi \Leftrightarrow (w, i, \lambda) \models t_\kappa(\psi).$$

Then, by the definition of the semantics of HTL$^\sim$ and LTL$^\Downarrow$, it follows for every data word w that

$$w \models \varphi \Leftrightarrow w \models t_{\kappa_\perp}(\varphi).$$

We omit the straightforward cases in the definition of t_κ:

- $t_\kappa(p) = p$
- $t_\kappa(\downarrow^{x_j}.\psi) =$
  $\bigvee_{L \subseteq \{1,\ldots,m\}} \Big[\big(\bigwedge_{\ell \in L} \Downarrow^{r_{(j,1)}}_{\texttt{@a}_\ell} \cdot \Uparrow^{r_{(j,1)}}_{\texttt{@a}_\ell} \wedge \bigwedge_{\ell \in \{1,\ldots,m\} \setminus L} \neg \Downarrow^{r_{(j,1)}}_{\texttt{@a}_\ell} \cdot \Uparrow^{r_{(j,1)}}_{\texttt{@a}_\ell} \big) \rightarrow \Downarrow^{j}_{L} .t_{\kappa[j \mapsto L]}(\psi) \Big]$
- $t_\kappa(\texttt{@a}_\ell \sim x_j.\texttt{@a}_s) = \begin{cases} \Uparrow^{r_{(j,s)}}_{\texttt{@a}_\ell}, & \text{if } \kappa(j) \text{ is defined and } s \in \kappa(j) \\ \perp, & \text{otherwise} \end{cases}$
- $t_\kappa(\mathbf{X}\psi) = \mathbf{X}t_\kappa(\psi)$
- $t_\kappa(\psi_1 \mathbf{U} \psi_2) = t_\kappa(\psi_1)\mathbf{U}t_\kappa(\psi_2)$

where p is a proposition, $j \in \{1, \ldots, k\}$ and $\ell, s \in \{1, \ldots, m\}$. □

The proof of the following proposition is an extension of the last one.

Proposition 12. For every $k \geq 1$ and $m \geq 0$, every closed formula in HTL$^\sim_k(\mathbf{X}, \mathbf{U})^{-\{\texttt{on}\}}$ using m attributes can be translated into an equivalent LTL$^\Downarrow_{mk}(\mathbf{X}, \mathbf{U})$-formula.

Proof. For some $k \geq 1$ and $m \geq 0$, let φ be a closed formula in $\text{HTL}_k^{\sim}(\mathbf{X}, \mathbf{U})^{-\{\texttt{on}\}}$ using m attributes. We will first give the translation for the case where the $\text{HTL}_k^{\sim}(\mathbf{X}, \mathbf{U})^{-\{\texttt{on}\}}$-formula does not use any attributes, i.e., $m = 0$. Note that in this case, no atomic formula of the form $\texttt{@a}{\sim}x.\texttt{@b}$ can occur in φ. Then, we will explain, how, using the ideas in the proof of Proposition 11, the translation can be extended to the case $m \geq 1$.

The idea is that in a formula $\downarrow^x.\psi$ atomic formulas x evaluate to true as long as some temporal operator does not "move" the current position. Thus, it suffices to keep track of whether a sub-formula of φ is evaluated on a position bound to a variable or not. Depending on this, atomic formulas x can be replaced by $\top$ or $\bot$. For instance, the formula $\downarrow^x.\mathbf{X}x$ is equivalent to the formula $\mathbf{X}\bot$.

Now, we explain the details of the translation. Without loss of generality, we assume that at most variables from $V = \{x_1, \ldots, x_k\}$ occur in φ. For every subset $S \subseteq \{1, \ldots, k\}$, we define a translation t_S on sub-formulas of φ such that for every sub-formula ψ, every data word w, every position i, every register assignment λ and every variable assignment μ with $S = \{j \mid \mu(x_j) = i\}$, it holds

$$w, i, \mu \models \psi \Leftrightarrow w, i, \lambda \models t_S(\psi).$$

Then, it obviously follows for every data word w,

$$w \models \varphi \Leftrightarrow w \models t_\emptyset(\varphi).$$

In the definition of t_S we content ourselves with the interesting cases:

- $t_S(p) = p$
- $t_S(x_j) = \begin{cases} \top, & \text{if } j \in S \\ \bot, & \text{otherwise} \end{cases}$
- $t_S(\downarrow^{x_j}.\psi) = t_{S \cup \{j\}}(\psi)$
- $t_S(\mathbf{X}\psi) = \mathbf{X} t_\emptyset(\psi)$
- $t_S(\psi_1 \mathbf{U} \psi_2) = t_S(\psi_2) \vee \Big(t_S(\psi_1) \wedge \mathbf{X}\big(t_\emptyset(\psi_1) \mathbf{U} t_\emptyset(\psi_2)\big)\Big)$

for every proposition p and $j \in \{1, \ldots, k\}$.

If attributes $\texttt{a}_1, \ldots, \texttt{a}_m$ with $m \geq 1$ occur in φ, we translate the formula into an $\text{LTL}^{\Downarrow}_{mk}(\mathbf{X}, \mathbf{U})$-formula which uses registers from $\{r_{(j,\ell)} \mid 1 \leq j \leq k \text{ and } 1 \leq \ell \leq m\}$. The idea how attributes are handled is the same as in the proof of Proposition 11. We extend the translation t_S to a translation $t_{(S,\kappa)}$ where for every variable x_j pointing to some position i, (i) the registers $r_{(j,1)}, \ldots, r_{(j,m)}$ store all data values at position i and (ii) $\kappa(j)$ keeps track of all ℓ such that the value of $\texttt{a}_\ell$ is defined at position i. We only consider sub-formulas whose translations differ from the case without attributes:

- $t_{(S,\kappa)}(\downarrow^{x_j}.\psi) =$
  $\bigvee_{L \subseteq \{1,\ldots,m\}} \Big[\big(\bigwedge_{\ell \in L} \Downarrow^{r_{(j,1)}}_{\texttt{@a}_\ell} . \Uparrow^{r_{(j,1)}}_{\texttt{@a}_\ell} \wedge \bigwedge_{\ell \in \{1,\ldots,m\} \setminus L} \neg \Downarrow^{r_{(j,1)}}_{\texttt{@a}_\ell} . \Uparrow^{r_{(j,1)}}_{\texttt{@a}_\ell}\big) \rightarrow \Downarrow^j_L . t_{(S \cup \{j\}, \kappa[j \mapsto L])}(\psi)\Big]$
- $t_{(S,\kappa)}(\texttt{@a}_\ell {\sim} x_j.\texttt{@a}_s) = \begin{cases} \Uparrow^{r_{(j,s)}}_{\texttt{@a}_\ell}, & \text{if } \kappa(j) \text{ is defined and } s \in \kappa(j) \\ \bot, & \text{otherwise} \end{cases}$

where $j \in \{1, \ldots, k\}$ and $\ell, s \in \{1, \ldots, m\}$.

Thus, the formula φ is equivalent to the $\text{LTL}^{\Downarrow}_{mk}(\mathbf{X}, \mathbf{U})$-formula $t_{(\emptyset,\kappa_\bot)}(\varphi)$.

$\square$

We note that for $\text{HTL}^{\sim}$ similar observations can be made as for hybrid temporal logic on linear frames in [100]. In particular, for $k \geq 1$, every $\text{HTL}^{\sim}_{k}$ formula can be converted into an equivalent $\text{HTL}^{\sim}_{k+2}(\mathbf{X}, \mathbf{U})$-formula. The idea is to fix an additional variable at the first position of the word and to bind the second additional variable to the current position whenever a past formula has to be evaluated. For instance, given that the additional variables are x and y, the sub-formula $p\mathbf{U}^{\leftarrow}q$ can be expressed by $\downarrow^{y}.\mathtt{on}(x).\mathbf{F}[\mathbf{F}y \wedge q \wedge (y \vee \mathbf{XG}(\mathbf{F}y \rightarrow p))]$. A similar technique was used in [200] in the context of branching time logics.

7.2.1.2 One Variable

In this section, we will focus on $\text{HTL}^{\sim}_{1}$, i.e., $\text{HTL}^{\sim}$ with a single variable. In the last section, we have seen that $\text{HTL}^{\sim}$ is strictly more expressive than $\text{LTL}^{\Downarrow}$ and that there is even an $\text{HTL}^{\sim}_{2}$-formula that cannot be translated into any equivalent $\text{LTL}^{\Downarrow}$-formula. In contrast to this result, we will show in this section that every $\text{HTL}^{\sim}_{1}$-formula can be translated into an equivalent $\text{LTL}^{\Downarrow}$-formula. However, the resulting formula uses as many registers as there are attributes in the original formula. Together with Lemma 1 it follows that, in the setting with a single attribute, $\text{HTL}^{\sim}_{1}$ is expressively equivalent to $\text{LTL}^{\Downarrow}_{1}$. A generalization of this result to multiple attributes is presumably not possible, but remains as an open question.

We will first give the translation from $\text{HTL}^{\sim}_{1}$-formulas with at most one attribute into $\text{LTL}^{\Downarrow}_{1}$-formulas. Then, we will explain how, using the technique in the proofs of Propositions 11 and 12, this result can be extended to multiple attributes. Finally, we will say a few words about the question whether $\text{LTL}^{\Downarrow}_{1}$ and $\text{HTL}^{\sim}_{1}$ are expressively equivalent in general.

We start with the translation of $\text{HTL}^{\sim}_{1}$-formulas with at most one attribute to $\text{LTL}^{\Downarrow}_{1}$-formulas. The translation relies on a kind of separation property. We will show that in formulas of the form $\downarrow^{x}.\chi$ the *top level* of χ can be rewritten into a Boolean combination of future and past formulas. This makes it easy to eliminate sub-formulas of the forms x and $\mathtt{on}(x).\psi$.

We introduce some new notation needed for the proof. For a set Φ of $\text{HTL}^{\sim}_{1}$-formulas, let $\mathtt{Prop}^{\Phi}$ denote the set $\{p_{\psi} \mid \psi \in \Phi\}$ of fresh propositions disjunct from the propositions occurring in the formulas in Φ. For a data word w, a position j of w and a set Φ of $\text{HTL}^{\sim}_{1}$-formulas, we denote by $(w, j)^{\Phi}$ the word that is obtained from w by removing the attributes (along with the data values) and adding to each position i all propositions p_{ψ} from $\mathtt{Prop}^{\Phi}$ for which $(w, i, j) \models \psi$. A sub-formula ψ of an $\text{HTL}^{\sim}_{1}$-formula φ is called a *top-level sub-formula* of φ if ψ is not in the scope of any $\downarrow$-operator. For every $\text{HTL}^{\sim}_{1}$-formula φ using at most one variable x and one attribute, let $\overrightarrow{T}(\varphi)$ be a set such that for every top-level sub-formula ψ of φ, we have: (i) if ψ is in one of the forms $\downarrow^{x}.\chi$, $\sim x$ or $\mathtt{on}(x).\chi$, then, ψ is contained in $\overrightarrow{T}(\varphi)$, and (ii) if ψ is in one of the forms $\mathbf{X}^{\leftarrow}\chi$ or $\chi\mathbf{U}^{\leftarrow}\xi$, then, $\mathtt{on}(x).\psi$ is contained in $\overrightarrow{T}(\varphi)$. We define $\overrightarrow{T}_{x}(\varphi)$ analogously, but with the additional atomic formula x.

We say that an $\text{HTL}^{\sim}_{1}$-formula is in *normal form* if in every sub-formula of the type $\downarrow^{x}.\psi$, the formula ψ is of one of the forms $\sim x$, $\mathbf{X}\chi$, $\chi_1\mathbf{U}\chi_2$, $\mathbf{X}^{\leftarrow}\chi$ or $\chi_1\mathbf{U}^{\leftarrow}\chi_2$. Due to the equivalences

- $\downarrow^{x}.p \equiv p$,
- $\downarrow^{x}.x \equiv \top$,
- $\downarrow^{x}.\mathtt{on}(x).\chi \equiv \downarrow^{x}.\chi$,
- $\downarrow^{x}.\neg\chi \equiv \neg \downarrow^{x}.\chi$ and
- $\downarrow^{x}.(\chi_1 \wedge \chi_2) \equiv \downarrow^{x}.\chi_1 \wedge \downarrow^{x}.\chi_2$

every $\text{HTL}^{\sim}_{1}$-formula can be translated into an equivalent formula in normal form.

The following two lemmas will help to rewrite a sub-formula ψ in formulas $\downarrow^x.\psi$ as a Boolean combination of past and future formulas.

Lemma 2. *For every HTL$^{\sim}_1$-formula φ using at most the variable x and a single attribute, there is an LTL-formula $\overrightarrow{\varphi}$ such that for every data word w and all positions i and j on w, it holds*

$$(w, i, j) \models \downarrow^x.\varphi \;\Leftrightarrow\; ((w, i)^{\overrightarrow{T}(\varphi)}, i) \models \overrightarrow{\varphi}.$$

Proof. Let φ be an HTL$^{\sim}_1$-formula φ using at most the variable x and one attribute. We inductively define for every top-level sub-formula ψ of φ, a PLTL-formula $t(\psi)$ such that for all positions i, j with $j \leq i$, it holds

$$(w, i, j) \models \psi \;\Leftrightarrow\; ((w, j)^{\overrightarrow{T_x}(\varphi)}, i) \models t(\psi). \tag{7.1}$$

We make a case distinction on the structure of ψ:

- $t(p) = p$ for propositions p
- $t(\neg\chi) = \neg t(\chi)$
- $t(\chi_1 \wedge \chi_2) = t(\chi_1) \wedge t(\chi_2)$
- $t(\chi) = p_\chi$ if χ is of one of the forms x, $\sim x$, $\downarrow^x.\xi$ and $\mathtt{on}(x).\xi$
- $t(\mathbf{X}\chi) = \mathbf{X}t(\chi)$
- $t(\chi_1 \mathbf{U} \chi_2) = t(\chi_1)\mathbf{U}t(\chi_2)$
- $t(\mathbf{X}^{\leftarrow}\chi) = (\neg p_x \wedge \mathbf{X}^{\leftarrow}t(\chi)) \vee (p_x \wedge p_{\mathtt{on}(x).\mathbf{X}^{\leftarrow}\chi})$
- $t(\chi_1 \mathbf{U}^{\leftarrow} \chi_2) = (\neg p_x \wedge t(\chi_1))\mathbf{U}^{\leftarrow}[(\neg p_x \wedge t(\chi_2)) \vee (p_x \wedge p_{\mathtt{on}(x).\chi_1 \mathbf{U}^{\leftarrow}\chi_2})]$

That is, the usual evaluation of past operators is restricted to positions greater or equal to j. If this is insufficient, then the new propositions are used. It is straightforward to show by induction that the equivalence 7.1 indeed holds.

Let now φ_1 be the formula that results from $t(\varphi)$ by replacing every occurrence of p_x with $\neg\mathbf{X}^{\leftarrow}\top$. Clearly, for all data words w and positions $j \leq i$, it holds $((w, j)^{\overrightarrow{T_x}(\varphi)}, i) \models t(\varphi)$ if and only if $((w, j)^{\overrightarrow{T}(\varphi)}[j, \ldots), (i - j + 1)) \models \varphi_1$ where $(w, j)^{\overrightarrow{T}(\varphi)}[j, \ldots)$ is the suffix of $(w, j)^{\overrightarrow{T}(\varphi)}$ starting at j.

By [101, Theorem 2.4] the PLTL-formula φ_1 can be effectively translated into an LTL-formula $\overrightarrow{\varphi}$ that is initially equivalent to φ_1. As the positions smaller than j are irrelevant for the validity of $\overrightarrow{\varphi}$ at position j, altogether the lemma follows. ☐

Similarly, we let for every HTL$^{\sim}_1$-formula φ with variable x and a single attribute, $\overleftarrow{T}(\varphi)$ be the set of all top-level sub-formulas of φ that are of one of the forms $\downarrow^x.\chi$, $\sim x$, $\mathtt{on}(x).\chi$ and of all formulas $\mathtt{on}(x).\mathbf{X}\chi$ and $\mathtt{on}(x).\chi\mathbf{U}\xi$ for which $\mathbf{X}\chi$ or $\chi\mathbf{U}\xi$, respectively, are top-level sub-formulas of φ. The proof of the following lemma is analogous to the proof of Lemma 2.

Lemma 3. *For every HTL$^{\sim}_1$-formula φ using at most the variable x and one attribute, there is a PLTL-formula $\overleftarrow{\varphi}$ which does not use any future operator such that for every data word w and all positions i, j on w it holds*

$$(w, i, j) \models \downarrow^x.\varphi \;\Leftrightarrow\; ((w, i)^{\overleftarrow{T}(\varphi)}, i) \models \overleftarrow{\varphi}.$$

Now, we are prepared to define the translation from HTL$^{\sim}_1$-formulas with a single attribute to equivalent LTL$^{\Downarrow}_1$-formulas.

Theorem 14. *Every closed HTL$^{\sim}_{1}$-formula using at most one attribute can be translated into an equivalent LTL$^{\Downarrow}_{1}$-formula.*

Proof. Let φ be a closed HTL$^{\sim}_{1}$-formula using at most one variable x and one attribute. Without loss of generality, we assume that φ is in normal form. We translate φ into an LTL$^{\Downarrow}_{1}$- formula using at most one register r. The translation is defined by mutual recursion between three translations t, $t_\bot$ and $t_\top$ where (i) t is responsible for the translation of sub-formulas which are not in the scope of any $\downarrow^x$, (ii) $t_\bot$ translates sub-formulas which are in the scope of $\downarrow^x$ but there is no data value defined at the x-position and (iii) $t_\top$ deals with sub-formulas which are in the scope of $\downarrow^x$ and the data value at the x-position is defined. The desired formula is $t(\varphi)$.

The definition of t is straightforward. We omit the the Boolean cases:

- $t(p) = p$ for propositions p
- $t(\downarrow^x.\psi) = \big[(\Downarrow^r.\Uparrow^r) \rightarrow \Downarrow^r.t_\top(\psi)\big] \wedge \big[(\neg \Downarrow^r.\Uparrow^r) \rightarrow t_\bot(\psi)\big]$
- $t(\mathbf{X}\psi) = \mathbf{X}t(\psi)$
- $t(\psi_1\mathbf{U}\psi_2) = t(\psi_1)\mathbf{U}t(\psi_2)$
- $t(\mathbf{X}^{\leftarrow}\psi) = \mathbf{X}^{\leftarrow}t(\psi)$
- $t(\psi_1\mathbf{U}^{\leftarrow}\psi_2) = t(\psi_1)\mathbf{U}^{\leftarrow}t(\psi_2)$

We now define $t_\top(\psi)$ and $t_\bot(\psi)$ for formulas ψ which follow immediately after $\downarrow^x$. As the original formula φ is in normal form, we can assume that ψ is of one of the forms $\sim x$, $\mathbf{X}\chi$, $\chi_1\mathbf{U}\chi_2$, $\mathbf{X}^{\leftarrow}\chi$ or $\chi_1\mathbf{U}^{\leftarrow}\chi_2$. We consider the first three cases and benefit from Lemma 2. The other cases are handled analogously and involve Lemma 3. For $b \in \{\top, \bot\}$, we reach $t_b(\psi)$ in three steps. First, let $\overrightarrow{\psi}$ be the LTL-formula guaranteed by Lemma 2. Remember that $\overrightarrow{\psi}$ can contain atomic formulas of the forms $p_{\downarrow^x.\chi}$, $p_{\sim x}$ and $p_{\mathtt{on}(x).\chi}$, but no atomic formula p_x. Note further that due to Lemma 2, for every data word w and all positions i and j of w, it holds

$$w, i, j \models \downarrow^x.\psi \;\Leftrightarrow\; (w,i)^{\overrightarrow{T}(\psi)}, i \models \overrightarrow{\psi}.$$

Now, let $\widehat{\psi}$ be the formula which results from $\overrightarrow{\psi}$ by replacing

- each $p_{\downarrow^x.\chi}$ by $\big[(\Downarrow^r.\Uparrow^r) \rightarrow \Downarrow^r.t_\top(\chi)\big] \wedge \big[(\neg \Downarrow^r.\Uparrow^r) \rightarrow t_\bot(\chi)\big]$, and
- each $p_{\sim x}$ by $\Uparrow^r$ if $b = \top$ and by $\bot$, otherwise.

Note that in case $b = \bot$, there is no data value defined at the x-position. Since in this case $\sim x$ cannot hold at any position, we replace $p_{\sim x}$ with $\bot$. It only remains to eliminate atomic formulas of the kind $p_{\mathtt{on}(x).\chi}$ in $\widehat{\psi}$. For every assignment $\alpha \in [\overrightarrow{T}(\psi) \rightarrow \{\top, \bot\}]$ let $\widehat{\psi}_\alpha$ be the formula resulting from $\widehat{\psi}$ by replacing every occurrence of an atomic formula $p_{\mathtt{on}(x).\chi}$ with $\alpha(\mathtt{on}(x).\chi)$. We finally get

$$t_b(\psi) = \bigvee_{\alpha \in [\overrightarrow{T}(\psi) \rightarrow \{\top,\bot\}]} \widehat{\psi}_\alpha \wedge \Big(\bigwedge_{\substack{\mathtt{on}(x).\chi \in \overrightarrow{T}(\psi) \\ \alpha(\mathtt{on}(x).\chi)=\top}} t_b(\chi) \wedge \bigwedge_{\substack{\mathtt{on}(x).\chi \in \overrightarrow{T}(\psi) \\ \alpha(\mathtt{on}(x).\chi)=\bot}} \neg t_b(\chi)\Big).$$

Note that $t_b(\psi)$ is evaluated at the x-position. Informally, $t_b(\psi)$ guesses for every formula $\mathtt{on}(x).\chi \in \overrightarrow{T}(\psi)$ whether χ holds at the x-position or not, replaces the occurrences of $\mathtt{on}(x).\chi$ in $\widehat{\psi}$ according to its guess by $\top$ or $\bot$ and checks at the "current" position - which is the x-position - that its guess is correct. ∎

The combination of Theorem 14 and Lemma 1 delivers:

Corollary 4. *In the setting where at most one attribute is allowed,* $\mathrm{HTL}^{\sim}_{1}$ *is expressively equivalent to* $\mathrm{LTL}^{\Downarrow}_{1}$*.*

The translation given in the proof of Theorem 14 can be extended easily to a translation from $\mathrm{HTL}^{\sim}_{1}$ with $m > 1$ attributes to $\mathrm{LTL}^{\Downarrow}_{m}$ by applying the technique used in Propositions 11 and 12. In a nutshell, we simulate an $\mathrm{HTL}^{\sim}_{1}$-formula φ with attributes $\mathtt{a}_1, \ldots, \mathtt{a}_m$ by an $\mathrm{LTL}^{\Downarrow}$-formula with registers $r_1, \ldots, r_m$ such that for every ℓ with $1 \leq \ell \leq m$, register r_ℓ is responsible for attribute $\mathtt{a}_\ell$. During the translation, for every sub-formula $\downarrow^x.\psi$, we have to take into account which attributes are defined at the x-position. To be more precise, we define for every $D \subseteq \{1, \ldots, m\}$, a translation t_D such that sub-formulas which are not in the scope of any $\downarrow^x$ are translated by $t_\emptyset$ and sub-formulas in the scope of $\downarrow^x$ are translated by some t_D such that the attributes whose values are defined at the x-position are exactly $\{\mathtt{a}_\ell \mid \ell \in D\}$. The overall translation basically differs from that given in the proof of Theorem 14 only with regard to the handling of formulas of the forms $\downarrow^x.\psi$ and $@\mathtt{a}_\ell \sim x.@\mathtt{a}_k$:

- $t_D(\downarrow^x.\psi) =$
  $\bigvee_{L=\{\ell_1,\ldots,\ell_s\}\subseteq\{1,\ldots,m\}} \left[\left(\bigwedge_{\ell\in L} \Downarrow^{r_\ell}_{@\mathtt{a}_\ell}.\Uparrow^{r_\ell}_{@\mathtt{a}_\ell} \wedge \bigwedge_{\ell\in\{1,\ldots,m\}\setminus L} \neg \Downarrow^{r_\ell}_{@\mathtt{a}_\ell}.\Uparrow^{r_\ell}_{@\mathtt{a}_\ell}\right) \rightarrow \Downarrow^{r_{\ell_1}}_{@\mathtt{a}_{\ell_1}} \cdots \Downarrow^{r_{\ell_s}}_{@\mathtt{a}_{\ell_s}}.t_L(\psi)\right]$
- $t_D(@\mathtt{a}_\ell \sim x.@\mathtt{a}_k) = \begin{cases} \Uparrow^{r_k}_{@\mathtt{a}_\ell}, & \text{if } k \in D \\ \bot, & \text{otherwise} \end{cases}$

Thus, we conclude:

Corollary 5. *Every closed* $\mathrm{HTL}^{\sim}_{1}$*-formula using* m *attributes can be translated into an equivalent* $\mathrm{LTL}^{\Downarrow}_{m}$*-formula.*

The question whether the result of Theorem 14 can be generalized to multiple attributes, i.e., whether $\mathrm{HTL}^{\sim}_{1}$ is expressively equivalent to $\mathrm{LTL}^{\Downarrow}_{1}$ remains as an open question. We assume that the answer is negative. We think that Property CS9 saying that

every client sending a request to a server gets an acknowledgement from the same server after some time

and expressed by the $\mathrm{HTL}^{\sim}_{1}$-formula $\mathbf{G}(\mathtt{req} \rightarrow \downarrow^x.\mathbf{F}(\mathtt{ack} \wedge @\mathtt{receiver} \sim x.@\mathtt{sender} \wedge @\mathtt{sender} \sim x.@\mathtt{receiver}))$ cannot be expressed in $\mathrm{LTL}^{\Downarrow}_{1}$. It seems that even the property that there are two positions agreeing on the values of two attributes is not expressible by using only one freeze register. The fact that the logic $\mathrm{FO}^{\sim}_{2}(\mathtt{Suc}, <)$ with multiple attributes is strictly more expressive than with a single attribute [41] also supports our belief.

7.2.2 Succinctness

In Section 7.2.1.1 we have seen that $\mathrm{HTL}^{\sim}_{2}(\mathbf{X}, \mathbf{U})$ can express properties that cannot be expressed by any $\mathrm{LTL}^{\Downarrow}$-formula. In this section we will show that there are $\mathrm{LTL}^{\Downarrow}$-expressible properties which can be expressed non-elementarily more succinct in $\mathrm{HTL}^{\sim}_{2}$, even in $\mathrm{HTL}^{\sim}_{2}(\mathbf{X}, \mathbf{F})$. Furthermore, even though $\mathrm{HTL}^{\sim}_{1}$ is not more expressive than $\mathrm{LTL}^{\Downarrow}$ (Section 7.2.1.2), we will prove that it can express properties exponentially more succinct than $\mathrm{LTL}^{\Downarrow}$.

We will consider properties which do not set any constraints on data values and we will make use of the simple observation that for the formulation of such properties the expressive power of $\mathrm{LTL}^{\Downarrow}$ does not go beyond PLTL. More formally, we call a property P *data insensitive* if for every data word $w = (S_1, v_1)(S_2, v_2)\ldots$ over some attribute set $\mathtt{Att}$ and every sequence $v'_1, v'_2, \ldots$ of partial attribute-value mappings from $\mathtt{Att}$ to $\mathcal{D}$, w satisfies P if and only if $(S_1, v'_1)(S_2, v'_2)\ldots$ satisfies P.

In other words, data values are irrelevant for the satisfaction of P. For instance, Property CS1 from Chapter 2 is data insensitive. Now, let φ be an $\mathrm{LTL}^{\Downarrow}$-formula describing a data insensitive property. As operations of the form $\Downarrow^r_{@\mathtt{a}}$ require that the value of attribute $\mathtt{a}$ must be defined at the "current" position, the formula φ must be equivalent to the possibly shorter PLTL-formula resulting from φ by replacing all sub-formulas of the form $\Downarrow^r_{@\mathtt{a}}.\psi$ by $\bot$. Thus, we observe:

Observation 6. For every data insensitive property P, the shortest $\mathrm{LTL}^{\Downarrow}$-formula expressing P is a PLTL-formula.

Based on this observation we can reuse some known results on PLTL in the following proofs. We start with the succinctness of $\mathrm{HTL}^{\sim}$ with two variables.

Proposition 13. The logic $\mathrm{HTL}^{\sim}_2(\mathbf{X},\mathbf{F})$ is non-elementarily more succinct than $\mathrm{LTL}^{\Downarrow}$.

Proof. The result is basically a corollary from results in [189], [188] and [184]. In [189], it is shown that there are star-free regular expressions $(\alpha_n)_{n\geq 1}$ built from union, concatenation, and negation such that there is no elementary function f for which $f(n)$ bounds the length of the smallest string satisfying α_n, for every $n \geq 1$. In [94], it is explained how one can build an equivalent FO-formula for every star-free regular expression. Following a similar technique, [184] gives a translation from star-free regular expressions α to formulas φ_α of hybrid temporal logic on linear frames such that φ_α is satisfied by a model if and only if the model encodes a string satisfying α. We can translate φ_α easily into an $\mathrm{HTL}^{\sim}_2(\mathbf{X},\mathbf{F})$-formula of size linear in $|\alpha|$ expressing the data insensitive property that the propositional part of the underlying data word matches α. This means that for every n, the expression α_n can be translated into a corresponding $\mathrm{HTL}^{\sim}_2(\mathbf{X},\mathbf{F})$-formula φ_n of size linear in $|\alpha_n|$.

On the other hand, as PLTL is expressively complete [128, 102], the expressions α_n can be translated into PLTL-formulas and hence, by definition, also into $\mathrm{LTL}^{\Downarrow}$-formulas. For every $n \geq 1$, let ψ_n be the shortest $\mathrm{LTL}^{\Downarrow}$-formula equivalent to φ_n. By Observation 6, every ψ_n must be a PLTL-formula. Every PLTL-formula in turn can be translated into an equivalent LTL-formula of length at most triply exponential in the size of the original one [159]. Moreover, [188] proves that every satisfiable LTL formula can be satisfied by a word of length at most exponential in the size of the formula. It follows that there is no elementary function bounding the length of the formulas ψ_n. □

We now turn to $\mathrm{HTL}^{\sim}$ with a single variable.

Proposition 14. The logic $\mathrm{HTL}^{\sim}_1(\mathbf{F})$ is exponentially more succinct than $\mathrm{LTL}^{\Downarrow}$.

Proof. The proof essentially follows the proof of [94, Theorem 3 (1)] that FO_2 is exponentially more succinct than *unary LTL*. We consider for every $n \geq 1$, the following data insensitive property P_n on data words over the proposition set $\{p_0,\ldots,p_n\}$:

Any two positions of the word that agree on propositions $p_1, p_2, \ldots, p_n$ also agree on proposition p_0.

Let for every $n \geq 1$, the set $\mathcal{L}_n$ be the language of all data words fulfilling P_n. For every $n \geq 1$, the language $\mathcal{L}_n$ is characterized by the following $\mathrm{HTL}^{\sim}_1(\mathbf{F})$-formula φ_n of length $\mathcal{O}(n)$:

$$\varphi_n = \mathbf{G}\Big[\downarrow^x.\mathbf{G}\big(\bigwedge_{i=1}^{n}(p_i \leftrightarrow \mathtt{on}(x).p_i) \rightarrow (p_0 \leftrightarrow \mathtt{on}(x).p_0)\big)\Big].$$

Since each φ_n uses only one variable and does not refer to any attribute, due to Theorem 14 there must be an equivalent $\mathrm{LTL}^{\Downarrow}_1$-formula ψ_n. Let for every $n \geq 1$, ψ'_n be the shortest $\mathrm{LTL}^{\Downarrow}$-formula equivalent to ψ_n. By Observation 6, every ψ'_n must be a PLTL-formula. For the sake of

contradiction, let us assume now that for every $n \geq 1$, we have $|\psi'_n| = 2^{o(n)}$. It follows from [199] that for every $n \geq 1$, there must be a non-deterministic automaton for ψ'_n of size $2^{|\psi'_n|} = 2^{2^{o(n)}}$. However, it can be shown as in [94] that every automaton deciding P_n requires at least 2^{2^n} states which results in a contradiction. □

7.3 Hierarchy Results

In this section, we will show that the variable hierarchy for HTL$^{\sim}$ and the register hierarchy for LTL$^{\Downarrow}$ are infinite, i.e., there is no $k \geq 1$ such that for all $i \geq k$ we have $\text{HTL}^{\sim}_i \leq \text{HTL}^{\sim}_k$ or such that for all $i \geq k$ we have $\text{LTL}^{\Downarrow}_i \leq \text{LTL}^{\Downarrow}_k$. It will turn out that this can be concluded from the result that the variable hierarchy of first order logic on finite undirected ordered graphs is strict [178]. We will first introduce undirected ordered graphs and define first order logic on such structures. Then, we will provide an encoding of these structures by data words. Subsequently, we will describe how first order formulas on these structures can be simulated by HTL$^{\sim}$-formulas on the encodings and how HTL$^{\sim}$-formulas on the encodings can be converted back to first order formulas on the original graphs. Finally, we will explain how from this and the mentioned result in [178] the infinity of the hierarchies for LTL$^{\Downarrow}$ and HTL$^{\sim}$ can be followed.

A *finite undirected ordered graph* $G = (V, E, <)$ consists of a finite set V of nodes, a finite set of undirected edges $E \subseteq \{\{u, v\} \mid u, v \in V \text{ and } u \neq v\}$ and a strict ordering $<$ on the node set V. Note that G cannot contain self-loops, that is, an edge from a node to itself. *First order logic (FO) on undirected ordered graphs* can use, besides universal and existential quantification over graph nodes and Boolean connectives, the edge relation and the ordering on nodes. For instance, the formula

$$\varphi_{big2small} = \forall x_1(\exists x_2(x_2 < x_1) \to \exists x_2(x_2 < x_1 \land E(x_1, x_2)))$$

expresses for every node u that if there exists a smaller node, then, u has an edge to one such node. As usual, for $k \geq 1$, we denote the fragment of FO on finite undirected ordered graphs where at most k variables are allowed, by FO_k. Our results in this section rely on the following theorem:

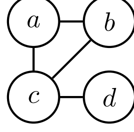
Theorem 15. *[178] For every $k \geq 1$, $\text{FO}_k \lneq \text{FO}_{k+1}$ on finite undirected ordered graphs.*

Next, we define *canonical encodings* of finite undirected ordered graphs by 1-complete data words. A 1-complete data word w is a canonical encoding of a finite ordered undirected graph $G = (V, E, <)$ if the following conditions hold:

- The word w has a node position $n(u)$, for every $u \in V$ and two edge positions $e(u, v)$ and $e(v, u)$, for every edge $\{u, v\} \in E$. Thus, w consists of $|V| + 2|E|$ positions.
- Node positions $n(u)$ have a unique data value and carry only the proposition n.
- For every $u, v \in V$ with $\{u, v\} \in E$, the edge positions $e(u, v)$ and $e(v, u)$ carry only proposition e and have the same data value which does not occur anywhere else.
- The order of the positions obeys for every $u, u', v, v' \in V$ and $\{u, v\}, \{u, v'\}, \{u', v'\} \in E$ the following rules:
  - if $u \leq u'$ then $n(u) < e(u', v')$
  - if $v < v'$ then $e(u, v) < e(u, v')$
  - if $u < v$ then $n(u) < n(v)$
  - if $u < u'$ then $e(u, v) < n(u')$.

  Note that these rules define a unique order on every canonical encoding w.

Example 12. Let $G = (\{a,b,c,d\}, \{\{a,b\},\{a,c\},\{b,c\},\{c,d\}\}, <)$ be the following undirected ordered graph where the nodes are ordered by $a < b < c < d$.



The following data word

Positions:	1	2	3	4	5	6	7	8	9	10	11	12
	n	e	e	n	e	e	n	e	e	e	n	e
	7	8	3	2	8	1	6	3	1	5	4	5

is a canonical encoding of G. The nodes a, b, c, and d are represented by the positions 1, 4, 7 and 11, respectively. The correspondence between edges and word positions is as follows:

- The edge $\{a,b\}$ is represented by postions 2 and 5,
- $\{a,c\}$ by 3 and 8,
- $\{b,c\}$ by 6 and 9 and
- $\{c,d\}$ by 10 and 12.

It should be observed that the underlying linear order on data values is not relevant for the encoding. ☐

A maximal sub-word in a canonical encoding where only the first position is labelled by n is called a *node block*. To give an example, the sub-word from the 4th to the 6th position of the above encoding constitutes a node block.

The following two Lemmas will be helpful to derive from Theorem 15 the infinity of the variable hierarchies for $\text{HTL}^{\sim}$ and $\text{LTL}^{\Downarrow}$.

Lemma 4. *For every formula $\varphi \in FO_k$,there is a formula $\varphi' \in LTL^{\Downarrow}_{k+1}$ such that for every finite undirected ordered graph G and every canonical encoding w_G of G, it holds $G \models \varphi \Leftrightarrow w_G \models \varphi'$.*

Proof. Let for some $k \geq 1$, φ be a formula from FO_k on finite undirected ordered graphs. Without loss of generality, we assume that φ uses at most the variables $\{x_1, \ldots, x_k\}$. We will show how φ can be translated into an $\text{LTL}^{\Downarrow}_{k+1}$-formula $t(\varphi)$ such that for all graphs G and canonical encodings w_G, it holds $G \models \varphi \Leftrightarrow w_G \models t(\varphi)$.

The formula $t(\varphi)$ uses for every variable x_i with $1 \leq i \leq k$, a register r_i simulating x_i. Additionally, it uses an auxiliary register r which helps to determine edge connections. For each variable x_i pointing to some node u, the register r_i stores the data value of the first position of the node block associated with u. The question whether two nodes bounded to variables x_i and x_j are connected by an edge can be tested by checking whether the blocks whose initial data values are stored in r_i and r_j, respectively, share some data value. We define the translation t inductively.

- $t(\exists x_i \psi) = \mathbf{F}\mathbf{F}^{\leftarrow} \Downarrow^{r_i}.(n \wedge t(\psi))$
- $t(x_i = x_j) = \mathbf{F}\mathbf{F}^{\leftarrow}(\Uparrow^{r_i} \wedge \Uparrow^{r_j})$
- $t(x_i < x_j) = \mathbf{F}\mathbf{F}^{\leftarrow}(\Uparrow^{r_i} \wedge \mathbf{X}\mathbf{F} \Uparrow^{r_j})$

- $t(E(x_i, x_j)) = \mathbf{F}\mathbf{F}^{\leftarrow}\Big(\Uparrow^{r_i} \wedge \mathbf{X}\Big(e\mathbf{U}(e\wedge \Downarrow^r.\mathbf{F}(\Uparrow^{r_j} \wedge \mathbf{X}(e\mathbf{U}(e\wedge \Uparrow^r))))\Big)\Big)$
- $t(\neg\psi) = \neg t(\psi)$
- $t(\psi_1 \wedge \psi_2) = t(\psi_1) \wedge t(\psi_2)$

where $i, j \in \{1, \ldots, k\}$. □

Lemma 5. *For every formula $\varphi \in$ HTL$^{\sim}_{k}$, there is a formula $\varphi' \in$ FO$_{2k+6}$ such that for every finite undirected ordered graph G and every canonical encoding w_G of G, it holds $w_G \models \varphi \Leftrightarrow G \models \varphi'$.*

Proof. Given a $\varphi \in$ HTL$^{\sim}_{k}$ for some k, the translation consists of two steps. First, we construct, as in the proof of Theorem 9, an FO$_{k+3}$-formula $\widehat{\varphi}$ that is equivalent to φ on data words. Then, we transform $\widehat{\varphi}$ into φ' by means of a quantifier-free logical interpretation [90, Section 11.2] that defines, for every finite undirected ordered graph $G = (V, E, <)$, a (unique) representation of the canonical encodings of G on the set $V \times V$. However, the translation of $\widehat{\varphi}$ into φ' requires two variables x' and x'' for every variable x of $\widehat{\varphi}$. Thus, the resulting formula potentially has $2k + 6$ variables. More precisely, the logical interpretation $\Phi = (\varphi_U, \varphi_n, \varphi_e, \varphi_<, \varphi_\sim)$ is defined as follows:

- $\varphi_U(x_1, x_2)$ defines the set of pairs that are actual positions of the representation of the canonical encodings. Thus, it is just $x_1 = x_2 \vee (E(x_1, x_2) \wedge E(x_2, x_1))$.
- $\varphi_n(x_1, x_2)$ defines the positions that carry the proposition n and is, thus, $x_1 = x_2$.
- $\varphi_e(x_1, x_2)$ defines the positions carrying e and is $E(x_1, x_2) \wedge E(x_2, x_1)$.
- $\varphi_<(x_1, x_2, y_1, y_2)$ defines the linear order on positions. It is
  $[x_1 = x_2 \wedge y_1 = y_2 \wedge x_1 < y_1]\vee$
  $[x_1 = x_2 \wedge E(y_1, y_2) \wedge (x_1 \leq y_1 \vee x_1 \leq y_2)]\vee$
  $[E(x_1, x_2) \wedge y_1 = y_2 \wedge (x_1 < y_1 \vee x_2 < y_1)]\vee$
  $[E(x_1, x_2) \wedge E(y_1, y_2) \wedge (x_1 \leq y_1 \vee x_1 \leq y_2 \vee x_2 \leq y_1 \vee x_2 \leq y_2)]$.
- Finally, $\varphi_\sim(x_1, x_2, y_1, y_2)$ defines the pairs of positions that have the same data value. It is $[x_1 = y_1 \wedge x_2 = y_2] \vee [x_1 = y_2 \wedge x_2 = y_1]$.

It is not hard to see that Φ indeed defines, for every graph G, the unique representation of the canonical encodings of G. □

From the last two lemmas and Theorem 15 we can derive the main theorem of this section:

Theorem 16. *The LTL$^{\Downarrow}_{k}$-hierarchy and the HTL$^{\sim}_{k}$-hierarchy are infinite.*

Proof. For the sake of contradiction, assume that there is some $k \geq 1$ such that

(1) for every $i > k$ we have LTL$^{\Downarrow}_{i} \equiv$ LTL$^{\Downarrow}_{k}$, or

(2) for every $i > k$ we have HTL$^{\sim}_{i} \equiv$ HTL$^{\sim}_{k}$.

Let $\varphi \in$ FO$_{2k+7}$ be an arbitrary formula on finite undirected ordered graphs. By Lemma 4 there is a formula $\psi \in$ LTL$^{\Downarrow}_{2k+8}$ such that for all finite undirected ordered graphs G and canonical encodings w_G, it holds $G \models \varphi \Leftrightarrow w_G \models \psi$. No matter which case in the assumption above holds, it follows from Lemma 1 that there must be a formula $\psi' \in$ HTL$^{\sim}_{k}$ equivalent to ψ:

- In case (1) there is an LTL$^{\Downarrow}_{k}$-formula equivalent to ψ which we can convert to an equivalent HTL$^{\sim}_{k}$-formula by Lemma 1.

- In the other case, ψ can first be converted into an equivalent formula $\widehat{\psi} \in \mathrm{HTL}^{\sim}_{2k+8}$ using Lemma 1, then, by assumption, there must be an $\mathrm{HTL}^{\sim}_{k}$-formula equivalent to $\widehat{\psi}$.

Then, by Lemma 5 there is a formula $\varphi' \in \mathrm{FO}_{2k+6}$ such that for every finite undirected ordered graph G and every canonical encoding w_G, it holds $w_G \models \psi' \Leftrightarrow G \models \varphi'$. Thus, φ is equivalent to φ' on finite undirected ordered graphs. As φ was chosen arbitrarily from FO_{2k+7}, we can conclude that FO_{2k+7} is expressively equivalent to FO_{2k+6} on finite undirected ordered graphs, a contradiction. □

We conjecture that both hierarchies are even strict, that is, for every $k \geq 1$, it holds $\mathrm{HTL}^{\sim}_{k+1} > \mathrm{HTL}^{\sim}_{k}$ and $\mathrm{LTL}^{\Downarrow}_{k+1} > \mathrm{LTL}^{\Downarrow}_{k}$. We believe that the strictness of the hierarchies can be concluded from their infinity by means of an Ehrenfeucht-Fraïssé game in a similar way as it was done for the FO_k-hierarchy in [178].

7.4 Discussion

We introduced $\mathrm{HTL}^{\sim}$, i.e., Hybrid Temporal Logic on data words, and compared it to $\mathrm{LTL}^{\Downarrow}$. One of our main results is that $\mathrm{HTL}^{\sim}$ is strictly more expressive than $\mathrm{LTL}^{\Downarrow}$ and that it only needs two variables to express a property which is not expressible in $\mathrm{LTL}^{\Downarrow}$. This is in contrast to the equivalence of these logics on classical words without data values. A further surprising result is that in the case of one variable, $\mathrm{HTL}^{\sim}$ looses its additional expressive power with respect to $\mathrm{LTL}^{\Downarrow}$. While every $\mathrm{HTL}^{\sim}$-formula with only one variable can be transformed into an equivalent $\mathrm{LTL}^{\Downarrow}$-formula, 1-variable-$\mathrm{HTL}^{\sim}$ and 1-register-$\mathrm{LTL}^{\Downarrow}$ coincide in terms of expressivity if only one attribute is allowed. The question whether this equivalence can be generalized to multiple attributes remains open. As we stated in the corresponding section we do not assume that this question can be answered affirmatively, because we do not believe that the property that an underlying word contains two arbitrary positions which agree on the values of two attributes can be expressed in the 1-register fragment of $\mathrm{LTL}^{\Downarrow}$.

In terms of succinctness, we showed that $\mathrm{HTL}^{\sim}$-formulas in the 1-variable fragment can be exponentially more succinct than $\mathrm{LTL}^{\Downarrow}$-formulas. Formulas with multiple variables can even be non-elementarily more succinct than $\mathrm{LTL}^{\Downarrow}$-formulas.

We finally derived from the variable hierarchy of FO on finite ordered undirected graphs [178] that the variable hierarchy for $\mathrm{HTL}^{\sim}$ and the register hierarchy for $\mathrm{LTL}^{\Downarrow}$ are both infinite. We conjecture that the strictness of both hierarchies can be concluded from their infinity in a similar way as it was done for FO in [178].

We conclude by mentioning that all our results, besides the results on succinctness, carry over to data ω-words. The translations in Lemmas 1-3, Propositions 9, 11, 12, Theorem 14 and Corollary 5 work on ω-words without any changes. The expressibility results in Corollaries 1-3 and Theorem 13 and the hierarchy results in Theorem 16 also carry over to ω-words, because every separating language can be turned into a language of ω-words by padding every word in the language with an infinite number of positions labelled by a special proposition.

> The results presented in this chapter are generalizations of the results in [129] which was a joint work with Thomas Schwentick. While in [129] we studied $\mathrm{HTL}^{\sim}$ on 1-complete data words, in this work I considered $\mathrm{HTL}^{\sim}$ on general data words. Unlike the 1-complete case, the containment of $\mathrm{LTL}^{\Downarrow}$ in $\mathrm{HTL}^{\sim}$ on general data words is not given by definition. This made the adaption of some results like Proposition 12 and Theorem 14 a bit more complicated than in [129]. Moreover, some results like Lemma 1, Proposition 11 and Corollary 5 which are in the 1-complete case self-evident or follow from other results, needed a proof in the general framework.

Chapter 8

Automata for Two-Variable Logic

As announced in Chapter 5, we will introduce and analyze *Weak Data Automata* which are a restriction of Data Automata and expressively equivalent to $\mathrm{EMSO}^{\sim}_2(\mathtt{Suc})$. Due to the reason that Data Automata are defined on 1-complete data words, we will restrict our studies in this chapter to this kind of structures. A Weak Data Automaton contains, instead of a class automaton, *data constraints* which set simpler conditions on the data values of the underlying words.

In Section 8.1, we will give the formal definition of Weak Data Automata on finite data words and explain their semantics. In Section 8.2.1, we will show that Weak Data Automata are strictly less expressive than Data Automata and incomparable with RA. Section 8.2.2 is devoted to the proof that Weak Data Automata are expressively equivalent to $\mathrm{EMSO}^{\sim}_2(\mathtt{Suc})$. From this result and the fact that the language separating Data Automata from Weak Data Automata can be expressed in $\mathrm{FO}^{\sim}_2(\mathtt{Suc}, <)$, it follows that $\mathrm{FO}^{\sim}_2(\mathtt{Suc}, <)$ is strictly more expressive than $\mathrm{FO}^{\sim}_2(\mathtt{Suc})$. In Section 8.3, we will see that it can be derived from [73] that the non-emptiness problem for Weak Data Automata can be solved in non-deterministic doubly exponential time.

In Section 8.4, we will introduce *Weak Büchi Data Automata* on infinite data words and ask whether the expressivity and complexity results for Weak Data Automata carry over to this model. Our answers will be positive. Weak Büchi Data Automata extend the base automaton of Weak Data Automata by a Büchi acceptance condition. We will show that this model can be characterized logically by the extension of $\mathrm{EMSO}^{\sim}_2(\mathtt{Suc})$ by existential quantifiers over infinite sets. Moreover, Weak Büchi Data Automata are strictly less expressive than Büchi Data Automata and incomparable with Büchi Register Automata. We can also show that the non-emptiness for Weak Büchi Data Automata can be polynomially reduced to the non-emptiness for Weak Data Automata resulting in a non-deterministic doubly exponential upper bound for the complexity of Weak Büchi Data Automata. However, this result will not be presented here, but can be found in [130]. In the last section of this chapter we will discuss some open questions.

8.1 Weak Data Automata

In this section, we will introduce a new automata model called Weak Data Automata (WDA) which follows a similar approach as Data Automata (DA). Just like a DA, a WDA contains a non-deterministic Letter-To-Letter Transducer (LLT) which reads and transforms the marked word projection of the input data word. However, unlike a DA which requires all class words of the resulting data word to be accepted by a finite automaton, a WDA imposes some weaker conditions called data constraints on the class words.

We start with the formal definition data constraints. Let w be a simple data word over some alphabet Γ. That is, w carries at each position a single symbol from Γ and a single data value. For some $\gamma \in \Gamma$, let $\texttt{Values}(w, \gamma)$ be defined as $\{d \mid \text{there is some position } i \text{ labelled by } \gamma \text{ which carries data value } d\}$. We define three kinds of data constraints.

- *Key constraints* $\texttt{key}(\gamma)$ with $\gamma \in \Gamma$: These constraints express that a symbol does not occur at two different positions with the same data value. Thus, given a $\gamma \in \Gamma$, the key constraint $\texttt{key}(\gamma)$ holds on w (written as $w \models \texttt{key}(\gamma)$) if for all distinct positions i and j of w labelled by γ, the data values at positions i and j are distinct.

- *Inclusion constraints* $V(\gamma) \subseteq V(\Gamma')$ with $\gamma \in \Gamma$ and $\Gamma' \subseteq \Gamma$: This kind of constraints state that the data values occurring at positions labelled by some symbol occur also at positions labelled by some other symbols. To define it formally, for some $\gamma \in \Gamma$ and $\Gamma' \subseteq \Gamma$, the inclusion constraint $V(\gamma) \subseteq V(\Gamma')$ holds on w (written as $w \models V(\gamma) \subseteq V(\Gamma')$) if $\texttt{Values}(w, \gamma) \subseteq \bigcup_{\gamma' \in \Gamma'} \texttt{Values}(w, \gamma')$.

- *Denial constraints* $V(\gamma) \cap V(\gamma') = \emptyset$ with $\gamma, \gamma' \in \Gamma$: This type of constraints require that two symbols do not share the same data values. In formal terms, for two symbols $\gamma, \gamma' \in \Gamma$, the denial constraint $V(\gamma) \cap V(\gamma') = \emptyset$ is satisfied by w (written as $w \models V(\gamma) \cap V(\gamma') = \emptyset$) if $\texttt{Values}(w, \gamma) \cap \texttt{Values}(w, \gamma') = \emptyset$.

It should be observed that the satisfaction of a data constraint on a data word does not depend on the order of positions in the word. For a set $\mathcal{C}$ of constraints, we write $w \models \mathcal{C}$ if for every $C \in \mathcal{C}$, we have $w \models \mathcal{C}$.

Note that the key constraints defined above are a restricted version of those introduced in [170]. In the notation of [170], the constraint $\texttt{key}(\gamma)$ can be expressed by $(\{\gamma, \bullet\})$.

A WDA $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ consists of a *base automaton* $\mathcal{B}$ and a finite set $\mathcal{C}$ of data constraints. Like in the case for DA, the base automaton is a non-deterministic LLT with some input alphabet $\Sigma \times \{\bot, \top\}$ and some output alphabet Γ. We consider Σ as the input alphabet of $\mathcal{A}$. Remember from Section 4.2.2 that the marked word projection of a 1-complete data word w contains at every position i, besides the propositions at position i of w, the symbol $\top$ if and only if position $i+1$ of w exists and it carries the same data value as position i. A data word $w = \begin{smallmatrix} P_1 \\ d_1 \end{smallmatrix} \cdots \begin{smallmatrix} P_n \\ d_n \end{smallmatrix}$ over some proposition set $\texttt{Prop}$ is accepted by a WDA $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ over the input alphabet $\Sigma = 2^{\texttt{Prop}}$ if

- there is a transduction $v = \gamma_1 \ldots \gamma_n$ of $\mathcal{B}$ on the marked word projection of w, and
- all data constraints in $\mathcal{C}$ are satisfied by $\begin{smallmatrix} \gamma_1 \\ d_1 \end{smallmatrix} \cdots \begin{smallmatrix} \gamma_n \\ d_n \end{smallmatrix}$.

We give some example WDA:

Example 13. In Example 3 of Section 4.2.2, we described how a DA can check the property:

All data values of the input word are pairwise distinct.

This property can also be checked by a simple WDA $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ where $\mathcal{B}$ outputs at every position the same symbol γ and $\mathcal{C}$ consists of the single key constraint $\texttt{key}(\gamma)$. □

We will see in later sections that the DA-expressible property that every p-position is followed by some q-position with the same data value, considered in Example 4 of Section 4.2.2, is not expressible with WDA. Nevertheless, the next examples show that it is easy to describe properties with WDA in which the order of positions does not play any role.

Example 14. We consider the following property:

For every position labelled by proposition p, there is a position labelled by proposition q which carries the same data value.

To check this property, the base automaton outputs at every p-position a symbol γ and at every q-position a symbol γ'. The only constraint we need is the inclusion constraint $V(\gamma) \subseteq V(\{\gamma'\})$. □

Example 15. Let us have a look at the following property:

For every position labelled by proposition p, there is a position labelled by proposition q which carries a different data value.

The base automaton first guesses whether the data word contains (1) no q-positions, (2) at least one q-position and all q-positions are in the same class, or (3) at least two q-positions in distinct classes. In the following, we say that the base automaton *marks* some position i by some symbol γ if it outputs γ at i. In Case (1), the base automaton just checks that the word neither contains any q- nor any p-position. In Case (2), it assures that there is at least one q position and no position labelled by p and q. It additionally marks the first q-position by a symbol α, all other q-positions by α' and all p-positions by β. The constraint set contains the inclusion constraint $V(\alpha') \subseteq V(\{\alpha\})$ and the denial constraints $V(\beta) \cap V(\alpha) = \emptyset$ and $V(\beta) \cap V(\alpha') = \emptyset$. Observe that in Case (3), it just has to be checked that the guess of the base automaton is correct. To this end, the base automaton chooses two q-positions, marks the first one by γ and the second one by γ'. By the denial constraint $V(\gamma) \cap V(\gamma') = \emptyset$, it is assured that γ and γ' are in different classes.

It is worth to mention that whatever the base automaton guesses, the data constraints do not disturb each other. If, for instance, it guesses Case (2), the constraints of Case (3) hold, because no position is marked by γ or γ'.

□

8.2 Expressivity of Weak Data Automata

In this section, we will devote our attention to the expressive power of WDA. In Section 8.2.1, we will compare WDA to DA and RA. It will turn out that WDA are strictly less expressive than DA. We will also show that WDA and RA are not comparable in terms of expressivity. The focus of Section 8.2.2 will be on the logical characterization of WDA. We will prove that WDA are expressively equivalent to $\text{EMSO}_2^{\sim}(\texttt{Suc})$.

8.2.1 Comparison with other Automata Models

First, we will prove that WDA are strictly less expressive than DA. Then, we will show that they are incomparable with RA in terms of expressivitiy.

We consider the two properties $E_{p<q}$ and E_{p2q} on data words over the proposition set $\{p, q\}$:

$E_{p<q}$: *For every position i labelled by p, there is a position $j > i$ such that j is labelled by q and carries the same data value as j.*

E_{p2q}: *For every position i labelled by p there is a position j with $j = i+2$ such that j is labelled by q and carries the same data value as j.*

Note that every data word fulfilling E_{p2q} satisfies $E_{p<q}$, too. Let $\mathcal{L}_{p2q}$ and $\mathcal{L}_{p<q}$ be the languages of data words satisfying E_{p2q} and $E_{p<q}$, respectively.

Lemma 6. *None of the languages $\mathcal{L}_{p<q}$ and $\mathcal{L}_{p2q}$ can be decided by WDA.*

Proof. Our argumentation uses some kind of pumping property for WDA. We first show that there cannot be any WDA deciding $\mathcal{L}_{p2q}$. For the sake of contradiction, assume that $\mathcal{L}_{p2q}$ is decided by some WDA $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ with some output alphabet Γ for $\mathcal{B}$. Let $n = |\Gamma|^4 + 1$ and $d_1, d'_1, d_2, d'_2, \ldots, d_n, d'_n$ be pairwise different data values. We consider the data word

$$w = \begin{array}{ccccccccccccc} p & p & q & q & p & p & q & q & & p & p & q & q \\ d_1 & d'_1 & d_1 & d'_1 & d_2 & d'_2 & d_2 & d'_2 & \cdots & d_n & d'_n & d_n & d'_n \end{array}$$

of length $4n$. Obviously, w is contained in $\mathcal{L}_{p2q}$ and its marked word projection is $((p, \perp)\ (p, \perp)\ (q, \perp)\ (q, \perp))^n$. As w is accepted by $\mathcal{A}$, there must exist a transduction $\gamma_1 \ldots \gamma_{4n}$ of $\mathcal{B}$ on the marked word projection of w such that all constraints in $\mathcal{C}$ are satisfied by

$$u = \begin{array}{ccccccccccccc} \gamma_1 & \gamma_2 & \gamma_3 & \gamma_4 & \gamma_5 & \gamma_6 & \gamma_7 & \gamma_8 & & \gamma_{4n-3} & \gamma_{4n-2} & \gamma_{4n-1} & \gamma_{4n} \\ d_1 & d'_1 & d_1 & d'_1 & d_2 & d'_2 & d_2 & d'_2 & \cdots & d_n & d'_n & d_n & d'_n \end{array}.$$

Due to the choice of n, there must exist natural numbers i, j with $0 \leq i < j < n$ such that $\gamma_{4i+1}\gamma_{4i+2}\gamma_{4i+3}\gamma_{4i+4} = \gamma_{4j+1}\gamma_{4j+2}\gamma_{4j+3}\gamma_{4j+4}$. Let w' be the data word obtained from w by interchanging the data values of positions $4i+3$ and $4i+4$ with those of $4j+3$ and $4j+4$. That is,

$$w' = \begin{array}{ccccccccccccccccccc} p & p & q & q & & p & p & q & q & & p & p & q & q & & p & p & q & q \\ d_1 & d'_1 & d_1 & d'_1 & \cdots & d_{i+1} & d'_{i+1} & d_{j+1} & d'_{j+1} & \cdots & d_{j+1} & d'_{j+1} & d_{i+1} & d'_{i+1} & \cdots & d_n & d'_n & d_n & d'_n \end{array}.$$

Clearly, $w' \notin \mathcal{L}_{p2q}$. However, as the marked word projections of w and w' are identical, the string $\gamma_1 \ldots \gamma_{4n}$ is also a possible output of $\mathcal{B}$ on the marked word projection of w'. Let

$$u' = \begin{array}{ccccccccccccccccccc} \gamma_1 & \gamma_2 & \gamma_3 & \gamma_4 & & \gamma_{4i+1} & \gamma_{4i+2} & \gamma_{4i+3} & \gamma_{4i+4} & & \gamma_{4j+1} & \gamma_{4j+2} & \gamma_{4j+3} & \gamma_{4j+4} & & \gamma_{4n-3} & \gamma_{4n-2} & \gamma_{4n-1} & \gamma_{4n} \\ d_1 & d'_1 & d_1 & d'_1 & \cdots & d_{i+1} & d'_{i+1} & d_{j+1} & d'_{j+1} & \cdots & d_{j+1} & d'_{j+1} & d_{i+1} & d'_{i+1} & \cdots & d_n & d'_n & d_n & d'_n \end{array}$$

be a data word whose sequence of data values is equal to that of w'. Observe that u' results from u by changing the order of positions. As u satisfies all constraints in $\mathcal{C}$ and the satisfaction of data constraints does not depend on the order of positions of the underlying data word, u' satisfies all constraints in $\mathcal{C}$, too. Thus, $w' \in \mathcal{L}(\mathcal{A}, \mathcal{C})$ which is a contradiction.

The proof for $\mathcal{L}_{p<q}$ is exactly the same. Obviously, $w \in \mathcal{L}_{p<q}$. Moreover, $w' \notin \mathcal{L}_{p<q}$, since for the p-position $4j+1$ there is no subsequent position with the same data value. □

With the help of the last lemma we can prove the following theorem:

Theorem 17. *(a) In terms of expressivity, RA and WDA are incomparable.*

(b) DA are strictly more expressive than WDA.

Proof. (a) From Lemma 6 we know that the language $\mathcal{L}_{p2q}$ cannot be recognized by a WDA whereas it can be decided by an RA using two registers. At every p-position i, the RA stores the data value at i and checks whether position $i+2$ exists, is labelled by q and carries the same data

value as position i. After position $i+2$ the data value of position i is not needed any more and the corresponding register can be overwritten. Thus, two registers suffice to decide $\mathcal{L}_{p2q}$.

On the other side, as mentioned before, it follows from Proposition 4 in [124] that the language of all data words in which every data value occurs at most once, cannot be decided by any RA. However, as demonstrated in Example 13 this language can be decided by a WDA.

(b) We show that every WDA can be translated into an equivalent DA. The strictness follows from (a) and the result that the class of all languages decided by RA is included in the class of languages decided by DA [39].

Let $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ be an arbitrary WDA. We construct a DA $\mathcal{A}' = (\mathcal{B}, \mathcal{C}')$ whose base automaton is the same as that of $\mathcal{A}$ and whose class automaton $\mathcal{C}'$ results from the intersection of all finite automata in $\{\mathcal{C}'_C | C \in \mathcal{C}\}$ where for every $C \in \mathcal{C}$ the automaton $\mathcal{C}'_C$ behaves as follows:

- If C is a key constraint $\texttt{key}(\gamma)$, then $\mathcal{C}'_C$ checks that the input string contains at most one position labelled by γ.
- If C is an inclusion constraint $V(\gamma) \subseteq V(\Gamma')$, then, $\mathcal{C}'_C$ tests that if the input string contains a γ-position, then it contains also a γ'-position for some $\gamma' \in \Gamma'$.
- If C is a denial constraint $V(\gamma) \cap V(\gamma') = \emptyset$, then, $\mathcal{C}'_C$ ensures that the input word does not contain two positions where one of them is labelled by γ and the other by γ'.

□

8.2.2 Logical Characterization

In this section, we will show that the class of all languages decided by WDA can be characterized by $\text{EMSO}_2^{\sim}(\texttt{Suc})$. Our result can be considered as an analogue of the characterizations of Büchi, Elgot and Trakhtenbrot [57, 92, 195] for string languages. A corresponding result on strings is that the regular languages are characterized by $\text{EMSO}_2(\texttt{Suc})$. Since the empty word ε cannot be expressed in the logic, we follow the common approach of ignoring the empty word in logical characterizations. That is, we associate every WDA with an $\text{EMSO}_2^{\sim}(\texttt{Suc})$-formula such that if the automaton accepts the empty word, then, its language is the language of the formula augmented by the empty word.

First, we prove that for every WDA, there is a corresponding $\text{EMSO}_2^{\sim}(\texttt{Suc})$-formula.

Lemma 7. *For every WDA $\mathcal{A}$, a corresponding $\text{EMSO}_2^{\sim}(\texttt{Suc})$-formula φ with $\mathcal{L}(\mathcal{A}) - \{\epsilon\} = \mathcal{L}(\varphi)$ is constructible in polynomial time.*

Proof. Let $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ be a WDA with $\mathcal{B} = (\Sigma, \Gamma, S, s_0, \delta, F)$ such that $\Sigma = 2^{\texttt{Prop}}$ for some proposition set $\texttt{Prop}$, $S = \{s_0, s_1, \ldots, s_n\}$ and $\Gamma = \{\gamma_1, \ldots, \gamma_\ell\}$. We will construct an $\text{EMSO}_2^{\sim}(\texttt{Suc})$-formula φ over $\texttt{Prop}$ with $\mathcal{L}(\mathcal{A}) - \{\epsilon\} = \mathcal{L}(\varphi)$. The construction is very similar to the classical translation of finite automata into MSO-formulas (see, e.g., in [194]).

In the sequel, for $\sigma \in \Sigma$ and a position variable x, we use $\sigma(x)$ as an abbreviation for $\bigwedge_{p\in\sigma} p(x) \wedge \bigwedge_{p\in\texttt{Prop}\setminus\sigma} \neg p(x)$. Moreover, we assume, without loss of generality, that $\mathcal{A}$ uses the initial state s_0 only once. The formula φ is of the form

$$\exists R_{s_1} \ldots \exists R_{s_n} \exists R_{\gamma_1} \ldots \exists R_{\gamma_\ell} (\varphi^S_{\texttt{part}} \wedge \varphi^\Gamma_{\texttt{part}} \wedge \varphi_{\texttt{start}} \wedge \varphi_{\texttt{trans}} \wedge \varphi_{\texttt{accept}} \wedge \varphi_{\texttt{constr}}).$$

We give an informal description of the formula:

- The variables $R_{s_1}, \ldots, R_{s_n}, R_{\gamma_1} \ldots R_{\gamma_\ell}$ are set variables with the following intended meaning: Each R_{s_i} with $1 \le i \le n$, consists of all positions of the underlying word which, after being read, move the automaton $\mathcal{B}$ into state s_i. Each R_{γ_j} with $1 \le j \le \ell$, consists of all positions at which $\mathcal{B}$ outputs γ_j.

- The formulas $\varphi^S_{\texttt{part}}$ and $\varphi^\Gamma_{\texttt{part}}$ assert that $R_{s_1},\ldots,R_{s_n}$ as well as $R_{\gamma_1},\ldots,R_{\gamma_\ell}$ partition the position set of the input word.
- The formula $\varphi_{\texttt{start}}$ expresses that the automaton starts in state s_0.
- The formula $\varphi_{\texttt{trans}}$ ensures that the sets $R_{s_1},\ldots,R_{s_n},R_{\gamma_1}\ldots R_{\gamma_\ell}$ are consistent with the transition relation of $\mathcal{B}$.
- The formula $\varphi_{\texttt{accept}}$ requires that the marked word projection of the input word is accepted by $\mathcal{B}$.
- The formula $\varphi_{\texttt{constr}}$ guarantees that the data constraints in $\mathcal{C}$ are fulfilled.

Now, we give the formal definition of φ.

- The definitions of $\varphi^S_{\texttt{part}}$ and $\varphi^\Gamma_{\texttt{part}}$ are easy:

$$\varphi^S_{\texttt{part}} = \forall x\Big[\bigvee_{1\le i\le n} R_{s_i}(x) \wedge \bigwedge_{1\le i\le n}\big(R_{s_i}(x) \rightarrow \bigwedge_{j\in\{1,\ldots,n\}\setminus\{i\}} \neg R_{s_j}(x)\big)\Big],$$

$$\varphi^\Gamma_{\texttt{part}} = \forall x\Big[\bigvee_{1\le i\le \ell} R_{\gamma_i}(x) \wedge \bigwedge_{1\le i\le \ell}\big(R_{\gamma_i}(x) \rightarrow \bigwedge_{j\in\{1,\ldots,\ell\}\setminus\{i\}} \neg R_{\gamma_j}(x)\big)\Big].$$

- The formula $\varphi_{\texttt{start}}$ ensures that the state and output of $\mathcal{B}$, after reading the first position of the marked word projection of the input word, are in accordance with the label at that position and the transition relation. Remember that the label of a position in the marked word projection depends on the equality relation between its own data value and that of its neighbour.

$$\begin{aligned}\varphi_{\texttt{start}} =&\exists x\Bigg[\neg\exists y\texttt{Suc}(y,x)\wedge\\ &\bigwedge_{\sigma\in\Sigma}\Bigg(\Big[\big(\sigma(x)\wedge\exists y(\texttt{Suc}(x,y)\wedge x{\sim}y)\big)\rightarrow \bigvee_{(s_0,(\sigma,\top),\gamma,s)\in\delta}\big(R_s(x)\wedge R_\gamma(x)\big)\Big]\wedge\\ &\Big[\big(\sigma(x)\wedge\neg\exists y(\texttt{Suc}(x,y)\wedge x{\sim}y)\big)\rightarrow \bigvee_{(s_0,(\sigma,\bot),\gamma,s)\in\delta}\big(R_s(x)\wedge R_\gamma(x)\big)\Big]\Bigg)\Bigg]\end{aligned}$$

- The formula $\varphi_{\texttt{trans}}$ is a generalization of the assertions above to all positions. That is, it checks that the states and output symbols associated with the positions are consistent with the transition relation.

$$\begin{aligned}\varphi_{\texttt{trans}} =&\forall x\forall y\Bigg[\texttt{Suc}(x,y)\rightarrow\\ &\bigwedge_{\sigma\in\Sigma,s\in S}\Bigg(\Big[\big(R_s(x)\wedge\sigma(y)\wedge\exists x(\texttt{Suc}(y,x)\wedge x{\sim}y)\big)\rightarrow \bigvee_{(s,(\sigma,\top),\gamma,s')\in\delta}\big(R_{s'}(y)\wedge R_\gamma(y)\big)\Big]\wedge\\ &\Big[\big(R_s(x)\wedge\sigma(y)\wedge\neg\exists x(\texttt{Suc}(y,x)\wedge x{\sim}y)\big)\rightarrow \bigvee_{(s,(\sigma,\bot),\gamma,s')\in\delta}\big(R_{s'}(y)\wedge R_\gamma(y)\big)\Big]\Bigg)\Bigg]\end{aligned}$$

- The formula $\varphi_{\texttt{accept}}$ only checks that the last position is contained in some R_s with $s \in F$.

$$\exists x \big[\neg \exists y \texttt{Suc}(x, y) \wedge \bigvee_{s \in F} R_s(x)\big]$$

- Remember that the data constraints are formulated on the output alphabet Γ of $\mathcal{B}$ and that the outputs of the latter are determined by the sets $R_{\gamma_1}, \ldots, R_{\gamma_\ell}$. The formula $\varphi_{\texttt{constr}}$ is a conjunction $\bigwedge_{C \in \mathcal{C}} \psi_C$ such that
  - if C is a key constraint $\texttt{key}(\gamma)$, then

$$\psi_C = \forall x \forall y \big[(R_\gamma(x) \wedge R_\gamma(y) \wedge x \sim y) \rightarrow x = y\big],$$

  - if C is an inclusion constraint $V(\gamma) \subseteq V(\Gamma')$, then

$$\psi_C = \forall x \exists y \big[R_\gamma(x) \rightarrow \bigvee_{\gamma' \in \Gamma'} (R_{\gamma'}(y) \wedge x \sim y)\big], \textit{and}$$

  - if C is a denial constraint $V(\gamma) \cap V(\gamma') = \emptyset$, then

$$\psi_C = \forall x \forall y \big[(R_\gamma(x) \wedge R_{\gamma'}(y)) \rightarrow \neg x \sim y\big].$$

The length of φ is $\mathcal{O}(|\Sigma||S||\delta| + |\mathcal{C}|)$. The correctness of the formula is straightforward and, thus, omitted. ☐

In [43] it is shown that every $\text{EMSO}_2^{\sim}(\texttt{Suc}, <)$-formula can be translated into an equivalent DA in doubly exponential time. We give an analogous result for $\text{EMSO}_2^{\sim}(\texttt{Suc})$ and WDA.

Lemma 8. *For every $\text{EMSO}_2^{\sim}(\texttt{Suc})$-formula, an equivalent WDA $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ is constructible in doubly exponential time. The size of the output alphabet of $\mathcal{B}$ and the number of the constraints in $\mathcal{C}$ are at most exponential.*

Proof. Let φ be an $\text{EMSO}_2^{\sim}(\texttt{Suc})$-formula using propositions from the set $\texttt{Prop} = \{p_1, \ldots, p_\ell\}$. We will give an algorithm running in doubly exponential time which translates φ into an equivalent WDA $\mathcal{A} = (\mathcal{B}, \mathcal{C})$.

First we explain some terms and notational elements which we are going to use in the proof. The base automaton $\mathcal{B}$ uses a finite input alphabet $\Sigma \times \{\bot, \top\}$ where each $\sigma \in \Sigma$ represents a subset of $\texttt{Prop}$. Similarly, each symbol from the output alphabet Γ of $\mathcal{B}$ represents a finite set of unary relation symbols. When we say that $\mathcal{B}$ *outputs* some relation symbol R at some position, we mean that it outputs a symbol representing a set including R. Likewise, we use boolean combinations of relation symbols to describe output symbols within data constraints. For instance, for three relation symbols R_1, R_2, R_3, the expression $\texttt{key}(R_1 \wedge R_2 \wedge \neg R_3)$ is an abbreviation for the set of key constraints $\texttt{key}(\gamma)$ with $R_1, R_2 \in \gamma$ and $R_3 \notin \gamma$. Given a set $\mathcal{R}$ of unary relation symbols and a data word position i, we say that a subset $\mathcal{R}' \subseteq \mathcal{R}$ is the *full atomic type* of i with respect to $\mathcal{R}$, if $i \in R$ for all $R \in \mathcal{R}'$ and $i \notin R$ for all $R \in \mathcal{R} \backslash \mathcal{R}'$. Likewise, a quantifier-free FO-formula $\alpha(x)$ is called a *full atomic type* with respect to $\mathcal{R}$ if there is a subset $\mathcal{R}' \subseteq \mathcal{R}$ such that $\alpha(x) = \bigwedge_{R \in \mathcal{R}'} R(x) \wedge \bigwedge_{R \in \mathcal{R} \backslash \mathcal{R}'} \neg R(x)$.

The sequel of the proof is structured as follows. First, we will convert φ into an equivalent formula φ' in normal form. Then, we will introduce some relations on word positions which will help to describe the strategy of $\mathcal{A}$. Finally, we will explain how, by means of these relations and some data constraints, the base automaton $\mathcal{B}$ tests that an input data word satisfies φ'.

We start with the normal form. Following [43], we first transform φ into an equivalent formula

$$\psi = \exists R_1 \dots \exists R_n \; [\forall x \forall y \; \chi' \wedge \bigwedge_{i=1}^{m} \forall x \exists y \; \chi'_i]$$

in Scott normal form [107] where χ' and each χ'_i are quantifier-free $\mathrm{FO}^{\sim}$-formulas and the size of ψ is linear in the size of φ. Note that the $\mathrm{FO}^{\sim}$-part of ψ can contain, besides the unary relation symbols representing propositions, the relation symbols $R_1, \dots, R_n$. For simplicity, we refer to the relation symbols for the propositions $p_1, \dots, p_\ell$ as $R_{n+1}, \dots, R_{n+\ell}$. In the next step, we rewrite the formula χ' into a conjunction

$$\chi = \bigwedge_{j=1}^{k} \neg(\alpha_j(x) \wedge \beta_j(y) \wedge \delta_j(x,y) \wedge \varepsilon_j(x,y))$$

where k is at most exponential and for every j, we have that

- $\alpha_j(x)$ and $\beta_j(y)$ are full atomic types with respect to $\mathcal{R} = \{R_1, \dots, R_{n+\ell}\}$,
- δ_j is either $x \sim y$ or $\neg x \sim y$ and
- $\varepsilon_j(x,y)$ is of one of the forms $x = y$, $\mathtt{Suc}(x,y)$, $\mathtt{Suc}(y,x)$ and $F(x,y)$, where $F(x,y)$ is an abbreviation for the formula $\neg\mathtt{Suc}(x,y) \wedge \neg\mathtt{Suc}(y,x) \wedge \neg x = y$ expressing that the distance of x and y is at least two.

Likewise, we rewrite every χ'_i into a disjunction

$$\chi_i = \bigvee_{j=1}^{h} (\alpha^i_j(x) \wedge \beta^i_j(y) \wedge \delta^i_j(x,y) \wedge \varepsilon^i_j(x,y))$$

where h is again at most exponential and each conjunct is of the respective form as above.

The base automaton $\mathcal{B}$ guesses for each position to which of the following relations the position does belong to and outputs the corresponding symbol at that position.

- Relations $R_1, \dots, R_{n+\ell}$ have the expected semantics. We refer to the full atomic type of a position with respect to the relations $R_1, \dots, R_{n+\ell}$ as its *SNF-type*.
- Relations $P_1, P_2, P_3, P^{\#1}, P^{\#2}, P^{\#3}$ with the following intention: If a class contains at least three positions of some SNF-type α, then exactly one of them is in P_3. If it contains at least two α-positions, then exactly one of them is in P_2, but not in P_3. If it contains at least one α-position, then exactly one of them is contained in P_1, but not in P_2 or P_3.

  Moreover, if a class contains at least three α-positions, then *all* α-positions of the class are contained in $P^{\#3}$, but not in any other P_k with $k \neq 3$. If it contains exactly k α-positions with $1 \leq k \leq 2$, all α-positions of the class are contained in $P^{\#k}$, but not in any other P_r with $r \neq k$.
- Relations $C_1, C_2, C_3, C^{\#1}, C^{\#2}, C^{\#3}$ with the following meaning: If there are at least three classes containing some position of some SNF-type α, then in exactly one of this classes, all α-positions are in C_3. If there are exactly k, $1 \leq k \leq 2$, classes containing α-positions, then all α-positions of exactly one of this classes are contained in C_k, but in no other C_r with $r \neq k$.

  Furthermore, if there are at least three classes containing α-positions, then all α-positions of the whole word are in $C^{\#3}$, but not in any $C^{\#r}$ with $k \neq r$. If there are exactly $k \in \{1,2\}$

classes containing α-positions, then all α-positions in the entire word are in $C^{\#3}$, but not in any other $C^{\#r}$ with $k \neq r$.

We refer to the full atomic type of a position with respect to all P_k, $P^{\#k}$, C_k and $C^{\#k}$ with $1 \leq k \leq 3$ as its *occurrence type*.

- Relations $\overleftarrow{E}, \overrightarrow{E}$ with the following intention: A position is in $\overleftarrow{E}$ (respectively, $\overrightarrow{E}$) if its left (respectively, right) neighbour exists and has the same data value.
- The relations $\overleftarrow{R}_k, \overrightarrow{R}_k$ for $k \in \{1, \ldots, n+\ell\}$ and $\overleftarrow{P}_k, \overleftarrow{P}^{\#k}, \overleftarrow{C}_k, \overleftarrow{C}^{\#k}, \overrightarrow{P}_k, \overrightarrow{P}^{\#k}, \overrightarrow{C}_k, \overrightarrow{C}^{\#k}$ for $k \in \{1, \ldots, 3\}$ with the following intention: For each position i, it holds that i is in a relation with an arrow pointing to the left (respectively, right), if its left (respectively, right) neighbour is in the corresponding relation without an arrow. We refer to the full atomic type of a position with respect to these relations and the relations $\overleftarrow{E}$ and $\overrightarrow{E}$ as its *neighbourhood type*.

Now, we explain how it can be ensured by $\mathcal{B}$ and $\mathcal{C}$ that the guesses of $\mathcal{B}$ are consistent with regard to the intentions described above.

- The consistency with respect to the relations $R_{n+1}, \ldots, R_{n+\ell}$ can be checked easily by $\mathcal{B}$ with the help of the read input symbols. Note that there are no consistency conditions with respect to $R_1, \ldots, R_n$.
- The consistency with respect to the P_k- and $P^{\#k}$-relations can be tested as follows: For every α, the automaton $\mathcal{B}$ can ensure:
  - Each α-position is in at most one P_k- and exactly one $P^{\#k}$-relation.
  - Each α-position which is in P_3 or not in any P_k is contained in $P^{\#3}$.
  - Each α-position in P_2 is contained in $P^{\#2}$ or $P^{\#3}$.
  - Each α-position in P_1 is contained in $P^{\#1}$, $P^{\#2}$ or $P^{\#3}$.

  The following inclusion constraints enforce for every α and every class that (i) the class contains an α-position in P_3 if it contains an α-position which is not in any P_k-relation and (ii) it contains an α-position in P_{k-1} if it contains an α-position in P_k with $2 \leq k \leq 3$:

  - $V(\alpha \wedge \neg P_1 \wedge \neg P_2) \subseteq V(\alpha \wedge P_3)$
  - $V(\alpha \wedge P_3) \subseteq V(\alpha \wedge P_2 \wedge P^{\#3})$
  - $V(\alpha \wedge P_2 \wedge P^{\#3}) \subseteq V(\alpha \wedge P_1 \wedge P^{\#3})$
  - $V(\alpha \wedge P_2 \wedge P^{\#2}) \subseteq V(\alpha \wedge P_1 \wedge P^{\#2})$

  The next inclusion constraints guarantee that if a class contains an α-position in some $P^{\#k}$, then it contains *at least* k α-positions:

  - $V(\alpha \wedge P_1 \wedge P^{\#3}) \subseteq V(\alpha \wedge P_2 \wedge P^{\#3})$
  - $V(\alpha \wedge P_1 \wedge P^{\#2}) \subseteq V(\alpha \wedge P_2 \wedge P^{\#2})$
  - $V(\alpha \wedge P_2 \wedge P^{\#3}) \subseteq V(\alpha \wedge P_3 \wedge P^{\#3})$

  To complete the correctness, we have to assure that each class contains for every α and relation P_k, at most one α-position in P_k. This can be done by key constraints. Note that by these constraints we implicitly avoid that a class contains an α-position in some $P^{\#k}$ and another α-position in some $P^{\#r}$ with $k \neq r$.

- The consistency with respect to the C_k- and $C^{\#k}$-relations can be tested in a similar fashion. For every α, the base automaton checks that if there is at least one α-position, then
  - every α-position is in at most one C_k,
  - there is exactly one k such that all α-positions are in $C^{\#k}$, but not in any other $C^{\#r}$ with $k \neq r$,
  - if all α-positions are in $C^{\#3}$, then there are α-positions in C_3, α-positions in C_2 and α-positions in C_1,
  - if all α-positions are in $C^{\#2}$, then there are α-positions in C_2 and α-positions in C_1 and every α-position is either in C_2 or C_1, and
  - if all α-positions are in $C^{\#1}$, then all α-positions are in C_1.

  The property that in every class containing an α-position, either no α-position belongs to any C_k or all α-positions belong to exactly one C_k can be ensured by denial constraints. The requirement that a C_k-relation does not contain two α-positions from different classes can be guaranteed by a joint work between the base automaton $\mathcal{B}$ and some inclusion constraints: For every C_k containing at least one α-position, the base automaton chooses exactly one α-position in C_k and outputs a special relation symbol $\widehat{C}_k$ at that position. Moreover, $\mathcal{C}$ involves the inclusion constraint $V(\alpha \wedge C_k) \subseteq V(\alpha \wedge \widehat{C}_k)$.

- The consistency with respect to neighbourhood types can be tested easily by $\mathcal{B}$. Remember that the marked word projection contains for every position the information about whether the next position exists and has the the same data value or not.

Now we describe how it can be tested by $\mathcal{B}$ and $\mathcal{C}$ that for all positions x and y of the input word, the formula χ is satisfied. For every conjunct $\neg(\alpha_j(x) \wedge \beta_j(y) \wedge \delta_j(x,y) \wedge \varepsilon_j(x,y))$ with $1 \leq j \leq k$, we distinguish between the following cases:

- $\varepsilon_j(x,y)$ is $x = y$: Note that in the case that $\delta_j(x,y)$ is $\neg x \sim y$ or α_j and β_j are different SNF-types, the formula obviously holds. In the remaining case, such a formula just forbids the occurrence of the SNF-type α_j which can be checked easily by $\mathcal{B}$.

- $\varepsilon_j(x,y)$ is $\texttt{Suc}(x,y)$ or $\texttt{Suc}(y,x)$: Such formulas state that the SNF-types α_j and β_j are forbidden as neighbours with equal or different data values. As this is a question of consistency between the SNF-type and the neighbourhood type of a position, it can be tested by $\mathcal{B}$.

- $\varepsilon_j(x,y)$ is $F(x,y)$ and $\delta_j(x,y)$ is $x \sim y$: This kind of formulas state that there should not be an α_j-position s and a β_j-position t in the same class with $|s-t| > 1$. This kind of constraints can be tested by the base automaton, by using neighbourhood types and some denial constraints. We exemplarily show for some cases how the validity of such a constraint can be ensured.

  Let $\alpha_j \neq \beta_j$ and s be an α_j-position such that the left neighbour is a β_j-position with the same data value as s and the right neighbour is not a β_j-position. In such a case, there should not be a further β_j-position in the class of s. Thus, the base automaton just checks that the left neighbour of s is contained in $P^{\#1}$.

  If s is an α_j-position such that neither the left nor the right neighbour is a β_j-position, it has to be assured that there is not any β_j-position in the class of s. The base automaton does not check anything, but $\mathcal{C}$ includes the denial constraint $V(\alpha_j \wedge \neg\overleftarrow{\beta_j} \wedge \neg\overrightarrow{\beta_j}) \cap V(\beta_j) = \emptyset$; here we use $\overleftarrow{\beta_j}$ (respectively, $\overrightarrow{\beta_j}$) as an abbreviation for the Boolean combination of neighbourhood types expressing that the left (respectively, the right) neighbour is not a β_j-positions.

- $\varepsilon_j(x,y)$ is $F(x,y)$ and $\delta_j(x,y)$ is $\neg x \sim y$: Such formulas postulate that α_j-positions s and β_j-positions t with $|s-t| > 1$ need to be in the same class. This can be tested by $\mathcal{B}$ and some inclusion constraints. We describe some example scenarios:

  If $\alpha_j = \beta_j$ and s is an α_j-position such that neither the left nor the right neighbour of s is an α_j-position, then it has to be ensured that all α_j-positions are in the same class. Thus, the base automaton just checks that s is in $C^{\#1}$.

  Assume that $\alpha_j \neq \beta_j$. Moreover, let s be an α_j-position whose left neighbour is a β_j-position with the same data value as s and the right neighbour is a β_j-position with a different data value. Then, the task is to check that there is no further β_j-position in the class of the right neighbour of s and there are no more than two different classes containing β_j-positions. Therefore, the base automaton assures that the left neighbour of s is in $C^{\#2}$ and the right neighbour is in $C^{\#2}$ and $P^{\#1}$.

  In cases where the data values of both neighbours of s differ from the data value of s, we use inclusion constraints.

What remains to be shown is how the formulas χ_i can be tested. For each $i \leq m$ and each position s, it has to be checked that there is a witness position t such that some disjunct $(\alpha^i_j(x) \wedge \beta^i_j(y) \wedge \delta^i_j(x,y) \wedge \varepsilon^i_j(x,y))$ of χ_i is satisfied for $x = s$ and $y = t$. This can be accomplished in the following way. For every $i \leq m$ and every position s, the base automaton $\mathcal{B}$ guesses for which j there is a witness position t such that the formula $(\alpha^i_j(x) \wedge \beta^i_j(y) \wedge \delta^i_j(x,y) \wedge \varepsilon^i_j(x,y))$ holds for $x = s$ and $y = t$. Clearly, if $\varepsilon^i_j(x,y)$ is equal to $x = y$, $\texttt{Suc}(x,y)$ or $\texttt{Suc}(y,x)$, the existence of a witness position can be checked by SNF- and neighbourhood types, hence, by $\mathcal{B}$ without using any constraints. In the case that $\varepsilon^i_j(x,y)$ is equal to $F(x,y)$ and $\delta^i_j(x,y)$ is equal to $x \sim y$, the existence of witness positions can be guaranteed by the base automaton and by inclusion constraints using a new relation symbol $W^{\sim}$. The case where $\varepsilon^i_j(x,y)$ is equal to $F(x,y)$ and $\delta^i_j(x,y)$ is equal to $\neg x \sim y$ can be handled by $\mathcal{B}$ and some denial constraints using a symbol $W^{\not\sim}$. We describe the behaviour of $\mathcal{B}$ in the last two cases in more detail.

- $\varepsilon^i_j(x,y)$ is $F(x,y)$ and $\delta^i_j(x,y)$ is $x \sim y$: For every α^i_j-position s, it has to be ensured that there is a β^i_j-position t with $|s-t| > 1$ such that s and t have the same data value. For the case that $\alpha^i_j \neq \beta^i_j$, we consider some example positions s.

  Let s be an α^i_j-position whose left neighbour has a different data value and the right neighbour is a β^i_j-position with the same data value as s. In this case, it is required that there is a further β^i_j-position in the class of s. Thus, the base automaton checks that the right neighbour of s is in $P^{\#2}$ or $P^{\#3}$.

  If s is an α^i_j-position such that both neighbours have different data values than s, it has to be guaranteed that there is a β^i_j-position in the class of s. To this end, the automaton outputs $W^{\sim}$ at position s. Moreover, $\mathcal{C}$ includes the inclusion constraint $V(\alpha^i_j \wedge \neg\overleftarrow{E} \wedge \neg\overrightarrow{E} \wedge W^{\sim}) \subseteq V(\beta^i_j)$.

- $\varepsilon^i_j(x,y)$ is $F(x,y)$ and $\delta^i_j(x,y)$ is $\neg x \sim y$: Such a formula holds for an α^i_j-position s if there is a β^i_j-position t such that $|s-t| > 1$ and t ha s a different data value than s.

  Let $\alpha^i_j \neq \beta^i_j$ and s be an α^i_j-position for which $\mathcal{B}$ assumes that such a formula holds. Let the left neighbour of s be a β^i_j-position with the same data value as s and the right neighbour be a β^i_j-position with a different data value. In this case, either there must be a third class containing a β^i_j-position or the class of the right neighbour of s has to include a further β^i_j-position. Consequently, the base automaton just checks that the right neighbour of s is in $C^{\#3}$, $P^{\#2}$ or $P^{\#3}$.

If neither the left nor the right neighbour of s is a β^i_j-position or has the same data value as s, it must be assured that there is a β^i_j-position outside the class of s. In this case, the base automaton either ensures that there is a β^i_j-position in the whole word included in $C^{\#2}$ or it outputs $W^{\not\sim}$ at s. Additionally, we have the denial constraint $V(\alpha^i_j \wedge \neg\overleftarrow{E} \wedge \neg\overrightarrow{E} \wedge \neg\overleftarrow{\beta_j} \wedge \neg\overrightarrow{\beta_j} \wedge W^{\not\sim}) \cap V(\beta^i_j \wedge C_1) = \emptyset$ in $\mathcal{C}$.

It should be noted that the number of the relations used by $\mathcal{B}$ and $\mathcal{C}$ is $\mathcal{O}(|\varphi|)$. Thus, the size of the output alphabet of $\mathcal{B}$ and the number of constraints are at most exponential in $|\varphi|$. As the normal form in the first part of the proof has at most exponential length, the number of the states of $\mathcal{B}$ and the overall construction time of $\mathcal{A}$ are at most doubly exponential. □

From Lemmas 7 and 8, we conclude:

Theorem 18. *Weak data automata and* $\text{EMSO}_2^{\sim}(\texttt{Suc})$ *are expressively equivalent.*

It is known that $\text{EMSO}_2(\texttt{Suc})$ and $\text{EMSO}_2(\texttt{Suc}, <)$ are expressively equivalent on usual strings. Due to the above characterization and Lemma 6, this equivalence does not carry over to data words: $\text{EMSO}_2^{\sim}(\texttt{Suc})$ is expressively equivalent to WDA (Theorem 18) by which the language $\mathcal{L}_{p<q}$ cannot be decided (Lemma 6). However, as $\mathcal{L}_{p<q}$ can be expressed by the $\text{EMSO}_2^{\sim}(\texttt{Suc}, <)$-formula

$$\forall y \exists y (p(x) \rightarrow (x < y \wedge q(y) \wedge x \sim y)),$$

it follows:

Corollary 6. *The logic* $\text{EMSO}_2^{\sim}(\texttt{Suc}, <)$ *is strictly more expressive than* $\text{EMSO}_2^{\sim}(\texttt{Suc})$.

8.3 Complexity of Weak Data Automata

While an elementary upper bound for the complexity of the non-emptiness problem for DA is not known yet, we will show in this section that in the case of WDA this problem can be solved in non-deterministic doubly exponential time. It will turn out that this result can be derived easily from [73].

We first discuss some extended versions of key and inclusion constrains studied in [73]. Let Γ be a finite alphabet:

- *Disjunctive key constraints* $\texttt{key}(\Gamma')$ with $\Gamma' \subseteq \Gamma$: Such a constraint is satisfied by a data word w if each of its classes has at most one position with a symbol from Γ'. That is, $\texttt{key}(\Gamma')$ holds on w (written as $w \models \texttt{key}(\Gamma')$) if w does not contain any class with distinct positions i and j such that both are labelled by some symbol from Γ'.
- *Disjunctive inclusion constraints* $V(\Gamma') \subseteq V(\Gamma'')$ with $\Gamma', \Gamma'' \subseteq \Gamma$: This constraint is satisfied by a data word w if each class containing a position labelled by a symbol from Γ', contains also a position labelled by a symbol from Γ''. More formally, we write $w \models V(\Gamma') \subseteq V(\Gamma'')$ if $\bigcup_{\gamma \in \Gamma'} \texttt{Values}(w, \gamma) \subseteq \bigcup_{\gamma \in \Gamma''} \texttt{Values}(w, \gamma)$.

A WDA is attributed with the term *extended* if it allows disjunctive key and inclusion constraints, besides the usual key, inclusion and denial constraints.

Before proving the upper complexity bound for WDA, we introduce a further automata model with constraints. A *Profile Automaton* $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ consists of a usual NFA $\mathcal{B}$ with some input alphabet $\Gamma \times \{\bot, \top\}$ and a set $\mathcal{C}$ of (extended) data constraints over Γ. We call Γ the input alphabet of $\mathcal{A}$. A simple data word w over Γ is accepted by $\mathcal{A}$ if $\mathcal{B}$ accepts the marked word projection of w and w satisfies all constraints in $\mathcal{C}$. We formulate a simple observation:

Observation 7. The non-emptiness problem for extended WDA can be polynomially reduced to the non-emptiness problem for Profile Automata.

Given an extended WDA $(\mathcal{B},\mathcal{C})$, one can construct a Profile Automaton $(\mathcal{B}',\mathcal{C})$ whose input alphabet equals the output alphabet of $\mathcal{B}$ and $\mathcal{B}'$ simply guesses a word projection and simulates $\mathcal{B}$ on it. To describe it technically, for every transition $(s_1,(\sigma,b),\gamma,s_2)$ of $\mathcal{B}$, the NFA $\mathcal{B}'$ has a transition $(s_1,(\gamma,b),s_2)$.

With the help of the last observation we can derive from [73] the upper complexity bound for WDA.

Theorem 19. *The non-emptiness problem for WDA is in* 2NExpTime*.*

Proof. In [73], it is shown that the non-emptiness problem for Profile Automata whose sets of constraint include only disjunctive key and disjunctive inclusion constraints, can be decided in non-deterministic doubly exponential time. By Observation 7, the non-emptiness problem for extended WDA without denial constraints can be polynomially reduced to the non-emptiness problem of Profile Automata without denial constraints. It, thus, only remains to show that the non-emptiness problem for WDA can be polynomially reduced to the non-emptiness problem for extended WDA without denial constraints.

To this end, let $\mathcal{A}=(\mathcal{B},\mathcal{C})$ be a WDA with $\mathcal{B}=(\Sigma,\Gamma,S,s_0,\delta,F)$. We will construct an equivalent extended $\mathcal{A}'=(\mathcal{B}',\mathcal{C}')$ without denial constraints. The idea is that for every γ occurring in some denial constraint in $\mathcal{C}$ and every class in which $\mathcal{B}$ outputs γ at some position, the automaton $\mathcal{B}'$ guesses exactly one γ-position and outputs some fresh symbol $\widehat{\gamma}$ instead of γ. That the $\widehat{\gamma}$-positions cover all γ-classes can be ensured by inclusion constraints. Then, each denial constraint $V(\gamma_1)\cap V(\gamma_2)=\emptyset$ in $\mathcal{C}$ can be replaced by a disjunctive key constraint $\mathtt{key}(\{\widehat{\gamma}_1,\widehat{\gamma}_2\})$ stating that no class contains two positions with symbols from $\{\widehat{\gamma}_1,\widehat{\gamma}_2\}$.

In the following, we explain the technical details. Let $\Gamma_{\mathtt{den}}=\{\gamma \mid \gamma$ occurs in some denial constraint in $\mathcal{C}\}$. The base automaton $\mathcal{B}'$ is defined as $(\Sigma,\Gamma',S,s_0,\delta',F)$ with $\Gamma'=\Gamma\cup\{\widehat{\gamma}\mid\gamma\in\Gamma_{\mathtt{den}}\}$ and $\delta'=\delta\cup\{(s_1,(\sigma,b),\widehat{\gamma},s_2)\mid(s_1,(\sigma,b),\gamma,s_2)\in\delta$ and $\gamma\in\Gamma_{\mathtt{den}}\}$. The constraint set $\mathcal{C}'$ results from $\mathcal{C}$ as follows:

- For every $\gamma\in\Gamma_{\mathtt{den}}$ we add the inclusion constraint $V(\gamma)\subseteq V(\widehat{\gamma})$.
- Every inclusion constraint $V(\gamma)\subseteq V(\Gamma')$ in $\mathcal{C}$ is replaced by the disjunctive inclusion constraint $V(\{\gamma\}\cup\{\widehat{\gamma'}\mid\gamma'\in\{\gamma\}\cap\Gamma_{\mathtt{den}}\})\subseteq V(\Gamma'\cup\{\widehat{\gamma'}\mid\gamma'\in\Gamma'\cap\Gamma_{\mathtt{den}}\})$.
- Every denial constraint $V(\gamma_1)\cap V(\gamma_2)=\emptyset$ in $\mathcal{C}$ is replaced by the disjunctive key constraint $\mathtt{key}(\{\widehat{\gamma_1},\widehat{\gamma_2}\})$.

Now, let w be a data word accepted by $(\mathcal{B},\mathcal{C})$. This means that there is a data word u which satisfies all constraints in $\mathcal{C}$ and results from w after $\mathcal{B}$ transforms the marked word projection of w. Let u' be the data word resulting from u after replacing for every $\gamma\in\Gamma_{\mathtt{den}}$ and every class word of u containing a γ-position, exactly one occurrence of γ by $\widehat{\gamma}$. By construction, the word projection of u' is a possible transduction of $\mathcal{B}'$ on the marked word projection of w. Furthermore, by construction of $\mathcal{C}'$, all constraints in $\mathcal{C}'$ are satisfied by u'. Thus, w is accepted by $(\mathcal{A}',\mathcal{C}')$.

The opposite direction is along the same lines. Let w be a data word accepted by $(\mathcal{B}',\mathcal{C}')$. Then, there is some data word u which satisfies all constraints in $\mathcal{C}'$ and results from w after $\mathcal{B}'$ converts the marked word projection of w. Let u' be the data word resulting from u by replacing all symbols $\widehat{\gamma}$ with $\gamma\in\Gamma_{\mathtt{den}}$ by γ. Obviously, the word projection of u' is a possible transduction of $\mathcal{B}$ on the marked word projection of w. As by construction all constraints in $\mathcal{C}$ are satisfied by u', the word w is accepted by $(\mathcal{A},\mathcal{C})$. ☐

We conclude by noting that the doubly exponential term in the complexity of the upper bound for Profile Automata given in [73], depends only on the alphabet size. The combination of this result with the translation in Lemma 8 delivers an upper complexity bound of 3NExpTime for the satisfiability for $\text{FO}^{\sim}_2(\texttt{Suc})$ which is worse than the bound in [170] (2NExpTime).

8.4 Weak Data Automata on Infinite Data Words

In this section, we will present a straightforward adaption of WDA to data ω-words. Moreover, we will give a logical characterization for this automata model by a simple extension of $\text{EMSO}^{\sim}_2(\texttt{Suc})$ on data ω-words. Finally, we will explain that all expressivity results from Section 8.2 easily carry over to infinite words.

A *Weak Büchi Data Automaton (WBDA)* is a tuple $(\mathcal{B}, \mathcal{C})$ where the base automaton $\mathcal{B}$ is a non-deterministic Büchi Letter-To-Letter Transducer over some input alphabet $\Sigma \times \{\bot, \top\}$ and some output alphabet Γ. Furthermore, $\mathcal{C}$ is a set of data constraints over Γ, defined in the same way as for WDA. The semantics of WDA is defined as expected. The extension of $\text{EMSO}^{\sim}_2(\texttt{Suc})$ on data ω-words is denoted by $\text{E}_\infty\text{MSO}^{\sim}_2(\texttt{Suc})$ and consists of all formulas of the form

$$\exists_\infty R_1 \ldots \exists_\infty R_m \exists S_1 \ldots \exists S_\ell \varphi$$

where $m, \ell \geq 0$ and $\varphi \in \text{FO}^{\sim}_2(\texttt{Suc})$. The semantics of $\text{E}_\infty\text{MSO}^{\sim}_2(\texttt{Suc})$ is defined in the same way as for $\text{EMSO}^{\sim}_2(\texttt{Suc})$ with the additional constraint that relation symbols quantified by $\exists_\infty$ have to be bound to infinite sets.

The generalization of Lemmas 7 and 8 to WBDA and $\text{E}_\infty\text{MSO}^{\sim}_2(\texttt{Suc})$ is an easy task:

Theorem 20. *Weak Büchi Data Automata and $\text{E}_\infty\text{MSO}^{\sim}_2(\texttt{Suc})$ are expressively equivalent.*

Proof. Compared to the translation in Lemma 7, the translation from WBDA to $\text{E}_\infty\text{MSO}^{\sim}_2(\texttt{Suc})$ uses an additional relation symbol quantified by $\exists_\infty$, in order to describe the acceptance condition of the base automaton. More precisely, a WBDA $\mathcal{A} = (\mathcal{B}, \mathcal{C})$ where F is the set of accepting states of $\mathcal{B}$, is converted into a formula

$$\exists_\infty R \exists R_{s_1} \ldots \exists R_{s_n} \exists R_{\gamma_1} \ldots \exists R_{\gamma_\ell} (\varphi^S_{\texttt{part}} \wedge \varphi^\Gamma_{\texttt{part}} \wedge \varphi_{\texttt{start}} \wedge \varphi_{\texttt{trans}} \wedge \varphi_{\texttt{accept}} \wedge \varphi_{\texttt{constr}})$$

where all sub-formulas are defined as in Lemma 7 except $\varphi_{\texttt{accept}}$ which is defined as

$$\forall x\ (R(x) \rightarrow \bigvee_{s \in F} R_s(x)).$$

The only basic difference of the translation in the opposite direction and the translation given in Lemma 8 is that the base automaton checks with the help of the Büchi condition that sets quantified with $\exists_\infty$ are indeed infinite. □

Let $\mathcal{L}^{\texttt{inf}}_{p<q}$ and $\mathcal{L}^{\texttt{inf}}_{p2q}$ be the languages of infinite data words satisfying, respectively, properties $E_{p<q}$ and E_{p2q} from Section 8.2.1. The proof of Lemma 6 can be turned into a proof that WBDA can neither decide $\mathcal{L}^{\texttt{inf}}_{p<q}$ nor $\mathcal{L}^{\texttt{inf}}_{p2q}$, because the words used in that proof can easily be turned into ω-words by padding them with infinitely many positions with a dummy symbol. Combining this with Theorem 20 and the fact that the $\text{FO}^{\sim}_2(\texttt{Suc}, <)$-formula given at the end of Section 8.2.2 describes $\mathcal{L}^{\texttt{inf}}_{p<q}$, we get that $\text{E}_\infty\text{MSO}^{\sim}_2(\texttt{Suc})$ is strictly less expressive than $\text{EMSO}^{\sim}_2(\texttt{Suc}, <)$.

Moreover, in the same way as an RA decides $\mathcal{L}_{p2q}$, a BRA can decide $\mathcal{L}^{\texttt{inf}}_{p2q}$. Furthermore, it follows from [124] that BRA cannot check the property that all data values of a data word are pairwise distinct. However, the job done by the WDA of Example 13 to check this property can also be done by a WBDA. Thus, WBDA and BRA are incomparable in terms of expressivity.

Additionally, as BRA can be converted into equivalent BDA [39, 41], the language $\mathcal{L}_{p2q}^{\texttt{inf}}$ can be decided by a BDA. As, moreover, the translation from WDA to DA in the proof of Theorem 17 works smoothly for WBDA and BDA, it follows that BDA are strictly more expressive than WBDA.

In [130] we show that the non-emptiness problem for WBDA can be polynomially reduced to the non-emptiness problem for WDA. The reduction is based on the insight that if a data ω-word is accepted by a WBDA, then there is a finite data word uv on which the base automaton has an accepting run looping over v. From the combination of this result with Theorem 19, it follows that the non-emptiness problem for WBDA is in 2NExpTime.

8.5 Discussion

We introduced and studied the automata model WDA on data words. We showed that the model is strictly less expressive than DA, incomparable with RA and logically characterized by $\text{EMSO}_2^{\sim}(\texttt{Suc})$. We followed from our results that $\text{FO}_2^{\sim}(\texttt{Suc})$ is strictly less expressive than $\text{FO}_2^{\sim}(\texttt{Suc}, <)$ which is a contrast to the equivalence of these logics on classical strings. Moreover, we showed that the non-emptiness problem for WDA is in 2NExpTime. We finally introduced WBDA, an extension of WDA to ω-words and showed that all expressivity and complexity results for WDA carry over to WBDA. We conclude with some open questions and the results of a recent work generalizing our results.

The precise complexity of WDA is still open. Recall that testing satisfiability of a $\text{FO}_2^{\sim}(\texttt{Suc})$-formula by translating it to a WDA and testing the latter for non-emptiness, results in an algorithm running in non-deterministic triply exponential time. With regard to the fact that satisfiability for $\text{FO}_2^{\sim}(\texttt{Suc})$ can be solved in 2NExpTime, it may be possible that the procedure translating $\text{FO}_2^{\sim}(\texttt{Suc})$-formulas to WDA or the procedure solving the non-emptiness problem for WDA can be improved complexity-wise.

We mentioned in Chapter 5 that one motivation for the design of an automata model corresponding to $\text{FO}_2^{\sim}(\texttt{Suc})$ comes from the area of verification of systems with unboundedly many processes. It would be interesting to experience whether WDA are indeed suitable for the usage in model checking procedures for $\text{FO}_2^{\sim}(\texttt{Suc})$.

In [202], the author asks for the genuine reason accounting for the elementary complexity of WDA. His answer is the commutativity of class conditions. To underpin his claim, he designs and investigates *Commutative Data Automata (CDA)* which are Data Automata where class conditions are restricted to commutative regular languages. Commutative Data Automata are strictly more expressive than WDA, since they can express that in each class of a data word, a particular proposition occurs an even number of times. They are strictly less expressive than Data Automata, since they cannot decide the language $\mathcal{L}_{p<q}$ from Section 8.2.1. It is shown that the upper bound for non-emptiness of CDA is indeed elementary, namely 3NExpTime. The author also gives a logical characterization for CDA in terms of Presburger logic. Finally, for a straightforward Büchi extension of CDA, it is shown that non-emptiness is in 4NExpTime.

> The results presented in this chapter originate from [130] which was a joint work with Thomas Schwentick and Tony Tan. The only difference to [130] is that I considered data words which at each position can carry multiple propositions, instead of a single symbol.

Part C

Models and Model Checking

This part of the work is devoted to formalisms describing systems with unboundedly many concurrent processes and the investigation of their model checking with data logics. In Chapter 9, we will give an overview on existing finite- and infinite-state models and mention important model checking techniques proposed in the literature. Our focus will be on models for concurrent systems. Some questions arising from the work done until now will be formulated in Chapter 10. These questions will concentrate on open problems with regard to *Dynamic Communicating Automata* defined in [47], *Process Register Automata* which are a restriction of a model introduced in [45] and *Branching High-Level Message Sequence Charts* which was designed by Benedikt Bollig in a joint work [46]. The first reason for the choice of these particular models is that they generate system traces which can be represented by structures with data values, particularly with data words. Moreover, each of the models provide a different view to the described systems. Dynamic Communicating Automata allow to look at the systems from the point of view of single processes, why they are often assigned to the category of *implementation models*. Process Register Automata provide a global view to the systems and are, therefore, considered as a *specification model* to be used in early design stages. While these two formalisms generate traces which impose a strict linear order on the executed actions, the structures generated by Branching High-Level Message Sequence Charts define only a partial order on actions.

In Chapter 11, we will define these three models formally and analyze their computational properties with respect to basic problems like non-emptiness, state reachability and executability. In the last chapter of this part, we will deal with the model checking problem of these models with regard to different data logics. While for Dynamic Communicating Automata and Process Register Automata we will use logics introduced in Parts A and B, in case of Branching High-Level Message Sequence Charts, we will introduce in Section 12.3.1 a logic which is similar to DNL, but suited for the navigation on partial orders.

Chapter 9

From Finite-State towards Infinite-State Model Checking - A Brief Review

The word *model checking* is a generic term for the design of algorithms testing whether a system model meets its formal specification. In this wide area of research, *finite-state model checking* builds a well-studied sub field [30, 155, 67, 156, 68, 199, 37]. A successful approach in the latter field is to model (concurrent) systems as transition systems, also called *Kripke structures*, with finitely many states. Each state of a Kripke structure contains some atomic propositions from a finite set which stand for relevant system properties in the corresponding situation of the system. The *computation tree* of a Kripke structure is a labelled tree which results from the "unraveling" of the Kripke structure at the initial state and where tree nodes represent states of the Kripke structure. Each node in the computation tree is labelled by the propositions of the corresponding state and the children of a node correspond to immediate follower states in the Kripke structure. Each path in the computation tree starting at the root is called a system *trace* and corresponds to a possible system run. The linear-time logic LTL and the branching-time logic CTL are two popular logics proposed for the formulation of system properties. While LTL-formulas are evaluated on single traces in computation trees, CTL-formulas can specify branching structures and are, therefore, interpreted on whole computation trees. Given a Kripke structure $\mathcal{K}$ and a formula φ specifying desired system properties, the *model checking problem* asks whether the model meets its specification, i.e., whether φ holds on the computation tree of $\mathcal{K}$. If φ is an LTL-like linear-time formula, the usual task is to check whether φ holds on all traces in the computation tree. If it is a formula of a branching-time logic like CTL, it is checked whether the formula is satisfied on the entire computation tree. Finite-state model checking has good computational properties which build the basis for successful applications of model checking algorithms in practically useful verification tools. For more details on the practical aspects of finite-state model checking we refer the reader to [69, 36, 91].

However, finite-state models often do not suffice to represent software or hardware systems adequately. For instance, if a system involves values from infinite domains and the properties to be checked refer to these value, there might not always be an adequate finite abstraction for the unbounded values. Possible sources for infinity are data structures like integers and unbounded stacks and channels. A further reason which gives rise to infinite-state models and is in the main focus of our studies in this part of the work, is the unboundedness of the number of processes involved in concurrent systems. In the following, we first give an overview of some models and model checking techniques for systems where the primary source of infinity are unbounded stacks, channels

or system values on which the systems operate. Then, we turn to systems with unboundedly many processes.

In several works, (classes of) systems of the first category are modeled by Relational Automata [60, 61], Timed Automata [20, 23], Pushdown Systems [58, 99, 49], Petri Nets [120, 121] and Lossy Channel Systems [7, 13]. In a recently published work [110], the authors aim to find a straightforward extension of the classical approach of LTL model checking described above to infinite domains. On the side of system models, the work introduces *abstract systems* which are extensions of Kripke structures where propositions are parameterized by variables. These variables can be assigned to values from infinite domains during system executions. Transitions of abstract systems are equipped with inequality conditions and reset actions which allow to set constraints on the values assigned to variables. On the specification side, the authors propose *Variable LTL*, an extension of usual LTL by propositions parameterized with variables which can be quantified universally and existentially. To give an example, assume that a system deals with an infinite set of message symbols. The Variable LTL-formula $\forall x \mathbf{G}(\mathtt{req}.x \rightarrow \mathtt{ack}.x)$ expresses that each request is not only followed by some acknowledgement, but also that the contents of the sent and received messages agree with each other. While model checking of abstract systems with Variable LTL is undecidable in general, it turns out that in the case where existential quantification is not allowed in the logic, the problem becomes PSPACE-complete, thus, not harder than LTL model checking in the classical case. In [111], the authors develop an automata-theoretic approach to model checking with Variable LTL and design some kind of generalized register automata into which abstract systems and formulas of Variable LTL can be converted.

In [86, 87], the authors model systems operating on infinite data by Minsky Machines with one counter (1-MCMs, for definition see Section 3.2.2) and consider their model checking with the data logics $\mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$ and $\mathrm{FO}^{\sim}(\mathtt{Suc}, <)$. The main results are that model checking of deterministic 1-MCMs with these logics is PSPACE-complete. For non-deterministic 1-MCMs, the problems are undecidable. An interesting further direction which arose in recent years and is also closely related to the topics studied in this work, is based on the approach of representing data structures like arrays and unbounded lists by data words and using register and data automata for the analysis of programs accessing these structures [19, 17, 18, 108].

An important specification model for systems with unbounded channels are *Message Sequence Charts (MSCs)* [179] which were standardized by the International Telecommunication Union (ITU) [119] and occur also in some modified version in UML where they are called *Sequence Diagrams*. Message Sequence Charts provide a convenient graphical representation and are used to describe communication protocols for finitely many processes communicating through unbounded channels. A single MSC describes a finite execution of a system by a partially ordered set of events caused by the involved processes. In its graphical representation, an MSC contains for each process a single vertical line modeling the lifetime of the process. Horizontal lines stand for message sending between processes. To describe infinite sets of MSCs, one can use formalism like automata or graphs where each transition or node is labelled by some MSC. Similar to finite automata describing regular languages, these formalisms allow to combine MSCs by choice, concatenation and repetition. The literature proposes several versions of such formalisms under different names like *Hierarchical MSCs* or *MSC Graphs* [22, 103, 113, 150, 166, 167]. In this work, we refer to all of them by the term *High-Level Message Sequence Charts (HMSCs)*. There are different approaches to check HMSCs against temporal properties. One approach is to describe the requirements to be checked as *template MSCs* or *template HMSCs* and to test whether a given template matches a given model [168, 165]. Loosely speaking, a template MSC or HMSC $\mathcal{T}$ is defined like a usual MSC or HMSC and it matches a model HMSC $\mathcal{H}$ if the events in $\mathcal{T}$ occur in $\mathcal{H}$ in the relative order as defined in $\mathcal{T}$ while allowing other events in between. Model checking by templates can be useful if the purpose is to test whether an unwanted execution occurs in the MSCs generated by $\mathcal{H}$. A second approach, which is for instance studied in [25], considers linearizations of the partially ordered events of MSCs. Given an HMSC $\mathcal{H}$ and a

property specified by a finite automaton or a linear-time temporal formula, it is checked whether the property holds on all linearizations of all MSCs generated by $\mathcal{H}$. Yet, another approach, which will be interesting for our studies in the following chapters, is to use languages like *Temporal Logic for Causalities (TLC)* [24] or *Propositional Dynamic Logic (PDL)* [48] whose formulas are interpreted directly on the partial order of MSCs why they are also called *structural logics* [173, 154, 153]. To put it simply, formulas of these logics are able to navigate along the process and message axes of MSCs, but cannot distinguish between different (order respecting) interleavings of the events of the same MSC. While MSCs and HMSCs are formalisms proposed for early system design stages, *Communicating Automata (CA)* [54], a further popular model for communication protocols, are seen as a formalism which is closer to the implementation phase. An important problem considered in the literature is the *realizability* of HMSCs, i.e., the question whether for a given HMSC one can construct a CA describing the same set of executions [22, 150, 105, 104, 114, 151, 164].

Now, we turn to model checking techniques for concurrent systems with unboundedly many processes. Systems where the number of processes is given as a parameter are called *parameterized systems* and verification methods for them are subsumed under the term *parameterized verification* [93, 33, 8]. We observe that many protocols like mutual exclusion or leader election algorithms presented in the literature are designed for an unbounded number of processes and it is expected that the protocols work correctly regardless the number of participating processes. In an early work, it was proven that the verification of parameterized systems with identical finite-state processes is in general undecidable [27]. This result motivated the search for classes of systems and verification problems where parameterized verification becomes decidable. One important research branch was the design of general frameworks which cover many parameterized systems and deliver decidability results for interesting verification questions. One such framework is that of Well-Structured Transition Systems (WSTS) [5, 98] which we briefly introduced in Section 3.2.3. Successful applications can be found in [77, 78, 2] where decidability for the *state reachability* problem for classes of ad-hoc networks of homogenous communicating processes is shown. State reachability is concerned with the question whether there is a number of processes organized in some communication topology such that after some time the system enters a situation in which one of the processes is in some certain (unwanted) state. Many safety problems for parameterized systems can be reduced to state reachability. Another widely used framework is called *Regular Model Checking* [201, 133, 52, 12]. In this framework, a configuration involving an unbounded number of finite-state processes is modeled as a word-, ring- or tree-like structure over a finite alphabet. This view makes it possible to represent infinite sets of configurations by regular languages of these structures. To simulate transitions on configurations, regular transducers are used. The computation of (an approximation of) the set of all reachable configurations is carried out mostly by computing the transitive closure of the transducers. The most considered model checking tasks in this framework are tests for safety and liveness properties which again reduce to the computation of reachable configurations. Applications on parameterized systems with linear or ring-formed topologies can be found in [122, 6] and on those with tree-like topologies in [14, 9, 51]. Algorithms based on the computation of the transitive closure of transducers are developed in [122, 10, 11].

A further framework in the context of parameterized systems are *Population Protocols* [26, 29, 64]. Originally motivated by mobile sensor networks and collections of molecules undergoing chemical reactions, these protocols serve as a model for large collections of tiny identical finite-state devices which are interacting with each other in order to carry out computations. The computational power of Population Protocols depends less on the computational power of individual agents, but rather on the synergy of interaction. In the original model, the agents cannot send any messages and do not share memory. An interaction between two agents just leads to a change of the states of the agents. The question which agents may interact with each other is answered by an interaction graph, usually modeling distances between agents. The input to a population protocol is distributed across the initial states of the agents. Moreover, each state of an agent is related to some output

symbol. The mostly considered problem on Population Protocols is whether for some input spread over initial states, the outputs of the agents converge to a correct answer. For the basic model of Population Protocols it is shown that the predicates computable on input values are exactly the semi-linear predicates. In [34, 112, 190], stronger self-stabilizing models are considered which can deal with failures. An extended model, called *Mediated Population Protocols*, in which interaction links between agents are enriched by memories, is introduced in [63]. It turns out that the latter model is strictly more expressive than the classical one and is able to compute interesting properties with regard to maximal matchings and transitive closures of the underlying interaction graphs.

Another model for concurrent systems with unboundedly many processes are *Data Multi-Pushdown Automata (DMPA)* which were introduced in [45]. These are Register Automata equipped with finitely many stacks and, similar to HMSCs, used to describe the "global" behaviour of concurrent systems. A DMPA can dynamically create new processes and store them in its registers and stacks. To preserve decidability of the model, the access to stacks underlies some restrictions. Nevertheless, the authors show that the model is expressive enough to describe token-based leader election protocols (see, e.g., in [152]) for unboundedly many processes. In contrast to HMSCs, the runs of DMPA generate traces with a strict linear order. These traces are modeled by data words. Similar to the model checking approach explained in the finite case, the authors use data logics for the formulation of requirements on system traces. It is shown that model checking against full $\mathrm{MSO}^{\sim}$ is decidable.

Further models for systems where the number of processes is not fixed a priori, but grows dynamically during system runs are introduced in [145] and [47]. In [145], the authors define *MSC-Grammars* which, as the name already indicates, generate sets of MSCs over an unbounded number of processes. They show that model checking against MSO-formulas describing structural properties is decidable. In [47], the authors introduce *Dynamic Communicating Automata (DCA)* which are basically classical CA extended by the ability to create new processes. Following the studies on HMSCs for finitely many processes, the main question considered in [47] is the realizability of MSC-Grammars by DCA.

Chapter 10

Motivating Questions on System Models and Model Checking

Local control of behaviour: Dynamic Communicating Automata

In the last chapter, we talked about Communicating Automata (CA) [54] which are a well-known computational model for systems with a finite number of processes communicating asynchronously through unbounded FIFO-channels. As mentioned before, one research branch deals with the realizability of High-Level Message Sequence Charts (HMSCs) by CA [22, 15, 114]. Communicating Automata also appear as a model for wireless ad-hoc networks [187, 186, 77, 78, 76, 2]. Usually, an ad-hoc network consists of a fixed number of processes performing rendezvous-based communication which means that messages are delivered directly from sender to receiver without any intermediate storage. Even though there are works considering settings where the communication topology between processes can change dynamically during system executions (see, e.g., in [187, 77]), in most of the settings the number of processes as well as the communication topology are a priori fixed. An intensively studied problem in the realm of ad-hoc networks is the parameterized control state reachability problem. For a given process description (in form of a communicating automaton) and a process state `target`, this problem asks whether there is an arbitrary number of processes and an arbitrary communication topology such that the network reaches a situation where at least one of the processes is in state `target`.

Just like in the areas of HMSCs and ad-hoc networks mentioned above, many frameworks using CA restrict to a fixed number of processes. This makes it difficult to apply CA in areas like mobile computing where the number of interacting processes is not known in advance. Due to this shortcoming of the classical CA-model, the authors in [47] introduced *Dynamic Communicating Automata (DCA)* which extend CA by the ability to spawn processes. In a network induced by a DCA $\mathcal{A}$, all processes behave according to $\mathcal{A}$. Each process possesses a unique process ID within the network and is, similar to Register Automata introduced in Section 4.2.1, equipped with finitely many registers in which IDs of other processes can be stored. Starting from an initial network configuration consisting of a single process, each process can create new processes and send messages to processes whose IDs are stored in its registers. Besides sending message symbols from a finite alphabet like in the case of classical CA, a process can also send IDs from its registers. Processes receiving messages can store incoming IDs in their registers. Thus, the number of processes within the network as well as the communication structure is flexible and can change during system executions. Since DCA describe the behaviour of single processes and allow to look at the whole network from the point of view of a process, they are considered as an implementation model suitable for late stages of system design. Hence, following the classical approach for CA and HMSCs, the authors

in [47] deal with the realizability of MSC-grammars by DCA, that is, the question whether for a given MSC-grammar there is a DCA which generates the same set of executions.

In this work, we are interested in the computational properties of DCA which were neither considered in [47], nor in any other follow-up paper. In particular, our focus is on the verification and model checking of DCA. Due to the tight links between non-emptiness and model checking, we first consider the non-emptiness problem. In [47], DCA are equipped with accepting states. Hence, the non-emptiness problem for DCA asks whether there is at least one system execution which reaches a configuration where all processes are in accepting states. The second direction attracting our attention is the state reachability question for DCA. As argued in [77, 78], the state reachability problem covers many interesting properties, like safety conditions, arising in ad-hoc networks. This suggests to transfer the studies on ad-hoc networks with classical CA to DCA-networks and to investigate the state reachability problem in the latter framework. Finally, we are interested in the model checking of DCA with data logics. In [47], the authors define a trace of a DCA-system as a finite sequence of actions leading to an accepting configuration. Each position of the trace carries the IDs of the processes involved in the corresponding action. Hence, traces of DCA can be seen as data words in which propositions indicate executed actions and data values represent process IDs. This view makes DCA quite suitable for the framework of model checking with data logics. In analogy to the classical finite-state model checking with LTL, our aim is to use DCA as a specification model for systems and to use data logics as a specification language for system requirements. Then, the model checking question that has to be considered is as follows: Given a DCA and a formula of some suitable data logic, does the formula hold on all traces of the DCA?

In Section 11.2, we introduce the version of DCA which we take as a basis for our investigations in this work. The basic difference to the model in [47] is that in our definition the communication is not asynchronous, but based on rendezvous without intermediate storage. One reason why we choose such a definition is that we want to keep our initial investigations on DCA simple. Indeed, it is well-known that even for networks with finitely many processes communicating through unbounded perfect channels, many interesting verification problems are undecidable [54]. Moreover, since our version of DCA is close to the model used in classical ad-hoc networks, it is easier to transfer proof concepts from that area.

In Section 11.2.1, we study the non-emptiness problem for DCA. We first show that this problem is undecidable, even in the case where processes have only one register. Then, we turn our attention to *selective* DCA, i.e., DCA where in each send action not only the sender has to know the ID of the receiver, but also vice-versa. While non-emptiness remains undecidable for selective DCA with two registers, we get decidability in the case where we restrict to one register. We prove that in the latter case the problem is solvable in PTime.

Control state reachability is considered in Section 11.2.2. This section also starts with bad news: Control state reachability is not decidable for DCA, even in the case of one register. In oder to find decidable restrictions, we follow here a different direction than in the case of non-emptiness. Inspired by recent works on the verification of ad-hoc networks [77, 2], we focus on the analysis of DCA where actions are only allowed if they lead to network configurations in which the maximum length of simple paths in the induced communication graph is bounded by a given natural number. Unfortunately, even in the case where all (un-)directed simple paths in communication graphs are bounded by some constant, state reachability remains undecidable (in contrast to results in ad-hoc networks [77, 78, 76, 2]). Then, inspired by lossy channel systems [7, 13], we consider *degenerative DCA* where every processes can loose non-deterministically register inputs. This kind of DCA can be used to model unexpected loss of communication links in mobile ad-hoc networks. While reachability for degenerative DCA is in general undecidable, we show that the problem becomes decidable if all allowed configurations are strongly bounded. We close our considerations on state reachability by summarizing our results from [4] on *buffered DCA* which, in terms of communication, are closer to

the original DCA-model from [47]. Each process described by a buffered DCA is equipped with a FIFO-mailbox so that communication is carried out asynchronously.

In Section 12.1, we consider the model checking of DCA with data logics introduced in previous parts of this work. We concentrate on fragments of Basic Data LTL (B-DLTL) and Freeze LTL ($\mathrm{LTL}^{\Downarrow}$). It is easy to see that all fragments of DCA with an undecidable non-emptiness problem must also have an undecidable model checking problem with these logics. As our results on non-emptiness do not leave many choices, we consider selective 1-register DCA for model checking. On the logic side, we first consider a restriction of B-DLTL where all shift values ℓ in formulas of the form $\mathcal{C}^{\ell}_{@\mathtt{a}}\psi$ equal 0. We show in Section 12.1.1 that the model checking problem for selective 1-register DCA is decidable for this fragment of B-DLTL. We assume that our decision procedure can be easily extended such that it covers full B-DLTL. However, this question as well as the question whether model checking with full Extended Data Navigation Logic is decidable remain open. We show in Section 12.1.2 that model checking with $\mathrm{LTL}^{\Downarrow}_1(\mathbf{X}, \mathbf{U})$, i.e., the future fragment of $\mathrm{LTL}^{\Downarrow}$ with a single freeze register, is undecidable.

These results indicate that DCA have hard decision problems. Our model checking results, moreover, illustrate how unlike the finite-state setting, model checking with a decidable logic (in this case $\mathrm{LTL}^{\Downarrow}_1(\mathbf{X}, \mathbf{U})$) can become undecidable if the interplay between system model and logic allows to encode undecidable problems.

Global control of behaviour: Process Register Automata

In the last chapter, we mentioned Data Multi-Pushdown Automata (DMPA) which were introduced in [45] as a model whose traces are data words with multiple data values at each position. A DMPA is a finite automaton equipped with finitely many registers and stacks to store data values. Each action of a DMPA can use, on the one hand, data values from registers and stacks, and, on the other hand, values which are fresh with respect to the whole run leading to the current action. Furthermore, an action can update register contents and push data values to the stacks. The data word resulting from a run carries at each position, besides a symbol associated with the corresponding action in the run, also the data values used by the action. Equipped with mechanisms to store and create new data values, DMPA are proposed as a suitable formalism to model concurrent systems with dynamic process creation [45]. In this context, DMPA provide, unlike DCA, a more global view to systems. Therefore, they are considered by the authors as a model to be used in early design stages. In the mentioned work, the authors investigate the model checking problem for DMPA with $\mathrm{MSO}^{\sim}$. By a reduction to the satisfiability of classical MSO over nested words [139], they show that model checking is decidable for DMPA where in each run the number of switches between stacks is bounded by some constant. In contrast to our model checking results for DCA, this result shows that model checking with an undecidable logic like $\mathrm{MSO}^{\sim}$ can be decidable when the structures, generated by the underlying system model, are restrictive enough.

The authors in [45] do not give any complexity bounds for the model checking problem for DMPA and $\mathrm{MSO}^{\sim}$. However, we can conclude from [139] that the problem has non-elementary complexity. Our primary question is whether there are expressive fragments of $\mathrm{MSO}^{\sim}$ for which model checking of DMPA delivers good complexity results. Furthermore, can we determine the borders after which the problem becomes non-elementary? Like in the case of DCA, we cannot give complete answers, but provide some first insights. As a starting point, we investigate *Process Register Automata (PRA)*, the fragment of DMPA which does not contain any stacks. On the logic side, we consider fragments of $\mathrm{LTL}^{\Downarrow}$ and $\mathrm{HTL}^{\sim}$ introduced in the first two parts of this work.

We give in Section 11.3 the formal definition of PRA and show in Section 11.3.1 that its non-emptiness problem is NP-complete. In Section 12.2.1, we prove that the model checking of PRA with $\mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$ can be solved in ExpSpace. Although we cannot show that this upper bound is tight, we conclude from our decision procedure that for every $k \geq 1$, the model checking problem

with $\mathrm{LTL}^{\Downarrow}_{k}(\mathbf{X},\mathbf{U})$ is PSPACE-complete, thus, not harder than satisfiability for LTL. In Section 12.2.2, we turn to model checking with $\mathrm{HTL}^{\sim}$. Here, things get more complicated. While for $\mathrm{HTL}^{\sim}_{1}$, i.e., $\mathrm{HTL}^{\sim}$ with only one variable, the problem is EXPSPACE-complete, it becomes already non-elementary as soon as a second variable is added. The last results raise hopes that $\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$ and HTL_1 can deliver elementary complexity for the model checking of DMPA.

From linear to partial orders: Branching High-Level Message Sequence Charts

An execution of a concurrent system can contain simultaneous actions of distinct processes. Thus, from a temporal point of view, actions are basically partially ordered. A classical approach is to represent traces of concurrent systems as *linearizations* of actions respecting the partial order. The two models DCA and PRA, which we explained in the previous paragraphs, are based on this approach. Each trace of a system execution represents exactly one linearization of the underlying partial order. A second well-known approach is to model system traces as partially ordered structures. As mentioned in the last chapter, a popular formalism in this context are MSCs [119, 179] which come along with a convenient graphical representation. A single MSC describes a single execution of a system with finitely many concurrent processes. The literature proposes different versions of High-Level Message Sequence Charts (HMSCs) in forms of automata and graphs which allow to describe infinite sets of MSCs [22, 103, 113, 150, 166, 167]. In the last chapter, we already gave an overview of different model checking techniques for HMSCs. Here, we would like to emphasize some specific approaches. An important question considered for HMSCs is their realizability by Communicating Automata (CA), i.e., the question whether for a given HMSC, there is a CA describing the same set of executions. The model checking of HMSCs by logics is divided into two approaches. The first one uses logics on linear structures and investigates the question whether all linearizations of all MSCs, generated by a given HMSC, satisfy a given formula. The second approach works with so-called structural logics whose formulas cannot distinguish between different linearizations of the same MSC. It turns out that, compared to the first approach, the second one delivers quite good decidability and complexity results [25, 22, 48, 173, 153, 154].

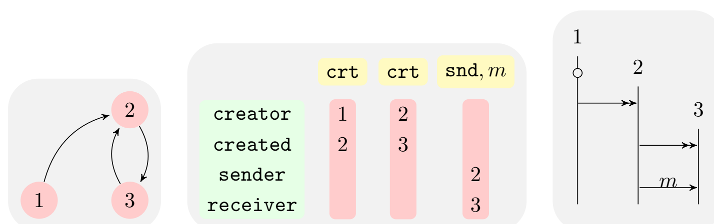
In [46], Benedikt Bollig designed Branching High-Level Message Sequence Charts (BHMSCs) which, similar to the generalization of CA to DCA, extend HMSCs by process creation. Just like DCA, BHMSCs use finitely many registers to store process IDs and are based on branching automata [148, 149]. Following the lines of the studies on classical HMSCs, we investigate the non-emptiness of BHMSCs, their realizability by DCA and their model checking by appropriate structural logics.

We first show that the non-emptiness problem is EXPTIME-complete. It follows from known results that realizability is not decidable for BHMSCs, even not in the case of finitely many processes [22, 114]. Therefore, we introduce and study the notion of executability for BHMSCs which is a necessary condition for realizability. Informally, the executability problem asks whether in each MSC, generated by a given BHMSCs, every sending process is aware of the ID of the receiver at the time of communication. We prove that also the executability problem is EXPTIME-complete. Moreover, we design a CTL-like logic called *MSC Navigation Logic (MNL)*. Similar to Data Navigation Logic considered in Chapter 6, it allows to navigate along the actions of a single process. Moreover, like in Propositional Dynamic Logic [48] and Temporal Logic of Causalities [24, 173], formulas can distinguish between process axes, on the one hand, and message and create axes, on the other hand. Furthermore, the logic allows quantification and navigation over paths in MSCs. We prove that model checking of BHMSCs with MNL is as hard as non-emptiness and executability for this model, namely EXPTIME-complete.

In Section 11.4, we give the formal definition of BHMSCs and consider non-emptiness and executability. The definition of MNL and the study of the model checking problem with this logic can be found in Section 12.3.

Chapter 11

Three Models - Three Views



In this chapter, we will introduce three models for concurrent systems with unboundedly many processes and study their basic computational properties with respect to decision problems like non-emptiness, reachability and executability. In all three models, processes are equipped with unique process IDs and are able to spawn new processes. Moreover, they can send messages to each other consisting of message symbols and IDs. The first model, called Dynamic Communicating Automata and originally introduced in [47], is used to describe the behaviour of a single process within a network. Therefore, it is usually considered as a model to be used in design phases which are close to implementation. The second model, namely Process Register Automata, is a restriction of Data Multi-Pushdown Automata [45], provides a more global view to the designed systems and abstracts from implementation details. While the first two models generate linear traces which will be represented by usual data words, the structures generated by the third model, called Branching High-Level Message Sequence Charts [46], are MSCs which are based on partial orders on actions. Before defining these modes formally, we will prepare in the next section a repertoire of useful notions and notations.

11.1 Notational Conventions

A *message alphabet* A is a finite set of symbols such that every symbol $m \in A$ has an arity $\mathtt{ar}(m) \in \mathbb{N}_0$. Given a set N of *process names* and a message alphabet A, we denote by $A(N)$ the set of all messages of the form $m(n_1, \ldots, n_{\mathtt{ar}(m)})$ with $m \in A$ and $n_1, \ldots, n_{\mathtt{ar}(m)} \in N$. By $\mathtt{Actions}(A, N)$, we denote the set of all *create actions* $\mathtt{crt}(n, n')$ and *send actions* $\mathtt{snd}(n, n', \mathtt{msg})$ such that $n, n' \in N$ and $\mathtt{msg} \in A(N)$. Informally, $\mathtt{crt}(n, n')$ means that process n creates a process n', and $\mathtt{snd}(n, n', \mathtt{msg})$ stands for the sending of the message $\mathtt{msg}$ from n to n'. For create actions $\mathtt{act} = \mathtt{crt}(n, n')$, we define the two parameters $\mathtt{creator}$ and $\mathtt{created}$ with $\mathtt{creator}(\mathtt{act}) = n$ and $\mathtt{created}(\mathtt{act}) = n'$. A send action $\mathtt{act} = \mathtt{snd}(n, n', m(n_1, \ldots, n_{\mathtt{ar}(m)}))$ has the parameters $\mathtt{sender}$, $\mathtt{receiver}$, $\mathtt{msym}$ and $\mathtt{mpar}_1, \ldots, \mathtt{mpar}_{\mathtt{ar}(m)}$ with $\mathtt{sender}(\mathtt{act}) = n$, $\mathtt{receiver}(\mathtt{act}) = n'$, $\mathtt{msym}(\mathtt{act}) = m$ and $\mathtt{mpar}_i(\mathtt{act}) = n_i$ for every $i \in \{1, \ldots, \mathtt{ar}(m)\}$. Other parameters are not

defined for actions.

We fix an infinite supply $\mathbb{P}$ of *process IDs*. For convenience, we often represent process IDs by natural numbers. The set N in $A(N)$ and $\mathtt{Actions}(A, N)$ will often be instantiated by the set $\mathbb{P}$. In case of $N = \mathbb{P}$, the messages in $A(N)$ and the actions in $\mathtt{Actions}(A, N)$ are usually called *concrete* messages and *concrete* actions, respectively. Otherwise, they are called *symbolic* messages and actions. In the next sections, we will represent traces of Dynamic Communicating Automata and Process Register Automata by data words. These traces result from sequences of concrete actions. We define a *data word representation* for such actions. Let $\mathtt{Actions}(A, \mathbb{P})$ be a set of concrete actions for some message alphabet A. Each action $\mathtt{act}$ in this set is represented by a data word position $\mathtt{dwrep}(\mathtt{act})$ over the proposition set $\mathtt{Prop}^A_{\mathtt{act}} = \{\mathtt{crt}, \mathtt{snd}\} \cup A$ and the attribute set $\mathtt{Attr}^A_{\mathtt{act}} = \{\mathtt{creator}, \mathtt{created}, \mathtt{sender}, \mathtt{receiver}\} \cup \{\mathtt{mpar}_1, \dots, \mathtt{mpar}_a\}$ where a is the highest arity assigned to a symbol in A. Figure 11.1 demonstrates how two actions from $\mathtt{Actions}(A, \mathbb{P})$ for some message alphabet A are represented by data word positions in the case that the highest arity in A is 2. We de-

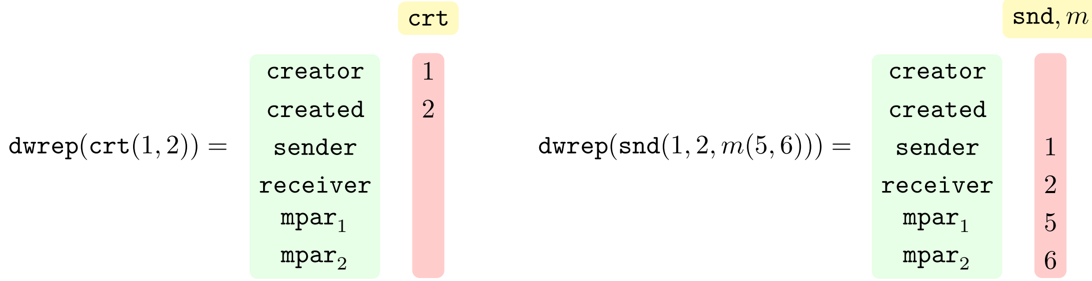


Figure 11.1: Representing actions by data word positions

fine the data word representation $\mathtt{dwrep}(\mathtt{act})$ of a concrete $\mathtt{act}$ formally as follows: If $\mathtt{act}$ is a create action of the form $\mathtt{crt}(p, p')$, then, $\mathtt{dwrep}(\mathtt{act}) = \langle\{\mathtt{crt}\}, \{\mathtt{creator} \mapsto p, \mathtt{created} \mapsto p'\}\rangle$. If $\mathtt{act}$ is a send action of the form $\mathtt{snd}(p, p', m(p_1, \dots, p_{\mathtt{ar}(m)}))$, then, $\mathtt{dwrep}(\mathtt{act}) = \langle\{\mathtt{send}, m\}, \{\mathtt{sender} \mapsto p, \mathtt{receiver} \mapsto p', \mathtt{mpar}_1 \mapsto p_1, \dots, \mathtt{mpar}_{\mathtt{ar}(m)} \mapsto p_{\mathtt{ar}}(m)\}\rangle$.

We close this section by the definition of *transition systems*. A transition system $\mathcal{T}$ is a triple $(\mathtt{Conf}, \mathtt{Conf}_{\mathtt{init}}, \longrightarrow, \mathtt{Conf}_{\mathtt{acc}})$, where $\mathtt{Conf}$ is a (possibly infinite) set of *configurations*, $\mathtt{Conf}_{\mathtt{init}} \subseteq \mathtt{Conf}$ is a set of *initial* configurations, $\longrightarrow \subseteq \mathtt{Conf} \times \mathtt{Conf}$ is a *transition relation* and $\mathtt{Conf}_{\mathtt{acc}} \subseteq \mathtt{Conf}$ is a set of *accepting* configurations. For $i \in \mathbb{N}_0$, we use $\longrightarrow^i$ to denote the i-times composition of $\longrightarrow$. We use $\longrightarrow^*$ to denote the reflexive and transitive closure of $\longrightarrow$. In cases where the set of accepting configurations of a transition system does not play any role for our considerations (for instance, in the context of reachability questions), we usually skip this set in the formal description of the transition system.

11.2 Dynamic Communicating Automata

In this section, we introduce Dynamic Communicating Automata (DCA) and analyze the non-emptiness and the reachability problem for this model. This formalism was originally defined in [47] as an implementation model for MSC-Grammars. Since our investigations in this work are first steps with regard to the verification of DCA, we here define a more simplified version of DCA. While in the original model, processes communicate asynchronously through unbounded channels and messages can contain multiple process IDs, our DCA-version is based on rendezvous communication and messages are restricted to at most one ID.

A DCA is a finite automaton which is equipped with finitely many registers and describes the behaviour of processes within a dynamic network. Each process can perform a *local* action that changes its current state. It can also spawn new processes and store their IDs in its registers. Moreover, it can send messages to processes whose IDs are stored in its registers. Similar to our introductory example in Chapter 2, the type of communication is rendezvous-based, i.e., messages are not stored intermediately, but delivered directly from sender to receiver. A message contains a symbol from a finite message alphabet and possibly a process ID which can be the ID of the sending process or an ID stored in one of its registers. A receiving process can store incoming IDs in its registers. Thus, the number of processes as well as the communication topology are not fixed, but change dynamically. In the sequel, we will first give the formal syntax of DCA and illustrate their semantics through an example. Then, we will describe the semantics formally and define some computational problems for DCA.

Formally, a DCA $\mathcal{A} = (A, R, S, s_0, \delta, F)$ consists of a finite *message alphabet* A in which the arity of message symbols is at most one, a finite set R of *registers*, a finite set S of *states*, an *initial state* $s_0 \in S$, a set $F \subseteq S$ of *accepting* states and a finite set δ of *transitions* of the form (s_1, a, s_2) where $s_1, s_2 \in S$. For each form the argument a can take, we give its informal interpretation:

- $a = \lambda$: The process performs a local action.
- $a = r \leftarrowtail \mathtt{crt}(s, r')$ for $r, r' \in R$ and $s \in S$: The process creates a new process with a fresh ID in state s. The ID of the new process is stored in register r of the creating process and the ID of the creating process is stored in register r' of the new process.
- $a = \mathtt{snd}(r, m)$ for $r \in R$ and $m \in A$ with $\mathtt{ar}(m) = 0$: The process sends message symbol m to the process whose ID is stored in register r.
- $a = \mathtt{snd}(r, m(r'))$ for $r \in R$, $r' \in R \cup \{\mathtt{self}\}$ and $m \in A$ with $\mathtt{ar}(m) = 1$: The process sends a message to the process whose ID is stored in register r. The message contains symbol m and either the ID contained in register r' of the sending process or the ID of the sending process itself ($\mathtt{self}$).
- $a = \mathtt{rcv}(r, m)$ for $r \in R$ and $m \in A$ with $\mathtt{ar}(m) = 0$: The process receives message symbol m from the process whose ID is stored in register r. As the process from which the message comes is determined by the input of register r, such actions are called *selective symbol reception*.
- $a = \mathtt{rcv}(\star, m)$ for $m \in A$ with $\mathtt{ar}(m) = 0$: The process receives message symbol m from some other process. We call such actions *non-selective symbol reception*.
- $a = \mathtt{rcv}(r, m(r'))$ for $r, r' \in R$ and $m \in A$ with $\mathtt{ar}(m) = 1$: The process receives a message from the process whose ID is stored in register r. The message contains message symbol m along with an ID. In case that the incoming ID is not the ID of the receiver, it is stored in register r'. Such actions are called *selective ID reception*.
- $a = \mathtt{rcv}(\star, m(r'))$ for $r' \in R$ and $m \in A$ with $\mathtt{ar}(m) = 1$: The process receives a message from some other process. The message contains message symbol m along with an ID to be stored in register r'. We refer to such actions as *non-selective ID reception*.
- $a = \mathtt{res}(r)$ for $r \in R$: The process resets its register r so that it becomes empty.

Before presenting the formal semantics of DCA, we illustrate the semantics through an example DCA:

Example 16. We design a DCA implementing the client-and-server protocol from our introductory example in Chapter 2. The DCA uses two registers r_1 and r_2, two message symbols `serv` and `req` of arity 1 and one message symbol `ack` of arity 0. The symbol `serv` is used by the root process to inform clients about the server ID. By means of the symbol `req`, client processes send to the server requests along with their own IDs. The symbol `ack` represents acknowledgements from server to clients. The graphical representation of the DCA is given in Figure 11.2. The sub automaton

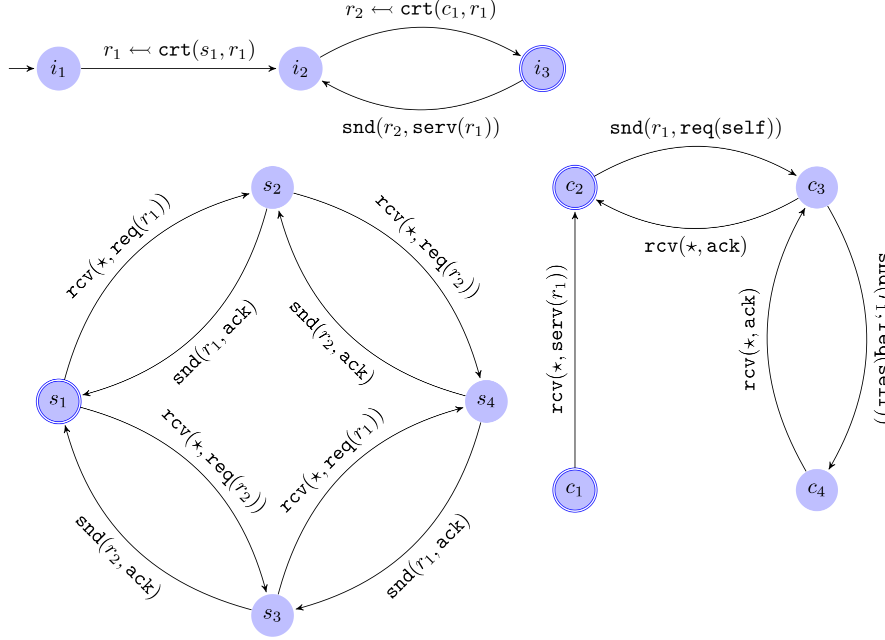


Figure 11.2: An example DCA implementating the client-and-server protocol from Chapter 2

consisting of the states i_1 to i_3 describes the behaviour of the initial root process. The part consisting of states s_1 to s_4 belongs to the server process. The part which consists of states c_1 to c_4 describe client processes. Note that the initial state is marked by an incoming sourceless arrow. Accepting states have surrounding circles. The root process first creates a server process starting in state s_1 and stores the ID of the server in its first register r_1. The ID of the root process is stored in register r_1 of the server process. Then, arbitrarily often the root process creates a new client, stores its ID in register r_2 and sends him the ID of the server along with the message symbol `serv`. Each client stores the received server ID in its register r_1. Holding the server ID, a client can send arbitrarily often a request to the server along with its own ID. Observe that in the designed DCA, at any time, each client can have at most two open requests. After being created by the root process, the server can receive requests associated with the IDs of the senders. In order not to lose the connection to the sending clients, the server stores the incoming sender IDs in its registers. Using these IDs, it acknowledges the requests of the clients. Similar to clients, a server can never have more than two open requests. □

Configurations and their graph encodings

A *configuration* $c = (P, \mathbf{s}, \mathbf{r})$ of a DCA $\mathcal{A} = (A, R, S, s_0, \delta, F)$ is a tuple where $P \subseteq \mathbb{P}$ is a finite set of processes, $\mathbf{s} \in [\mathbb{P} \rightharpoonup S]$ with $\mathtt{dom}(\mathbf{s}) = P$ maps each process $p \in P$ to its current state and

$\mathbf{r} \in [\mathbb{P} \rightharpoonup [R \rightharpoonup P]]$ with $\mathtt{dom}(\mathbf{r}) = P$ maps every process $p \in P$ to its register contents. For two processes $p_1, p_2 \in P$ and $r \in R$, $\mathbf{r}(p_1)(r) = p_2$ means that register r of p_1 contains the ID of p_2. If $\mathbf{r}(p_1)(r)$ is not defined, then register r of p_1 is empty. We use $s \in c$ to denote that there exists a process $p \in \mathbb{P}$ in c such that $\mathbf{s}(p) = s$. The set of all configurations of $\mathcal{A}$ is denoted by $\mathtt{Conf}(\mathcal{A})$.

Given a configuration $c = (P, \mathbf{s}, \mathbf{r})$ and a set $P' \subseteq P$, the sub configuration of c induced by P' is defined as $c' = (P', \mathbf{s}', \mathbf{r}')$ where $\mathbf{s}'$ results from $\mathbf{s}$ by restricting the domain of $\mathbf{s}$ to P' and $\mathbf{r}'$ results from $\mathbf{r}$ by restricting the the domain of $\mathbf{r}$ to P' and setting $\mathbf{r}'(p)(r) = \perp$ for every $p \in P'$, $r \in R$ and $\mathbf{r}(p)(r) \notin P'$. Two configurations $c_1 = (P_1, \mathbf{s}_1, \mathbf{r}_1)$ and $c_2 = (P_2, \mathbf{s}_2, \mathbf{r}_2)$ are called *isomorphic* if one of them can be obtained from the other by process renaming. That is, there is a bijection $b : P_1 \mapsto P_2$ such that (i) for every process $p \in P_1$, we have $\mathbf{s}(p) = \mathbf{s}(b(p))$, and (ii) for all processes $p_1, p_2 \in P_1$ and all registers $r \in R$, we have $\mathbf{r}(p_1)(r) = p_2$ if and only if $\mathbf{r}(b(p_1))(r) = b(p_2)$.

In our proofs in the next sections it will sometimes be useful to work with graph encodings of configurations. Before defining such encodings, we introduce some notions on graphs. A *labelled directed graph* G is a tuple $(V, \Sigma_v, \Sigma_e, \lambda, E)$ where V is a finite set of vertices, Σ_v is a set of vertex labels, Σ_e is a set of edge labels, $\lambda : V \to \Sigma_v$ is a vertex labelling function, and $E \subseteq V \times \Sigma_e \times V$ is a set of labelled edges. A *path* in G is a finite sequence $\pi = v_1 v_2 \ldots v_k$ of vertices where for every i with $1 \leq i < k$, there is an $a \in \Sigma_e$ such that $(v_i, a, v_{i+1}) \in E$. We say that π is *simple* if all vertices in π are pairwise different. The length $\mathtt{length}(\pi)$ of π is defined as $k - 1$. We set the *diameter* $\mathtt{diameter}(G)$ of G as the largest k such that there is a simple path π in G with $\mathtt{length}(\pi) = k$. We also consider *node-labelled directed* and *node-labelled undirected graphs* where edge labels are skipped. In the first kind of structures, an edge is described by an ordered pair of nodes and in the latter one, as a set of two distinct nodes. The notions of simple path and diameter are adapted in a straightforward way to node-labelled directed and undirected graphs.

In the graph encoding of a configuration, every process is represented by a vertex labelled with the state of the process. Furthermore, there is an edge from vertex u to vertex v labelled with $r \in R$ if the process corresponding to u has the ID of the process corresponding to v in its register r. Formally, the encoding of a configuration $c = (P, \mathbf{s}, \mathbf{r})$ is defined as the labelled directed graph $enc(c) = (P, S, R, \mathbf{s}, E = \{(p, r, q) \mid \mathbf{r}(p)(r) = q\}$. For the sake of simplicity, we will often skip node and edge labels in figures depicting graph encodings.

The transition relation on configurations

We define a transition relation $\rightarrow_{\mathcal{A}}$ on the set $\mathtt{Conf}(\mathcal{A})$ of configurations of $\mathcal{A}$. Given two configurations $c, c' \in \mathtt{Conf}(\mathcal{A})$ with $c = (P, \mathbf{s}, \mathbf{r})$ and $c' = (P', \mathbf{s}', \mathbf{r}')$, we have $c \rightarrow_{\mathcal{A}} c'$ if one of the following conditions holds:

`Local` There is a transition $(s_1, \lambda, s_2) \in \delta$ and a process $p \in P$ such that (i) $P' = P$ and $\mathbf{r}' = \mathbf{r}$, i.e., the processes and registers are left unchanged, (ii) $\mathbf{s}(p) = s_1$, and (iii) $\mathbf{s}' = \mathbf{s}[p \mapsto s_2]$. A local transition changes the state of one process. If c' results from c by the execution of a local action, we also write $c \xrightarrow{\varepsilon}_{\mathcal{A}} c'$.

`Create` There is a transition $(s_1, r \hookleftarrow \mathtt{crt}(s, r'), s_2) \in \delta$ and a process $p \in P$ such that (i) $\mathbf{s}(p) = s_1$, (ii) $P' = P \cup \{q\}$ for some process $q \notin P$, (iii) $\mathbf{s}' = \mathbf{s}[p \mapsto s_2][q \mapsto s]$, i.e., process q is spawned in state s, while the new state of process p is s_2, and (iv) $\mathbf{r}' = \mathbf{r}[p \mapsto \mathbf{r}(p)[r \mapsto q]][q \mapsto \{r' \mapsto p\}]$, i.e., register r of process p is assigned the ID of the new process q and register r' of q is assigned the ID of p. If c' results from c by such a create action, we also write $c \xrightarrow{\mathtt{crt}(p,q)}_{\mathcal{A}} c'$.

`Selective symbol sending` There are two different processes p and q in P and two transitions $(s_1, \mathtt{snd}(r, m), s_2)$ and $(s_3, \mathtt{rcv}(r', m), s_4)$ in δ such that (i) $\mathbf{s}(p) = s_1$ and $\mathbf{s}(q) = s_3$, (ii) $\mathbf{r}(p)(r) = q$ and $\mathbf{r}(q)(r') = p$, i.e., the sender p has the ID of q in its register r and the

receiver q has the ID of p in its register r', (iii) $\mathbf{s}' = \mathbf{s}[p \mapsto s_2][q \mapsto s_4]$, and (iv) $\mathbf{r}' = \mathbf{r}$. Such a transition on configurations is also denoted as $c \xrightarrow{\mathtt{snd}(p,q,m)}_{\mathcal{A}} c'$.

`Selective ID sending` The set P contains two different processes p and q and δ contains two transitions $(s_1, \mathtt{snd}(r_1, m(r_1')), s_2)$ and $(s_3, \mathtt{rcv}(r_2, m(r_2')), s_4)$ such that (i) $\mathbf{s}(p) = s_1$ and $\mathbf{s}(q) = s_3$, (ii) $\mathbf{r}(p)(r_1) = q$ and $\mathbf{r}(q)(r_2) = p$, (iii) $\mathbf{s}' = \mathbf{s}[p \mapsto s_2][q \mapsto s_4]$, (iv) either $r_1' = \mathtt{self}$ (in this case, we set $p' = p$) or r_1' is a register such that $\mathbf{r}(p)(r_1')$ is defined (in this case, we set $p' = \mathbf{r}(p)(r_1')$), i.e., the ID to be sent should be the ID of some process, and (v) if $p' \neq q$, then, $\mathbf{r}' = \mathbf{r}[q \mapsto \mathbf{r}(q)[r_2' \mapsto p']]$, otherwise, $\mathbf{r}' = \mathbf{r}$, i.e., if q does not receive its own ID, it updates its register r_2' with the incoming ID. If c' results from c by such an action, we also write $c \xrightarrow{\mathtt{snd}(p,q,m(p'))}_{\mathcal{A}} c'$.

`Register resetting` There is a transition $(s_1, \mathtt{res}(r), s_2) \in \delta$ and a process $p \in P$ such that (i) $\mathbf{s}(p) = s_1$ and $\mathbf{s}' = \mathbf{s}[p \mapsto s_2]$, and (ii) $\mathbf{r}' = \mathbf{r}[p \mapsto \mathbf{r}(p)[r \mapsto \bot]]$, i.e., register r of process p is reset. Such a transition is also notated as $c \xrightarrow{\varepsilon}_{\mathcal{A}} c'$

There are also transitions between configurations caused by non-selective symbol or ID reception. The only difference to `Selective symbol sending` and `Selective ID sending` is that the receiver does not need to have the ID of the sender in its registers. We skip the formal definition of these kinds of transitions.

The transition system, runs and traces

A configuration $c = (P, \mathbf{s}, \mathbf{r})$ is said to be *initial* if it contains exactly one process (i.e., $P = \{p\}$ for some $p \in \mathbb{P}$), the process is in the initial state (i.e., $\mathbf{s}(p) = s_0$) and the registers of the process are empty (i.e., $\mathbf{r}(p) = R_{\mapsto\bot}$). The set of initial configurations of $\mathcal{A}$ is denoted by $\mathtt{Conf}_{\mathtt{init}}(\mathcal{A})$. A configuration $c = (P, \mathbf{s}, \mathbf{r})$ is *accepting* if all processes in c are in accepting states, i.e., $\mathbf{s}(p) \in F$ for all $p \in P$. We denote the set of accepting configurations by $\mathtt{Conf}_{\mathtt{acc}}(\mathcal{A})$. The transition system induced by $\mathcal{A}$ is defined by $\mathcal{T}(\mathcal{A}) = (\mathtt{Conf}(\mathcal{A}), \mathtt{Conf}_{\mathtt{init}}(\mathcal{A}), \longrightarrow_{\mathcal{A}}, \mathtt{Conf}_{\mathtt{acc}}(\mathcal{A}))$.

A sequence $\tau = c_0 \xrightarrow{\mathtt{act}_1}_{\mathcal{A}} c_1 \xrightarrow{\mathtt{act}_2}_{\mathcal{A}} \ldots \xrightarrow{\mathtt{act}_{n-1}}_{\mathcal{A}} c_{n-1} \xrightarrow{\mathtt{act}_n}_{\mathcal{A}} c_n$ of configurations and labelled transitions is called a *run* of $\mathcal{A}$ if $c_0 \in \mathtt{Conf}_{init}(\mathcal{A})$. A run which ends up in a configuration from $\mathtt{Conf}_{\mathtt{acc}}(\mathcal{A})$ is called *accepting*. As signalized in Section 11.1, we model the *traces* of the DCA $\mathcal{A}$ by data words over the proposition set $\mathtt{Prop}^{A}_{\mathtt{act}}$ and the attribute set $\mathtt{Attr}^{A}_{\mathtt{act}}$. We define the traces of $\mathcal{A}$ in such a way that only create and send actions are visible. If τ, as given above, is an accepting run, we define the trace of τ as the data word $trace(\tau) = \mathtt{dwrep}(\mathtt{act}_1) \ldots \mathtt{dwrep}(\mathtt{act}_n)$ where $\mathtt{dwrep}(\varepsilon) = \varepsilon$ is the empty word. The *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is defined as the set of all non-empty traces (i.e., traces of length at least 1) resulting from accepting runs of $\mathcal{A}$. We refer to the traces in $\mathcal{L}(\mathcal{A})$ as the traces of $\mathcal{A}$. Observe that it follows from definition that each trace of $\mathcal{A}$ must start with a create action.

Example 17. Figure 11.3 depicts a trace of the DCA given in Example 16. The ID 1 belongs to the root process, ID 2 belongs to the server process and the remaining IDs 3 and 4 belong to client processes. First, the root process creates the server and a client with ID 3. Then, it sends the ID of the server to client 3. After that, client 3 sends to the server a request along with its own ID. This is followed by the creation of client 4 by the root process. Then, the server ID is sent from the root process to the newly created client 4. Just like client 3, also client 4 sends a request to the server. Finally, the server first acknowledges the request of client 4 and then that of client 3.

□

A DCA which does not contain non-selective symbol or ID reception is called a *selective DCA*. If a DCA contains exactly k registers, we call it a k-DCA.

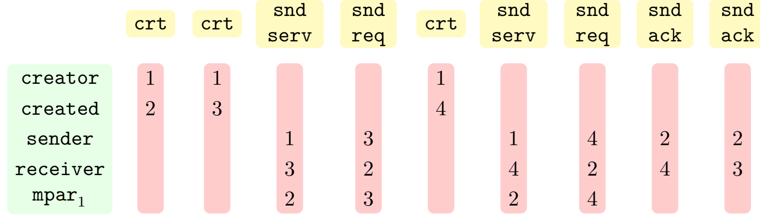


Figure 11.3: A trace of the DCA given in Example 16

As expected, the *non-emptiness problem* for DCA asks whether the language of a given DCA is non-empty. We further define the *state reachability problem* for DCA. A state `target` of a DCA $\mathcal{A}$ is called *reachable* in the transition system of $\mathcal{A}$ if there exists a run $c_0 \rightarrow^*_{\mathcal{A}} c_n$ with $\mathtt{target} \in c_n$. The state reachability problem $\textsc{StateReach}(\mathcal{C})$ for a class $\mathcal{C}$ of DCA asks the following question: Given a DCA $\mathcal{A}$ from $\mathcal{C}$ and a state `target` of $\mathcal{A}$, is `target` reachable in the transition system of $\mathcal{A}$?

11.2.1 Non-Emptiness

First, we will show that the non-emptiness problem for DCA is undecidable, even in the case with one register. Then, we will concentrate on selective DCA, i.e., DCA where a message can only be sent if the ID of the sender is stored in the registers of the receiver. We will see that the construction used in the undecidability proof for general 1-register DCA can be easily transferred to the setting of selective 2-register DCA, which means that for the latter case non-emptiness is also not decidable. Finally, we will prove that we get decidability if we restrict to selective 1-register DCA.

We start with general 1-DCA. The proof idea of the following theorem stems from our work [3].

Theorem 21. *The non-emptiness problem for 1-DCA is undecidable.*

Proof. The proof is by reduction from TransProb which we defined in Section 3.2.4. As this problem is undecidable [2], the result follows. By definition, a DCA is non-empty if and only if it has an accepting run. Given an instance of TransProb, i.e., two NFA $\mathcal{A}$ and $\mathcal{B}$ and a non-deterministic Letter-To-Letter Transducer $\mathcal{T}$ over the same alphabet Σ, the encoding of TransProb into the non-emptiness problem of DCA consists of constructing a *transduction chain* where the first element $p_{\mathcal{A}}$ of the chain is a process simulating $\mathcal{A}$, the last one is a process $p_{\mathcal{B}}$ simulating $\mathcal{B}$ and all intermediate processes $p^i_{\mathcal{T}}$ encode $\mathcal{T}$. Figure 11.4 illustrates the graph representation of a transduction chain with 4 transducer processes (for simplicity, we skip the state and register labels).

Figure 11.4: A transduction chain constructed by a 1-DCA

In order to construct such a chain, the initial process $p_{\mathcal{A}}$ first spawns a new process in one of the two states $s_{\mathcal{T}}$ and $s_{\mathcal{B}}$ and then starts simulating $\mathcal{A}$. The choice between $s_{\mathcal{T}}$ and $s_{\mathcal{B}}$ is made non-deterministically. If the new process starts from state $s_{\mathcal{T}}$, it spawns, similarly to the initial process, a new process in $s_{\mathcal{T}}$ or $s_{\mathcal{B}}$ and then starts simulating transducer $\mathcal{T}$ and so on. If a process is set to start from $s_{\mathcal{B}}$, it does not spawn any new process and simulates $\mathcal{B}$. Thus, we obtain a chain

of processes whose head simulates $\mathcal{A}$, the tail simulates $\mathcal{B}$ and all processes in between simulate transducer $\mathcal{T}$. Note that one register suffices to construct such a transduction chain.

The simulation of $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{T}$ works as follows: The first process $p_{\mathcal{A}}$ sends a word from Σ^*, symbol by symbol, to its successor in the chain. If the word is accepted by $\mathcal{A}$, it sends a special acceptance symbol to its successor and moves to an accepting state. Meanwhile, each intermediate process $p^i_{\mathcal{T}}$ sends to its successor for every incoming symbol $\sigma \in \Sigma$, a symbol corresponding to the output of $\mathcal{T}$ when reading σ. If an intermediate process gets the acceptance symbol, it checks whether the so far received word is accepted by $\mathcal{T}$. If it is the case, it transmits the acceptance symbol to the next process and enters an accepting state. Otherwise, it enters an error state. At the reception of the acceptance symbol, the last process $p_{\mathcal{B}}$ checks whether the so far received word is accepted by $\mathcal{B}$. If yes, it moves to an accepting state, otherwise, it moves to an error state. Note that if there are no intermediate processes simulating $\mathcal{T}$, process $p_{\mathcal{A}}$ sends the symbols directly to $p_{\mathcal{B}}$. It can easily be shown by induction that there exists an $i \geq 0$ with $\mathcal{T}^i(\mathcal{L}(\mathcal{A})) \cap \mathcal{L}(\mathcal{B}) \neq \emptyset$ if and only if a transduction chain of length $i+2$ where all processes reach an accepting state can be constructed. □

DCA with selective communication

Note that the transduction chain, given in the proof of Theorem 21, uses non-selective communication, i.e., each element in the chain can receive information from its predecessor without having the ID of the sender in its register. A similar transduction chain can be established by selective DCA with two registers where the first chain element maintains a link to its successor, the last one maintains a link to its predecessor and every inner chain element maintains a link to its predecessor as well as a link to its successor. Thus, we conclude:

Corollary 7. *The non-emptiness problem for selective 2-DCA is undecidable.*

If we restrict our consideration to selective 1-register DCA, we get decidability for the non-emptiness problem. We will show that the problem is solvable in polynomial time. Before presenting the proof, we would like to state some easy observations on the shape of configurations of selective 1-register DCA.

Given a configuration containing two processes p_1 and p_2, we say that there is a *one-directed link* from p_1 to p_2 if the register of p_1 holds p_2, but not vice-versa. First, observe that in the setting of selective 1-register DCA, a process can never receive (via a receive-action) an ID, besides its own and the one which is already in its register. To be convinced, think of two processes p_1 and p_2. To make communication between p_1 and p_2 possible, the register of p_1 must hold p_2 and vice-versa. In such a situation p_2 can only send its own ID which is already in the register of p_1 or the ID in its register which is the ID of p_1 itself. As the only remaining way for a process to get a new ID is to create a process, we conclude that in each configuration c of a selective 1-register DCA, a process p_1 is enabled to send (or receive) some information to (or from) a process p_2, if (i) there is some previous configuration where one of the processes p_1 and p_2 spawned the other one, and (ii) none of them has executed a spawn or reset action until configuration c. A further conclusion is that one-directed links within configurations are redundant in the sense that a process with a one-directed link to another process behaves like a process whose register is empty. Eliminating all one-directed links results in configurations where every weakly connected component in the corresponding graph encodings is either a single node or consists of two nodes with edges to each other. The observation that configurations of selective 1-register DCA can be simulated by such simplified configurations paves the way for deciding the non-emptiness problem.

We proceed with a second simple observation which also holds for general DCA. Let c be a configuration and let c_1 and c_2 be two isomorphic sub configurations corresponding to two weakly connected components in the graph encoding of c. Note that all actions executable by a process

in c_1 are also executable by the corresponding process in c_2 and vice-versa. Moreover, all configurations evolving from c_1 can (up to isomorphism) also be derived from c_2 and vice-versa. Now, let $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a partitioning of the set of all sub configurations of c which correspond to weakly connected components such that sub configurations in the same S_i are isomorphic and sub configurations from distinct sets are not. It is easy to see that an accepting configuration can be reached from c if and only if for every $i \in \{1, \ldots, n\}$, there is a sub configuration $c' \in S_i$ such that an accepting configuration can be reached from c'.

Now, let us put our observations into some formal shape. Let $\mathcal{A} = (A, \{r\}, S, s_0, \delta, F)$ be a selective DCA with one register r. Given a configuration $c = (P, \mathbf{s}, \mathbf{r})$ of $\mathcal{A}$, the register content of some process p_1 in c is called *redundant* if $\mathbf{r}(p_1)(r) = p_2$ for some process p_2, but $\mathbf{r}(p_2)(r) \neq p_1$. In the following, we use brackets in form of $\{\!|$ and $|\!\}$ to define multisets which can contain multiple copies of the same element. We call a process p in c a *single* if $\mathbf{r}(p)(r) = \bot$ and there is no process $p' \in P$ with $\mathbf{r}(p')(r) = p$. A pair p_1, p_2 of two distinct processes in c is called a *couple* if $\mathbf{r}(p_1)(r) = p_2$, $\mathbf{r}(p_2)(r) = p_1$ and there is no other process p', besides p_1 and p_2 with $\mathbf{r}(p')(r) = p_1$ or $\mathbf{r}(p')(r) = p_2$. The *type* of a single p is defined as $\mathbf{s}(s)$ and that of a couple consisting of p_1 and p_2 as $\{\!|\mathbf{s}(p_1), \mathbf{s}(p_2)|\!\}$. For a single type s, an s-configuration is a configuration consisting of a single of type s. Analogously, for a couple type $\{\!|s_1, s_2|\!\}$, an $\{\!|s_1, s_2|\!\}$-configuration is a configuration consisting of a couple of type $\{\!|s_1, s_2|\!\}$. Given a configuration c, we define $\texttt{Types}(c) = \{s \mid c$ contains a single of type $s\} \cup \{\{\!|s_1, s_2|\!\} \mid c$ contains a couple of type $\{\!|s_1, s_2|\!\}\}$. We call a configuration c' the *simplification* of some configuration c if c' results from c by deleting the inputs of all redundant registers. Figure 11.5 illustrates the graph representations of an exemplary configuration and its simplification.

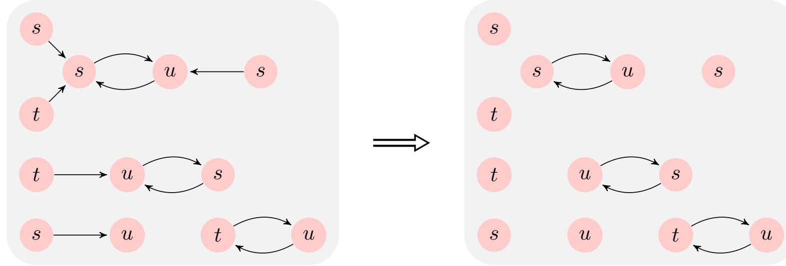


Figure 11.5: The simplification of a configuration

We formulate:

Observation 8. For every configuration c of a selective 1-DCA, its simplification consists of singles and couples.

Now, we define a *simplified transition relation* $\rightarrow^{\texttt{sim}}_{\mathcal{A}}$ on the set of configurations of $\mathcal{A}$. For two configurations c_1 and c_2, we set $c_1 \rightarrow^{\texttt{sim}}_{\mathcal{A}} c_2$ if there is some configuration c_1' with $c_1 \rightarrow_{\mathcal{A}} c_1'$ and c_2 is the simplification of c_1'. We call a sequence $\tau = c_0 \rightarrow^{\texttt{sim}}_{\mathcal{A}} c_1 \ldots c_{n-1} \rightarrow^{\texttt{sim}}_{\mathcal{A}} c_n$ a *simplified run* of $\mathcal{A}$ *starting* at c_0. The sequence τ is called a *simplified run* if c_0 is an initial configuration. A simplified run is *accepting* if it ends up in an accepting configuration. According to our informal explanations above, we observe:

Observation 9. A selective 1-DCA has an accepting run if and only if it has a simplified accepting run.

Our discussion on isomorphic sub configurations implies that the problem of finding an accepting run can even be made more simple.

Observation 10. Let $\mathcal{A}$ be a selective 1-DCA and c a simplified configuration of $\mathcal{A}$. There is a simplified accepting run of $\mathcal{A}$ starting at c if and only if for every $t \in \texttt{Types}(c)$, there is a simplified accepting run of $\mathcal{A}$ starting at an arbitrary t-configuration.

Using Observations 8-10, we can easily show that non-emptiness for selective 1-register DCA is solvable in polynomial time.

Theorem 22. *The non-emptiness problem for selective 1-DCA is in* PTIME.

Proof. Recall that it follows from the definition of DCA-traces that the language of a DCA is non-empty if and only if the DCA has an accepting run containing at least one create action. Given a selective 1-DCA, we will describe a polynomial time algorithm which, starting from a representation of all accepting configurations, computes a representation of all configurations from which there is an accepting run containing at least one create action. Then, the language of the DCA is non-empty if and only if an initial configuration is contained in the computed set.

Let $\mathcal{A} = (A, R, \{r\}, S, s_0, \delta, F)$ be a selective 1-DCA with a single register r. It follows from Observations 8-10 that an accepting configuration is reachable from a simplified configuration c if and only if there is a (simplified) configuration c' with $c \rightarrow_{\mathcal{A}}^{\texttt{sim}} c'$ and for every $t \in \texttt{Types}(c')$, there is an arbitrary t-configuration reaching an accepting configuration. Hence, the non-emptiness problem for $\mathcal{A}$ can be solved by (i) computing the set of all singles and couples from which there is a simplified accepting run containing at least one create action, and (ii) checking whether the set contains at least one single representing an initial configuration. Observe that the mentioned set is in general infinite, because we have an infinite supply of process IDs. However, since we do not have to distinguish between singles or couples of the same type, we represent the set by the finite set of occurring types.

Now, we explain the details of the algorithm. Besides usual types of the forms s and $\{\!|s_1, s_2|\!\}$, the set computed by the algorithm can also contain elements of the forms $\widehat{s}$ and $\widehat{\{\!|s_1, s_2|\!\}}$ with the following meaning: If a single type s (or a couple type $\{\!|s_1, s_2|\!\}$, respectively) is contained in the set, then, there is an accepting run of $\mathcal{A}$ which starts at an s-configuration (or an $\{\!|s_1, s_2|\!\}$-configuration, respectively) and does not contain any create action. If the set contains an $\widehat{s}$ (or an $\widehat{\{\!|s_1, s_2|\!\}}$, respectively) then, there is an accepting run of $\mathcal{A}$ which starts at an s-configuration (or an $\{\!|s_1, s_2|\!\}$-configuration, respectively) and contains at least one create action. The algorithm first constructs the set $N = S \cup \{\{\!|s_1, s_2|\!\} \mid s_1, s_2 \in S\}$ of all possible types and the set $M_0 = F \cup \{\{\!|s_1, s_2|\!\} \mid s_1, s_2 \in F\}$ of all types representing accepting configurations. Then, it iteratively computes successor sets M_i for $i \geq 1$ until it reaches a fixed point, i.e., an $i \geq 1$ with $M_i = M_{i-1}$. Let for some $i \geq 0$, M_i be the current set. The successor set M_{i+1} is defined by $M_i \cup M_i'$ where M_i' contains all possible predecessor types for the types in M_i. Formally, the set M_i' is the smallest set fulfilling the conditions given below. We start by the enumeration of cases implying the containment of single types s in M_i':

- If there is a single type $s \in N$, there is a local transition (s, λ, s') or a reset transition $(s, \texttt{res}(r), s')$ in δ and s' is contained in M_i, then, s is contained in M_i'.
- If there is a single type $s \in N$, there is a create transition $(s, r \hookleftarrow \texttt{crt}(s'', r), s') \in \delta$ and $\{\!|s', s''|\!\}$ is contained in M_i, then, s is contained in M_i'.

We now enumerate cases in which a couple type must be contained in M_i'. Please keep in mind that in the framework of selective 1-register DCA, symbol sending and ID sending have the same effect.

- If there is a couple type $\{\!|s_1, s_2|\!\} \in N$, there is a local transition $(s_1, \lambda, s_1') \in \delta$ and $\{\!|s_1', s_2|\!\}$ is contained in M_i, then, $\{\!|s_1, s_2|\!\}$ is contained in M_i'.

- If there is a couple type $\{\!|s_1, s_2|\!\} \in N$, there is a reset transition $(s_1, \texttt{res}(r), s_1') \in \delta$ and s_1' and s_2 are contained in M_i, then, $\{\!|s_1, s_2|\!\}$ is contained in M_i'.
- If there is a couple type $\{\!|s_1, s_2|\!\} \in N$, there is a create transition $(s_1, r \hookleftarrow \texttt{crt}(s, r), s_1') \in \delta$ and $\{\!|s_1', s|\!\}$ and s_2 are contained in M_i, then, $\{\!|s_1, s_2|\!\}$ is contained in M_i'.
- If there is a couple type $\{\!|s_1, s_2|\!\} \in N$, there are transitions $(s_1, \texttt{snd}(r, m), s_1')$, $(s_2, \texttt{rcv}(r, m), s_2')$ $\in \delta$ or transitions $(s_1, \texttt{snd}(r, m(r')), s_1')$, $(s_2, \texttt{rcv}(r, m(r)), s_2') \in \delta$ and $\{\!|s_1', s_2'|\!\}$ is contained in M_i, then, $\{\!|s_1, s_2|\!\}$ is contained in M_i'.

The cases in which elements of the forms $\widehat{s}$ or $\{\!|\widehat{s_1, s_2}|\!\}$ must be contained in M_i' are straightforward extensions of the ones above. We exemplarily give two cases implying the containment of such elements in M_i':

- If there is a single type $s \in N$, there is a create transition $(s, r \hookleftarrow \texttt{crt}(s'', r), s') \in \delta$ and $\{\!|s', s''|\!\}$ or $\{\!|\widehat{s', s''}|\!\}$ is contained in M_i, then, $\widehat{s}$ is contained in M_i'.
- If there is a couple type $\{\!|s_1, s_2|\!\} \in N$, there is a reset transition $(s_1, \texttt{res}(r), s_1') \in \delta$ and $\widehat{s_1'}, s_2 \in M_i$ or $s_1', \widehat{s_2} \in M_i$ or $\widehat{s_1'}, \widehat{s_2} \in M_i$, then, $\{\!|\widehat{s_1, s_2}|\!\}$ is contained in M_i'.

If the algorithm reaches a set M_i with $i \geq 1$ and $M_i = M_{i-1}$, it just checks whether $\widehat{s_0}$ is contained in M_i and outputs that the language of $\mathcal{A}$ is non-empty if and only if this element is available in M_i.

Observe that the sizes of N and M_0 are at most $|S| + |S|^2$ and $|F| + |F|^2$, respectively. Moreover, after at most $2|N|$ iterations, the algorithm must reach a fixed point. In each iteration, the computation of a new set M_i can be performed in time $\mathcal{O}(|N||\delta|)$. The test whether $\widehat{s_0}$ is contained in the final set can be done in time $\mathcal{O}(|N|)$. Thus, the overall algorithm works in polynomial time. ☐

11.2.2 State Reachability

In this section, we will consider the state reachability problem for DCA. As accepting states do not play any role in reachability questions, throughout this section we will skip the sets of accepting states and accepting configurations in the definitions of DCA and their corresponding transition systems. We will first explain that it easily follows from the undecidability of the non-emptiness of 1-DCA that reachability for this fragment is also not decidable. Searching for fragments with a decidable reachability problem, we will follow a different path than in the case of non-emptiness. Inspired by recent approaches in the verification of ad-hoc networks [77, 2], we will focus on DCA with bounded transition systems, i.e., transition systems where transitions are only allowed if they lead to configurations where the length of directed paths in the underlying communication graphs are bounded by some constant. It will turn out that also in this case, state reachability remains undecidable. Decidability cannot even be achieved for strongly-bounded transition systems which restrict to configurations where communication paths are bounded regardless the directions of edges. Therefore, we will introduce degenerative DCA, i.e., DCA where processes can lose register contents non-deterministically. While reachability for degenerative DCA with bounded transition systems is still undecidable, we will show that this problem is decidable for degenerative DCA if the corresponding transition systems are strongly bounded. Our proof is by a non-trivial instantiation of the framework of Well-Structured Transition Systems introduced in Section 3.2.3. We will close our studies on reachability by a summary of our results on buffered DCA in [4]. In terms of communication, this kind of DCA are closer to the original DCA-model [47], because processes of buffered DCA communicate asynchronously via unbounded FIFO-mailboxes.

Now, we describe our results in detail. The undecidability of the state reachability for 1-DCA can be derived easily from the proof of the undecidability of the non-emptiness of 1-DCA (Theorem

21). Remember that the result was obtained by reduction from the problem TransProb. We briefly recall the idea of that reduction. For two NFA $\mathcal{A}$ and $\mathcal{B}$ and a transducer $\mathcal{T}$, we gave a 1-DCA which constructs a transduction chain where the first process $p_{\mathcal{A}}$ simulates $\mathcal{A}$, the last process $p_{\mathcal{B}}$ simulates $\mathcal{B}$ and all intermediate processes $p_{\mathcal{T}}$ imitate transducer $\mathcal{T}$. After the construction phase, $p_{\mathcal{A}}$ sends a word accepted by $\mathcal{A}$, symbol by symbol, to its successor and finally moves to an accepting state. Each process $p_{\mathcal{T}}$ forwards for each received symbol an output symbol to its successor according to the behaviour of $\mathcal{T}$. If the entire received word is accepted by $\mathcal{T}$, the process $p_{\mathcal{T}}$ enters an accepting state. Finally, process $p_{\mathcal{B}}$ checks whether the received word is accepted by $\mathcal{B}$ and goes into an accepting state if the answer is yes. We can easily turn this construction into an encoding of TransProb into StateReach(1-DCA). Given two NFA $\mathcal{A}$, $\mathcal{B}$ and a transducer $\mathcal{T}$, we designate a special state `target` and design a 1-register DCA which allows the construction of the same transduction chain with the difference that $p_{\mathcal{A}}$ and each intermediate process $p_{\mathcal{T}}$, after confirming that the received word is accepted by the corresponding automaton ($\mathcal{A}$ or $\mathcal{T}$, respectively), moves to an idle state, instead of an accepting one. Furthermore, process $p_{\mathcal{B}}$ enters state `target` if and only if it decides that the received word is accepted by $\mathcal{B}$. Thus, $\mathcal{T}^i(\mathcal{L}(\mathcal{A})) \cap \mathcal{L}(\mathcal{B}) \neq \emptyset$ for some $i \geq 0$ if and only if the transduction chain reaches a configuration where one process is in state `target`. We conclude:

Corollary 8. *The problem* StateReach*(1-DCA) is undecidable.*

An important point in the reduction from TransProb to the non-emptiness or the state reachability of 1-DCA is that the communication paths in the constructed transduction chains are allowed to be as long as necessary. With communication path we mean a sequence $p_1, \ldots, p_n$ of processes such that every p_i with $1 \leq i < n$ holds the ID of p_{i+1} in its register. If we forbid such paths of unbounded length, our reductions do not work in this form. Next, we will show that even in the case where the length of communication paths is bounded by some constant, state reachability remains undecidable.

Bounded state reachability

We define bounded configurations and bounded state reachability for DCA. Let $\mathcal{A}$ be a DCA and $\mathcal{T}(\mathcal{A}) = (\mathtt{Conf}(\mathcal{A}), \mathtt{Conf}_{init}(\mathcal{A}), \longrightarrow_{\mathcal{A}})$ its corresponding transition system. For a natural number k, we say that a configuration $c \in \mathtt{Conf}(\mathcal{A})$ is *k-bounded* if the diameter of its graph encoding is bounded by k, i.e., $\mathtt{diameter}(enc(c)) \leq k$. Given a set $B \subseteq \mathtt{Conf}(\mathcal{A})$ of configurations, we use $B^{\oslash k}$ to denote the set of k-bounded configurations in B. The restriction of $\longrightarrow_{\mathcal{A}}$ to the set $\mathtt{Conf}(\mathcal{A})^{\oslash k}$ of k-bounded configurations is defined as $\longrightarrow_{\mathcal{A}}^{\oslash k} = \longrightarrow_{\mathcal{A}} \cap ((\mathtt{Conf}(\mathcal{A})^{\oslash k}) \times (\mathtt{Conf}(\mathcal{A})^{\oslash k}))$. We use $\mathcal{T}^{\oslash k}(\mathcal{A})$ to denote the resulting transition system defined by $\mathcal{T}^{\oslash k}(\mathcal{A}) = (\mathtt{Conf}(\mathcal{A})^{\oslash k}, \mathtt{Conf}_{init}(\mathcal{A})^{\oslash k}, \longrightarrow_{\mathcal{A}}^{\oslash k})$.

For a class $\mathcal{C}$ of DCA and a natural number k, we denote by BoundStateReach$(\mathcal{C}, k)$ the following *k-bounded state reachability problem*: Given a DCA $\mathcal{A} \in \mathcal{C}$ and a state `target` of $\mathcal{A}$, is there a reachable configuration c in $\mathcal{T}^{\oslash k}(\mathcal{A})$ with $\mathtt{target} \in c$?

By an adaption of the transduction chain construction introduced in the proof of Theorem 21 and adjusted for Corollaries 7 and 8, we can show that even for DCA with two registers the 2-bounded state reachability problem is not decidable.

Theorem 23. *The problem* BoundStateReach*(2-DCA,2) is not decidable.*

Proof. We reduce TransProb to the 2-bounded state reachability problem for 2-register DCA. Given an instance of TransProb, we construct a DCA which builds configurations reproducing the purpose of the transduction chain that we described in the proof of Theorem 21. The challenge of the encoding is to keep the simple path length in the graph encodings of configurations bounded by 2. In order to do that, we make use of additional *relay* processes as well as reset transitions.

Let $\mathcal{A}$, $\mathcal{B}$ be some NFA and $\mathcal{T}$ a transducer over some alphabet Σ which serve as inputs for TransProb. We construct a DCA with two registers and a designated state `target` such that there is some $i \geq 0$ with $\mathcal{T}^i(\mathcal{L}(\mathcal{A})) \cap \mathcal{L}(\mathcal{B}) \neq \emptyset$ if and only if `target` is reachable in the 2-bounded transition system of the DCA. To give the overall idea about the structure of the configurations produced by the DCA, we present in Figure 11.6 exemplarily the final shape of configurations in the case of $i = 3$ transductions. The processes $p_{\mathcal{A}}, p^1_{\mathcal{T}}, p^2_{\mathcal{T}}, p^3_{\mathcal{T}}$ and $p_{\mathcal{B}}$ in the graphic encode a transduction chain in

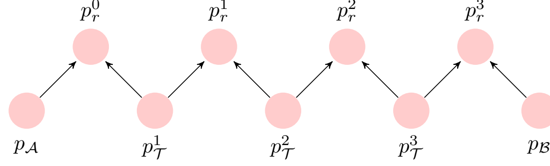


Figure 11.6: A transduction chain with relay processes constructed by a 2-DCA

the sense that the head $p_{\mathcal{A}}$ simulates $\mathcal{A}$, the tail $p_{\mathcal{B}}$ simulates $\mathcal{B}$ and all intermediate processes $p^i_{\mathcal{T}}$ imitate $\mathcal{T}$. The remaining processes p^i_r with $0 \leq i \leq 3$ are *relay* processes which are responsible for forwarding messages between each two consecutive processes of the chain.

Now, we explain how configurations of this shape are constructed. In the beginning, the initial process $p_{\mathcal{A}}$ creates a process p^0_r. The latter process proceeds by reseting its register containing the ID of process $p_{\mathcal{A}}$ and creating a new process, either $p_{\mathcal{B}}$ or $p^1_{\mathcal{T}}$. The choice is made non-deterministically. After spawning the new process, p^0_r resets its register containing the ID of the new process and moves to a certain relay state s_r from which it will relay messages. If the process that p^0_r spawned is $\mathcal{B}$, no further processes are created and the construction of the chain ends. Otherwise, i.e., if $p^1_{\mathcal{T}}$ has been created, the construction of the chain continues by reproducing the same scheme: The process $p^1_{\mathcal{T}}$ spawns the second relay process p^1_r and so on until $p_{\mathcal{B}}$ is spawned. Thus, we obtain shapes with simple paths of length at most two.

Given that at the end of the chain construction phase, a configuration where the number of processes simulating $\mathcal{T}$ is m ($m = 3$ in the example of Figure 11.6) is obtained, we define our transduction chain as the sequence of processes $p_{\mathcal{A}}, p^1_{\mathcal{T}}, \ldots, p^m_{\mathcal{T}}, p_{\mathcal{B}}$. Even though there is no simple path between two consecutive processes in the transduction chain, we will show how symbols can be transmitted from one process to the next one in the chain along relay processes p^i_r. Once the final shape of the chain is built, each relay process must be in state s_r. Consider the first two processes $p_{\mathcal{A}}$ and $p^1_{\mathcal{T}}$ in the chain. Process p^0_r plays the role of a relay between these processes. Let $\sigma \in \Sigma$ be a symbol that has to be sent from $p_{\mathcal{A}}$ to $p^1_{\mathcal{T}}$. The sending happens in two steps. First, process $p_{\mathcal{A}}$ sends (σ, out) to p^0_r which moves p^0_r to some state s_σ. Process p^0_r stays in that state until it receives a symbol (σ, in) from $p^1_{\mathcal{T}}$. Meanwhile, the latter process tries to send a symbol (σ', in) to p^0_r where σ' is a symbol which transducer $\mathcal{T}$ can read as an input in its particular state. If the symbols match, i.e., if the symbol that $p^1_{\mathcal{T}}$ sends is (σ, in), then (i) p^0_r receives the symbol from $p^1_{\mathcal{T}}$ and returns to the relay state s_r, and (ii) $p^1_{\mathcal{T}}$ moves to a temporary state from which it will send to the next relay process p^1_r the output symbol of $\mathcal{T}$ corresponding to the input symbol which it has just synchronized with p^0_r. Otherwise, i.e., if the symbols do not match, process p^0_r stays in state s_σ, and so does process $p^1_{\mathcal{T}}$. Thus, instead of executing receive and send transitions for $\mathcal{T}$-transitions, process $p^1_{\mathcal{T}}$ simulates a transducer transition by two send actions, the first one to the previous relay process, the second one to the next relay process.

Following this mode of communication, symbols are handed over from the head $p_{\mathcal{A}}$ to the tail $p_{\mathcal{B}}$. Finally, $p_{\mathcal{B}}$ moves to state `target` if and only if the entire word, which it receives via p^m_r, is accepted by $\mathcal{B}$. It is easy to see that there is some $i \geq 0$ with $\mathcal{T}^i(\mathcal{L}(\mathcal{A})) \cap \mathcal{L}(\mathcal{B}) \neq \emptyset$ if and only if the 2-register DCA produces a transduction chain consisting of processes $p_{\mathcal{A}}, p^1_{\mathcal{T}}, \ldots, p^i_{\mathcal{T}}, p_{\mathcal{B}}$ along

with relay processes $p_r^0, \ldots, p_r^i$ such that $p_{\mathcal{B}}$ reaches state $\texttt{target}$.

□

Hence, bounding the diameter of the graph encodings of configurations does not provide decidability for state reachability. Next, we consider a stronger restriction on the set of allowed configurations. It sets a bound on the length of paths in the graph representations regardless the edge direction.

Strongly bounded state reachability

Before introducing *strongly bounded state reachability* for DCA, we define the *closure* of labelled directed graphs. Given a directed labelled graph $G = (V, \Sigma_v, \Sigma_e, \lambda, E)$, the closure $\mathit{closure}(G)$ of G is the node-labelled undirected graph obtained from G by removing labels and directions of edges, i.e., $\mathit{closure}(G) = (V, \Sigma_v, \lambda, \{\{u,v\} | \, (u,\sigma,v) \in E\})$. We say that a $c \in \texttt{Conf}(\mathcal{A})$ of a DCA $\mathcal{A}$ is *k-strongly bounded* for some natural number k if $\texttt{diameter}(\mathit{closure}(\mathit{enc}(c))) \leq k$. Given a $B \subseteq \texttt{Conf}(\mathcal{A})$, we use $B^{\otimes k}$ to denote the set of k-strongly bounded configurations in B, i.e., $B^{\otimes k} = \{c \in B | \, \texttt{diameter}(\mathit{closure}(\mathit{enc}(c))) \leq k\}$. We define the transition relation $\longrightarrow_{\mathcal{A}}^{\otimes k}$ by $\longrightarrow_{\mathcal{A}}^{\otimes k} = \longrightarrow_{\mathcal{A}} \cap (\texttt{Conf}(\mathcal{A})^{\otimes k} \times \texttt{Conf}(\mathcal{A})^{\otimes k})$ and the transition system $\mathcal{T}^{\otimes k}(\mathcal{A})$ by $\mathcal{T}^{\otimes k}(\mathcal{A}) = (\texttt{Conf}(\mathcal{A})^{\otimes k}, \texttt{Conf}_{init}(\mathcal{A})^{\otimes k}, \longrightarrow_{\mathcal{A}}^{\otimes k})$.

For a class $\mathcal{C}$ of DCA and a natural number k, we denote by StrongBoundStateReach$(\mathcal{C}, k)$ the following *k-strongly bounded state reachability problem*: Given a DCA $\mathcal{A} \in \mathcal{C}$ and a state $\texttt{target}$ of $\mathcal{A}$, is there a reachable configuration c in $\mathcal{T}^{\otimes k}(\mathcal{A})$ with $\texttt{target} \in c$?

In [3], we show that even in this restrictive setting, state reachability remains undecidable:

Theorem 24. *The problem* StrongBoundStateReach*(2-DCA,4) is not decidable.*

The proof is carried out by a reduction from the reachability problem for Minsky 2-Counter Machines (2-MCM, for definition, see Section 3.2.2). Here, we do not give the full proof.

In the following, we will show that strongly bounded state reachability becomes decidable if we restrict to *degenerative DCA*, i.e., DCA which are allowed to execute non-deterministic reset transitions at each state. Formally, a degenerative DCA $\mathcal{A} = (A, R, S, s_0, \delta)$ is a DCA where for every state $s \in S$ and register $r \in R$, the transition $(s, \texttt{res}(r), s)$ is contained in δ. The degenerative counterpart $\texttt{Deg}(\mathcal{A})$ of a DCA $\mathcal{A}$ results from $\mathcal{A}$ by adding to the transition relation of $\mathcal{A}$ a transition $(s, \texttt{res}(r), s)$ for every state s and register r.

The rest of this section is devoted to the proof of the following theorem:

Theorem 25. *For every $k \geq 1$, the problem* StrongBoundStateReach*(degenerative DCA,k) is decidable.*

The proof is carried out by a non-trivial instantiation of the framework of Well-Structured Transition Systems (WSTS) defined in Section 3.2.3. Let $k \geq 1$ be a natural number, $\mathcal{A} = (A, R, S, s_0, \delta)$ a degenerative DCA and $\texttt{target}$ a state from S. We first fix some notations. For the sake of readability, we set $C_{\texttt{init}} = \texttt{Conf}_{\texttt{init}}(\mathcal{A})^{\otimes k}$ and $C = \texttt{Conf}(\mathcal{A})^{\otimes k}$. Thus, the k-strongly bounded transition system $\mathcal{T}^{\otimes k}(\mathcal{A})$ induced by $\mathcal{A}$ is described by $(C, C_{\texttt{init}}, \longrightarrow_{\mathcal{A}}^{\otimes k})$. We use $\xrightarrow{\texttt{reset}}_{\mathcal{A}} \subseteq C \times C$ to denote an arbitrary reset-transition, i.e., for two configurations c and c', it holds $c \xrightarrow{\texttt{reset}}_{\mathcal{A}} c'$ if c' result from c by the execution of a reset transition from $\{(s, \texttt{res}(r), s) \mid s \in S, r \in R\} \subseteq \delta$. The *reset prefix* transition relation $\rightsquigarrow$ is defined as $\xrightarrow{\texttt{reset}}_{\mathcal{A}}^{*} \circ \longrightarrow_{\mathcal{A}}^{\otimes k}$. Note that the reflexive transitive closures of $\rightsquigarrow$ and $\longrightarrow_{\mathcal{A}}^{\otimes k}$ are identical. Hence, $\texttt{target}$ is reachable in $\mathcal{T}^{\otimes k}(\mathcal{A}) = (C, C_{\texttt{init}}, \longrightarrow_{\mathcal{A}}^{\otimes k})$ if and only if it is reachable in $\mathcal{T}_{\texttt{ext}}^{\otimes k}(\mathcal{A}) = (C, C_{init}, \rightsquigarrow)$.

We will prove that the reachability of $\texttt{target}$ in $\mathcal{T}_{\texttt{ext}}^{\otimes k}(\mathcal{A})$ is decidable. To this end, we will first show that $\mathcal{T}_{\texttt{ext}}^{\otimes k}(\mathcal{A})$ is a Well-Structured Transition System equipped with some well-quasi

ordering $\preccurlyeq$ on C. Then, we will explain that one can fix a configuration $c_{\texttt{target}} \in C$ such that the coverability of $c_{\texttt{target}}$ in $\mathcal{T}_{\texttt{ext}}^{\otimes k}(\mathcal{A})$ is equivalent to the reachability of $\texttt{target}$ in the same transition system. Finally, we will prove that $\preccurlyeq$ is decidable, $\mathcal{T}_{\texttt{ext}}^{\otimes k}(\mathcal{A})$ has computable predecessor bases and for every $c \in C$, it is decidable whether $\uparrow\{c\} \cap C_{\texttt{init}}$ is non-empty. By Theorem 1, these conditions suffice to conclude that coverability and, hence, reachability in $\mathcal{T}_{\texttt{ext}}^{\otimes k}(\mathcal{A})$ is decidable.

For the proof that $\mathcal{T}_{\texttt{ext}}^{\otimes k}(\mathcal{A}) = (C, C_{\texttt{init}}, \rightsquigarrow)$ is a Well-Structured Transition System, we have to define a well-quasi ordering $\preccurlyeq$ on C and to show that $\rightsquigarrow$ is monotonic with respect to $\preccurlyeq$.

A well-quasi ordering on C. We use $\sqsubseteq_{sub}$ to denote the *sub-graph embedding* relation defined on labelled graphs which is defined as follows: For two labelled graphs $(V_1, \Sigma_v, \Sigma_e, \lambda_1, E_1)$ and $(V_2, \Sigma_v, \Sigma_e, \lambda_2, E_2)$, we have $(V_1, \Sigma_v, \Sigma_e, \lambda_1, E_1) \sqsubseteq_{sub} (V_2, \Sigma_v, \Sigma_e, \lambda_2, E_2)$ if there exists an injective mapping $t : V_1 \to V_2$ that is label and edge preserving, i.e. $\forall v, u \in V_1$ and $\forall a \in \Sigma_e$ we have $\lambda_1(v) = \lambda_2(t(v))$ and $(v, a, u) \in E_1 \Rightarrow (t(v), a, t(u)) \in E_2$. The embedding relation over node-labelled undirected graphs is defined analogously. The ordering $\preccurlyeq$ over the set of configurations is defined as follows: Given two configurations $c_1 = (P_1, \mathbf{s}_1, \mathbf{r}_1)$ and $c_2 = (P_2, \mathbf{s}_2, \mathbf{r}_2)$, we have $c_1 \preccurlyeq c_2$ if $enc(c_1) \sqsubseteq_{sub} enc(c_2)$. Note that $c_1 \preccurlyeq c_2$ is equivalent to saying that there exists an injective mapping $g : P_1 \to P_2$ such that (i) for every $p \in P_1$, it holds $\mathbf{s}_1(p) = \mathbf{s}_2(g(p))$, and (ii) for every $p_1, p_2 \in P_1$ and every $r \in R$, it holds $\mathbf{r}_1(p_1)(r) = p_2 \Longrightarrow \mathbf{r}_2(g(p_1))(r) = g(p_2)$. Figure 11.7 shows the graph encodings of three configurations $c_1 \preccurlyeq c_2 \preccurlyeq c_3$.

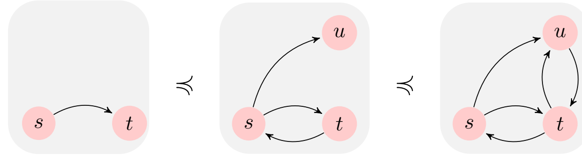


Figure 11.7: The well-quasi ordering $\preccurlyeq$ on configurations

Lemma 9. *The relation $\preccurlyeq$ is a well-quasi ordering on C.*

Proof. We have to show that for every infinite sequence $(c_i)_{i \geq 0}$ of configurations from C, there are two indices $i < j$ with $c_i \preccurlyeq c_j$. We will do this by making use of a theorem in [88] which says that sub-graph ordering on node-labelled directed graphs is a well-quasi ordering, given that the underlying undirected graphs, namely the closures of the directed graphs, have a bounded diameter.

First, we define an operation *lf* which converts labelled directed graphs into node-labelled ones by replacing each edge by a node labelled with the label of the edge. Formally, given a labelled directed graph $G = (V, \Sigma_v, \Sigma_e, \lambda, E)$, the node-labelled directed graph $lf(G)$ is defined as $lf(G) = (V', \Sigma_v \cup \Sigma_e, \lambda', E')$, where $V' = V \cup \left\{v_{(u_1,a,u_2)} \mid (u_1, a, u_2) \in E\right\}$, the vertex labelling function λ' is defined by $\lambda'(v) = \lambda(v)$ for $v \in V$ and $\lambda'\left(v_{(u_1,a,u_2)}\right) = a$ for $v_{(u_1,a,u_2)} \in V' \backslash V$ and the set E' of edges is given by $E' = \left\{(u_1, v_{(u_1,a,u_2)}), (v_{(u_1,a,u_2)}, u_2) \mid (u_1, a, u_2) \in E\right\}$. Note that for two labelled graphs G_1 and G_2, we have $G_1 \sqsubseteq_{sub} G_2$ if and only if $lf(G_1) \sqsubseteq_{sub} lf(G_2)$. Note also that if the diameter of the closure of a labelled graph G is k, then, the diameter of the closure of $lf(G)$ is at most $2k + 2$. For the proof of this, let G be a labelled graph, $closure(G) = (V, \Sigma_v, \lambda, E)$ be of diameter k and $closure(lf(G)) = (V', \Sigma_v, \lambda', E')$. For the sake of contradiction, assume now that there is a simple path $\pi = v_1 \ldots v_{2k+4}$ of length $2k + 3$ in $closure(lf(G))$. By construction of $closure(lf(G))$, the nodes in π must alternate between nodes from V and nodes from $V' \backslash V$ which arose after eliminating labelled edges in G. We assume that v_1 is from V (the other case is handled analogously). Then, by construction of $closure(lf(G))$, for every odd i with $1 \leq i \leq 2k + 1$, we have $\{v_i, v_{i+2}\} \in E$. Thus, the subsequence of π consisting of all odd positions builds a simple

path of length $k+1$ in $closure(G)$ which is a contradiction to our assumption that the diameter of $closure(G)$ is k.

Now, let $(c_i)_{i\geq 0}$ be an infinite sequence of configurations from C. For every $i \geq 0$, the diameter of the graph $closure(enc(c_i))$ must be bounded by k, because c_i is k-strongly bounded. We consider the sequence $(lf(enc(c_i)))_{i\geq 0}$. Due to our explanations above, for every $i \geq 0$, $closure(lf(enc(c_i)))$ is a graph of diameter at most $2k+2$. By Theorem 2.6 in [88], subgraph ordering on node-labelled directed graphs is a well-quasi ordering[1], under the condition that the closures of the underlying undirected graphs have a bounded diameter. Thus, as the closure of every graph in $(lf(enc(c_i)))_{i\geq 0}$ has a diameter of at most $2k+2$, the subgraph relation on these graphs is a well-quasi ordering. Hence, there are $i < j$ with $lf(enc(c_i)) \sqsubseteq_{sub} lf(enc(c_j))$. It follows that there are $i < j$ such that $enc(c_i) \sqsubseteq_{sub} enc(c_j)$, and thus $c_i \preccurlyeq c_j$. □

Monotonicity. Now, we turn towards the second condition expected from Well-Structured Transition Systems.

Lemma 10. *The transition relation $\rightsquigarrow$ is monotonic with respect to $\preccurlyeq$.*

Proof. In order to proof that $\rightsquigarrow$ is monotonic with respect to $\preccurlyeq$, we have to show that for every three configurations c_1, c_2 and c_3 from C with $c_1 \rightsquigarrow c_2$ and $c_1 \preccurlyeq c_3$, there is a fourth configuration $c_4 \in C$ with $c_3 \rightsquigarrow c_4$ and $c_2 \preccurlyeq c_4$.

To this end, let c_1, c_2 and c_3 be three configurations from C such that $c_1 \rightsquigarrow c_2$ and $c_1 \preccurlyeq c_3$. From $c_1 \preccurlyeq c_3$ it follows that the graph encoding of c_1 can be embedded into the graph encoding of c_3. Hence, by the execution of several reset transitions, one can obtain from c_3 a configuration c_3° which consists of isolated single processes and an isolated sub configuration $c_{\mathtt{sub}}$ which is isomorphic to c_1 (see Figure 11.8). Observe that the diameters of c_1 and c_3° must be equal. Moreover, as c_3°

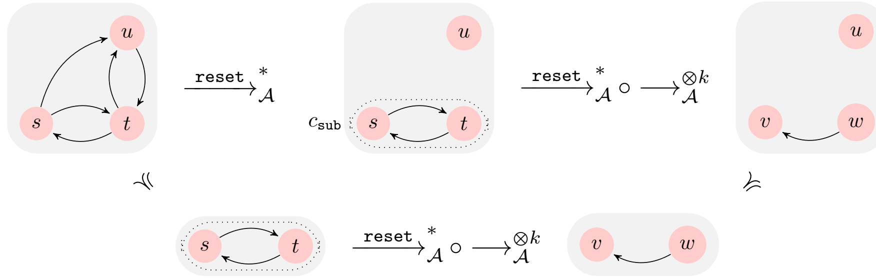


Figure 11.8: Simulation of transitions on small configurations on greater ones

contains a sub configuration which is isomorphic to c_1, it can execute the same transitions which led from c_1 to c_2. This results in a configuration c_4 such that c_4 does not violate bound k and c_2 can be embedded into c_4.

□

From state reachability to coverability. From Lemmas 9 and 10 we derive that $\mathcal{T}^{\otimes k}_{\mathtt{ext}}(\mathcal{A})$ is a well-structured transition system equipped with the well-quasi-ordering $\preccurlyeq$. We now explain that one can construct a configuration $c_{\mathtt{target}} \in C$ such that $\mathtt{target}$ is reachable in $\mathcal{T}^{\otimes k}_{\mathtt{ext}}(\mathcal{A})$ if and only if

[1] The definition of subgraph in [88] assumes the existence of a well-quasi ordering on the labels of vertices and requires that each vertex of the smaller graph has, w.r.t. the label ordering, a smaller label than the label of the corresponding vertex in the bigger graph. The equality ordering over the finite set $\Sigma_v \cup \Sigma_e = S \cup R$ of vertex labels is in fact a well-quasi ordering.

$c_{\texttt{target}}$ is coverable in $\mathcal{T}_{\texttt{ext}}^{\otimes k}(\mathcal{A})$. We consider the configuration $c_{\texttt{target}} = (\{p\}, \mathbf{s}, \mathbf{r})$ which is composed of a single process p whose state is `target` (i.e., $\mathbf{s}(p) = \texttt{target}$) and whose registers are empty (i.e., for all $r \in R$, $\mathbf{r}(p)(r)$ is undefined). Note that the upward closure $\uparrow\{c_{\texttt{target}}\} = \{c \in C \mid c_{\texttt{target}} \preccurlyeq c\}$ of $c_{\texttt{target}}$ consists exactly of those configurations in C where at least one process is in state `target`. Moreover, recall that the coverability of $c_{\texttt{target}}$ in $\mathcal{T}_{\texttt{ext}}^{\otimes k}(\mathcal{A})$ means that there is an initial configuration c_0 and some configuration c in $\uparrow\{c_{\texttt{target}}\}$ such that $c_0 \rightsquigarrow^* c$. Thus, the state reachability of `target` in $\mathcal{T}_{\texttt{ext}}^{\otimes k}(\mathcal{A})$ is equivalent to the coverability of $c_{\texttt{target}}$ in the same transition system.

We conclude the proof of Theorem 25 by showing that the remaining conditions listed in Theorem 1 hold for $\mathcal{T}_{\texttt{ext}}^{\otimes k}(\mathcal{A})$. These conditions require that (i) $\preccurlyeq$ is decidable, (ii) the non-emptiness of $\uparrow\{c\} \cap C_{\texttt{init}}$ is decidable for every configuration c, and (iii) $\mathcal{T}_{\texttt{ext}}^{\otimes k}(\mathcal{A})$ has computable predecessor bases. For (i), note that testing $c_1 \preccurlyeq c_2$ for two configurations c_1 and c_2 amounts to checking whether a graph is embeddable in another graph. It is well-known that this problem is decidable. For (ii), observe that for every configuration c, non-emptiness of $\uparrow\{c\} \cap C_{\texttt{init}}$ can be easily decided by testing that c consists of a single process in the initial state with empty registers. For (iii), we prove:

Lemma 11. *The transition system $\mathcal{T}_{ext}^{\otimes k}(\mathcal{A})$ has computable predecessor bases.*

Proof. We have to show that for every configuration $c \in C$, a basis for $\texttt{Pre}(\uparrow\{c\}) \cup \uparrow\{c\}$ is computable. Let $c = (P, \mathbf{s}, \mathbf{r}) \in C$ be a configuration. Given a transition $t \in \delta$ of $\mathcal{A}$, we use $\texttt{Pre}_t\,(\uparrow\{c\})$ to denote the set of configurations c' such that the execution of t at c' leads to some configuration $c'' \succcurlyeq c$. Let *min* be a function which for every upward closed set returns a basis. Observe that the set of bases of $\cup_{t\in\delta}\ min\,(\texttt{Pre}_t(\uparrow\{c\}) \cup \uparrow\{c\})$ is equal to the set of bases of $\texttt{Pre}(\uparrow\{c\}) \cup \uparrow\{c\}$. Hence, it suffices to show that for every $t \in \delta$, a finite basis B_t for $\texttt{Pre}_t(\uparrow\{c\}) \cup \uparrow\{c\}$ is computable.

Computing a set B_t for $\texttt{Pre}_t(\uparrow\{c\}) \cup \uparrow\{c\}$ where t corresponds to a local action, a symbol sending or a symbol reception is rather simple, because register mappings are not affected by these transitions. Conversely, transitions corresponding to create actions, ID sending, ID reception or register resetting can affect register mappings. Please keep in mind that a send transition always needs a receiving counterpart and vice-versa. In this proof, we concentrate on the computability of a finite basis B_t for $\texttt{Pre}_t(\uparrow\{c\}) \cup \uparrow\{c\}$ where t corresponds to a create action or an ID sending which is paired with a non-selective ID reception. The other cases can be handled analogously.

Let $t = (s_1, r \leftarrow \texttt{crt}(s, r'), s_2) \in \delta$. We construct B_t as the smallest set of k-strongly bounded configurations which contains c and configurations $c' = (P', \mathbf{s}', \mathbf{r}')$ such that one of the following properties is satisfied:

- Case where the creating as well as the spawned process is in c: There are two processes $p_1, p_2 \in P$ such that
  - $\mathbf{s}\,(p_1) = s_2$ and $\mathbf{s}\,(p_2) = s$,
  - $\mathbf{r}\,(p_1)\,(r) = p_2$, $\mathbf{r}\,(p_2)\,(r') = p_1$, $\mathbf{r}\,(p_2)\,(\hat{r})$ is undefined for all registers $\hat{r} \neq r'$, and for all processes $p \in P$ and registers $\hat{r} \in R$, it holds: if $\mathbf{r}\,(p)\,(\hat{r}) = p_2$, then $p = p_1$ and $\hat{r} = r$,
  - $P' = P \setminus \{p_2\}$,
  - $\mathbf{s}' = \mathbf{s}[p_1 \mapsto s_1][p_2 \mapsto \bot]$, and
  - $\mathbf{r}' = \mathbf{r}[p_2 \mapsto \bot][p_1 \mapsto \mathbf{r}(p_1)[r \mapsto \bot]]$.
- Case where only the creating process is in c: There is some process $p_1 \in P$ such that
  - $\mathbf{s}\,(p_1) = s_2$,
  - $\mathbf{r}\,(p_1)\,(r)$ is undefined,
  - $P' = P$

  - $\mathbf{s}' = \mathbf{s}[p_1 \mapsto s_1]$, and
  - $\mathbf{r}'(p) = \mathbf{r}(p)$ for all processes $p \neq p_1$ and $\mathbf{r}'(p_1)(r'') = \mathbf{r}(p_1)(r'')$ for all registers $r'' \in R$ with $r'' \neq r$.

- Case where only the created process is in c: There are processes $p_2 \in P$ and $p_1 \in P'$ such that
  - $\mathbf{s}\,(p_2) = s$,
  - $\mathbf{r}(p_2) = R_\perp$ and there is no process $p \in P$ and no regster $\hat{r} \in R$ with $\mathbf{r}\,(p)\,(\hat{r}) = p_2$,
  - $P' = (P \cup \{p_1\}) \setminus \{p_2\}$,
  - $\mathbf{s}' = \mathbf{s}[p_1 \mapsto s_1][p_2 \mapsto \perp]$, and
  - $\mathbf{r}' = \mathbf{r}$.

The case where neither the spawning, nor the created process is in c is captured by the fact that c is contained in B_t.

Now, let $(s_1, \texttt{snd}(r, m(r')), s_2) \in \delta$ be an ID sending transition from δ. We construct B_t as the smallest set of k-strongly bounded configurations which contains c and configurations $c' = (P', \mathbf{s}', \mathbf{r}')$ such that there is a non-selective ID receiving transition $(s_3, \texttt{rcv}(\star, m(r'')), s_4) \in \delta$ and one of the following properties is satisfied. For simplicity, we only consider the case where the sent process is in c:

- Case where the sending and the receiving process are in c: There are processes $p_1, p_2 \in P$ with $p_1 \neq p_2$ such that
  - $\mathbf{s}(p_1) = s_2$ and $\mathbf{s}(p_2) = s_4$,
  - $\mathbf{r}(p_1)(r) = p_2$, and
    * if $r = \texttt{self}$ then $\mathbf{r}(p_2)(r'') = p_1$,
    * otherwise $\mathbf{r}(p_1)(r')$ is defined and $\mathbf{r}(p_2)(r'') = \mathbf{r}(p_1)(r')$,
  - $P' = P$,
  - $\mathbf{s}' = \mathbf{s}[p_1 \mapsto s_1][p_2 \mapsto s_3]$, and
  - $\mathbf{r}' = \mathbf{r}[p_2 \mapsto \mathbf{r}(p_2)[r'' \mapsto \perp]]$.
- Case where the sending process is in configuration c, but not the receiving one: There are processes $p_1 \in P$ and $p_2 \in P'$ with $p_1 \neq p_2$ such that
  - $\mathbf{s}(p_1) = s_2$,
  - $\mathbf{r}(p_1)(r)$ is not defined and if $r' \neq \texttt{self}$ then $\mathbf{r}(p_1)(r')$ is defined,
  - $P' = P \cup \{p_2\}$,
  - $\mathbf{s}' = \mathbf{s}[p_1 \mapsto s_1][p_2 \mapsto s_3]$, and
  - $\mathbf{r}' = \mathbf{r}[p_1 \mapsto \mathbf{r}(p_1)[r \mapsto p_2]][p_2 \mapsto R_\perp]$.
- Case where the receiving process is in configuration c, but not the sending one: There are processes $p_2, p_3 \in P$ and $p_1 \in P'$ with $p_1 \neq p_2$ such that
  - $\mathbf{s}(p_2) = s_4$,
  - $\mathbf{r}(p_2)(r'') = p_3$ (note that due to our assumption that the sent ID must be in c, it cannot be the ID of the sender),
  - $P' = P \cup \{p_1\}$,

  - $\mathbf{s}' = \mathbf{s}[p_1 \mapsto s_1][p_2 \mapsto s_3]$, and
  - $\mathbf{r}' = \mathbf{r}[p_2 \mapsto \mathbf{r}(p_2)[r'' \mapsto \bot]][p_1 \mapsto \{r \mapsto p_2, r' \mapsto p_3\} \cup \{\hat{r} \mapsto \bot \mid \hat{r} \in R \backslash \{r, r'\}\}]$.

- Case where neither the sending, nor the receiving process is in c: There is a process $p_3 \in P$ and processes $p_1, p_2 \in P'$ with $p_1 \neq p_2$ such that
  - $P' = P \cup \{p_1, p_2\}$,
  - $\mathbf{s}' = \mathbf{s}[p_1 \mapsto s_1][p_2 \mapsto s_3]$, and
  - $\mathbf{r}' = \mathbf{r}[p_2 \mapsto R_\bot][p_1 \mapsto \{r \mapsto p_2, r' \mapsto p_3\} \cup \{\hat{r} \mapsto \bot \mid \hat{r} \in R \backslash \{r, r'\}\}]$.

□

We conclude this section by a summary of some further insights from our work [3]. The undecidability results from Corollary 8 and Theorem 23 also hold for degenerative DCA. Furthermore, by a reduction from the reachability problem for Lossy Counter Machines [183], we can show that strongly bounded reachability for degenerative DCA is non-primitive recursive.

It is obvious that for every DCA $\mathcal{A}$, its degenerative counterpart $\texttt{Deg}(\mathcal{A})$ is an over-approximation in terms of reachable states, i.e., every state reachable in the transition system of $\mathcal{A}$ is also reachable in the transition system of $\texttt{Deg}(\mathcal{A})$. We can even show that the reachable sets are equal. Moreover, for every k, the set of reachable states in the k-strongly bounded transition system of $\mathcal{A}$ is included in the set of reachable states in the k-strongly bounded transition system of $\texttt{Deg}(\mathcal{A})$. Furthermore, the set of k-strongly bounded reachable states by $\texttt{Deg}(\mathcal{A})$ is included in the set of all reachable states in the full transition system of $\mathcal{A}$. Thus, strongly bounded reachability for $\texttt{Deg}(\mathcal{A})$ can be considered as a good under-approximation of reachability for $\mathcal{A}$.

By a simple graph theoretical observation, one remarks that any k-bounded configuration of a DCA with 1 register must be $2k$-strongly bounded. Thus, using Theorem 25, we can directly conclude that for every $k \geq 1$, the problem BoundStateReach(degenerative 1-DCA,k) is decidable.

Recall that k-(strongly) bounded transition systems forbid transitions to configurations which are not k-(strongly) bounded. An interesting question is whether the undecidability results in Theorems 23 and 24 still hold if we consider DCA where all reachable configurations are k-(strongly) bounded. We call a DCA *k-safe* (or *k-strongly safe*) if every reachable configuration in the corresponding transition system is k-bounded (or k-strongly bounded). It turns out that while StateReach(k-safe DCA) and StateReach(degenerative k-safe DCA) remain in general undecidable, StateReach(k-strongly safe DCA) is decidable for every $k \geq 1$.

11.2.2.1 Dynamic Communicating Automata with Buffers

In [4], we considered *Buffered Dynamic Communicating Automata* (bDCA) which, compared to the model presented here, is closer to the original model in [47, 46] in terms of communication. In this section, we will briefly describe the differences between DCA and bDCA and summarize our results on the latter model.

While DCA-communication is rendezvous-based, the communication of bDCA-processes is asynchronous and realized through the usage of buffers. Besides finitely many registers, each process described by a bDCA is equipped with an unbounded FIFO-buffer. Like in the case of DCA, a process can create new processes and communicate with other processes whose IDs are stored in its registers. It can send messages (symbols as well as IDs) to the buffer of other processes, read messages from its own buffer and store incoming IDs in its own registers. Thus, message sending and receiving occur asynchronously. The other major difference is that instead of reset actions, processes can execute disconnect actions which detach them from the whole network. The result of a disconnect action by a process p is that the contents of all registers belonging to p or containing

p are reset, the buffer of p is emptied and the ID of p is deleted in the buffers of all other processes in the network. We also considered *lossy* bDCA, a version of bDCA in which each process can non-deterministically disconnect itself from the network.

As there is no obvious simulation of reset actions by disconnect actions or vice-versa, there is no simple reduction of the state reachability problem from one to the other model. We first showed that, in terms of reachable states, every bDCA is equivalent to its lossy counterpart. Then, we proved that the state reachability problem for (lossy) bDCA is undecidable, even in the case where only configurations with a single communication edge are allowed. Therefore, we considered a restriction on (lossy) bDCA that diminishes the power of the model with regard to buffers: we set a bound on the length of buffers. However, even if the capacity of the buffers is restricted to at most one message, the problem remains undecidable. The undecidability result still holds if we bound simple paths in communication graphs.

Then, we concentrated on the *strongly bounded* reachability problem for bDCA with bounded buffers. The definition of strongly bounded configurations takes, besides communication edges, also edges into account which come from the containment of IDs in buffers. While strongly bounded reachability for bDCA with bounded buffers is still undecidable, we obtained decidability in the case of lossy bDCA. Finally, we proved the decidability of the strongly bounded reachability problem for full bDCA in the case that communication graphs are acyclic. Such a restriction was not considered for DCA.

11.3 Process Register Automata

In this section, we define Process Register Automata (PRA) and analyze their non-emptiness problem. The model is basically a restriction of Data Multi-Pushdown Automata [45] where stacks are skipped. Recall that a DCA describes a single template according to which each process of the designed system behaves. Compared to DCA, PRA provide a more global view to systems. A PRA is a finite automaton equipped with finitely many registers in which process IDs can be stored. In contrast to DCA, these registers do not belong to single processes, but are some kind of global system registers. Only processes which are stored in some registers are able to participate in actions. A PRA has only two kinds of transitions: create transitions and send transitions. A create transition enforces that a process contained in some register creates a new process which is again stored in some register. A send transition enforces that a process in some register sends some message to a process in some other register. Besides a message symbol, a message can contain a list of processes which are currently stored in registers. Thus, although a PRA is able to produce unboundedly many processes during a run, at any time the number of processes which are able to participate in actions is bounded by the number registers.

Now, we define PRA formally. A PRA $\mathcal{A} = (A, R, r_0, S, s_0, \delta, F)$ consists of a finite *message alphabet* A, a finite set R of *registers* with *initial register* $r_0 \in R$, a finite set S of *states* with *initial state* $s_0 \in S$, a finite set $F \subseteq S$ of *accepting states* and a set δ of transitions of the form $(s_1, \texttt{act}, s_2)$ where s_1 and s_2 are states from S and $\texttt{act}$ is a symbolic action from the set $\texttt{Actions}(A, R)$ as defined in Section 11.1.

We demonstrate the semantics of PRA by an example:

Example 18. We construct a PRA modeling the client-and-server protocol from Chapter 2. The PRA has four registers, r_0 to r_3, and uses the message symbols `serv`, `req`, `ack` and `noti`. The first two symbols have arity 1 and the latter two have arity 0. Register r_0 is reserved for the root process, register r_1 is reserved for the server process and the remaining two registers are kept for client processes. The message symbols `serv`, `req` and `ack` are used for the same purpose as in the DCA implementation in Example 16. The additional symbol `noti` is used by clients to inform the

root process that there are no more requests to be sent to the server. The PRA is depicted in Figure 11.9.

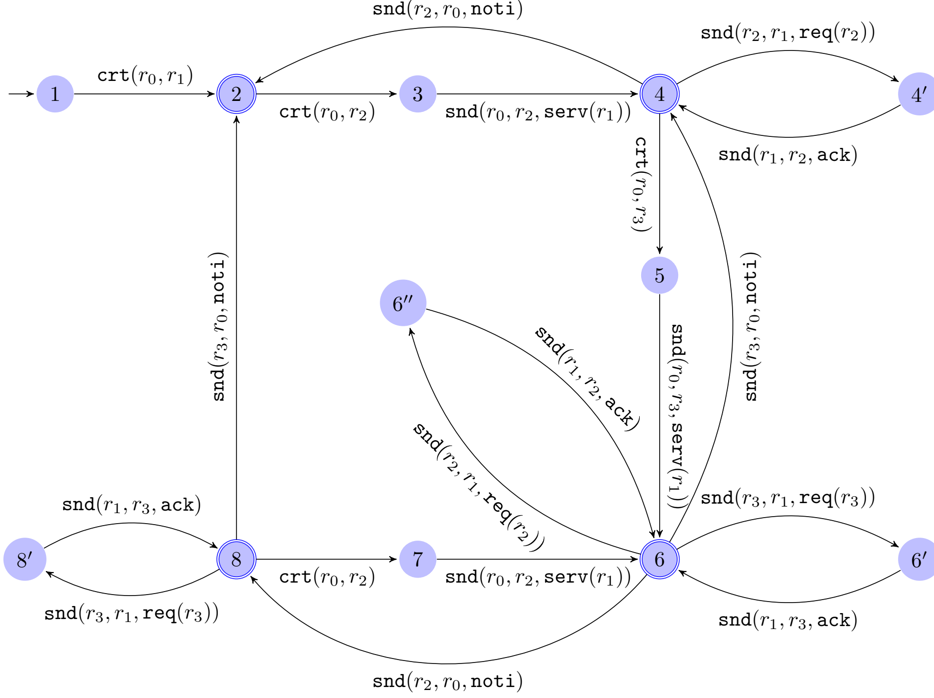


Figure 11.9: An example PRA modeling the client-and-server protocol from Chapter 2

At the beginning, all registers except r_0, containing the root process, are empty. The overall idea is that the root process first creates a server whose ID is stored in register r_1 and then creates client processes to be stored in registers r_2 and r_3. Each client receives from the root process the server ID is stored in register r_1. By means of this ID and the message symbols `req` and `ack`, the clients can communicate with the server. Moreover, they can send to the root process a notification that no further communication with the server is needed. After the reception of such a notification from a client from some register r_i with $i \in \{2, 3\}$, the root process creates a new client which is stored in r_i. Thus, the ID of the former client in r_i is overwritten and cannot participate in actions anymore.

State 2 represents a situation where both client registers are empty or have already informed the root process that they have stopped communication with the server. In state 4, register r_2 is occupied by a client sending requests to the server and register r_3 is either empty or the corresponding client has already announced that it has stopped sending requests. In state 6, both clients of registers r_2 and r_3 are still active and sending requests. Finally, state 8 stands for a situation where only the client in r_3 is communicating with the server. □

Configurations

Before defining the configurations of a PRA $\mathcal{A} = (A, R, r_0, S, s_0, \delta, F)$, we introduce some notations. Let $\texttt{crt}(r, r')$ and $\texttt{snd}(r, r', m(r_1, \ldots, r_{\texttt{ar}(m)}))$ be, respectively, a symbolic create and a symbolic send action with $r, r', r_1, \ldots, r_{\texttt{ar}(m)} \in R$. Given a partial register assignment $\nu \in$

$[R \rightharpoonup \mathbb{P}]$ defined on all registers $r, r', r_1, \ldots, r_{\mathtt{ar}(m)}$, we set $\nu(\mathtt{crt}(r,r')) = \mathtt{crt}(\nu(r), \nu(r'))$ and $\nu(\mathtt{snd}(r, r', m(r_1, \ldots, r_{\mathtt{ar}(m)}))) = \mathtt{snd}(\nu(r), \nu(r'), m(\nu(r_1), \ldots, \nu(r_{\mathtt{ar}(m)})))$. A configuration of $\mathcal{A}$ consists of three components s, ν and E where $s \in S$ is the current state of the system, $\nu \in [R \rightharpoonup \mathbb{P}]$ describes the current contents of the registers and E is the set of all processes created so far during the system run. Thus, we define the set of configurations of $\mathcal{A}$ as $\mathtt{Conf}(\mathcal{A}) = S \times [R \rightharpoonup \mathbb{P}] \times 2^{\mathbb{P}}$.

The transition relation on configurations

We say that a create transition $(s_1, \mathtt{crt}(r, r'), s_2) \in \delta$ is *enabled* at some configuration $c = (s, \nu, E)$ if $s = s_1$ and $\nu(r)$ is defined. Likewise, a send transition $(s_1, \mathtt{snd}(r, r', m(r_1, \ldots, r_{\mathtt{ar}(m)})), s_2) \in \delta$ is enabled at c if $s = s_1$ and $\nu(\hat{r})$ is defined for every $\hat{r} \in \{r, r', r_1, \ldots, r_{\mathtt{ar}(m)}\}$. We will see that the execution of an enabled transition at a configuration can lead to infinitely many possible *successor configuration*. Now, we define a transition relation $\rightarrow_{\mathcal{A}}$ on configurations of $\mathcal{A}$. For two configurations $c, c' \in \mathtt{Conf}(\mathcal{A})$ with $c = (s, \nu, E)$ and $c' = (s', \nu', E')$, we have $c \rightarrow_{\mathcal{A}} c'$ if one of the following conditions holds:

- There is a transition $(s_1, \mathtt{crt}(r, r'), s_2) \in \delta$ enabled at c and there is a process $p' \in \mathbb{P}$ with $p' \notin E$ such that (i) $\nu' = \nu[r' \mapsto p']$, (ii) $E' = E \uplus \{p'\}$, and (iii) $s' = s_2$.
- There is a transition $(s_1, \mathtt{snd}(r, r', m(r_1, \ldots, r_{\mathtt{ar}(m)})), s_2) \in \delta$ enabled at c such that (i) $\nu' = \nu$, (ii) $E' = E$, and (iii) $s' = s_2$.

If a transition $c \rightarrow_{\mathcal{A}} c'$ with $c' = (s', \nu', E')$ is caused by a δ-transition $(s_1, \mathtt{act}, s_2)$, we also write $c \xrightarrow{\nu'(\mathtt{act})}_{\mathcal{A}} c'$. Observe that while a send transition leads to exactly one successor configuration, a create transition gives rise to infinitely many successor configurations, because there are infinitely many processes in $\mathbb{P}$ which can be chosen as new process.

The transition system, runs and traces

A configuration $c = (s, \nu, E)$ is called *initial* if $s = s_0$, $E = \{p\}$ for some arbitrary process p and $\nu = \{r_0 \mapsto p\}$. It is called *accepting* if $s \in F$. Thus, the transition system of $\mathcal{A}$ is defined as $\mathcal{T}(\mathcal{A}) = (\mathtt{Conf}(\mathcal{A}), \mathtt{Conf}_{\mathtt{init}}(\mathcal{A}), \longrightarrow_{\mathcal{A}}, \mathtt{Conf}_{\mathtt{acc}}(\mathcal{A}))$ where $\mathtt{Conf}_{\mathtt{init}}(\mathcal{A})$ and $\mathtt{Conf}_{\mathtt{acc}}(\mathcal{A})$ are, respectively, the set of initial and the set of accepting configurations. A sequence $\tau = c_0 \xrightarrow{\mathtt{act}_1}_{\mathcal{A}} \ldots \xrightarrow{\mathtt{act}_n}_{\mathcal{A}} c_n$ (where transitions are labelled by concrete actions over $\mathbb{P}$) is called a *run* of $\mathcal{A}$ if c_0 is an initial configuration. Observe that each register assignment within a run must be injective which means that PRA cannot contain the same ID in two different registers. A run which ends up in an accepting configuration is called *accepting*. Like in the case of DCA, we define the traces of PRA as data words over the proposition set $\mathtt{Prop}^{\mathcal{A}}_{\mathtt{act}}$ and the attribute set $\mathtt{Attr}^{\mathcal{A}}_{\mathtt{act}}$. For an accepting run $\tau = c_0 \xrightarrow{\mathtt{act}_1}_{\mathcal{A}} \ldots \xrightarrow{\mathtt{act}_n}_{\mathcal{A}} c_n$ of $\mathcal{A}$, we call the data word $\mathit{trace}(\tau) = \mathtt{dwrep}(\mathtt{act}_1) \ldots \mathtt{dwrep}(\mathtt{act}_n)$ the *trace* of τ. A trace, resulting from an accepting run, is called a *trace of* $\mathcal{A}$ if its length is at least 1, i.e. the data word contains at least one position. The language $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ consists of all traces of $\mathcal{A}$.

Example 19. Figure 11.3 shows a trace of the PRA designed in Example 18. The ID 1 identifies the root process, the ID 2 belongs to the server process and the IDs 3 to 5 identify clients. First, the root process creates the server and then it creates client 3. Afterwards, the root process sends the server ID to client 3. Having communicated with the server via a request and an acknowledgement, client 3 informs the root process (by the message symbol `noti`) that it stops the communication with the server. Thereafter, the root process creates two new clients 4 and 5. Finally, client 5 sends a request to the server and receives an acknowledgement. □

	crt	crt	snd serv	snd req	snd ack	snd noti	crt	snd serv	crt	snd serv	snd req	snd ack
creator	1	1					1		1			
created	2	3					4		5			
sender			1	3	2	3		1		1	5	2
receiver			3	2	3	1		4		5	2	5
$\mathtt{mpar}_1$			2	3				2		2	5	

Figure 11.10: A trace of the PRA given in Example 18

The symbolic behaviour of PRA

When dealing with the algorithmic properties of a PRA, we will often make use of its *symbolic* transition system. A symbolic transition system contains symbolic configurations, runs and traces which are not defined over processes, but registers. The distinction between usual transition systems and symbolic ones is analogous to the distinction between concrete and symbolic actions. To avoid confusion, we will often use the prefix “concrete” for usual transition systems, configurations, runs and traces.

We first give the formal definition of symbolic configurations of a PRA $\mathcal{A} = (A, R, r_0, S, s_0, \delta, F)$. A symbolic configuration $sc = (s, D)$ consists of a state s from S and a set $D \subseteq R$ of registers which, intuitively, represents the domain of register assignments of possible concrete instantiations of sc. The set of all symbolic configurations of $\mathcal{A}$ is denoted as $\mathtt{SConf}(\mathcal{A})$. The conditions for the execution of a transition at a symbolic configuration are defined similarly to the case of concrete configurations: A create transition $(s_1, \mathtt{crt}(r, r'), s_2)$ is *enabled* at a symbolic configuration $sc = (s, D)$ if $s = s_1$ and $r \in D$. A send transition $(s_1, \mathtt{snd}(r, r', m(r_1, \dots, r_{\mathtt{ar}(m)})), s_2)$ is *enabled* at sc if $s = s_1$ and $r, r', r_1, \dots, r_{\mathtt{ar}(m)} \in D$. The definition of the transition relation $\rightarrow^s_{\mathcal{A}}$ on symbolic configurations is much more simpler than in the concrete case. The execution of a create transition $(s_1, \mathtt{crt}(r, r'), s_2)$ at a symbolic configuration $sc = (s, D)$ leads to the symbolic configuration $(s_2, D \cup \{r'\})$. When a send transition $(s_1, \mathtt{snd}(r, r', m(r_1, \dots, r_{\mathtt{ar}(a)})), s_2)$ is executed at sc, we obtain the successor configuration (s_2, D). If a symbolic configuration sc' results from sc by the execution of a transition $(s_1, \mathtt{act}, s_2)$, we write $sc \xrightarrow{\mathtt{act}}^s_{\mathcal{A}} sc'$. A symbolic configuration $sc = (s, D)$ is called *initial* if $s = s_0$ and $D = \{r_0\}$, it is accepting if $s \in F$. The sets of initial and accepting symbolic configurations of $\mathcal{A}$ are denoted as $\mathtt{SConf}_{\mathtt{init}}(\mathcal{A})$ and $\mathtt{SConf}_{\mathtt{acc}}(\mathcal{A})$, respectively. Thus, we obtain the symbolic transitions system $\mathcal{ST}(\mathcal{A}) = (\mathtt{SConf}(\mathcal{A}), \mathtt{SConf}_{\mathtt{init}}(\mathcal{A}), \rightarrow^s_{\mathcal{A}}, \mathtt{SConf}_{\mathtt{acc}}(\mathcal{A}))$ for $\mathcal{A}$.

A sequence $\theta = sc_0 \xrightarrow{\mathtt{act}_1}^s_{\mathcal{A}} \dots \xrightarrow{\mathtt{act}_n}^s_{\mathcal{A}} sc_n$ of symbolic configurations and actions is called a *symbolic run* of $\mathcal{A}$ if sc_0 is initial. A symbolic run is *accepting* if the last configuration is accepting. *Traces* of symbolic runs are usual words with propositions (and without data values) that signalize at each position which symbolic action is currently executed with which parameters. For a message alphabet A and a register set R, we define $\mathtt{Prop}^{A,R}_{\mathtt{sact}} = \{\mathtt{snd}, \mathtt{crt}\} \cup \{[\mathtt{par}, \mathtt{par}(\mathtt{act})] \mid \mathtt{act}$ is an action in $\mathtt{Actions}(A, R)$ with parameter $\mathtt{par}\}$ as the set of propositions for symbolic traces. Recall from Section 11.1 that the parameters of create actions are $\mathtt{creator}$ and $\mathtt{created}$ and those of send actions with some message symbol m are $\mathtt{sender}$, $\mathtt{receiver}$, $\mathtt{msym}$ and $\mathtt{mpar}_i$ for every $i \in \{1, \dots, \mathtt{ar}(m)\}$. A symbolic action $\mathtt{act}$ within a trace is represented by a word position $\mathtt{wrep}(\mathtt{act})$ carrying exactly the propositions in $\{p\} \cup \{[\mathtt{par}, \mathtt{par}(\mathtt{act})] \mid \mathtt{par}$ is a parameter for $\mathtt{act}\}$ where $p = \mathtt{snd}$ if $\mathtt{act}$ is a send action and $p = \mathtt{crt}$, otherwise. The symbolic trace of an accepting symbolic run $\theta = sc_0 \xrightarrow{\mathtt{act}_1}^s_{\mathcal{A}} \dots \xrightarrow{\mathtt{act}_n}^s_{\mathcal{A}} sc_n$ is the word $strace(\theta) = \mathtt{wrep}(\mathtt{act}_1) \dots \mathtt{wrep}(\mathtt{act}_n)$. The set of symbolic traces of $\mathcal{A}$ consists of all symbolic traces which are induced by symbolic runs of $\mathcal{A}$ and are of length at least one. Finally, the symbolic language $\mathcal{SL}(\mathcal{A})$ of $\mathcal{A}$ consists of all symbolic

traces of $\mathcal{A}$.

Example 20. We give in Figure 11.11 the first four positions of the symbolic trace corresponding to the concrete trace in Figure 11.10 of the PRA in Figure 11.9. □

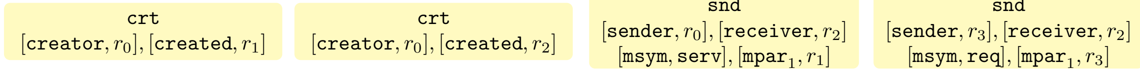


Figure 11.11: A prefix of the symbolic trace associated with the concrete trace in Figure 11.10 of the PRA in Figure 11.9

We define a straightforward mapping from concrete runs to their symbolic counterparts. Intuitively, from a concrete run we get its corresponding symbolic run by skipping the processes in configurations and replacing concrete actions by corresponding symbolic actions from δ. We first define a mapping $\mathtt{symb}$ from concrete configurations to symbolic ones and then extend the mapping to runs. For a concrete configuration $c = (s, \nu, E)$, we set $\mathtt{symb}(c) = (s, \mathtt{dom}(\nu))$. Given a concrete run

$$\tau = c_0 \xrightarrow{\nu_1(\mathtt{act}_1)}_{\mathcal{A}} \dots \xrightarrow{\nu_n(\mathtt{act}_n)}_{\mathcal{A}} c_n$$

with $c_i = (s_i, \nu_i, E_i)$ for every $i \in \{0, \dots, n\}$, we define

$$\mathtt{symb}(\tau) = \mathtt{symb}(c_0) \xrightarrow{\mathtt{act}_1}{}^{s}_{\mathcal{A}} \dots \xrightarrow{\mathtt{act}_n}{}^{s}_{\mathcal{A}} \mathtt{symb}(c_n).$$

Observe that it easily follows from the definition of symbolic runs τ that $\mathtt{symb}(\tau)$ is well-defined. The following observation is straightforward:

Observation 11. Let $\mathcal{A} = (A, R, r_0, S, s_0, \delta, F)$ be a PRA. For every accepting concrete run τ of $\mathcal{A}$, $\mathtt{symb}(\tau)$ is an accepting symbolic run of $\mathcal{A}$. Likewise, for every accepting symbolic run θ of $\mathcal{A}$, there is an accepting concrete run τ of $\mathcal{A}$ with $\mathtt{symb}(\tau) = \theta$.

11.3.1 Non-Emptiness

We show that the non-emptiness problem for PRA is NP-complete. The idea of the proof originates from our paper [46].

Theorem 26. *The non-emptiness problem for PRA is* NP*-complete.*

Proof. Let $\mathcal{A} = (A, R, r_0, S, s_0, \delta, F)$ be a PRA. We first consider the upper bound of the non-emptiness problem. It follows by definition that $\mathcal{A}$ is non-empty if and only if there is an accepting concrete run of $\mathcal{A}$ containing at at least one concrete action. Due to Observation 11, the latter holds if and only if there is an accepting symbolic run of $\mathcal{A}$ containing at at least one symbolic action. Thus, we can reduce the non-emptiness problem for $\mathcal{A}$ to the problem of finding a witness sequence

$$(s_0, D_0)(s_0, \mathtt{act}_1, s_1)(s_1, D_1)(s_1, \mathtt{act}_2, s_2) \dots (s_{n-1}, D_{n-1})(s_{n-1}, \mathtt{act}_n, s_n)(s_n, D_n)$$

of symbolic configurations and transitions such that

(a) $n \geq 1$, $D_0 = \{r_0\}$, $s_n \in F$, and

(b) for every i with $1 \leq i \leq n$,

- if $\mathtt{act}_i$ is a create action, then, $\mathtt{creator}(\mathtt{act}_i) \in D_{i-1}$ and $D_i = D_{i-1} \cup \{\mathtt{creator}(\mathtt{act}_i)\}$, and
- if $\mathtt{act}_i$ is a send action, then, $\mathtt{par}(\mathtt{act}_i) \in D_{i-1}$ for all parameters $\mathtt{par}$ of $\mathtt{act}_i$ besides $\mathtt{msym}$ and $D_i = D_{i-1}$.

First of all, note that the sets D_i are monotonically increasing. Furthermore, unlike accepting paths in NFA, the elimination of sub sequences leading from a state to itself within a (symbolic) accepting run does not necessarily result again in a run, since a loop may yield a configuration where more transitions are enabled than before. The reason is that the set of enabled transition at a configuration (s, D) also depends on the set D of registers and a loop can enlarge this set. However, taking the same loop twice does not bring any benefit. Thus, if there is a witness sequence for $\mathcal{A}$, then there is one of length polynomial in the size of $\mathcal{A}$. Furthermore, whether a given sequence constitutes a witness sequence can be tested in at most polynomial time: Whether the first state is initial, the states of consecutive transitions and configurations comply with each other and condition (a) holds can be assured in polynomial time. Moreover, for the test of condition (b), a comparison between all consecutive three tuples in the sequence suffices. Hence, the non-emptiness problem is in NP.

We show the lower bound by a reduction from the NP-complete problem 3-CNF-SAT. First, let us briefly recall this problem. Let $V = \{A_1, \ldots, A_n\}$ be a finite set of propositional variables. A variable A or its negation $\neg A$ is called a *literal*. A disjunction of literals constitutes a *clause*. A formula over V is a Boolean combination of variables in V. Given a formula $\varphi = \bigwedge_{i=1}^{k}(l_1^i \vee l_2^i \vee l_3^i)$ in conjunctive normal form where each l_j^i is a literal, the problem 3-CNF-SAT asks whether there is a truth assignment $\lambda \in [V \mapsto \{\mathtt{true}, \mathtt{false}\}]$ for the variables in V satisfying φ.

We reduce 3-CNF-SAT to the non-emptiness problem for PRA as follows. Given a formula $\varphi = \bigwedge_{i=1}^{k}(l_1^i \vee l_2^i \vee l_3^i)$ over V, we construct a PRA $\mathcal{A}_\varphi$ over a single message symbol m of arity 0. The automaton $\mathcal{A}_\varphi$ contains the register set $R = \{r_0\} \cup \{r_A, \overline{r_A} \mid A \in V\}$ where each register r_A represents the propositional variable A and each register $\overline{r_A}$ corresponds to $\neg A$. The automaton is shown in Figure 11.12. In the picture, for every i, j with $1 \leq i \leq k$ and $1 \leq j \leq 3$, register r_j^i represents r_A if $l_j^i = A$ and it represents $\overline{r_A}$ if $l_j^i = \neg A$.

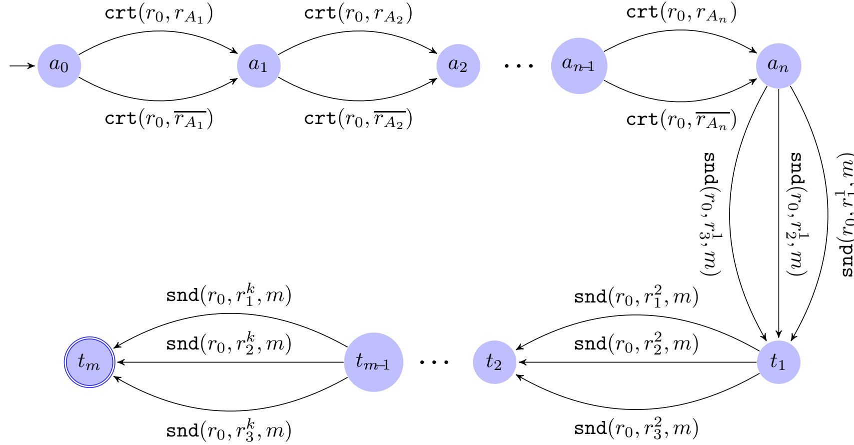


Figure 11.12: Encoding 3-CNF-SAT into the non-emptiness problem for PRA

The behaviour of the automaton is separated into an *assignment phase* and a *test phase*. The assignment phase, which starts in the initial state a_0 and ends in state a_n, constructs a truth

assignment for the variables in V and the test phase, starting in state a_n and continuing until the final state t_m, checks whether this truth assignment satisfies all clauses in φ. The more detailed description of the automaton is as follows. In the assignment phase, the root process in r_0 chooses for each ℓ with $1 \leq \ell \leq n$, non-deterministically one of the registers r_{A_ℓ} and $\overline{r_{A_\ell}}$, spawns a new process and stores it in the chosen register. Each register assignment ν obtained at the end of the assignment phase corresponds to a unique truth assignment λ for V in the sense that for every $A \in V$, it holds $\lambda(A) = \texttt{true}$ if and only if $\nu(r_A)$ is defined. In the second phase, the root process chooses for each i with $1 \leq i \leq k$, non-deterministically a register $r^i_j \in \{r^i_1, r^i_2, r^i_1\}$ and attempts to send a message to the process of that register. Obviously, the message can only be sent if the input of register r^i_j is defined by the register assignment at the end of the assignment phase. As register assignments correspond to truth assignments, the sending of a message to some r^i_j means that the truth assignment defined in the assignment phase satisfies the clause $(l^i_1 \vee l^i_2 \vee l^i_1)$.

The correctness of the construction can be shown easily. To this end, assume that $\mathcal{A}_\varphi$ is non-empty. This means that there is a trace $w = w_1 \dots w_n w_{n+1} \dots w_{n+m}$ of $\mathcal{A}_\varphi$ induced by an accepting run $\tau = c_0 \xrightarrow{w_1}_{\mathcal{A}_\varphi} \dots \xrightarrow{w_n}_{\mathcal{A}_\varphi} c_n \xrightarrow{w_{n+1}}_{\mathcal{A}_\varphi} \dots \xrightarrow{w_{n+m}}_{\mathcal{A}_\varphi} c_{n+m}$. By construction of $\mathcal{A}_\varphi$, the register assignment ν_n in c_n must have the property that for every ℓ with $1 \leq \ell \leq n$, $\nu_n(r_{A_\ell})$ is defined if and only if $\nu(\overline{r_{A_\ell}})$ is undefined. We define a truth assignment λ on V as follows: for every ℓ with $1 \leq \ell \leq n$, $\lambda(A_\ell) = \texttt{true}$ if and only if $\nu(r_{A_\ell})$ is defined. Due to the construction of the test phase of $\mathcal{A}_\varphi$, for every i with $1 \leq i \leq k$, there must be some register r^i_j such that $\nu(r^i_j)$ is defined. If $r^i_j = r_A$ for some variable A, then, by definition, $\lambda(A) = \texttt{true}$ which means that $(l^i_1 \vee l^i_2 \vee l^i_1)$ is satisfied by λ. Analogously, if $r^i_j = \overline{r_A}$, then, by definition, $l^i_j = \neg A$ and $\lambda(A) = \texttt{false}$ from which it again follows that λ satisfies $(l^i_1 \vee l^i_2 \vee l^i_1)$. Hence, λ satisfies φ.

Assume now that φ is satisfiable. By definition, there is a truth assignment λ on V satisfying φ. In particular, for every clause $C^i = l^i_1 \vee l^i_2 \vee l^i_3$, there must be a literal l^i_j satisfied by λ. We now show that there must be an accepting run for $\mathcal{A}_\varphi$. Let $\tau_1 = c_0 \xrightarrow{\texttt{act}_1}_{\mathcal{A}_\varphi} \dots \xrightarrow{\texttt{act}_n}_{\mathcal{A}_\varphi} c_n$ be an (incomplete) run of $\mathcal{A}_\varphi$ where each c_ℓ with $1 \leq \ell \leq n$ results from $c_{\ell-1}$ by the execution of $\texttt{crt}(r_0, r_{A_\ell})$ if $\lambda(A_\ell) = \texttt{true}$ and by the execution of $\texttt{crt}(r_0, \overline{r_{A_\ell}})$, otherwise. Hence, the register assignment ν in c_n must correspond to the truth assignment λ. Consequently, for every clause C^i, there must be a *witness literal* l^i_w and a variable A_ℓ such that either $l^i_w = A_\ell$ and $\nu(r_{A_\ell})$ is defined or $l^i_w = \neg A_\ell$ and $\nu(\overline{r_{A_\ell}})$ is defined. Thus, the run τ_1 can be extended to an accepting run $c_0 \xrightarrow{\texttt{act}_1}_{\mathcal{A}_\varphi} \dots \xrightarrow{\texttt{act}_n}_{\mathcal{A}_\varphi} c_n \xrightarrow{\texttt{act}_{n+1}}_{\mathcal{A}_\varphi} \dots \xrightarrow{\texttt{act}n+m}_{\mathcal{A}_\varphi} c_{n+m}$ where each c_i with $n+1 \leq i \leq k$ results from c_{i-1} by the execution of $\texttt{snd}(r_0, r_{A_\ell}, m)$ if $l^i_w = A_\ell$, and by the execution of $\texttt{snd}(r_0, \overline{r_{A_\ell}}, m)$ if $l^i_w = \neg A_\ell$. It follows that the language of $\mathcal{A}$ must be non-empty.

Note that the size of $\mathcal{A}$ is polynomial in the size of φ. Thus, 3-CNF-SAT is polynomially reducible to the satisfiability problem for PRA. We conclude that the latter problem is NP-hard and, together with the upper bound, NP-complete. □

11.4 Branching High-Level Message Sequence Charts

In this section, we will define Branching High-Level Message Sequence Charts (BHMSCs) and analyze the non-emptiness and the executability problem for this model. Unlike DCA and PRA which generate traces based on a liner order, the structures produced by BHMSCs are Message Sequence Charts (MSCs) which are based on partially ordered sets of events. First, we will introduce MSCs, then, we will explain the concatenation of MSCs. Finally, we will define BHMSCs. Non-emptiness and executability of BHMSCs will be studied in the two following subsections.

Message Sequence Charts. A single MSC describes the interaction between finitely many processes within a dynamic system. Message Sequence Charts have a convenient graphical representation. Before defining MSCs formally, we present an example MSC:

Example 21. Figure 11.13 depicts an MSC describing a possible execution of our introductory example system in Chapter 2. The natural numbers represent process IDs. A vertical line below an ID models the lifetime of the corresponding process. Horizontal arrows with a single arrow head stand for send actions, those with two heads describe process creation. The starting event of the initial process is modeled by a small circle.

In the presented MSC, the initial process with ID 1 models the root process of our example system. It first creates the server with ID 2 and then a client with ID 3. Thereafter, it sends to the client the server ID along with the message symbol `serv`. Then, the client sends to the server a request and its own ID. The execution ends with an acknowledgement from server to client.

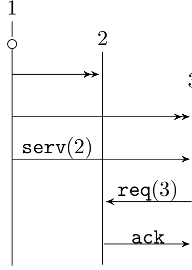


Figure 11.13: An MSC

□

We introduce some notations helpful for the formal definition of MSCs. The set of *(action) types* for MSCs is defined as $\mathcal{T} = \{\mathtt{start}, \mathtt{crt}, \mathtt{snd}, \mathtt{rec}\}$ where `start` stands for the initial events of processes and the other types correspond, respectively, to create, send and receive actions. For some finite set E of *events* and a set N of process names, let $\lambda \in [E \mapsto N \times \mathcal{T}]$ be a mapping assigning to every event a corresponding process name along with an action type. For a type $\theta \in \mathcal{T}$, we define $E^\lambda_\theta = \{e \in E \mid \lambda(e) \in N \times \{\theta\}\}$ as the subset of E consisting of all events assigned to type θ. Likewise, for a process name $n \in N$, we define $E^\lambda_n = \{e \in E \mid \lambda(e) \in \{n\} \times \mathcal{T}\}$ as the subset which consists of all events assigned to n. If λ is clear from the context, we skip the superscripts in E^λ_θ and E^λ_n.

Though the MSC in Figure 11.13 contains concrete process IDs, we will also consider MSCs containing registers. Therefore, we define MSCs over an abstract set of process names. An MSC M over some message set A and a set N of process names is a tupe $(E, \lhd, \lambda, \mu)$ where E is a nonempty finite set of *events*, $\lhd$ is the *edge relation* partitioned into $\lhd = \lhd_{\mathtt{proc}} \uplus \lhd_{\mathtt{crt}} \uplus \lhd_{\mathtt{msg}}$ of *process edges*, *create edges* and *message edges*, $\lambda \in [E \mapsto N \times \mathcal{T}]$ assigns to every event a corresponding process name and a type and $\mu \in [\lhd_{\mathtt{msg}} \mapsto A(N)]$ labels every message edge in M by a message. The MSC M has to fulfill the following conditions:

- The reflexive and transitive closure $\lhd^*$ of $\lhd$ constitutes a partial order on E with a unique minimal element $\mathtt{init}(M) \in E_{\mathtt{start}}$. As demonstrated in Figure 11.13, we symbolize $\mathtt{init}(M)$ in the graphical representation of M by a circle without any incoming or outgoing message or create edge.
- The relation $\lhd_{\mathtt{proc}}$ is a subset of $\bigcup_{n \in N}(E_n \times E_n)$ such that for every $n \in N$, $\lhd_{\mathtt{proc}} \cap (E_n \times E_n)$ is a total order on E_n, called the *process relation* for n.
- The set $E_{\mathtt{start}}$ of start events consists of all events e such that there is no event e' with $e' \lhd_{\mathtt{proc}} e$.

- The relations $\lhd_{\mathtt{crt}}$ and $\lhd_{\mathtt{msg}}$ are subsets of $\bigcup_{n,m\in N, n\neq m}(E_n \times E_m)$.
- The relation $\lhd_{\mathtt{crt}}$ induces a bijection between $E_{\mathtt{crt}}$ and $E_{\mathtt{start}} \setminus \{\mathtt{init}(M)\}$, i.e., for every create event, there is exactly one start event which is not $\mathtt{init}(M)$, and vice-versa.
- Similarly, $\lhd_{\mathtt{msg}}$ induces a bijection between $E_{\mathtt{snd}}$ and $E_{\mathtt{rec}}$ and satisfies the following FIFO-condition: for all process names n and m and all events $e_1, e_2 \in E_n$ and $e'_1, e'_2 \in E_m$ with $e_1 \lhd_{\mathtt{msg}} e'_1$ and $e_2 \lhd_{\mathtt{msg}} e'_2$, it holds $e_1 \lhd^*_{\mathtt{proc}} e_2$ if and only if $e'_1 \lhd^*_{\mathtt{proc}} e'_2$.

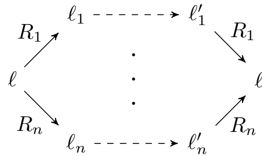
The set of all MSCs over A and N is denoted as $\mathbb{MSC}(A, N)$. We call two MSCs from $\mathbb{MSC}(A, N)$ *equivalent* if one can obtain one from the other by renaming processes. The equivalence class of an MSC M with respect to renaming processes is denoted as $[M]$. For a set $\mathcal{L}$ of MSCs, we set $[\mathcal{L}] = \bigcup_{M\in\mathcal{L}}[M]$. The set $\mathcal{L}$ is called *closed* if $\mathcal{L} = [\mathcal{L}]$.

Partial Message Sequence Charts. Given an MSC $M = (E, \lhd, \lambda, \mu)$, we call a (with respect to $\lhd^*$) downward closed subset $E' \subseteq E$ *complete with respect to message and create edges* if for all $(e, e') \in \lhd_{\mathtt{crt}} \cup \lhd_{\mathtt{msg}}$, we have that $e' \in E'$ implies $e \in E'$. If E' is downward closed with respect to $\lhd^*$ and complete with respect to message and create edges, the restriction of M to E' is called a *partial MSC*. Note that a partial MSC does not have to contain a *unique* minimal element. All notions and notations for (full) MSCs carry over to partial ones. The set of all partial MSCs over A and N is denoted by $\mathbb{PMSC}(A, N)$.

Branching High-Level Message Sequence Charts. We introduce some further notations. Given a (partial) MSC $M = (E, \lhd, \lambda, \mu) \in \mathbb{PMSC}(A, N)$, we denote by $\mathtt{MsgPar}(M)$ the set of process names occurring as parameters in messages in M, i.e., $\mathtt{MsgPar}(M) = \{n \mid \text{there is } m(n_1, \ldots, n_k) \in \mu(\lhd_{\mathtt{msg}}) \text{ with } n \in \{n_1, \ldots, n_k\}\}$. For every process name $n \in N$ with $E_n \neq \emptyset$, we denote the single minimal and the single maximal event with respect to $\lhd_{\mathtt{proc}} \cap E_n$ by $\mathtt{min}_n(M)$ and $\mathtt{max}_n(M)$, respectively. We further define $\mathtt{Min}(M) = \{\mathtt{min}_n(M) \mid n \in N \text{ and } E_n \neq \emptyset\}$. The set $\mathtt{Max}(M)$ is defined analogously. We set $\mathtt{Pids}(M) = \{n \in N \mid E_n \neq \emptyset\}$ and define the set $\mathtt{Free}(M)$ of *free* process names in M as $\{n \in \mathtt{Pids}(M) \mid E_{\mathtt{start}} \cap E_n = \emptyset\}$. The set $\mathtt{Bnd}(M)$ of *bounded* process names in M is defined, as expected, as $\mathtt{Pids}(M) \setminus \mathtt{Free}(M)$. Next, we define the *concatenation* of two partial MSCs. Two partial MSCs $M = (E, \lhd, \lambda, \mu)$ and $M' = (E', \lhd', \lambda', \mu')$ can be concatenated to a new partial MSC $M \circ M'$ if $\mathtt{Pids}(M) \cap \mathtt{Bnd}(M') = \emptyset$. To explain it visually, $M \circ M'$ is obtained by connecting the process edges of the same process names. Formally, $M \circ M'$ is defined as $(\widehat{E}, \widehat{\lhd}, \widehat{\lambda}, \widehat{\mu})$ where $\widehat{E} = E \uplus E'$, $\widehat{\lhd}_{\mathtt{proc}} = \lhd_{\mathtt{proc}} \cup \lhd'_{\mathtt{proc}}$, $\widehat{\lhd}_{\mathtt{crt}} = \lhd_{\mathtt{crt}} \cup \lhd'_{\mathtt{crt}}$, $\widehat{\lhd}_{\mathtt{msg}} = \lhd_{\mathtt{msg}} \cup \lhd'_{\mathtt{msg}}$, $\widehat{\lambda} = \lambda \cup \lambda'$ and $\widehat{\mu} = \mu \cup \mu'$. Note that $\mathtt{Pids}(M \circ M') = \mathtt{Pids}(M) \cup \mathtt{Pids}(M')$ and $\mathtt{Bnd}(M \circ M') = \mathtt{Bnd}(M) \cup \mathtt{Bnd}(M')$.

A *Branching High-Level Message Sequence Chart (BHMSC)* is a tuple $\mathcal{H} = (A, L, L_{\mathtt{init}}, L_{\mathtt{acc}}, R, r_0, \delta)$ where A is a *message alphabet*, L is a finite set of *locations*, $L_{\mathtt{init}}, L_{\mathtt{acc}} \subseteq L$ are sets of *initial* and *accepting locations*, R is a finite set of *registers* with *initial register* r_0 and δ is a finite set of *transitions*. There are two sorts of transitions, namely *sequential transitions* and *fork-and-join transitions*:

- A sequential transition (ℓ, M, ℓ') (also written as written as $\ell \xrightarrow{M} \ell)$ is an element of $L \times \mathbb{PMSC}(A, R) \times L$ such that $\mathtt{Free}(M) \neq \emptyset$ and $\mathtt{MsgPar}(M) \cap \mathtt{Bnd}(M) = \emptyset$.
- A fork-and-join transition is of the form $\ell \to \{(\ell_1, R_1, \ell'_1), \ldots, (\ell_n, R_n, \ell'_n)\} \to \ell'$ where $n \geq 1$ is the degree of the transition, $\ell, \ell_1, \ldots, \ell_n, \ell'_1, \ldots, \ell'_n, \ell' \in L$ are locations and $R_1, \ldots, R_n$ are pairwise disjoint nonempty subsets of R. We depict a fork-and-join transition by:



Informally, a sequential transition (ℓ, M, ℓ') enforces that an instantiation of M by concrete processes is appended to the MSC leading to location ℓ. A fork-and-join-transition $\ell \to \{(\ell_1, R_1, \ell'_1), \ldots, (\ell_n, R_n, \ell'_n)\} \to \ell'$ expresses that for each $i \in \{1, \ldots, n\}$, the processes stored in the register in R_i are sent to a sub computation starting in location ℓ_i. Within the sub computation, the registers in R_i can be updated. After the sub computation reaches location ℓ'_i, the entire system resumes its execution at location ℓ' by using the register contents obtained at location ℓ'_i.

Example 22. We give an example BHMSC modeling the client-and-server scenario from Chapter 2. To demonstrate the expressive power of BHMSCs, we extend the setting in Chapter 2 by a second server and by giving clients the ability to spawn sub processes. The BHMSC is depicted in Figure 11.14. It uses five registers, r_0 to r_4, and the message symbols `serv`, `req`, `ack` and `noti`

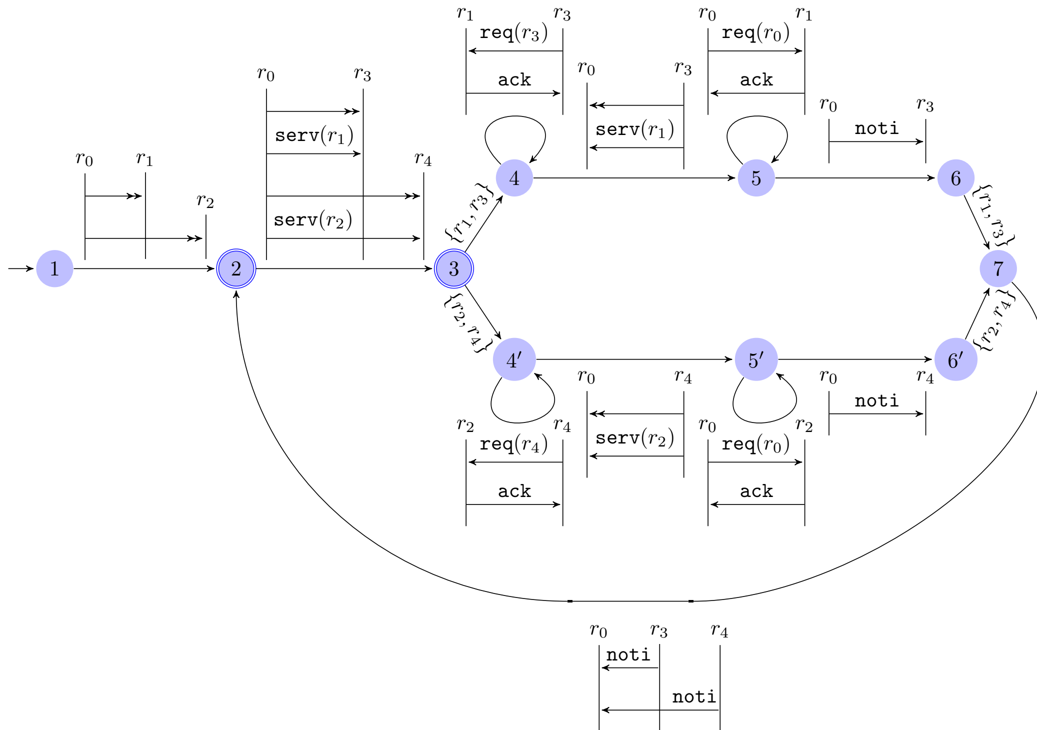


Figure 11.14: An example BHMSC modeling the client-and-server protocol from Chapter 2

with the same meaning as in the PRA in Example 18. The MSCs at the edges are defined over registers. The overall MSCs, generated by the BHMSC, are compositions of concrete instantiations of the MSCs on edges.

At the beginning, all registers, besides r_0 which contains the root process, are empty. The BHMSC starts with the execution of the MSC leading to location 2. In this MSC, the root process creates two servers and stores their IDs in registers r_1 and r_2. In the next MSC, the root process creates two clients to be stored in r_3 and r_4, sends to the first one the server ID in r_1 and to the latter one the server ID in r_2. Then, the computation splits into two parallel sub computations. Registers r_1 and r_3 are transmitted to the first sub computation and registers r_2 and r_4 to the second one. We explain the behaviour of the BHMSC in the first sub computation. The behaviour of the second one is analogous and can be obtained from the first one by replacing r_1 and r_3 with

r_2 and r_4, respectively. Recall that r_1 contains the ID of the first server and r_3 the ID of the first client. While going from location 3 to location 4, all registers, besides r_1 and r_3 are emptied. In the loop at location 4, the client in register r_1 sends arbitrarily often a request to the server in r_3 and gets acknowledgments. Then, it creates a sub client which is stored in r_0. Additionally, the sub client gets from the super client the server ID in r_1. After that, similar to the super client, the sub client communicates arbitrarily often with the server via requests and acknowledgements within the loop at location 5. Thereafter, it notifies the super client that it stops communication with the server. At location 6, the sub computation stops. When going from locations 6 and $6'$ to location 7, the contents of registers r_1 and r_3 are kept as in location 6 and the contents of registers r_2 and r_4 are kept as in location $6'$. However, the content of r_0 is rewritten by its content before entering the sub computations. This means that the sub client spawned in the meantime is overwritten by the root process. In the MSC at the edge leading from location 7 to 2, the clients in registers r_3 and r_4 send notifications to the root process meaning that the communication with the servers is completed. At state 2, the automaton can stop or continue. In the latter case, two new clients are crated and stored in registers r_3 and r_4 and new sub computations are started. $\square$

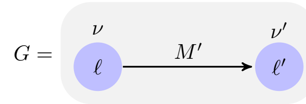
Runs and languages of BHMSCs

Just like in the case of Process Register Automata, we call an injective partial mapping $\nu \in [R \rightharpoonup \mathbb{P}]$ from the set of registers to the set of processes a *register assignment*. For a register assignment ν and a set $Q \subseteq R$, we define the *restriction* $\nu_{\restriction Q}$ of ν to Q as $\{r \mapsto \nu(r) \mid r \in \mathtt{dom}(\nu) \cap Q\}$. For two register assignments ν, ν' and a partial MSC $M \in \mathbb{PMSC}(A, R)$, we write $\nu \xrightarrow{M} \nu'$ if

- $\mathtt{Free}(M) \cup \mathtt{MsgPar}(M) \subseteq \mathtt{dom}(\nu)$,
- ν and ν' coincide on $R \backslash \mathtt{Bnd}(M)$, i.e., for every $r \in R \backslash \mathtt{Bnd}(M)$, $\nu(r) = \nu'(r)$, and
- $\mathtt{dom}(\nu') = \mathtt{dom}(\nu) \cup \mathtt{Bnd}(M)$ and $\nu'(\mathtt{Bnd}(M)) \cap \nu(R) = \emptyset$.

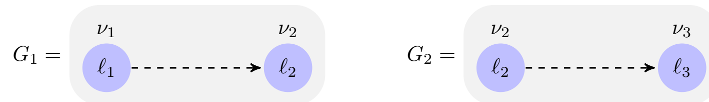
A *run* $G = (V, T, \mathtt{loc}, \mathtt{reg}, \rho)$ of the BHMSC $\mathcal{H}$ is a finite directed acyclic graph (V, T) with a unique *source node* $\mathtt{in}(G)$, *a unique sink node* $\mathtt{out}(G)$ and labelling functions $\mathtt{loc} : V \mapsto L$, $\mathtt{reg} : V \mapsto [R \rightharpoonup \mathbb{P}]$ and $\rho : T \mapsto 2^R \cup \mathbb{PMSC}(A, \mathbb{P})$. We define the set of all runs of $\mathcal{H}$ inductively. For the sake of simplicity, we mostly only give a convenient graphical representation of runs and skip their formal definitions.

- For two register assignments $\nu, \nu' \in [R \rightharpoonup \mathbb{P}]$ and a sequential transition $t = \ell \xrightarrow{M} \ell' \in \delta$ such that $\nu \xrightarrow{M} \nu'$, let $M' \in \mathbb{PMSC}(A, \mathbb{P})$ be the partial MSC obtained from M by replacing every register r by $\nu'(r)$. Then, the following graph G is an *atomic* run of $\mathcal{H}$.

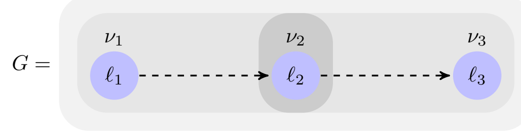
  

  We set $\mathtt{Pids}(G) = \nu(R) \cup \mathtt{Pids}(M')$ and $\mathtt{Bnd}(G) = \mathtt{Bnd}(M')$.

- Let G_1 and G_2 be the following two runs of $\mathcal{H}$ such that $\mathtt{Pids}(G_1) \cap \mathtt{Bnd}(G_2) = \emptyset$.

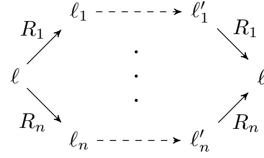
Then, the following graph G is a run of $\mathcal{H}$.



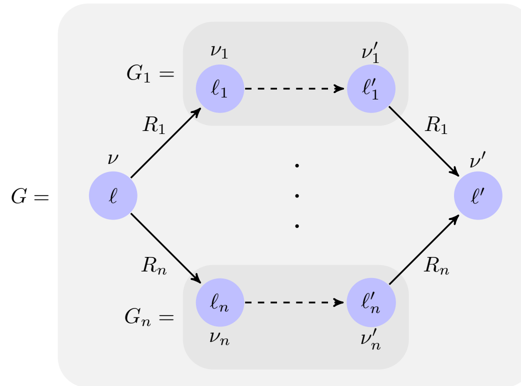
We set $\texttt{Pids}(G) = \texttt{Pids}(G_1) \cup \texttt{Pids}(G_2)$ and $\texttt{Bnd}(G) = \texttt{Bnd}(G_1) \cup \texttt{Bnd}(G_2)$.

- For $n \geq 1$, let

be runs of $\mathcal{H}$ and



a fork-and-join transition of $\mathcal{H}$. Furthermore, let ν and ν' be two register assignments such that $\texttt{Bnd}(G_i) \cap (\nu(R) \cup \bigcup_{j \neq i} \texttt{Pids}(G_j)) = \emptyset$ and $\nu_i = \nu_{\restriction R_i}$ for all i with $1 \leq i \leq n$. Furthermore, let $\nu' = \nu_{\restriction R_0} \cup \bigcup_{i \in \{1,\ldots,n\}} (\nu_i')_{\restriction R_i}$ where $R_0 = R \backslash (R_1 \cup \ldots \cup R_n)$. Then, the following graph G is also a run of $\mathcal{H}$.



We set $\texttt{Pids}(G) = \nu(R) \cup \bigcup_{i, 1 \leq i \leq n} \texttt{Pids}(G_i)$ and $\texttt{Bnd}(G) = \bigcup_{i, 1 \leq i \leq n} \texttt{Bnd}(G_i)$. Observe that, as illustrated in Example 22, each ν_i is the restriction of ν to R_i. Moreover, ν' results from ν and $\nu_1', \ldots, \nu_n'$ by taking the inputs of the registers in R_0 from ν and taking the inputs of the registers in R_i from ν_i', for every $i \in \{1, \ldots, n\}$.

Given a run G of $\mathcal{H}$, let $M_1, \ldots, M_n$ be an arbitrary enumeration of all MSCs occurring in G that respects the partial order induced by the edge relation of the run. We define the MSC $M(G) \in \mathbb{PMSC}(A, \mathbb{P})$ resulting from G as $M_1 \circ \ldots \circ M_n$. Since the sub computations in fork-and-join

transitions employ disjoint sets of process IDs, the MSC $M(G)$ is well defined and does not depend on the chosen enumeration. A run $G = (V, T, \texttt{loc}, \texttt{reg}, \rho)$ is called *accepting* if $\texttt{loc}(\texttt{in}(G)) \in L_{\texttt{init}}$, $\texttt{loc}(\texttt{out}(G)) \in L_{\texttt{acc}}$ and $\texttt{reg}(\texttt{in}(G)) = \{r_0 \mapsto p\}$ for some $p \in \mathbb{P}$. The *language* $\mathcal{L}(\mathcal{H})$ of $\mathcal{H}$ is defined as $\{ \overset{p}{\phi} \circ M(G) \mid G = (V, T, \texttt{loc}, \texttt{reg}, \rho)$ is an accepting run of $\mathcal{H}$ with $\texttt{reg}(\texttt{in}(G)) = \{r_0 \mapsto p\}\}$.

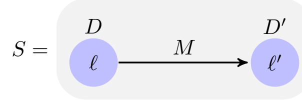
Observe that the language of $\mathcal{H}$ is closed, i.e., $\mathcal{L}(\mathcal{H}) = [\mathcal{L}(\mathcal{H})]$.

The symbolic behaviour of BHMSCs

Recall that in Section 11.3 we had defined symbolic configurations, runs and traces for Process Register Automata. They helped to solve the non-emptiness problem for that model. Similarly, we here define symbolic runs for BHMSCs which will later be used in several decision procedures for BHMSCs. Like in the case for Process Register Automata, usual runs will sometimes be called *concrete runs* in order to avoid confusion. Let $\mathcal{H} = (A, L, L_{\texttt{init}}, L_{\texttt{acc}}, R, r_0, \delta)$ be a BHMSC. A symbolic run $S = (V, T, \texttt{loc}, \texttt{def}, \pi)$ of $\mathcal{H}$ is a labelled graph which is almost defined like a concrete run of $\mathcal{H}$. However, instead of a function $\texttt{reg}$ that maps nodes to register assignments, the symbolic run S contains the mapping $\texttt{def}$ mapping nodes to sets of registers. Informally, these sets are the domains of register assignments in corresponding concrete runs. Moreover, the partial MSCs assigned to the edges of S are not defined over processes from $\mathbb{P}$, but over registers from R. To reflect this latter difference, we denote the edge labelling in S by π instead of ρ.

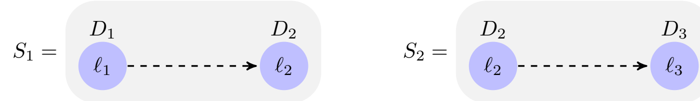
Just like in the concrete case, we define the set of symbolic runs of $\mathcal{H}$ inductively. Before diving into the definition, we introduce the convention that for a partial MSC $M \in \mathbb{PMSC}(A, R)$ and sets $D, D' \subseteq R$, we write $D \xrightarrow{M} D'$ if $\texttt{Free}(M) \cup \texttt{MsgPar}(M) \subseteq D$ and $D' = D \cup \texttt{Bnd}(M)$.

- Let D, D' be subsets of R and M a partial MSC such that $\ell \xrightarrow{M} \ell'$ is a sequential transition in δ with $D \xrightarrow{M} D'$. Then, the following graph

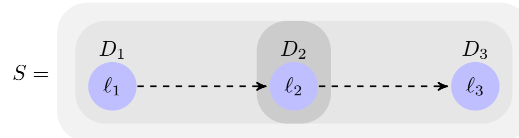


  is a symbolic atomic run of $\mathcal{H}$ with $\texttt{Bnd}(S) = \texttt{Bnd}(M)$.

- If S_1 and S_2 are the following symbolic runs of $\mathcal{H}$

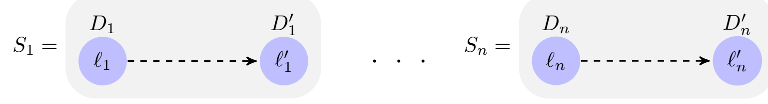


  then, the run

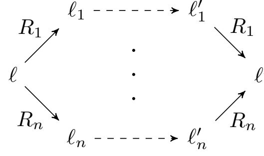


  is also a symbolic run of $\mathcal{H}$ with $\texttt{Bnd}(S) = \texttt{Bnd}(S_1) \cup \texttt{Bnd}(S_2)$.

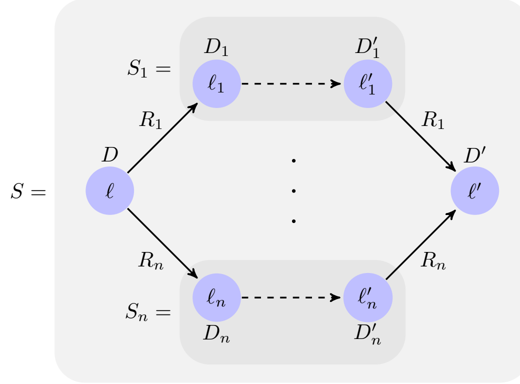
- Let for some $n \geq 1$, the graphs



be symbolic runs and



a fork-and-join transition of $\mathcal{H}$. Given two sets $D, D' \subseteq R$ with $D_i = R_i \cap D$ for $i \in \{1, \ldots, n\}$ and $D' = (D \cap R_0) \cup \bigcup_{i \in \{1,\ldots n\}} (D'_i \cap R_i)$ (again, we use R_0 for $R \backslash (R_1 \cup \ldots R_n)$), then the following graph S is also a symbolic run of $\mathcal{H}$.



We set $\texttt{Bnd}(S) = \bigcup_{i, 1 \leq i \leq n} (\texttt{Bnd}(S_i) \cap R_i)$.

We call a symbolic run $S = (V, T, \texttt{loc}, \texttt{def}, \pi)$ *accepting* if $\texttt{loc}(\texttt{in}(S)) \in L_{\texttt{init}}$, $\texttt{loc}(\texttt{out}(S)) \in L_{\texttt{acc}}$, and $\texttt{def}(\texttt{in}(S)) = \{r_0\}$.

Just like in the case with Process Register Automata, we define a mapping `symb` from concrete runs to symbolic runs of $\mathcal{H}$. Intuitively, from a concrete run G, we get the corresponding symbolic run $\texttt{symb}(G)$ by replacing register assignments at nodes by their domains and the MSCs at edges by the MSCs over R which belong to the corresponding sequential transitions from which the edges result. More formally, for a run $G = (V, T, \texttt{loc}, \texttt{reg}, \rho)$, we set $\texttt{symb}(G) = (V, T, \texttt{loc}, \texttt{def}, \pi)$ where `def` and π are defined as follows:

- $\texttt{def}(v) = \texttt{dom}(\texttt{reg}(v))$ for all $v \in V$, and
- $\pi(v, v') = \begin{cases} R', & \text{if } \rho(v, v') = R' \subseteq R \\ M, & \text{if } (v, v') \text{ results from some sequential transition } \ell \xrightarrow{M} \ell' \end{cases}$

  for all $(v, v') \in T$.

The following lemma, highlighting the relationship between concrete runs and symbolic ones, is an analogon of Observation 11 for BHMSCs.

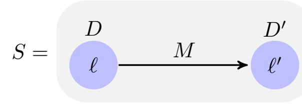
Lemma 12. *Let $\mathcal{H}$ be a BHMSC.*

(a) For every concrete run G of $\mathcal{H}$, $\mathtt{symb}(G)$ is a symbolic run of $\mathcal{H}$. Furthermore, $\mathtt{symb}(G)$ is accepting if and only if G is accepting.

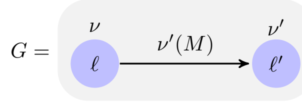
(b) For every symbolic run S of $\mathcal{H}$, there exists a concrete run G of $\mathcal{H}$ such that $S = \mathtt{symb}(G)$.

Proof. Statement (a) follows by straightforward induction. In order to prove (b), we argue by induction over the structure of symbolic runs that for every symbolic run S, one can construct a concrete run G with $\mathtt{symb}(G) = S$.

- In the base case, we consider a symbolic run

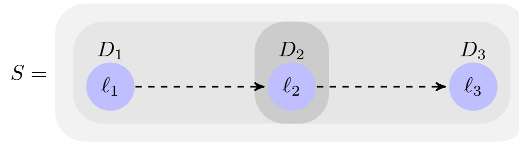


resulting from a sequential transition $\ell \xrightarrow{M} \ell'$. Let ν and ν' with $\mathtt{dom}(\nu) = D$ and $\mathtt{dom}(\nu') = D'$ be two register assignments such that (i) ν assigns pairwise distinct process IDs to the registers in D and ν' assigns pairwise distinct process IDs to the registers in D', (ii) ν and ν' coincide for $R \backslash \mathtt{Bnd}(M)$, and (iii) $\nu'(\mathtt{Bnd}(M)) \cap \nu(R) = \emptyset$. Since there is an infinite supply of process IDs, the existence of ν and ν' is guaranteed. Furthermore, note that due to the definitions of ν and ν', we have $\nu \xrightarrow{M} \nu'$. Thus, we can construct a concrete run



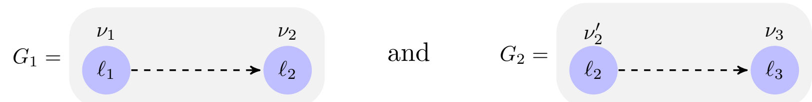
for which it clearly holds $\mathtt{symb}(G) = S$.

- Let

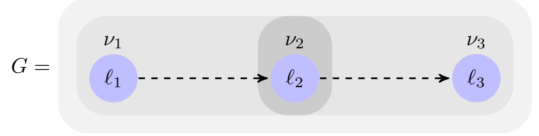


be a symbolic run resulting from the (sub) runs

By induction, there are runs

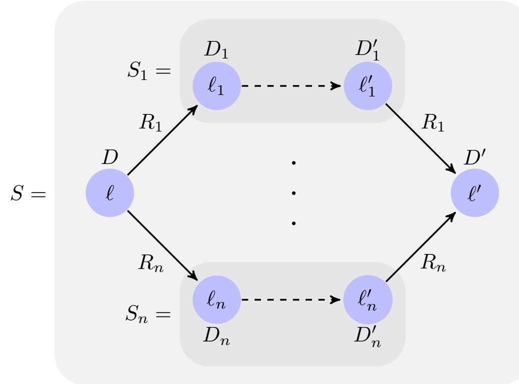


with $\texttt{symb}(G_1) = S_1$ and $\texttt{symb}(G_2) = S_2$. We can modify G_1 and G_2 in such a way that ν_2 equals ν'_2 and all process IDs appearing in G_2, but not in $\nu'_2(R)$, are different from all process IDs in G_1 (this is possible, since infinitely many IDs are available). Then, due to the construction rules for concrete runs, we can build

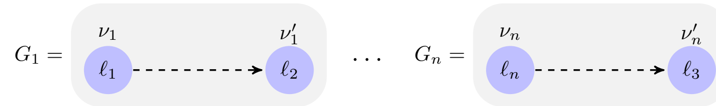


As $\texttt{symb}(G_1) = S_1$, $\texttt{symb}(G_2) = S_2$ and S results from the concatenation of S_1 and S_2, it easily follows $\texttt{symb}(G) = S$.

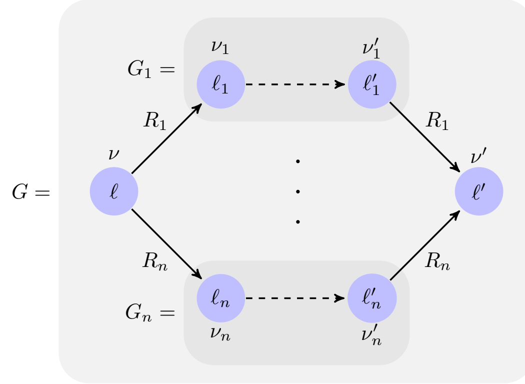
- Now, let



be a symbolic run resulting from sub runs $S_1, \ldots, S_n$ for some $n \geq 1$ by a fork-and-join transition t. By induction hypothesis, there are runs



such that $\texttt{symb}(G_i) = S_i$ for all i with $1 \leq i \leq n$. Again, we can assume that there is no ID occurring in some G_i which also occurs in some other $j \neq i$. Let ν and ν' be two register assignments with $\texttt{dom}(\nu) = D$ and $\texttt{dom}(\nu') = D'$ such that $\nu' = \nu_{\upharpoonright R_0} \cup \bigcup_{i \in \{1,\ldots,n\}} (\nu'_i)_{\upharpoonright R_i}$ and for every i with $1 \leq i \leq n$, $\nu_i = \nu_{\upharpoonright R_i}$ and $\texttt{Bnd}(G_i) \cap \nu(R) = \emptyset$. Using transition t, we can construct the concrete run



.

By the choice of ν and ν' and the fact that for every i, we have $\mathtt{symb}(G_i) = S_i$, it follows $\mathtt{symb}(G) = S$.

□

11.4.1 Non-Emptiness

The non-emptiness problem for BHMSCs is ExpTime-complete. In our work [46], the lower bound is shown by a reduction from the intersection non-emptiness problem for deterministic top-down automata on binary trees. Here, we present the proof of the upper bound. Thanks to Lemma 12, the question of non-emptiness for a BHMSC can be solved by searching for an accepting symbolic run. We show that such a search can be concluded in exponential time:

Lemma 13. *The non-emptiness problem for BHMSCs is in* ExpTime.

Proof. We first note that the non-emptiness problem for BHMSCs can be reduced to the problem of deciding whether a given BHMSC has an accepting symbolic run. Indeed, given a BHMSC $\mathcal{H}$, the following equivalences hold:

$\mathcal{L}(\mathcal{H}) \neq \emptyset \quad \Leftrightarrow \quad$ there is an MSC $M \in \mathcal{L}(\mathcal{H})$
$\quad \Leftrightarrow \quad$ there is an accepting run G of $\mathcal{H}$ (by definition of $\mathcal{L}(\mathcal{H})$)
$\quad \Leftrightarrow \quad$ there is an accepting symbolic run S of $\mathcal{H}$ (by Lemma 12)

It remains to give an algorithm which for every BHMSC $\mathcal{H} = (A, L, L_{\mathtt{init}}, L_{\mathtt{acc}}, R, r_0, \delta)$ decides in exponential time whether $\mathcal{H}$ has an accepting symbolic run. We first define *symbolic states* for $\mathcal{H}$. A symbolic state is a pair $s = (\ell, D)$ with $\ell \in L$ and $D \subseteq R$. Each symbolic state (ℓ, D) represents a node v in a symbolic run $S = (V, T, \mathtt{loc}, \mathtt{def}, \pi)$ of $\mathcal{H}$ with $\mathtt{loc}(\mathtt{in}(v)) = \ell$ and $\mathtt{def}(\mathtt{in}(v)) = D$. Our algorithm computes the set P of all pairs (s_1, s_2) of symbolic states for which there exists a symbolic run from s_1 to s_2. It decides that $\mathcal{H}$ is non-empty if and only if there is a pair $((\ell, \{r_0\})(\ell', D)) \in P$ with $\ell \in L_{\mathtt{init}}$, $\ell' \in L_{\mathtt{acc}}$ and some $D \subseteq R$.

The set P can be computed by a straightforward monotone fixed point computation. Since the number of symbolic states is at most exponential in the size of $\mathcal{H}$, an exponential number of iterations suffice to compute P. In each iteration, the algorithm checks for an (at most) exponential number of pairs (s_1, s_2) of symbolic states whether the pair can be obtained by a sequential transition, by concatenation or by parallel composition (i.e., by means of a fork-and-join transition) of given pairs. In the first case, it browses the transition relation δ which has at most linear size. In the second case, it has at most polynomially many choices among the given symbolic states. In the last case, the number of choices is at most exponential. Altogether, the running time of the algorithm is at most exponential. □

11.4.2 Executability

In Chapters 9 and 10, we gave several references to works studying the realizability problem for High-Level Message Sequence Charts (HMSCs) over finite sets of processes. Realizability deals with the question whether for a given HMSC, there is a communicating automaton (CA) describing the same sets of executions. As BHMSCs and DCA are basically dynamic extension of HMSCs and CA, it is quite natural to consider the realizability problem for BHMSCs with respect to DCA. Unfortunately, it follows from known results that this problem is not decidable [22, 114]. Therefore, we will analyze the executability problem, a necessary criterion for realizability. At an informal level, a BHMSC is called *executable* if at every sending event in the generated MSCs, the sender is "aware" of the receiver and the sent processes in its message. We will show that, just like non-emptiness, executability is ExpTime-complete. We will get the lower bound by a reduction from the non-emptiness problem. For the upper bound, we will first define a notion of executability on symbolic runs of BHMSCs. Then, we will prove that the executability of all symbolic runs of a BHMSC corresponds to the executability of all generated (concrete) MSCs. Finally, we will show that it can be checked in exponential time whether all symbolic runs of a given BHMSC are executable.

Executability of BHMSCs. Before defining executability formally, we introduce some notations. Given a mapping $\lambda \in [E \mapsto N \times \mathcal{T}]$ from an event set E to pairs of process names and action types, we define the mapping $\mathtt{pid}^\lambda \in [E \mapsto N]$ with $\mathtt{pid}^\lambda = \{e \mapsto n \mid \lambda(e) \in \mathcal{T} \times \{n\}\}$. If λ is clear from the context, we skip it in the notation of $\mathtt{pid}^\lambda$. Let $M = (E, \lhd, \lambda, \mu) \in \mathbb{MSC}(A, N)$ be an MSC over some message alphabet A and a set N of process names. For a process name n and an event e in M, we write $n \leadsto_M e$ if there is a path from $\min_n(M)$ to e in M. This path might involve the reversal of the create edge that starts n. Formally, $n \leadsto_M e$ if $(\min_n(M), e) \in (\lhd \cup \lhd_{\mathtt{crt}}^{-1})^*$. Intuitively, $n \leadsto_M e$ indicates that the process executing e is aware of n.

Let $M = (E, \lhd, \lambda, \mu) \in \mathbb{MSC}(A, \mathbb{P})$. A message edge $(e, e') \in \lhd_{\mathtt{msg}}$ in M with a message $m(p_1, \ldots, p_{\mathtt{ar}(m)})$ is executable if $p \leadsto_M e$, for every $p \in \{\mathtt{pid}(e'), p_1, \ldots, p_{\mathtt{ar}(m)}\}$. The MSC M is executable if each of its messages is executable. Finally, a BHMSC $\mathcal{H}$ is executable if all MSCs in $\mathcal{L}(\mathcal{H})$ are executable.

Example 23. In the MSC in Figure 11.15, the message edge from process 1 to 3 is not executable, because process 1 cannot be aware of the ID of process 3. It follows that the whole MSC is not executable.

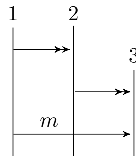


Figure 11.15: A non-executable MSC

□

We first prove the lower bound of the executability problem.

Lemma 14. *The executability problem for BHMSCs is* ExpTime*-hard.*

Proof. The proof is by reduction from the non-emptiness problem for BHMSCs. Since the latter problem is ExpTime-hard [46], the result follows.

Given an algorithm A solving the executability problem, we can easily extend A to an algorithm B for the non-emptiness problem. The latter algorithm works as follows. Given a BHMSC $\mathcal{H} = (A, L, L_{\mathtt{init}}, L_{\mathtt{acc}}, R, r_0, \delta)$ as input, it asks A whether $\mathcal{H}$ is executable or not. If A answeres "not executable", then, by definition of executability, this means that there *exists* an accepting run of $\mathcal{H}$ which is not executable. Thus, algorithm B outputs "non-empty". Otherwise, if algorithm A says "executable", this either means that (a) $\mathcal{H}$ has at least one accepting run and all accepting runs are executable, or (b) it has no accepting run at all. In order to check this, B extends $\mathcal{H}$ to a BHMSC $\mathcal{H}'$ with two new registers and a new initial non-executable transition. More formally, $\mathcal{H}' = (A, L \cup \{\ell_0\}, \{\ell_0\}, L_{\mathtt{acc}}, R \uplus \{r'_1, r'_2\}, r_0, \delta')$ where $\delta' \uplus \{\ell_0 \xrightarrow{M} \ell \mid \ell \in L_{\mathtt{init}}\}$ where M is an arbitrary non-executable MSC, for instance, the MSC in Figure 11.15. Note that every run in $\mathcal{H}'$ has to start with M. After the construction of $\mathcal{H}'$, B asks A whether $\mathcal{H}'$ is executable. If A answers "executable", this means that $\mathcal{H}'$, and thus $\mathcal{H}$, does not have any accepting run. Consequently, B outputs "empty". In the other case, it means that the existing runs of $\mathcal{H}$ turned to non-executable runs by appending them to M. Thus, B outputs "non-empty". □

Now, we turn to the upper bound of the executability problem. Like in the case for non-emptiness, we want to reduce the question of executability of a BHMSC to a test on its symbolic runs. But first, let us analyze how the executability of an MSC M, resulting from a concrete run, can be inferred from properties of the partial MSCs composing M. To find an answer to this question, we adapt the notion of executability to partial MSCs. Consider a partial MSC M' which is part of M. The executability of a message edge e in M' does not only depend on the relation $\rightsquigarrow_{M'}$ within M'. Rather, we have to take into account the set of all processes the process executing e is "aware of" when entering M'.

Executability of partial MSCs. We define the executability of partial MSCs with respect to *awareness relations* $K \subseteq \mathbb{P} \times \mathbb{P}$. Intuitively, $K(p, q)$ means that process p is aware of process q. We require that the awareness relation is reflexive. We say that a partial MSC $M = (E, \lhd, \lambda, \mu)$ over A and $\mathbb{P}$ is *executable at some awareness relation* K if for every message edge $(e, f) \in \lhd_{\mathtt{msg}}$ with message $m(p_1, \ldots, p_{\mathtt{ar}(m)})$ and for every $q \in \{\mathtt{pid}(f), p_1, \ldots, p_{\mathtt{ar}(m)}\}$, either $q \rightsquigarrow_M e$ or there is a q' such that $K(q', q)$ and $q' \rightsquigarrow_M e$. Similarly, a run G is called *executable at* K if $M(G)$ is executable at K. Clearly, after the execution of an MSC, the set of processes a process is aware of may be updated. We formalize this issue as follows: For a partial MSC M which is executable under an awareness relation K, we write $K \xrightarrow{M} K'$ where K' is the resulting awareness relation and is defined as $K \cup \{(p, q) \mid q \rightsquigarrow_M \mathtt{max}_p(M)$ or there is q' such that $K(q', q)$ and $q' \rightsquigarrow_M \mathtt{max}_p(M)\}$. For a concrete run G, we write $K \xrightarrow{G} K'$ if $K \xrightarrow{M(G)} K'$.

We formulate three observations which will be helpful in the proof of the upper bound of executability problem for BHMSCs.

Observation 12. A partial MSC M is executable at some K if and only if M is executable at $K_{\restriction \mathtt{Free}(M)}$.

Observation 13. For a partial MSC $M = M_1 \circ M_2$, we have $K \xrightarrow{M} K'$ for some K and K' if and only if there is some K_1 such that $K \xrightarrow{M_1} K_1$ and $K_1 \xrightarrow{M_2} K'$.

Observation 14. For a partial MSC $M = M_1 \circ \ldots \circ M_n$ where for every two $i, j \in \{1, \ldots, n\}$ with $i \neq j$, M_i and M_j do not share any process, it holds $K \xrightarrow{M} K'$ if and only if $K \xrightarrow{M_i} K_i$ for all $i \in \{1, \ldots, n\}$ where $K' = \bigcup_{i \in \{1,\ldots,n\}} K_i$.

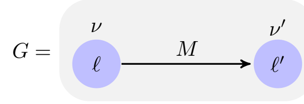
Notice that we have defined the notions of executability and awareness relations on MSCs and runs over $\mathbb{P}$. However, since we aim to check the executability of a BHMSC $\mathcal{H}$ on the basis of its symbolic runs, we have to understand what the executability of a BHMSC means for the symbolic

runs of $\mathcal{H}$. Therefore, we will define executability and awareness relations on symbolic runs and MSCs over R. When we deal with executability on symbolic runs, we have to be careful, because we have to take into account that the same register may represent different processes within a symbolic run.

Before defining executability and awareness on symbolic runs of a BHMSC $\mathcal{H} = (A, L, L_{\mathtt{init}}, L_{\mathtt{acc}}, R, r_0, \delta)$, we introduce some further notations. For an awareness relation K and a register assignment ν, we define the *induced symbolic awareness relation* $\mathtt{symb}_\nu(K)$ as the set $\{(r,s) \in R \times R \mid (\nu(r), \nu(s)) \in K\}$. For every partial MSC M over $\mathbb{P}$, we define the *flow relation* $\mathtt{flw}_M$ which, loosely speaking, describes the "information flow" within M. The set $\mathtt{flw}_M$ consists of all pairs (p,q) of processes such that (i) $p = q$ and $p \notin \mathtt{Pids}(M)$, or (ii) $q \in \mathtt{Free}(M)$ and there is a path from the minmial event of q to the maximal event of p in M. That is, $\mathtt{flw}_M = \{(p,p) \in \mathbb{P} \times \mathbb{P} \mid p \notin \mathtt{Pids}M\} \cup \{(p,q) \in \mathbb{P} \times \mathbb{P} \mid q \in \mathtt{Free}(M) \text{ and } q \rightsquigarrow_M \mathtt{max}_p(M)\}$. For a concrete run G with initial register assignment $\nu \in [R \rightharpoonup \mathbb{P}]$ and final register assignment $\nu' \in [R \rightharpoonup \mathbb{P}]$, we define $\mathtt{flw}_G = \{(r,s) \in R \times R \mid (\nu'(r), \nu(s)) \in \mathtt{flw}_{M(G)}\}$. Observe that $(r,s) \in \mathtt{flw}_G$ means that there is an "information flow" from the process in s at the time of enetring $M(G)$ to the process in r at the time of exiting $M(G)$. Furthermore, we define the set B_G of *refreshed registers* by $B_G = \{r \in R \mid \nu'(r) \neq \nu(r)\}$. Note that $\nu'(r) \neq \nu(r)$ holds in particular if $\nu'(r)$ is defined and $\nu(r)$ not.

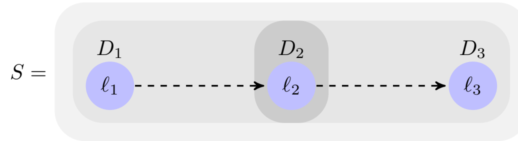
Executability of symbolic runs. The executability of symbolic runs is defined in dependency with symbolic awareness relations $SK \subseteq R \times R$. We also define the *effect* of the execution of a symbolic run which consists of a new symbolic awareness relation, a flow relation and a set of refreshed registers.

- Let

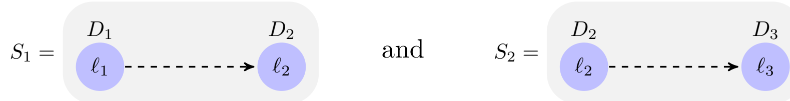


  be an atomic concrete run, resulting from a sequential transition $\ell \xrightarrow{M} \ell'$. If $M(G)$ is executable with $K \xrightarrow{M(G)} K'$ for awareness relations K and K', then $S = \mathtt{symb}(G)$ is executable at $\mathtt{symb}_\nu(K)$ with effect $(\mathtt{symb}_\nu(K), \mathtt{flw}_G, B_G)$ and write $\mathtt{symb}_\nu(K) \xrightarrow{S} (\mathtt{symb}_\nu(K), \mathtt{flw}_G, B_G)$

- Let

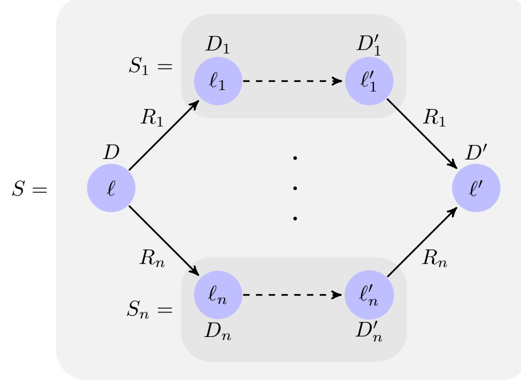


  be a symbolic run, resulting from the sub runs



  with $SK_1 \xrightarrow{S_1} (SK_2, \mathtt{flw}_1, B_1)$ and $SK_2 \xrightarrow{S_2} (SK_3, \mathtt{flw}_2, B_2)$. Then, S is executable at SK_1 with effect $(SK_3, \mathtt{flw}_2 \circ \mathtt{flw}_1, B_1 \cup B_2)$ (written as $SK_1 \xrightarrow{S} (SK_3, \mathtt{flw}_2 \circ \mathtt{flw}_1, B_1 \cup B_2)$).

- Let



be a symbolic run resulting from sub runs $S_1, \ldots, S_n$ for some $n \geq 1$ by a fork-and-join transition. The run S is executable at some SK if for each $i \in \{1, \ldots, n\}$, the sub run S_i is executable at $SK_{\restriction R_i}$. Moreover, if $SK_{\restriction R_i} \xrightarrow{S_i} (SK'_i, \texttt{flw}'_i, B'_i)$ for each $i \in \{1, \ldots, n\}$, then $SK \xrightarrow{S} (SK', \texttt{flw}', B')$ where

  - $B' = \bigcup_{i \in \{1,\ldots,n\}} (B'_i \cap R_i)$,
  - $\texttt{flw}' = \{(r, r) \mid (r, r) \in SK \text{ and } r \notin B'\} \cup \bigcup_{i \in \{1,\ldots,n\}} (\texttt{flw}'_i)_{\restriction R_i}$, and
  - SK' is the set of all pairs (r, s) for which one of the following conditions holds:
    (i) $r \in R_0$, $(r, s) \in SK$ and $y \notin B'$,
    (ii) for some $i \in \{1, \ldots, n\}$ and $t \in R$, $r \in R_i$, $(t, s) \in SK$, $(r, t) \in \texttt{flw}'_i$ and $s \notin B'$,
    (iii) for some $i \in \{1, \ldots, n\}$, $r, s \in R_i$ and $(r, s) \in SK'_i$.

We say that a symbolic run is executable if it is executable at $\{(r_0, r_0)\}$.

It should be noted that in the definition above, the information whether a register value is defined at some node is implicitly given by the corresponding symbolic awareness relation SK: r is defined if $(r, r) \in SK$. Furthermore, and crucial for the algorithm below, the definition only makes use of concrete MSCs in the base case.

In order to characterize the executability of a BHMSC in terms of symbolic runs, we will use the following lemma which will be proven at the end of this section. It clarifies the strong relationship between the executability of concrete runs and the executability of their symbolic counterparts.

Lemma 15. *Given a BHMSC $\mathcal{H}$, for every concrete run G of $\mathcal{H}$ with initial register assignment ν, final register assignment ν' and awareness relations K and K', it holds:*

(a) If $\texttt{symb}(G)$ is executable at $\texttt{symb}_\nu(K)$, then G is executable at K.

(b) If G is executable at K, then $\texttt{symb}(G)$ is executable at $\texttt{symb}_\nu(K)$.

(c) If $K \xrightarrow{G} K'$, then $\texttt{symb}_\nu(K) \xrightarrow{\texttt{symb}(G)} (\texttt{symb}_{\nu'}(K'), \texttt{flw}_G, B_G)$.

Using this lemma we can easily proof:

Lemma 16. *A BHMSC $\mathcal{H}$ is executable if and only if every accepting symbolic run of $\mathcal{H}$ is executable.*

Proof. Let $\mathcal{H} = (A, L, L_{\texttt{init}}, L_{\texttt{acc}}, R, r_0, \delta)$ be a BHMSC.

$\mathcal{H}$ is executable $\Leftrightarrow$ every MSC in $\mathcal{L}(\mathcal{H})$ is executable (by the definition of the executability of BHMSCs)
$\Leftrightarrow$ every accepting run G of $\mathcal{H}$ with $\texttt{reg}(\texttt{in}(G)) = \{r_0 \mapsto p\}$ for some process $p \in \mathbb{P}$ is executable at $\{(p, p)\}$ (by the definition of awareness relations)
$\Leftrightarrow$ every accepting symbolic run of $\mathcal{H}$ is executable at $\{(r_0, r_0)\}$ (by Lemmas 12 and 15)
$\Leftrightarrow$ every accepting symbolic run of $\mathcal{H}$ is executable (by the definition of the executability of symbolic runs)

□

Based on the last statement of the last lemma we can prove the upper bound complexity of the executability of BHMSCs:

Lemma 17. *The executability problem for BHMSCs is in* ExpTime.

Proof. By Lemma 16, the executability problem for BHMSCs can be reduced to the question whether all accepting symbolic runs of a given BHMSC are executable. We present an algorithm which checks in exponential time whether there is an accepting run which is *not* executable. The algorithm can be seen as an extension of the algorithm in Lemma 13, because it constructs inductively bigger symbolic runs from small ones.

Given a BHMSC $\mathcal{H} = (A, L, L_{\texttt{init}}, L_{\texttt{acc}}, R, r_0, \delta)$, the algorithm works in two steps. In the first step it computes the set ET of all tuples $(\ell, SK, \ell', SK', \texttt{flw}', B')$ for which there is a symbolic run S from location ℓ to location ℓ' with $SK \xrightarrow{S} (SK', \texttt{flw}', B')$. This can be done inductively using the rules defined for the executability of symbolic runs. It should be noted that only the base case explicitly involves MSCs. Due to Observation 12 and the definition of partial MSCs occurring in symbolic runs, the executability of atomic symbolic runs can be checked by simple reachability tests. Furthermore, as awareness relations implicitly contain the sets of defined registers, it is always easy to check which tuples from ET can be combined to obtain tuples for bigger symbolic runs.

In the second step, the algorithm computes the set NT of all tuples (ℓ, SK, ℓ', D') for which there is an accepting symbolic run S from location ℓ to ℓ' with $\texttt{def}(\texttt{out}(S)) = D'$ which is *not* executable at the symbolic awareness relation SK. For atomic symbolic runs, resulting from some sequential transition $\ell \xrightarrow{M} \ell'$, it has to be verified that M is not executable at $SK_{\restriction \texttt{Free}(M)}$. For symbolic runs, resulting from concatenation, the algorithm either combines (i) a tuple $(\ell_1, SK_1, \ell_2, D_2)$ from NT with a tuple $(\ell_2, SK_2, \ell_3, SK_3, \texttt{flw}_3, B_3)$ from ET resulting in a tuple $(\ell_1, SK_1, \ell_3, D_3)$ such that the set of registers occurring in D_2 is equal to the set of registers occurring in SK_2 and the set of registers occurring in D_3 is equal to the set of registers occurring in SK_3, or (ii) a tuple $(\ell_1, SK_1, \ell_2, SK_2, \texttt{flw}_2, B_2)$ from ET with a tuple $(\ell_2, SK_2, \ell_3, D_3)$ from NT with the resulting tuple $(\ell_1, SK_1, \ell_3, D_3)$, or (iii) a tuple $(\ell_1, SK_1, \ell_2, D_2)$ from NT with a further tuple $(\ell_2, SK_2, \ell_3, D_3)$ from NT such that D_2 and SK_2 have the same sets of registers and the resulting tuple is $(\ell_1, SK_1, \ell_3, D_3)$. To obtain symbolic runs constructed from fork-and-join transitions, it combines an arbitrary number of tuples $(\ell_i, SK_i, \ell'_i, SK'_i, \texttt{flw}'_i, B'_i)$ from ET with at least one tuple $(\ell_j, SK_j, \ell'_j, D'_j)$ from NT, resulting in a tuple (ℓ, SK, ℓ', D') where SK and D' are computed according to the rules given in the definition of executable symbolic runs.

The algorithm decides that $\mathcal{H}$ has an accepting, but non-executable run if it finds a tuple $(\ell, \{r_0, r_0\}, \ell', D')$ in NT with $\ell \in L_{\texttt{init}}$ and $\ell' \in L_{\texttt{acc}}$. As the number of tuples in ET and NT is at most exponential and each iteration adds at most one tuple in exponential time, the overall time is exponential. □

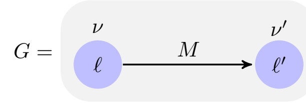
From Lemmas 14 and 17, we get the main result of this section:

Theorem 27. *The executability problem for BHMSCs is* EXPTIME*-complete.*

It remains to proof Lemma 15.

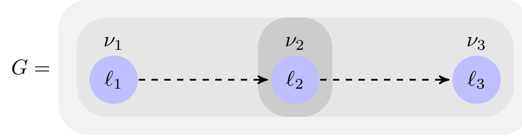
Proof of Lemma 15. Let $\mathcal{H}$ be a BHMSC and G a concrete run of $\mathcal{H}$. We argue by induction on the structure of G.

- For the base case, let

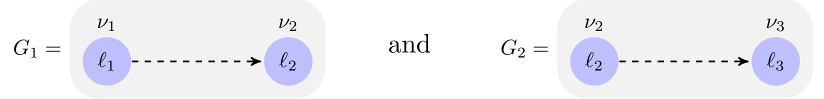
  

  be an atomic run resulting from a sequential transition $\ell \xrightarrow{M} \ell'$ and K and awareness relation. By the definition of the executability of symbolic runs, G is executable at K if and only if $\mathtt{symb}(G)$ is executable at $\mathtt{symb}_{\nu}(K)$. From this, we immediately get (a) and (b). Item (c) follows by the definition of the effects of the executions of symbolic runs.

- In the induction step, we first consider runs resulting from concatenation. Let

  be a symbolic run resulting from

  and K_1 an awareness relation.

  For the proof of (a), let us assume that $\mathtt{symb}(G)$ is executable at $SK_1 = \mathtt{symb}_{\nu_1}(K_1)$. By definition, it follows that $\mathtt{symb}(G_1)$ is executable at SK_1 with some effect $(SK_2, \mathtt{flw}_1, B_1)$ and $\mathtt{symb}(G_2)$ is executable at SK_2 with some effect $(SK_3, \mathtt{flw}_2, B_2)$. By induction,

  – G_1 is executable at K_1 with effect K_2 such that $\mathtt{symb}_{\nu_2}(K_2) = SK_2$, and
  – G_2 is executable at K_2.

  It follows by the definition of the executability of runs and Observation 13 that G is executable at K_1.

  Concerning (b), let G be executable at K_1. Then, again by Observation 13, G_1 must be executable at K_1 with $K_1 \xrightarrow{G_1} K_2$ for some effect K_2 and G_2 must be executable at K_2 with $K_2 \xrightarrow{G_2} K_3$ for some K_3. It follows by induction that $\mathtt{symb}(G_1)$ is executable at $\mathtt{symb}_{\nu_1}(K_1)$ with $\mathtt{symb}_{\nu_1}(K_1) \xrightarrow{\mathtt{symb}(G_1)} (\mathtt{symb}_{\nu_2}(K_2), \mathtt{flw}_{G_1}, B_{G_1})$ and $\mathtt{symb}(G_2)$ is executable at $\mathtt{symb}_{\nu_2}(K_2)$ with $\mathtt{symb}_{\nu_2}(K_2) \xrightarrow{\mathtt{symb}(G_2)} (\mathtt{symb}_{\nu_3}(K_3), \mathtt{flw}_{G_2}, B_{G_2})$. By the definition of executable symbolic runs, it follows that $\mathtt{symb}(G)$ is executable at $\mathtt{symb}_{\nu_1}(K_1)$.

Now, we show item (c). To this end, let $K_1 \xrightarrow{G} K_3$. By Observation 13, there is a K_2 with $K_1 \xrightarrow{G_1} K_2$ and $K_2 \xrightarrow{G_2} K_3$. By induction, it follows $\mathtt{symb}_{\nu_1}(K_1) \xrightarrow{\mathtt{symb}(G_1)} (\mathtt{symb}_{\nu_2}(K_2), \mathtt{flw}_{G_1}, B_{G_1})$ and $\mathtt{symb}_{\nu_2}(K_2) \xrightarrow{\mathtt{symb}(G_2)} (\mathtt{symb}_{\nu_3}(K_3), \mathtt{flw}_{G_2}, B_{G_2})$. From this and the definition of executable symbolic runs, it follows $\mathtt{symb}_{\nu_1}(K_1) \xrightarrow{\mathtt{symb}(G)} (\mathtt{symb}_{\nu_3}(K_3), \mathtt{flw}_{G_2} \circ \mathtt{flw}_{G_1}, B_{G_1} \cup B_{G_2})$. Thus, it remains to show that $\mathtt{flw}_{G_1} \circ \mathtt{flw}_{G_2} = \mathtt{flw}_G$ and $B_{G_1} \cup B_{G_2} = B_G$.

To see that $B_G = B_{G_1} \cup B_{G_2}$, just observe that a register is "refreshed" in G if and only if it is refreshed in G_1 or G_2.

For the proof of $\mathtt{flw}_G = \mathtt{flw}_{G_2} \circ \mathtt{flw}_{G_1}$, we first show that for every $r, s \in R$ with $(r, s) \in \mathtt{flw}_G$, it follows $(r, s) \in \mathtt{flw}_{G_2} \circ \mathtt{flw}_{G_1}$. Thus, assume for some $r, s \in R$ that $(r, s) \in \mathtt{flw}_G$. By definition of $\mathtt{flw}_G$, one of the following cases holds:

(i) $\nu_1(r) = \nu_3(s)$ and $\nu_1(s)$ does not occur in $M(G)$, or

(ii) $\nu_1(s) \in \mathtt{Free}(M(G))$ and $\nu_1(s) \rightsquigarrow_{M(G)} \mathtt{max}_{\nu_3}(r)(M(G))$.

First, we consider case (i). By the definition of runs, it must hold $r = s$ and $\nu_1(r) = \nu_2(r) = \nu_3(r)$. As $\nu_1(r)$ does not occur in $M(G)$ at all, it follows $(r, r) \in \mathtt{flw}_{G_1}$ and $(r, r) \in \mathtt{flw}_{G_2}$. Thus, $(r, r) \in \mathtt{flw}_{G_2} \circ \mathtt{flw}_{G_1}$. Case (ii) has three sub cases depending on where the path from $\mathtt{min}_{\nu_1(s)}(M(G))$ to $\mathtt{max}_{\nu_3(r)}(M(G))$ starts and ends:

(1) it starts and ends in $M(G_1)$,

(2) it starts and ends in $M(G_2)$, or

(3) it starts in $M(G_1)$ and ends in $M(G_2)$.

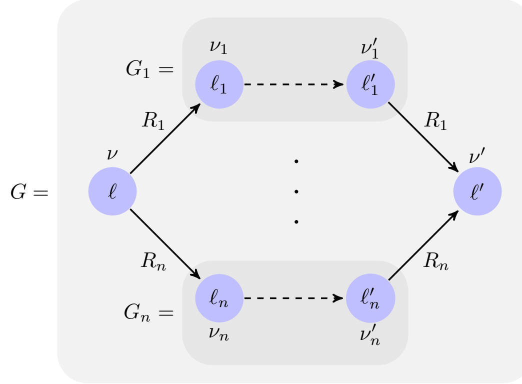
We first deal with case (1). As $\nu_1(s) \in \mathtt{Free}(M(G))$, it follows $\nu_1(s) \in \mathtt{Free}(M(G))$. Moreover, as $\nu_3(r)$ occurs in $M(G_1)$, by the definition of runs, it must hold $\nu_2(r) = \nu_3(r)$. Thus, we get $\nu_1(s) \in \mathtt{Free}(M(G_1))$ and $\nu_1(s) \rightsquigarrow_{M(G_1)} \mathtt{max}_{\nu_2(r)}(M(G_1))$ which means $(r, s) \in \mathtt{flw}_{G_1}$. As $\nu_2(r) = \nu_3(r)$ and $\nu_2(r)$ does not occur in $M(G_2)$ (otherwise $\mathtt{max}_{\nu_3(r)}(M(G))$ would not be in $M(G_1)$), we get $(r, r) \in \mathtt{flw}_{G_2}$. Together with $(r, s) \in \mathtt{flw}_{G_1}$, it follows $(r, s) \in \mathtt{flw}_{G_2} \circ \mathtt{flw}_{G_1}$.

Case (2) is analogue. As $\nu_1(s)$ occurs in $M(G_2)$, it must hold $\nu_1(s) = \nu_2(s)$ and, as it does not occur in $M(G_1)$, we get $(s, s) \in \mathtt{flw}_{G_1}$. Moreover, observe that it follows from $\nu_1(s) = \nu_2(s) \in \mathtt{Free}(M(G))$ and $\nu_1(s) \in \mathtt{Pids}(M(G_2))$ that $\nu_2(s)$ must be in $\mathtt{Free}(M(G_2))$. Furthermore, as $\nu_1(s) = \nu_2(s)$ and the path $\nu_1(s) \rightsquigarrow_{M(G)} \mathtt{max}_{\nu_3(r)}(M(G))$ starts and ends in $M(G_2)$, we get $\nu_2(s) \rightsquigarrow_{M(G_2)} \mathtt{max}_{\nu_3(r)}(M(G_2))$. Thus, $(r, s) \in \mathtt{flw}_{G_2}$. Together with $(s, s) \in \mathtt{flw}_{G_1}$, it follows $(r, s) \in \mathtt{flw}_{G_2} \circ \mathtt{flw}_{G_1}$.

For case (3), observe that there has to be some $t \in R$ such that $\nu_1(s) \rightsquigarrow_{M(G_1)} \mathtt{max}_{\nu_2(t)}(M(G_1))$, $\nu_2(t) \in \mathtt{Free}(M(G_2))$ and $\nu_2(t) \rightsquigarrow_{M(G_2)} \mathtt{max}_{\nu_3(r)}(M(G_2))$. Thus, $(t, s) \in \mathtt{flw}_{G_1}$ and $(r, t) \in \mathtt{flw}_{G_2}$. It follows $(r, s) \in \mathtt{flw}_{G_2} \circ \mathtt{flw}_{G_1}$.

For the other direction, namely that for every $r, s \in R$ with $(r, s) \in \mathtt{flw}_{G_2} \circ \mathtt{flw}_{G_1}$, it follows $(r, s) \in \mathtt{flw}_G$, the main observation is that two paths $\nu_1(s) \rightsquigarrow_{M(G_1)} \mathtt{max}_{\nu_2(t)}(M(G_1))$ and $\nu_2(t) \rightsquigarrow_{M(G_2)} \mathtt{max}_{\nu_3(r)}(M(G_2))$ can be put together to a single path $\nu_1(s) \rightsquigarrow_{M(G)} \mathtt{max}_{\nu_3(r)}(M(G))$.

- We finally consider runs resulting from fork-and-join transitions. Let



be a run resulting from the sub runs $G_1, \ldots, G_n$.

For the proof of (a), assume that $\mathtt{symb}(G)$ is executable at $\mathtt{symb}_\nu(K)$. By definition, each $\mathtt{symb}(G_i)$ is executable at $\mathtt{symb}_\nu(K)_{\upharpoonright R_i} = \mathtt{symb}_{\nu_i}(K)$. By induction, each G_i is executable at $K_{\upharpoonright R_i}$. By Observation 12 and because $\mathtt{Free}(M(G_i)) \subseteq R_i$, each G_i is executable at $K_{\upharpoonright \mathtt{Free}(M(G_i))}$. Again by Observation 12, each G_i is executable at K. As there are no i, j such that $M(G_i)$ and $M(G_j)$ share a process, we can follow by Observation 14 that G is executable at K.

The proof of statement (b) is similar. Assume that G is executable at K. Then, by Observation 14, each G_i must be executable at K. By Observation 12 and because $\mathtt{Free}(M(G_i)) \subseteq R_i$, for every i, each G_i must be executable at $K_{\upharpoonright R_i}$. Therefore, by induction, each $\mathtt{symb}(G_i)$ is executable at $\mathtt{symb}_\nu(K)_{\upharpoonright R_i}$. It follows by definition that $\mathtt{symb}(G)$ is executable at $\mathtt{symb}_\nu(K)$.

Now, we turn towards the proof of statement (c). Assume $K \xrightarrow{G} K'$ and $SK \xrightarrow{\mathtt{symb}(G)} (SK', \mathtt{flw}', B')$ with $SK = \mathtt{symb}_\nu(K)$. We have to show that $SK' = \mathtt{symb}_{\nu'}(K')$, $\mathtt{flw}' = \mathtt{flw}_{M(G)}$ and $B' = B_G$.

We start with the proof of the first equality. We show that for two registers $r, s \in \mathtt{dom}(\nu')$, it holds $(\nu'(r), \nu'(s)) \in K'$ if and only if $(r, s) \in SK'$.

To this end, let r and s be registers with $(\nu'(r), \nu'(s)) \in K'$. We consider three possible cases and conclude in each case that $(r, s) \in SK'$. In the following, by (i)-(iii) we refer to the three conditions on SK' in the fork-and-join case in the definition of executable symbolic runs

- $(\nu'(r), \nu'(s)) \in K$: Since $\nu'(r)$ and $\nu'(s)$ are already present in K, they cannot be in B'. Thus, $\nu(r) = \nu'(r)$ and $\nu(s) = \nu'(s)$ and, therefore, $(r, s) \in SK$. We distinguish two sub cases. If $r \in R_0$, then $(r, s) \in SK'$ by (i). Otherwise, let $r \in R_i$ for some $i \in \{1, \ldots, n\}$. As $r \notin B'$, we get $(r, r) \in \mathtt{flw}'_i$. Thus, $(r, s) \in SK'$ by (ii) with $t = r$.
- $\nu'(s) \rightsquigarrow_{M(G)} \mathtt{max}_{\nu'(r)}(M(G))$: In this case, the path is in some $M(G_i)$. Thus $(r, s) \in SK'_i$ and, by (iii), $(r, s) \in SK'$.
- $(q, \nu'(s)) \in K$ and $q \rightsquigarrow_{M(G)} \mathtt{max}_{\nu'(r)}(M(G))$ for some process q: In this case, there is a register t with $\nu'(t) = q$ such that r and t are in some R_i and $(\nu'(t), \nu'(s)) \in K$. By definition, we have $(r, t) \in \mathtt{flw}'_i$ and due to $(t, s) \in SK$, we get $(r, s) \in SK'$ by (ii).

Let us now assume $(r, s) \in SK'$. Due to the definition of SK', we distinguish three cases and show in each case that $(\nu'(r), \nu'(s)) \in K'$:

- $r \in R_0$, $(r, s) \in SK$ and $s \notin B'$: By definition of SK, we have $(\nu(r), \nu(s)) \in K$. However, as $r \in R_0$ and $s \notin B'$, we get $\nu'(r) = \nu(r)$ and $\nu'(s) = \nu(s)$. Therefore, $(\nu'(r), \nu'(s)) \in K'$.

- for some $i \in \{1, \dots, n\}$ and $t \in R$, $r \in R_i$, $(r, t) \in \mathtt{flw}'_i$, $(t, s) \in SK$ and $s \notin B'$: In this case, $\nu(t) \rightsquigarrow_{M(G_i)} \mathtt{max}_{\nu'(r)}(M(G_i))$ and $(\nu(t), \nu(s)) \in K$. As furthermore $\nu'(s) = \nu(s)$, we conclude $(\nu'(r), \nu'(s)) \in K'$ by the definition of K'.
- for some $i \in \{1, \dots, n\}$, $r, s \in R_i$ and $(r, s) \in SK'_i$: By induction, it follows $(\nu'_i(r), \nu'_i(s)) \in K'_i$. Due to the definition of K', we get $(\nu'(r), \nu'(s)) \in K'$.

Now, we show $B' = B_G$, that is, $\bigcup_{i \in \{1,\dots,n\}} (B'_i \cap R_i) = \{r \in R \mid \nu(r) \neq \nu'(r)\}$. By induction, for every i, $B'_i = B_{G_i}$. Thus, $r \in B'_i$ if and only if $\nu_i(r) \neq \nu'_i(r)$. This yields the desired equality, as $B_G = \bigcup_{i \in \{1,\dots,n\}} \{r \in R_i \mid \nu_i(r) \neq \nu'_i(r)\}$.

It remains to show that $\mathtt{flw}_G$ is the same as $\mathtt{flw}' = \{(r, r) \mid (r, r) \in SK, r \notin B'\} \cup \bigcup_{i \in \{1,\dots,n\}} (\mathtt{flw}'_i)_{\restriction R_i}$. We first show that for every pair $(r, s) \in R \times R$ with $(r, s) \in \mathtt{flw}_G$, it holds $(r, s) \in \mathtt{flw}'$. To this end, let for some $(r, s) \in R \times R$, $(r, s) \in \mathtt{flw}_G$. By definition, either (1) $r = s$, $\nu'(r) = \nu(r)$ and $\nu(r)$ does not occur in $M(G)$, or (2) $\nu(s) \in \mathtt{Free}(M(G))$ and there is a path from $\nu(s)$ to $\nu'(r)$ in $M(G)$. If (1) holds, then $(r, r) \in SK$ and $r \notin B'$. Thus, by definition, $(r, s) \in \mathtt{flw}'$. If, on the other hand, (2) holds, then $r, s \in R_i$ and the path from $\nu(s)$ to $\nu'(r)$ must be in $M(G_i)$ for some i. Thus, by induction, (r, s) is in $\mathtt{flw}'_i$, hence, in $\mathtt{flw}'$.

Now, let $(r, s) \in R \times R$ a pair such that $(r, s) \in \mathtt{flw}'$. There are two possibilities: either (1) $(r, s) \in (\mathtt{flw}'_i)_{\restriction R_i}$ for some i, or (2) $r = s$ with $(r, r) \in SK$ and $r \notin B'$. In the former case, we obtain by induction that $(r, s) \in \mathtt{flw}_{G_i}$. Therefore, by definition of $\mathtt{flw}_G$, it holds $(r, s) \in \mathtt{flw}_G$. The latter case has two sub cases: $r \in R_0$ and $r \in R_i$ for some i. If $r \in R_0$, then, $\nu(r) = \nu(s)$ and $\nu(r)$ has no event in $M(G)$. By definition, it follows $(r, r) \in \mathtt{flw}_G$. In the second case, it must hold $(r, r) \in (\mathtt{flw}'_i)_{\restriction R_i}$. By induction, we get $(r, r) \in \mathtt{flw}_G$.

□

11.5 Discussion

In this chapter, we introduced DCA, PRA and BHMSCs and analyzed some of their basic computational properties. We summarize our main results. For DCA, we first showed that the non-emptiness problem is not decidable, even not in the case of a single register. Then, we considered selective DCA, i.e., DCA where in each send action, the receiver has to be aware of the sender. While for selective 2-register DCA the problem remains undecidable, we were able to show that for selective 1-register DCA the problem is solvable in polynomial time. Inspired by recent works on the verification of ad-hoc networks, we also considered the state reachability problem for DCA. We first showed that, just like in the case of non-emptiness, state reachability is not decidable for 1-register DCA. Then, we focused on (strongly) bounded DCA, that is, DCA where executions of actions are only allowed if they lead to configurations where simple paths in the underlying (un-)directed communication graphs remain bounded by some constant. In contrast to results on ad-hoc networks, state reachability remains undecidable for strongly bounded DCA. As a further restriction, we considered degenerative DCA, i.e., DCA where processes are subjected to unexpected losses of register inputs. Surprisingly, each DCA is equivalent to its degenerative counterpart in terms of reachable configuration sets. While for bounded degenerative DCA, state reachability is undecidable, we showed by a non-trivial instantiation of the framework of Well-Structured Transition Systems that the problem becomes decidable if we restrict to strongly bounded degenerative DCA.

In the course of our considerations on PRA, we defined symbolic runs for this model and observed that for every usual run of a PRA, there is a corresponding symbolic run and vice-versa. This relationship between usual and symbolic runs established the basis for our decision

procedure solving the non-emptiness problem for which we proved NP-completeness. The latter result should be compared with the complexities of usual Register Automata and Fresh-Register Automata. Recall that when a PRA executes a create transition, a new process ID, which is fresh with respect to the whole run so far, is generated. The non-emptiness problem for Register Automata without such a freshness assumption is NP-complete on sequences of data values [124] and PSPACE-complete on simple data words [83]. For Fresh-Register Automata, which are equipped with transitions testing input data values for freshness, the authors in [196] only give the result that non-emptiness is decidable.

Similar to the case of PRA, we defined symbolic runs also for BHMSCs and worked with these structures in our decision procedures for this model. We proved EXPTIME-completeness for the non-emptiness of BHMSCs. In this work, we only gave the proof of the upper bound and referred for the lower bound to our paper [46]. Moreover, we studied the executability problem for BHMSCs which is a necessary criterion for the realizability of BHMSCs by DCA. It turned out that executability is EXPTIME-complete, thus, it has the same complexity as non-emptiness.

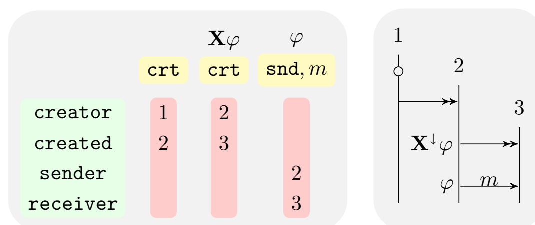
Our results contain several open questions attracting our attention. The open questions are in particular related to DCA. For instance, the decidability result on selective 1-register DCA relies heavily on the fact that configurations of this DCA-class have restricted shapes: each configuration can be represented by a set of isolated processes and pairs of processes. Presumably, decidability can be extended to DCA-classes where configurations can be represented by sets of more complex, but finitely many shapes. Then, an interesting question would be whether one can find syntactical restrictions for DCA whose configurations are as described. The second issue, leaving room for open questions, is that our searches for DCA-classes with decidable non-emptiness followed a different direction than in case of reachability. Thus, natural questions are, on the one hand, whether the decidability of reachability for strongly bounded degenerative DCA can be carried over to non-emptiness, and, on the other hand, whether we can obtain decidability for reachability if we relax the bounds on paths in configurations, but restrict to selective DCA. Finally, the DCA-version which we studied in this work is particularly with regard to two aspects a restriction of the original model in [47]: Firstly, messages in our model are restricted to at most one process ID. We assume that it should not be difficult to show that our results also hold in the case of multiple IDs in messages. Secondly, the style of communication in our model is rendezvous-based while the original model allows asynchronous communication through unbounded channels. It is well-known that, even for finitely many processes communicating via perfect unbounded channels, many verification problems are undecidable [54]. Nevertheless, it would be interesting to consider DCA with different kinds of asynchronous communication through bounded, lossy or unordered channels. Our results from [4] on DCA with buffers, which we summarized in Section 11.2.2.1, are first steps into this direction.

From our results in this chapter we conclude that DCA, a specification model for late design stages, are inherently harder to analyze than PRA and BHMSCs, which are suited for early design phases since they allow a more global view to systems. It is easy to see that the non-emptiness problem for our models is reducible to the model checking against logical formulas if the used logic is expressive enough to express $\bot$. In view of our non-emptiness results on DCA, this observation does not leave much room for decidability results on the model checking of DCA with logics. As the only DCA-class for which we proved decidable non-emptiness is the class of selective 1-register DCA, we will focus our studies on model checking of DCA in the next chapter on this class. Conversely, the non-emptiness results on PRA and BHMSCs are quite promising. Moreover, symbolic runs turned out to be a useful formalism in decision procedures for computational problems on these models. In the next chapter, we will show how symbolic runs can be used in the design of model checking algorithms for PRA and BHMSCs.

The results on state reachability for DCA in Section 11.2.2 of this chapter stem from [3, 4] which were joint works with Parosh Aziz Abdulla, Mohamed Faouzi and Othmane Rezine. In those papers, I defined the starting model and formulated initial questions. Furthermore, I was involved in discussions on proof strategies. The results on non-emptiness and executability of BHMSCs in Sections 11.4.1 and 11.4.2 originate from [46], a joint paper with Benedikt Bollig, Aiswarya Cyriac, Loïc Hélouët and Thomas Schwentick. The mentioned results are those on which I spent the most effort during the preparation of that paper. The remaining results in this chapter, namely the results on non-emptiness of DCA and PRA in Sections 11.2.1 and 11.3.1, are not published anywhere and I obtained them during the preparation of this thesis.

Chapter 12

New Results on Model Checking



In this chapter, we will study the verification of concurrents systems with unboundedly many processes by model checking. As system models, we will concentrate on the three automata models DCA, PRA and BHMSCs which were introduced in the last chapter. On the logic side, we will use for DCA and PRA data logics that were introduced in Parts A and B of this work. For BHMSCs, we will introduce in Section 12.3.1 a logic called MSC Navigation Logic (MNL) suited for the navigation on MSCs. It is inspired by Data Navigation Logic (DNL) from Chapter 6 and Temporal Logic of Causalities (TLC) [24, 173]. The model checking problem asks whether a given formula holds on all structures in the language of a given system description. As defined in the previous chapter, if the system description is given by an instance of DCA or PRA, the generated structures are data words and if the description is a BHMSC, the structures are MSCs. Recall that a formula of data logics is satisfied on a data word if it is satisfied at the first position of the word. For MNL, we will similarly define that a formula is satisfied on an MSC if it is satisfied at the initial event of the MSC. In the context of model checking, this means that we ask whether a given formula holds on the *first* position or event of each generated structure. This approach, often called *the anchored viewpoint* [182, 157], is very common in program verification.

Our results in this chapter do not give an exhaustive picture of model checking results for the mentioned system models. They should rather be considered as first insights. In terms of complexity, we will consider *combined complexity* where both, a system model as well as a formula, are parts of the input. Precise analyses of program and formula complexities where one of the components is fixed are left for future work.

In all of our (un-)decidability proofs, we will actually solve the *existential* model checking problem which checks for a system model and a formula whether there is at least one structure generated by the automaton and satisfying the formula. The term existential model checking occurs in the literature (see, e.g., in [38]) and the mentioned approach of solving the (general) model checking problem by considering its existential version is common in the area of finite-state model checking. Indeed, it is easy to see that a formula holds on all generated structures of a system automaton if and only if there is no generated structure satisfying the negation of the formula. Thus, as long as

the considered logics are closed under negating formulas, decidability results for one version of the model checking problem also hold for the other one. Additionally, if for a system model and a logic one version of the problem is contained in a complexity class which is closed under complementation, then also the other version is contained in the class.

In several decidability proofs, we will apply extensions of the well-known technique of labelling positions of the underlying structures by *consistent* sets of formulas. This approach is often used in model checking algorithms for finite-state systems and temporal logics (see, e.g., in [199]). However, as we deal with different system models and logics, in each case we have to redefine what consistency means. We finish this introduction by a formal definition of the usual and existential model checking problem.

The (existential) model checking problem. Let $\mathcal{C}$ be a class of system automata and $\mathcal{L}$ a logic. The *model checking problem* $\textsc{ModCheck}(\mathcal{C}, \mathcal{L})$ for $\mathcal{C}$ and $\mathcal{L}$ asks the following question: Given an automaton $\mathcal{A}$ from $\mathcal{C}$ and a formula φ from $\mathcal{L}$, does φ hold on *all* structures in the language of $\mathcal{A}$? The *existential model checking problem* $\textsc{EModCheck}(\mathcal{C}, \mathcal{L})$ asks whether for a given automaton $\mathcal{A}$ from $\mathcal{C}$ and a formula φ from $\mathcal{L}$, there *exists* a structure in the language of $\mathcal{A}$ satisfying φ.

12.1 Model Checking of Dynamic Communicating Automata

We will consider the model checking of DCA with restrictions of B-DLTL and $\text{LTL}^{\Downarrow}$. As the widest DCA-class for which we were able to show decidable non-emptiness was the class of selective 1-register DCA, this class will be subject of our investigations in this section. Remember from Chapter 6 that B-DLTL is the fragment of B-DNL using the temporal operators $\mathbf{X}$, $\mathbf{U}$, $\mathbf{X}_=$, $\mathbf{U}_=$ and their past counterparts instead of path expressions. The fragment with which we will work here is called *Restricted Basic Data LTL (RB-DLTL)* and results from B-DLTL by restricting all shift values ℓ in formulas of the form $\mathcal{C}^{\ell}_{\texttt{@a}}\varphi$ to 0. Recall that for a DCA with message alphabet A, we defined its traces as data words over the proposition set $\texttt{Prop}^{A}_{\texttt{act}} = \{\texttt{crt}, \texttt{snd}\} \cup A$ and the attribute set $\texttt{Attr}^{A}_{\texttt{act}} = \{\texttt{creator}, \texttt{created}, \texttt{sender}, \texttt{receiver}\} \cup \{\texttt{mpar}_1, \ldots, \texttt{mpar}_a\}$ where a is the highest arity assigned to a symbol in A. Hence, our logics in this section will be defined over the same proposition and attribute set.

We will first show that the the model checking problem for selective 1-register DCA is decidable for RB-DLTL. If the considered logic is $\text{LTL}^{\Downarrow}$, the problem is undecidable, even for the fragment $\text{LTL}^{\Downarrow}_1(\mathbf{X}, \mathbf{U})$. The latter result is interesting when we take into account that non-emptiness for selective 1-register DCA as well as satisfiability for $\text{LTL}^{\Downarrow}_1(\mathbf{X}, \mathbf{U})$ (on 1-complete data words) are both decidable.

12.1.1 Model Checking with Restricted Basic Data LTL

The main goal of this subsection is to prove the decidability of the model checking problem for selective 1-register DCA and RB-DLTL. Actually, we will show that the corresponding existential model checking problem is decidable and follow from this and the closure of RB-DLTL under negating formulas that the original model checking problem is decidable. Our proof is based on a reduction to reachability in Multicounter Automata (MCA, for definition, see Section 3.2.2).

Before starting with the technical part of the proof, we would like to explain why we do not chose the obvious proof strategy of reducing the existential model checking problem to the decidable non-emptiness problem for Data Automata (DA, for definition, see Section 4.2.2). Recall from Section 6.2 that every B-DNL-formula φ can be translated into a formula φ' simulating φ on encodings of general data words by 1-complete ones. Moreover, every B-DNL-formula on 1-complete data words can be translated into an equivalent DA $\mathcal{D}_{\varphi'}$[1]. This strategy suggests to solve the existential model

[1] To be precise, in Section 6.2 we actually showed that for every B-DNL-formula φ' on 1-complete data words,

checking problem for a selective 1-register DCA $\mathcal{A}$ and an RB-DLTL-formula φ according to an analogous plan: construct a DA $\mathcal{D}_{\varphi'}$ obtained from φ as above, construct a second DA $\mathcal{D}_{\mathcal{A}}$ deciding exactly the set of 1-complete encodings of all traces of $\mathcal{A}$ and check intersection non-emptiness of the languages of $\mathcal{D}_{\varphi'}$ and $\mathcal{D}_{\mathcal{A}}$. Since the latter problem is decidable [41], such a strategy would be an elegant way to solve the existential model checking problem. However, the snag in this strategy is that we do not see how a DA can check that an input word represents a correct DCA-trace. Recall that our 1-complete encoding requires to represent a single trace position by a block of at least four positions, one for each of the attributes `creator`, `created`, `sender` and `receiver`. Now, think of a selective 1-register DCA describing a system where each process created by the initial one creates a new process and sends him a message. On 1-complete encodings of traces, this means that for every block representing a create action (not performed by the initial process), there is some following block of arbitrary distance representing a send action such that the values at the `creator`- and `created`-positions in the create block are, respectively, equal to the values at the `sender`- and `receiver`-positions in the send block. As the ability of DA to check equality between data value tuples of arbitrary distance is limited, we believe that 1-complete encodings of such traces cannot be decided by DA.

Therefore, we follow here a different strategy to solve the existential model checking problem for selective 1-register DCA and RB-DLTL. We benefit from Observations 8 and 9 in Section 11.2.1 which state that runs of such automata can be simulated by runs with simplified configurations containing singles and couples. Using this, we reduce existential model checking to reachability for MCA where the constructed MCA simulates a run of a given DCA by counting the number of singles and couples in configurations and by assuring that all of them are accepting at the end.

We proceed with the technical preparations for our decidability proof. We first introduce a *negation normal form* for RB-DLTL-formulas where negation symbols can only occur in front of atomic formulas. In addition to the usual operators and atomic formulas, a formula in negation normal form can contain the *release* operators $\mathbf{R}$ and $\mathbf{R}_=$ (along with their past counterparts $\mathbf{R}^{\leftarrow}$ and $\mathbf{R}^{\leftarrow}_=$) and atomic formulas of the forms $\mathtt{start}$, $\mathtt{start}_=$, $\mathtt{end}$, $\mathtt{end}_=$ and $\bot_{\mathtt{a}}$ for attributes $\mathtt{a}$. The global atomic formula $\mathtt{start}$ is only true at the initial positions of data words and is equivalently expressed by $\neg\mathbf{X}^{\leftarrow}\top$. Analogously, the global atomic formula $\mathtt{end}$ only holds at the last position (if there exists one) of each data word and is equivalent to $\neg\mathbf{X}\top$. Their class counterparts $\mathtt{start}_=$ and $\mathtt{end}_=$ are expressed by $\neg\mathbf{X}^{\leftarrow}_=\top$ and $\neg\mathbf{X}_=\top$, respectively. The atomic formula $\bot_{\mathtt{a}}$ expresses that the value of attribute $\mathtt{a}$ at the current position is not defined and can be equivalently formulated as $\neg\mathcal{C}_{\mathtt{@a}} \sim\mathtt{@a}$. The temporal operator $\mathbf{R}$ has the following formal semantics: given a data word w, a position i in w and two global formulas φ_1 and φ_2, it holds $(w, i) \models \varphi_1\mathbf{R}\varphi_2$ if

- $(w, j) \models \varphi_2$ for all positions $j \geq i$, or
- there is a position $j \geq i$ such that $(w, j) \models \varphi_1$ and $(w, k) \models \varphi_2$ for all positions $i \leq k \leq j$.

It follows by definition that $\mathbf{R}$ is the dual of $\mathbf{U}$, i.e., for all formulas φ_1 and φ_2, we have $\varphi_1\mathbf{U}\varphi_2 \equiv \neg(\neg\varphi_1\mathbf{R}\neg\varphi_2)$ and $\varphi_1\mathbf{R}\varphi_2 \equiv \neg(\neg\varphi_1\mathbf{U}\neg\varphi_2)$. Likewise, $\mathbf{R}_=$, $\mathbf{R}^{\leftarrow}$ and $\mathbf{R}^{\leftarrow}_=$ are defined as the duals of $\mathbf{U}_=$, $\mathbf{U}^{\leftarrow}$ and $\mathbf{U}^{\leftarrow}_=$, respectively. Global formulas φ and class formulas ψ of RB-DLTL in negation normal form are constructed according to the following grammar:

$$\varphi := p \mid \neg p \mid \bot_{\mathtt{@a}} \mid \mathtt{start} \mid \mathtt{end} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \mathbf{X}^{\leftarrow}\varphi \mid \varphi\mathbf{U}\varphi \mid \varphi\mathbf{U}^{\leftarrow}\varphi \mid \varphi\mathbf{R}\varphi \mid \varphi\mathbf{R}^{\leftarrow}\varphi \mid \mathcal{C}_{\mathtt{@a}}\psi$$

$$\psi := \varphi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathbf{X}_=\psi \mid \mathbf{X}^{\leftarrow}_=\psi \mid \psi\mathbf{U}_=\psi \mid \psi\mathbf{U}^{\leftarrow}_=\psi \mid \psi\mathbf{R}_=\psi \mid \psi\mathbf{R}^{\leftarrow}_=\psi \mid \mathtt{@a} \mid \neg \sim\mathtt{@a}$$

there is a DA $\mathcal{D}_{\varphi'}$ which reads words with additional propositions and is, therefore, not equivalent to φ', but non-empty if and only if φ' is satisfiable. However, $\mathcal{D}_{\varphi'}$ can easily be turned into a DA $\mathcal{D}'_{\varphi'}$ equivalent to φ'. The base automaton of $\mathcal{D}'_{\varphi'}$ can "guess" the additional propositions and forward them as outputs to the class automaton such that base and class automaton can check in a joint work that the guess of the base automaton is correct.

where p is a proposition and $\mathtt{a}$ an attribute.
Observe that we explicitly add the $\vee$-operator to the syntax as it is not obvious how to express it with other operators in a formula in negation normal form. It is easy to see that thanks to De Morgan's rules and the equivalences

- $\neg\mathbf{X}\psi \equiv \mathtt{end} \vee \mathbf{X}\neg\psi$, $\neg\mathbf{X}^{\leftarrow}\psi \equiv \mathtt{start} \vee \mathbf{X}^{\leftarrow}\neg\psi$,
- $\neg\mathbf{X}_{=}\psi \equiv \mathtt{end}_{=} \vee \mathbf{X}_{=}\neg\psi$, $\neg\mathbf{X}^{\leftarrow}_{=}\psi \equiv \mathtt{start}_{=} \vee \mathbf{X}^{\leftarrow}_{=}\neg\psi$,
- $\neg\mathcal{C}_{@\mathtt{a}}\psi \equiv \bot_{@\mathtt{a}} \vee \mathcal{C}_{@\mathtt{a}}\neg\psi$,
- $\neg(\varphi_1\mathbf{U}\varphi_2) \equiv (\neg\varphi_1\mathbf{R}\neg\varphi_2)$, $\neg(\varphi_1\mathbf{U}^{\leftarrow}\varphi_2) \equiv (\neg\varphi_1\mathbf{R}^{\leftarrow}\neg\varphi_2)$,
- $\neg(\varphi_1\mathbf{U}_{=}\varphi_2) \equiv (\neg\varphi_1\mathbf{R}_{=}\neg\varphi_2)$, and $\neg(\varphi_1\mathbf{U}^{\leftarrow}_{=}\varphi_2) \equiv (\neg\varphi_1\mathbf{R}^{\leftarrow}_{=}\neg\varphi_2)$,

every RB-DLTL-formula can be converted into an equivalent formula in negation normal form by "pushing" the negation symbols inwards. Such a transformation yields a formula of at most linear length with respect to the size of the original one.

Given an RB-DLTL-formula φ in negation normal form over the proposition set $\mathtt{Prop}^{A}_{\mathtt{act}}$ and attribute set $\mathtt{Attr}^{A}_{\mathtt{act}}$ for some message alphabet A, we define the closure set $\mathtt{Closure}(\varphi)$ of φ as the smallest set containing

- $\mathtt{start}$, $\mathtt{end}$, $\mathtt{start}_{=}$, $\mathtt{end}_{=}$ and $\bot_{@\mathtt{a}}$ for every attribute $\mathtt{a}$,
- every sub-formula of φ,
- $\mathbf{X}(\psi_1\mathbf{U}\psi_2)$ for every sub-formula $\psi_1\mathbf{U}\psi_2$ of φ,
- $\mathbf{X}^{\leftarrow}(\psi_1\mathbf{U}^{\leftarrow}\psi_2)$ for every sub-formula $\psi_1\mathbf{U}^{\leftarrow}\psi_2$ of φ,
- $\mathbf{X}(\psi_1\mathbf{R}\psi_2)$ for every sub-formula $\psi_1\mathbf{R}\psi_2$ of φ,
- $\mathbf{X}^{\leftarrow}(\psi_1\mathbf{R}^{\leftarrow}\psi_2)$ for every sub-formula $\psi_1\mathbf{R}^{\leftarrow}\psi_2$ of φ,
- $\mathbf{X}_{=}(\psi_1\mathbf{U}_{=}\psi_2)$ for every sub-formula $\psi_1\mathbf{U}_{=}\psi_2$ of φ,
- $\mathbf{X}^{\leftarrow}_{=}(\psi_1\mathbf{U}^{\leftarrow}_{=}\psi_2)$ for every sub-formula $\psi_1\mathbf{U}^{\leftarrow}_{=}\psi_2$ of φ,
- $\mathbf{X}_{=}(\psi_1\mathbf{R}_{=}\psi_2)$ for every sub-formula $\psi_1\mathbf{R}_{=}\psi_2$ of φ, and
- $\mathbf{X}^{\leftarrow}_{=}(\psi_1\mathbf{R}^{\leftarrow}_{=}\psi_2)$ for every sub-formula $\psi_1\mathbf{R}^{\leftarrow}_{=}\psi_2$ of φ.

Note that for every formula φ, the size of the set $\mathtt{Closure}(\varphi)$ is at most polynomial in the length of φ. A set $C \subseteq \mathtt{Closure}(\varphi)$ is called *initial with respect to global formulas (or, respectively, class formulas)* if it does not contain any formula of the form $\mathbf{X}^{\leftarrow}\psi$ (or, respectively, $\mathbf{X}^{\leftarrow}_{=}\psi$). It is *final with respect to global formulas (or, respectively, class formulas)* if it does not contain any formula of the form $\mathbf{X}\psi$ (or, respectively, $\mathbf{X}_{=}\psi$). Given two sets $C_1, C_2 \subseteq \mathtt{Closure}(\varphi)$, C_2 is a *successor* of C_1 *with respect to global formulas* if for every formula $\mathbf{X}\psi \in C_1$, ψ is contained in C_2 and for every formula $\mathbf{X}^{\leftarrow}\psi \in C_2$, ψ is contained in C_1. Likewise, C_2 is a *successor* of C_1 *with respect to class formulas* if for every formula $\mathbf{X}_{=}\psi \in C_1$, the formula ψ is in C_2 and for every formula $\mathbf{X}^{\leftarrow}_{=}\psi \in C_2$, the formula ψ is in C_1. If C_2 is a successor of C_1 with respect to global formulas (or, respectively, class formulas), then, C_1 is called a *predecessor* of C_2 with respect to global formulas (or, respectively, class formulas).

Let $[C, \mathtt{act}, g]$ be a triple where $C \subseteq \mathtt{Closure}(\varphi)$, $\mathtt{act} \in \mathtt{Actions}(A, \mathbb{P})$ and g is a function which maps every process in $\mathtt{act}$ to some subset of $\mathtt{Closure}(\varphi)$. Furthermore, let $\mathcal{C}_g = \{g(p) \mid p$ is a process occurring in $\mathtt{act}\}$ be the set of all subsets of $\mathtt{Closure}(\varphi)$ assigned to processes in $\mathtt{act}$. The tuple $[C, \mathtt{act}, g]$ is called *consistent* if it respects the following rules:

`rule for global formulas`: If there is some $C' \in \mathcal{C}_g$ containing some global formula ψ, then, ψ is contained in C.

`rules for propositions`:
- If $\mathtt{act}$ is a create action, then, C does not contain $\mathtt{snd}$, $\neg\mathtt{crt}$ or any message symbol $m \in A$.
- if $\mathtt{act}$ is a send action with some message symbol m, then, C does not contain $\mathtt{crt}$, $\neg\mathtt{snd}$, $\neg m$ or any $m' \in A$ with $m \neq m'$.

$\bot$`-rule`: If for some $\mathtt{a} \in \mathtt{Attr}^A_{\mathtt{act}}$, the parameter $\mathtt{a}(\mathtt{act})$ is defined, then, $\bot_{\mathtt{@a}}$ is not contained in C.

$\mathcal{C}$`-rule`: If for some attribute $\mathtt{a} \in \mathtt{Attr}^A_{\mathtt{act}}$, the formula $\mathcal{C}_{\mathtt{@a}}\psi$ is contained in C, then, $\mathtt{a}(\mathtt{act})$ is defined and $\psi \in g(\mathtt{a}(\mathtt{act}))$.

`@-rules`: For every process p in $\mathtt{act}$ and attribute $\mathtt{a} \in \mathtt{Attr}^A_{\mathtt{act}}$:

- If $\sim\mathtt{@a} \in g(p)$, then, $\mathtt{a}(\mathtt{act})$ is defined and $p = \mathtt{a}(\mathtt{act})$.
- If $\neg \sim\mathtt{@a} \in g(p)$, then, $\mathtt{a}(\mathtt{act})$ is not defined or $p \neq \mathtt{a}(\mathtt{act})$.

$\wedge$`-rule`: If there is some $C' \in \{C\} \cup \mathcal{C}_g$ with $\psi_1 \wedge \psi_2 \in C'$, then, $\psi_1 \in C'$ and $\psi_2 \in C'$.

$\vee$`-rule`: If there is some $C' \in \{C\} \cup \mathcal{C}_g$ with $\psi_1 \vee \psi_2 \in C'$, then, $\psi_1 \in C'$ or $\psi_2 \in C'$.

$\mathbf{U}$`-rule`: If $\psi_1 \mathbf{U} \psi_2 \in C$, then,

- $\psi_2 \in C$ or
- $\psi_1 \in C$ and $\mathbf{X}(\psi_1 \mathbf{U} \psi_2) \in C$.

$\mathbf{U}^{\leftarrow}$`-rule`: If $\psi_1 \mathbf{U}^{\leftarrow} \psi_2 \in C$, then,

- $\psi_2 \in C$ or
- $\psi_1 \in C$ and $\mathbf{X}^{\leftarrow}(\psi_1 \mathbf{U}^{\leftarrow} \psi_2) \in C$.

$\mathbf{R}$`-rule`: if $\psi_1 \mathbf{R} \psi_2, \in C$, then,

- $\psi_2 \in C$ and
- $\mathtt{end} \in C$ or $\psi_1 \in C$ or $\mathbf{X}(\psi_1 \mathbf{R} \psi_2) \in C$.

$\mathbf{R}^{\leftarrow}$`-rule`: if $\psi_1 \mathbf{R}^{\leftarrow} \psi_2, \in C$, then,

- $\psi_2 \in C$ and
- $\mathtt{start} \in C$ or $\psi_1 \in C$ or $\mathbf{X}^{\leftarrow}(\psi_1 \mathbf{R}^{\leftarrow} \psi_2) \in C$.

$\mathbf{U}_{=}$`-rule`: If there is some $C' \in \mathcal{C}_g$ with $\psi_1 \mathbf{U}_{=} \psi_2 \in C'$, then,

- $\psi_2 \in C'$ or
- $\psi_1 \in C'$ and $\mathbf{X}_{=}(\psi_1 \mathbf{U}_{=} \psi_2) \in C'$.

$\mathbf{U}^{\leftarrow}_{=}$`-rule`: If there is some $C' \in \mathcal{C}_g$ with $\psi_1 \mathbf{U}^{\leftarrow}_{=} \psi_2 \in C'$, then,

- $\psi_2 \in C'$ or
- $\psi_1 \in C'$ and $\mathbf{X}^{\leftarrow}_{=}(\psi_1 \mathbf{U}^{\leftarrow}_{=} \psi_2) \in C'$.

Additionally, we have $\mathbf{R}_=$- and $\mathbf{R}^{\leftarrow}_=$-rules which are defined in analogy to the rules for $\mathbf{R}$ and $\mathbf{R}^{\leftarrow}$, respectively, by using the atomic class formulas $\mathtt{start}_=$ and $\mathtt{end}_=$.

Let $\tau = c_0 \xrightarrow{\mathtt{act}_1}_{\mathcal{A}} c_1 \ldots c_{n-1} \xrightarrow{\mathtt{act}_n}_{\mathcal{A}} c_n$ be a run of a DCA $\mathcal{A}$ and let $I = \{i_1 < \ldots < i_\ell\}$ be the set of all indices i_k such that $\mathtt{act}_{i_k} \neq \varepsilon$. Given $k, k' \in \{1, \ldots, \ell\}$ with $k < k'$ and a process p, we call k' the *p-successor* of k in τ if (i) both, $\mathtt{act}_{i_k}$ and $\mathtt{act}_{i_{k'}}$ contain p and (ii) there is no k'' with $k < k'' < k'$ such that $\mathtt{act}_{i_{k''}}$ also contains p. The notion of *p-predecessors* is defined analogously. Observe that p-successors and p-predecessors correspond to successors and predecessors in the p-class of the trace of τ. Let h be a function which maps every $k \in \{1, \ldots, \ell\}$ to some consistent tuple. The function h is called a *validity mapping* for τ if for each k, the action in $h(k)$ is $\mathtt{act}_{i_k}$, i.e., $h(k) = [C, \mathtt{act}_{i_k}, g]$ for some C and g. Given that h is a validity mapping for τ with $h(k) = [C_k, \mathtt{act}_{i_k}, g_k]$ for every $k \in \{1, \ldots, \ell\}$, we say that h is *correct with respect to successors* if for every k, the following conditions are fulfilled:

- If C_k is not final with respect to global formulas, then, $k + 1 \leq \ell$ and C_{k+1} is a successor of C_k with respect to global formulas.
- If C_k is not initial with respect to global formulas, then, $k - 1 \geq 1$ and C_{k-1} is a predecessor of C_k with respect to global formulas.
- For every process p occurring in $\mathtt{act}_{i_k}$:
  - If $g_k(p)$ is not final with respect to class formulas, then, k has a p-successor k' such that $g_{k'}(p)$ is a successor of $g_k(p)$ with respect to class formulas.
  - If $g_k(p)$ is not initial with respect to class formulas, then, k has a p-predecessor k' such that $g_{k'}(p)$ is a predecessor of $g_k(p)$ with respect to class formulas.

Observe that from these conditions it follows that C_1 must be initial with respect to global formulas, C_ℓ must be final with respect to global formulas and for every $k \in \{1, \ldots, \ell\}$ and process p in $\mathtt{act}_{i_k}$, it holds that (i) if k does not have a p-predecessor, then $g_k(p)$ is initial with respect to class formulas and (ii) if k does not have a p-successor, then $g_k(p)$ is final with respect to class formulas. Assume that h is a validity mapping for τ which is correct with respect to successors. The pair (τ, h) is called a φ-run of $\mathcal{A}$ if

- I is non-empty,
- $\varphi \in C_1$, and
- for all $k \in \{1, \ldots, \ell\}$, it holds that
  - if $\mathtt{start} \in C_k$, then, $k = 1$,
  - if $\mathtt{end} \in C_k$, then, $k = \ell$,
  - if there is some process p occurring in $\mathtt{act}_{i_k}$ such that $\mathtt{start}_= \in g_k(p)$, then, k has no p-successor, and
  - if there is some process p occurring in $\mathtt{act}_{i_k}$ such that $\mathtt{end}_= \in g_k(p)$, then, k has no p-predecessor.

We say that the pair (τ, h) is an *accepting* φ-run, if it is a φ-run and c_n is an accepting configuration of $\mathcal{A}$.

Lemma 18. *A DCA $\mathcal{A}$ has a trace satisfying an RB-DLTL-formula φ in negation normal form if and only if there is an accepting φ-run of $\mathcal{A}$.*

Proof. Let $\mathcal{A} = (A, R, S, s_0, \delta, F)$ be a DCA and φ an RB-DLTL-formula in negation normal form. We first prove the "only if"-direction. Let w be the trace of an accepting run $\tau = c_0 \xrightarrow{\texttt{act}_1}_{\mathcal{A}} c_1 \ldots c_{n-1} \xrightarrow{\texttt{act}_n}_{\mathcal{A}} c_n$ of $\mathcal{A}$ satisfying φ. This means that (i) the set $I = \{i_1 < \ldots < i_\ell\}$ of all indices i_k with $\texttt{act}_{i_k} \neq \varepsilon$ is non-empty (otherwise, τ would induce an empty word which cannot be satisfied by any formula), (ii) $w = w_1 \ldots w_\ell$ with $w_k = \texttt{dwrep}(\texttt{act}_{i_k})$ for every $\in \{1, \ldots, \ell\}$ and (iii) $w \models \varphi$. We will construct a validity mapping h and show that (τ, h) constitutes an accepting φ-run. For every k with $1 \le k \le \ell$, we set $h(k) = [C_k, \texttt{act}_{i_k}, g_k]$ where C_k consist of all global formulas $\psi \in \texttt{Closure}(\varphi)$ with $(w, k) \models \psi$ and for each process p in $\texttt{act}_{i_k}$, $g_k(p)$ consists of all class formulas ψ with $(w, k, p) \models \psi$. First, we will show that for every k, the tuple $h(k)$ is consistent. We will do this by explaining that all consistency rules defined above hold for $h(k)$. Observe that compliance with the rules for global formulas, those for propositions and the $\bot$-, $\wedge$-, $\vee$-rules follow directly from construction. Among the remaining rules, we pick out some interesting ones and proof that they must hold. The proofs for the omitted cases can be derived straightforwardly. Let $k \in \{1, \ldots, \ell\}$.

- $\mathcal{C}$-rule: Assume that for some attribute $\texttt{a} \in \texttt{Attr}^A_{\texttt{act}}$, the formula $\mathcal{C}_{\texttt{@a}}\psi$ is contained in C_{i_k}. By the construction of $h(k)$, we conclude $(w, k) \models \mathcal{C}_{\texttt{@a}}\psi$. By the semantics of the class operator, this means that $\texttt{a}(\texttt{act}_{i_k}) = p$ is defined and $(w, k, p) \models \psi$. Again by construction of $h(k)$, we get $\psi \in g_k(p)$.
- $\texttt{@}$-rules: Let p be a process occurring in $\texttt{act}_{i_k}$ and $\texttt{a} \in \texttt{Attr}^A_{\texttt{act}}$ an attribute.
  - If $\sim\!\texttt{@a} \in g_k(p)$, then, by construction of h_k, it must hold $(w, k, p) \models\sim\!\texttt{@a}$. Thus, by the semantics of RB-DLTL, $\texttt{a}(\texttt{act}_{i_k})$ must be defined and it must be equal to p.
  - If $\neg\sim\!\texttt{@a} \in g_k(p)$, then, by construction, it follows $(w, k, p) \models \neg\sim\!\texttt{@a}$. According to the semantics of RB-DLTL, either $\texttt{a}(\texttt{act}_{i_k})$ is not defined or it is not equal to p.
- $\mathbf{R}$-rule: Let $\psi_1 \mathbf{R} \psi_2 \in C_k$. By construction of $h(k)$, we have $(w, k) \models \psi_1 \mathbf{R} \psi_2$. By the semantics of the $\mathbf{R}$-operator, this means that (i) $(w, i) \models \psi_2$ for all positions i with $k \le i \le \ell$, or (ii) there is a position $i \ge k$ such that $(w, i) \models \psi_1$ and $(w, j) \models \psi_2$ for all positions j with $k \le j \le i$. This is equivalent to requiring that (i) $(w, k) \models \psi_2$ and (ii) $(w, k) \models \texttt{end}$ or $(w, k) \models \psi_1$ or $(w, k) \models \mathbf{X}(\psi_1 \mathbf{R} \psi_2)$. Again by construction of $h(k)$, we conclude that (i) $\psi_2 \in C_k$ and (ii) $\texttt{end} \in C_k$ or $\psi_1 \in C_k$ or $\mathbf{X}(\psi_1 \mathbf{R} \psi_2) \in C_k$.
- $\mathbf{U}^{\leftarrow}_{=}$-rule: Assume that there is some process p in $\texttt{act}_{i_k}$ with $\psi_1 \mathbf{U}^{\leftarrow}_{=} \psi_2 \in g_k(p)$. By construction, it must hold $(w, k, p) \models \psi_1 \mathbf{U}^{\leftarrow}_{=} \psi_2$. Thus, by the semantics of $\mathbf{U}^{\leftarrow}_{=}$, there is some $i \le k$ such that $(w, i, p) \models \psi_2$ and $(w, j, p) \models \psi_1$ for all positions j in the p-class of w with $i < j \le k$. The latter is equivalent to the requirement that (i) $(w, k, p) \models \psi_2$ or (ii) $(w, k, p) \models \psi_1$ and $(w, k, p) \models \mathbf{X}^{\leftarrow}_{=}(\psi_1 \mathbf{U}^{\leftarrow}_{=} \psi_2)$. Consequently, by construction of $g_k(p)$, it must hold (i) $\psi_2 \in g_k(p)$ or (ii) $\psi_1 \in g_k(p)$ and $\mathbf{X}^{\leftarrow}_{=}(\psi_1 \mathbf{U}^{\leftarrow}_{=} \psi_2) \in g_k(p)$.

It easily follows from construction that φ must be contained in C_1. Moreover, the atomic formula $\texttt{start}$ (or, respectively, $\texttt{end}$) can be contained at most in C_1 (or, respectively, C_ℓ). Likewise, $\texttt{start}_=$ (or, respectively, $\texttt{end}_=$) can occur in some $g_k(p)$ for some k and process p only if k has no p-predecessor (or, respectively, p-successor). Furthermore, as τ is an accepting run, c_n must be an accepting configuration. Thus, for the proof that (τ, h) is an accepting φ-run, it remains to show that h is correct with respect to successors. Also this follows more or less straightforwardly from the construction of h. Here, we content ourselves by showing that for every $k \in \{1, \ldots, \ell\}$ and process p occurring in $\texttt{act}_{i_k}$ such that $g_k(p)$ is not initial with respect to class formulas, it holds that k has a p-predecessor k' such that $g_{k'}(p)$ is a predecessor of $g_k(p)$ with respect to class formulas. To this end, suppose that $\mathbf{X}^{\leftarrow}_{=}\psi \in g_k(p)$ for some k and process p. We have to show that k

has a p-predecessor k' such that $\psi \in g_{k'}(p)$. By construction, it must hold $(w, k, p) \models \mathbf{X}^{\leftarrow}_{=}\psi$. By the semantics of $\mathbf{X}^{\leftarrow}_{=}$, there must exist a predecessor k' of k in the p-class of w with $(w, k', p) \models \psi$. By the definition of the p-predecessor and the construction of h, it follows that k' is the p-predecessor of k and $\psi \in g_{k'}(p)$. This concludes the proof of the "only if"-part of the lemma.

With regard to the proof of the "if"-direction, assume that there is an accepting run $\tau = c_0 \xrightarrow{\texttt{act}_1}_{\mathcal{A}} c_1 \ldots c_{n-1} \xrightarrow{\texttt{act}_n}_{\mathcal{A}} c_n$ of $\mathcal{A}$ and a validity mapping h on τ such that (τ, h) is an accepting φ-run of $\mathcal{A}$. We will show that the trace of τ must satisfy φ. Let $I = \{i_1 < \ldots < i_\ell\}$ be the set of all indices i_k such that $\texttt{act}_{i_k} \neq \varepsilon$. Just like in the proof of the "only if"-part, we denote for every $k \in \{1, \ldots, \ell\}$, the tuple $h(k)$ by $[C_k, \texttt{act}_{i_k}, g_k]$. Observe that by the definition of φ-runs, I must not be empty. Moreover, the trace of $\mathcal{A}$ is defined by $w = \texttt{dwrep}(\texttt{act}_{i_1}) \ldots \texttt{dwrep}(\texttt{act}_{i_\ell})$. We will show that for every formula $\psi \in \texttt{Closure}(\varphi)$ and every $k \in \{1, \ldots, \ell\}$, it holds:

- If ψ is a global formula and contained in C_k, then, $(w, k) \models \psi$.
- If ψ is a class formula, but not a global formula, and contained in $g_k(p)$ for some process p occurring in $\texttt{act}_{i_k}$, then, $(w, k, p) \models \psi$.

As φ is a global formula and, by the assumption that (τ, h) is a φ-run, contained in C_1, it will follow that $(w, 1) \models \varphi$ and, thus, w satisfies φ.

The proof is by induction on the structure of RB-DLTL-formulas ψ. We restrict ourselves to the consideration of some interesting cases. The skipped cases can be handled analogously. Observe that in the cases where ψ is a proposition, a negated proposition or one of the atomic formulas $\texttt{start}$, $\texttt{start}_=$, $\texttt{end}$ and $\texttt{end}_=$, our claim follows directly from the definition of consistent tuples and accepting φ-runs. We start with some cases where ψ is an atomic formula:

- $\psi = \bot_{\texttt{@a}}$ for some attribute $\texttt{a}$: Note that $\bot_{\texttt{@a}}$ is a global formula and assume that $\bot_{\texttt{@a}} \in C_k$ for some $k \in \{1, \ldots, k\}$ (otherwise, there is nothing to show). Due to the $\bot$-rule, $\texttt{a}(\texttt{act}_{i_k})$ cannot be defined. Thus, $(w, k) \models \bot_{\texttt{@a}}$.
- $\psi =\sim\texttt{@a}$ for some attribute $\texttt{a}$: The formula $\sim\texttt{@a}$ is a class but not a global formula. Suppose that $\sim\texttt{@a} \in g_k(p)$ for some $k \in \{1, \ldots, \ell\}$ and some process p in $\texttt{act}_{i_k}$. According to the $\texttt{@}$-rules, $\texttt{a}(\texttt{act}_{i_k}) = p$. In compliance with the semantics of RB-DLTL, we can derive $(w, k, p) \models \sim\texttt{@a}$.

The case for $\psi = \neg \sim\texttt{@a}$ is handled analogously. We now turn to the cases where ψ is neither an atomic, nor a negated atomic formula:

- $\psi = \mathcal{C}_{\texttt{@a}}\chi$ for some attribute $\texttt{a}$: The formula is a global formula. Assume that $\mathcal{C}_{\texttt{@a}}\chi \in C_k$ for some $k \in \{1, \ldots, \ell\}$. It follows from the $\mathcal{C}$-rule that $\texttt{a}(\texttt{act}_{i_k}) = p$ for some process p and $\chi \in g_k(p)$. If χ is a global formula, then, by definition of consistent tuples, $\chi \in C_k$. Then, by induction, we deduce that $(w, k) \models \chi$, thus, $(w, k) \models \mathcal{C}_{\texttt{@a}}\chi$. If χ is a class formula and not a global formula, it follows directly by induction that $(w, k, p) \models \chi$ and, therefore, $(w, k) \models \mathcal{C}_{\texttt{@a}}\chi$.
- $\psi = \mathbf{X}\chi$: Note that ψ is a global formula and, due to the formation rules for RB-DLTL-formulas, χ is also a global formula. Let us assume that $\mathbf{X}\chi \in C_k$ for some $k \in \{1, \ldots, \ell\}$. As h is correct with respect to successors, $k + 1 \leq \ell$ and $\psi \in C_{k+1}$. By induction, $(w, k + 1) \models \chi$ and, consequently, $(w, k) \models \mathbf{X}\chi$.
- $\psi = \chi_1\mathbf{U}\chi_2$: By the definition of RB-DLTL-formulas, the formulas ψ, χ_1 and χ_2 must be global formulas. Provided that ψ is contained in C_k for some $k \in \{1, \ldots, \ell\}$, it follows from the $\mathbf{U}$-rule that (i) $\chi_2 \in C_k$ or (ii) $\chi_1 \in C_k$ and $\mathbf{X}(\chi_1\mathbf{U}\chi_2) \in C_k$. By taking into account that h is correct with respect to successors and by applying the $\mathbf{U}$-rule repeatedly, we deduce

that there must be some j with $k \leq j \leq \ell$ such that (i) $\chi_2 \in C_j$ and (ii) $\chi_1 \in C_{j'}$ for all j' with $k \leq j' < j$. By induction, we infer that there is some j with $k \leq j \leq \ell$ such that (i) $(w, j) \models \chi_2$ and (ii) $(w, j') \models \chi_1$ for all j' with $k \leq j' < j$. By the semantics of the $\mathbf{U}$-operator, it directly follows $(w, k) \models \chi_1 \mathbf{U} \chi_2$.

- $\psi = \chi_1 \mathbf{U}^{\leftarrow}_{\equiv} \chi_2$: Note that in this case, ψ is a class, but not a global formula. Let us assume that χ_1 and χ_2 are also class formulas. The other cases can be solved analogously. Assume further that ψ is contained in $g_k(p)$ for some $k \in \{1, \ldots, \ell\}$ and some process p in $\texttt{act}_{i_k}$. By the $\mathbf{U}^{\leftarrow}_{\equiv}$-rule, we infer (i) $\chi_2 \in g_k(p)$ or (ii) $\chi_1 \in g_k(p)$ and $\mathbf{X}^{\leftarrow}_{\equiv}(\chi_1 \mathbf{U}^{\leftarrow}_{\equiv} \chi_2) \in g_k(p)$. Taking into account that h is correct with respect to successors, we deduce that there is a subset $\{j_1 < \ldots < j_m = k\} \subseteq \{1, \ldots, \ell\}$ such that (i) for every r with $1 < r \leq m$, j_{r-1} is the p-predecessor of j_r, (ii) $\chi_2 \in g_{j_1}(p)$ and (iii) $\chi_1 \in g_{j_r}(p)$ for every r with $1 < r \leq m$. By induction, the construction of w and the assumption that χ_1 and χ_2 are class formulas, we infer that there is some position $j < k$ in the p-class of w such that (i) $(w, j, p) \models \chi_2$ and (ii) $(w, j', p) \models \chi_1$ for every position j' in the p-class of w with $j < j' \leq k$. By the semantics of the $\mathbf{U}^{\leftarrow}_{\equiv}$-operator, we conclude $(w, k, p) \models \chi_1 \mathbf{U}^{\leftarrow}_{\equiv} \chi_2$.

This concludes the "if"-part of the proof. □

The latter lemma encourages to solve the question whether a given DCA $\mathcal{A}$ has a trace satisfying a given formula φ by trying to construct an accepting φ-run for $\mathcal{A}$. However, finding an accepting φ-run subsumes finding an accepting run which is in general not decidable for DCA (Theorem 21 in Section 11.2.1). By Theorem 22, the non-emptiness problem for selective 1-register DCA is decidable. Moreover, from Observations 8 and 9, we know that every run of such a DCA can be simulated by a simplified run where every configuration consists of isolated single processes and couples of processes. Due to these facts, loosely speaking, for a procedure constructing an accepting φ-run for selective 1-register DCA, it suffices to take care about the number of these singles and couples in configurations. This idea paves the way for solving the model checking problem for selective 1-register DCA and RB-DLTL through a reduction to reachability for Multicounter Automata (MCA) introduced in Section 3.2.2.

Theorem 28. *The problem* ModCheck*(selective 1-DCA,RB-DLTL) is decidable.*

Proof. We reduce EModCheck(selective 1-DCA,RB-DLTL) to the reachability problem for MCA which is decidable [160, 134]. As RB-DLTL is closed under negation, the decidability of ModCheck(selective 1-DCA,RB-DLTL) follows.

Recall that MCA are counter machines which contain only increment and decrement transitions, but no zero-tests. Given an MCA $\mathcal{M}$ and a state $\texttt{target}$ of $\mathcal{M}$, the reachability problem for $\mathcal{M}$ asks whether from the initial configuration, i.e., the unique configuration where the state of $\mathcal{M}$ is initial and all counters have value 0, the configuration where the state is $\texttt{target}$ and all counter values are 0 is reachable.

Let $\mathcal{A} = (A, \{r\}, S, s_0, \delta, F)$ be a selective 1-DCA with the single register r and let ψ be an RB-DLTL-formula. We will describe the construction of an MCA $\mathcal{M}_{\mathcal{A},\psi}$ which reaches a designated state $\texttt{target}$ if and only if $\mathcal{A}$ has a trace satisfying ψ. We first convert ψ into an equivalent formula φ in negation normal form. By Lemma 18, $\mathcal{A}$ has a trace satisfying φ if and only if there is an accepting φ-run of $\mathcal{A}$. Recall that an accepting φ-run consists of an accepting run of $\mathcal{A}$ and some suitable validity mapping associating the actions in the run with consistent tuples. We also recall that by Observation 9, $\mathcal{A}$ has an accepting run if and only if it has an accepting simplified run $\tau = c_0 \rightarrow^{\texttt{sim}}_{\mathcal{A}} c_1 \ldots c_{n-1} \rightarrow^{\texttt{sim}}_{\mathcal{A}} c_n$ where c_0 is initial and for every i with $0 < i \leq n$, c_i is a simplified successor of c_{i-1}. The latter means that there is some configuration c'_i with $c_{i-1} \xrightarrow{\texttt{act}_i}_{\mathcal{A}} c'_i$ and c_i results from c'_i by deleting the register input of every process p such that $\mathbf{r}(p)(r) = p'$ for some

process p', but $\mathbf{r}(p')(r) \neq p$. We further know from Observation 8 that every configuration in a simplified run of a selective 1-register DCA must consist of single processes called singles and pairs of processes called couples. A single p in a configuration $c = (P, \mathbf{s}, \mathbf{r})$ does not have any connection to other processes, i.e., $\mathbf{r}(p)(r) = \bot$ and there is no process $p' \in P$ with $\mathbf{r}(p')(r) = p$. The two processes p_1 and p_2 in a couple have only connection to each other, i.e., $\mathbf{r}(p_1)(r) = p_2$, $\mathbf{r}(p_2)(r) = p_1$ and there is no other process p', besides p_1 and p_2, with $\mathbf{r}(p')(r) = p_1$ or $\mathbf{r}(p')(r) = p_2$. These observations justify to construct $\mathcal{M}_{\mathcal{A},\psi}$ in such a way that it simulates an accepting φ-run (τ, h) of $\mathcal{A}$ where τ is simplified. In the sequel, we will first give a rough idea on how $\mathcal{A}$-configurations within a φ-run are represented by $\mathcal{M}_{\mathcal{A},\psi}$-configurations. Then, we will informally describe the overall behaviour of $\mathcal{M}_{\mathcal{A},\psi}$. Finally, we will explain in more detail how $\mathcal{A}$-transitions are simulated by $\mathcal{M}_{\mathcal{A},\psi}$.

For each pair $(s, C) \in S \times 2^{\texttt{Closure}(\varphi)}$, the counter machine is equipped with a counter $\texttt{cnt}_{(s,C)}$. Additionally, it has for each multiset $\{\!|\ (s_1, C_1), (s_2, C_2)\ |\!\}$ consisting of two pairs $(s_1, C_1), (s_2, C_2) \in S \times 2^{\texttt{Closure}(\varphi)}$, a counter which we denote by $\texttt{cnt}_{\{\!|(s_1,C_1),(s_2,C_2)|\!\}}$. Let $\tau = c_0 \rightarrow^{\texttt{sim}}_{\mathcal{A}} c_1 \ldots c_{n-1} \rightarrow^{\texttt{sim}}_{\mathcal{A}} c_n$ be a simplified run of $\mathcal{A}$ where for every i with $0 < i \leq n$, it holds that $c_{i-1} \xrightarrow{\texttt{act}_i}_{\mathcal{A}} c'_i$ for some action $\texttt{act}_i$ and some configuration c'_i such that c_i is a simplification of c'_i. Furthermore, let $I = \{i_1 < \ldots < i_\ell\}$ be the set of indices i_k with $\texttt{act}_{i_k} \neq \varepsilon$ and h be a validity mapping with $h(k) = [C_k, \texttt{act}_{i_k}, g_k]$ for every $k \in \{1, \ldots, \ell\}$ such that (τ, h) is a φ-run. We describe how configurations of such a φ-run are represented by $\mathcal{M}_{\mathcal{A},\psi}$-configurations. For every i with $i_1 \leq i \leq n$, we call a tuple $[C_k, \texttt{act}_{i_k}, g_k]$ the *closest predecessor of i in I* if i_k is the greatest index in I such that $i_k \leq i$. The tuple $[C_k, \texttt{act}_{i_k}, g_k]$ is called the *closest p-predecessor of i in I* if i_k is the greatest number in I such that $i_k \leq i$ and p occurs in $\texttt{act}_{i_k}$. Observe that for every i with $i < i_1$, the action $\texttt{act}_i$ is not visible in the trace of τ and configuration c_i consists of a single process. Such a configuration is represented by some $\mathcal{M}_{\mathcal{A},\psi}$-configuration $c_i^{\mathcal{M}}$ where all counters have value 0 and the state of the single process is encoded in the state of $c_i^{\mathcal{M}}$. We now consider a configuration c_i with $i \geq i_1$. Notice that every process in c_i must occur in some $\texttt{act}_{i_k}$ with $i_k \leq i$, because the initial process must have created at least one process and all processes, besides the initial one, must have been created at some time. Let $[C, \texttt{act}, g]$ be the closest predecessor of i in I and let $[C_p, \texttt{act}_p, g_p]$ the closest p-predecessor of i in I for every process p in c_i. The configuration c_i is represented by a $\mathcal{M}_{\mathcal{A},\psi}$-configuration $c_i^{\mathcal{M}}$ with the following properties:

- The set C is encoded in the state of $c_i^{\mathcal{M}}$. We call this set the *global set* encoded in $c_i^{\mathcal{M}}$.
- For every $\mathcal{A}$-state $s \in S$ and every set $C \subseteq \texttt{Closure}(\varphi)$, the value of counter $\texttt{cnt}_{(s,g_p(p))}$ corresponds to the number of singles p in c_i which are in state s and for which $g_p(p) = C$. The set C is called the *local set* of process p in $c_i^{\mathcal{M}}$.
- Likewise, for every two $\mathcal{A}$-states $s_1, s_2 \in S$ and sets $C_1, C_2 \subseteq \texttt{Closure}(\varphi)$, the value of counter $\texttt{cnt}_{\{\!|(s_1,C_1),(s_2,C_2)|\!\}}$ corresponds to the number of couples $\{\!|\ p_1, p_2\ |\!\}$ in c_i such that p_1 is in state s_1, p_2 is in state p_2, $g_{p_1}(p_1) = C_1$ and $g_{p_2}(p_2) = C_2$. Like above, we refer to C_1 and C_2 as the local sets of p_1 and p_2, respectively, in $c_i^{\mathcal{M}}$.

We now describe in an informal manner how $\mathcal{M}_{\mathcal{A},\psi}$ simulates an accepting φ-run of $\mathcal{A}$. At the beginning of the simulation, all counters are 0. As long as the initial process of the run does not create any new process, the machine $\mathcal{M}_{\mathcal{A},\psi}$ just takes care about the state of the initial process. When the initial process performs its first create action, the machine $\mathcal{M}_{\mathcal{A},\psi}$ chooses a global set C and two local sets C_1 and C_2 such that (i) C is initial with respect to global formulas and contains φ, (ii) C_1 and C_2 are initial with respect to class formulas, and (iii) the sets C, C_1 and C_2 represent a consistent tuple. The machine enters a configuration where the global set is C and the counter $\texttt{cnt}_{\{\!|(s_1,C_1),(s_2,C_2)|\!\}}$, where s_1 and s_2 are the new states of the two current processes, is incremented by 1. In each simulation of further transitions leading from an $\mathcal{A}$-configuration to the next one, the

machine possibly changes the encoded global set and updates some counters such that the tuple represented by the current global and local sets is consistent with the previous one. Later, we will explain in detail that for each simulated $\mathcal{A}$-transition, it suffices to update at most three counters. The machine further checks that the atomic formulas `start` and `end` occur at most in the first and last global sets, respectively. Likewise, it is checked that the formulas $\texttt{start}_=$ and $\texttt{end}_=$ occur at most in first and last local sets of processes, respectively. In a situation where the current global set is final with respect to global formulas, $\mathcal{M}_{\mathcal{A},\psi}$ decides non-deterministically that it has reached the final configuration of the simulated φ-run and enters the *decrement phase*. In this phase, it decrements arbitrarily often all counters $\texttt{cnt}_{(s,C)}$ and $\texttt{cnt}_{\{\!|(s_1,C_1),(s_2,C_2)|\!\}}$ where s, s_1 and s_2 are accepting states from S and C, C_1 and C_2 are final with respect to class formulas. Then, it moves to state `target`. It is easy to see that $\mathcal{M}_{\mathcal{A},\psi}$ reaches a configuration where the state is `target` and all counter values are 0 if and only if it reaches, just before entering the decrement phase, a configuration where (i) the global set is final, (ii) all counters $\texttt{cnt}_{(s,C)}$ where s is not accepting or C is not final with respect to class formulas have value 0, and (iii) all counters $\texttt{cnt}_{\{\!|(s_1,C_1),(s_2,C_2)|\!\}}$ where one of the states s_1 and s_2 is not accepting or one of the sets C_1 and C_2 is not final with respect to class formulas have value 0. Thus, `target` is reachable in $\mathcal{M}_{\mathcal{A},\psi}$ if and only there is an accepting φ-run for $\mathcal{A}$.

Now, we explain in more detail how $\mathcal{A}$-transitions within a simplified φ-run are simulated by $\mathcal{M}_{\mathcal{A},\psi}$. In order to facilitate the explanations, we make use of consistent tuples $[C, \texttt{act}, g]$ where $\texttt{act} \in \texttt{Actions}(A, N)$ for some set N of process names and, accordingly, g maps elements from N to subsets of $\texttt{Closure}(\varphi)$. We skip the description of the simulation of $\mathcal{A}$-transitions which are executed at configurations containing only the initial process and focus on transitions at configurations containing at least two processes. Now, let c_1 be an $\mathcal{A}$-configuration and $c_1^{\mathcal{M}}$ be the representing $\mathcal{M}_{\mathcal{A},\psi}$-configuration. Since we assume that c_1 contains at least two processes, the φ-run leading to c_1 has to contain at least one create action Therefore, $c_1^{\mathcal{M}}$ must encode some global set which we denote in the sequel by C. We distinguish between the different $\mathcal{A}$-transitions which can be fired at c_1. It is worth mentioning that the simulation of a transition leading from c_1 to some other $\mathcal{A}$-configuration c_2 can require several consecutive $\mathcal{M}_{\mathcal{A},\psi}$-transitions. In the following, we neglect the descriptions of the intermediate $\mathcal{M}_{\mathcal{A},\psi}$-configurations and give directly the configuration $c_2^{\mathcal{M}}$ encoding c_2.

`Local` Since local actions are not visible in traces, they do not require an update of global or local sets. Assume that there is a transition $(s_1, \lambda, s_2) \in \delta$. This means that the machine $\mathcal{M}_{\mathcal{A},\psi}$ can enter a configuration which has the same global set as $c_1^{\mathcal{M}}$ and results from $c_1^{\mathcal{M}}$ either (i) by decrementing a counter $\texttt{cnt}_{(s_1,C_1)}$ by 1 and incrementing $\texttt{cnt}_{(s_2,C_1)}$ by 1, or (ii) by decrementing a counter $\texttt{cnt}_{\{\!|(s_1,C_1),(s',C_2)|\!\}}$ by 1 and incrementing $\texttt{cnt}_{\{(s_2,C_1),(s',C_2)\}}$ by 1. Observe that case (i) corresponds to the execution of a local action by some single and the other case to the execution of a local action by some process within a couple.

`Register resetting` Reset actions also belong to those actions which are not visible in traces. Recall that the execution of a reset action by a single only changes the state of the single. In contrast, the execution of such an action by a process within a couple results in two new singles. We assume that there is a transition $(s_1, \texttt{res}(r), s_2) \in \delta$. The machine $\mathcal{M}_{\mathcal{A},\psi}$ can enter a configuration which has the same global set as $c_1^{\mathcal{M}}$ and results from $c_1^{\mathcal{M}}$ either (i) by decrementing a counter $\texttt{cnt}_{(s_1,C_1)}$ by 1 and incrementing $\texttt{cnt}_{(s_2,C_1)}$ by 1, or (ii) by decrementing a counter $\texttt{cnt}_{\{\!|(s_1,C_1),(s',C_2)|\!\}}$ by 1 and incrementing both counters $\texttt{cnt}_{(s_2,C_1)}$ and $\texttt{cnt}_{(s',C_2)}$, respectively, by 1.

`Create` Since create actions are visible in traces, they require an update of the global and local sets. We again distinguish between the cases whether a spawn action is performed by a single or by a process within a couple. Let $(s_1, r \longleftarrow \texttt{crt}(s, r), s_2)$ be a transition in δ and

$[C', \mathtt{crt}(n_1, n_2), g]$ a consistent tuple such that C' is a successor of (the current global set) C with respect to global formulas and $g(n_2)$ is initial with respect to class formulas. Moreover, let $\hat{C}$ be a predecessor of $g(n_1)$ with respect to class formulas. The machine $\mathcal{M}_{\mathcal{A},\psi}$ can enter a configuration which results from $c_1^{\mathcal{M}}$ by setting the global set to C' and (i) decrementing $\mathtt{cnt}_{(s_1,\hat{C})}$ by 1 and incrementing $\mathtt{cnt}_{\{\!|(s_2,g(n_1)),(s,g(n_2))|\!\}}$ by 1, or (ii) decrementing a counter $\mathtt{cnt}_{\{\!|(s_1,\hat{C}),(s',\hat{C}')|\!\}}$ by 1 and incrementing both counters $\mathtt{cnt}_{(s',\hat{C}')}$ and $\mathtt{cnt}_{\{\!|(s_2,g(n_1)),(s,g(n_2))|\!\}}$, respectively, by 1.

`Selective symbol sending` Send actions are also visible in traces and, therefore, require an update of global and local sets. Remember also that they can only be performed by processes within couples. Finally, recall that in configurations of selective 1-register DCA, the execution of a send action only changes the states of sender and receiver. Let $(s_1, \mathtt{snd}(r,m), s_2)$, $(s_3, \mathtt{rcv}(r,m), s_4) \in \delta$ and $[C', \mathtt{snd}(n_1, n_2, m), g]$ some consistent tuple such that C' is a successor of C with respect to global formulas. Furthermore, let C_1 be a predecessor of $g(n_1)$ and C_2 a predecessor of $g(n_2)$ with respect to class formulas. The machine $\mathcal{M}_{\mathcal{A},\psi}$ can enter a configuration which results from $c_1^{\mathcal{M}}$ by setting the global set to C', decrementing counter $\mathtt{cnt}_{\{\!|(s_1,C_1),(s_3,C_2)|\!\}}$ by 1 and incrementing counter $\mathtt{cnt}_{\{\!|(s_2,g(n_1)),(s_4,g(n_2))|\!\}}$ by 1.

The case for selective ID sending is handled analogously to the last case. ☐

12.1.2 Model Checking with Freeze LTL

Remember from the results in Part A that the satisfiability problem for $\mathrm{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$ (on 1-complete data words) is decidable. Although the non-emptiness problem for selective 1-register DCA is also decidable, the next result surprisingly states that the combination of these formalisms delivers an undecidable model checking problem.

Theorem 29. *The problem* ModCheck*(selective 1-DCA,$\mathrm{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$) is not decidable.*

Proof. We will give a reduction from the reachability problem for Minsky Counter Machines with two counters (2-MCMs, for definition, see Section 3.2.2) to the existential model checking of selective 1-DCA with $\mathrm{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$. As reachability for 2-MCMs is not decidable [162] and $\mathrm{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$ is closed under negation, the result follows.

We recall that, given a Minsky Counter Machine $\mathcal{M}$ and a state `target` of $\mathcal{M}$, the reachability problem for $\mathcal{M}$ and `target` asks whether $\mathcal{M}$ has a run reaching configuration $(\mathtt{target}, 0, 0)$, that is, the configuration where the state is `target` and both counter values are 0. Now, let $\mathcal{M} = (2, S, s_0, \delta)$ be a 2-MCM and `target` a state from S. We will construct a selective 1-register DCA $\mathcal{A}$ and an $\mathrm{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$-formula φ such that $\mathcal{M}$ reaches $(\mathtt{target}, 0, 0)$ if and only if $\mathcal{A}$ has a trace satisfying φ.

Without loss of generality, we assume that the initial state s_0 of $\mathcal{M}$ is not equal to `target` (otherwise, the construction of some DCA and a formula is trivial). In the sequel, we will first recall the consistency properties from Section 3.2.2 ensuring that a sequence of transitions of $\mathcal{M}$ induces a correct run reaching $(\mathtt{target}, 0, 0)$. Then, we will explain how sequences of transitions can be encoded as traces. After that, we will describe the construction of a selective 1-DCA $\mathcal{A}$ whose traces are such encodings. We will see that the traces of $\mathcal{A}$ already satisfy some of the consistency properties. Finally, we will construct an $\mathrm{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$-formula φ expressing the remaining properties. Hence, by construction, it will follow that $\mathcal{A}$ has a trace satisfying φ if and only if $\mathcal{M}$ reaches $(\mathtt{target}, 0, 0)$.

From Section 3.2.2 we know that a sequence $\tau = (s_1, \mathtt{act}_1, s'_1), \ldots, (s_n, \mathtt{act}_n, s'_n)$ of transitions from δ represents an $\mathcal{M}$-run reaching $(\mathtt{target}, 0, 0)$ if and only if the following consistency properties are satisfied.

- *Consistency with respect to states*: We have that $s_1 = s_0$, $s_n = \texttt{target}$ and for all i with $1 \leq i < n$, $s'_i = s_{i+1}$.
- *Consistency with respect to counters*: There is a bijection m from the set $\texttt{DECS}_\tau = \{i \mid 1 \leq i \leq n \text{ and } \texttt{act}_i \text{ is a decrement action}\}$ to the set $\texttt{INCS}_\tau = \{i \mid 1 \leq i \leq n \text{ and } \texttt{act}_i \text{ is an increment action}\}$ such that for each counter k and index $i \in \texttt{DECS}_\tau$ with $\texttt{act}_i = \texttt{dec}_k$, it holds $m(i) < i$ and $\texttt{act}_{m(i)} = \texttt{inc}_k$.
- *Consistency with respect to zero-tests*: For every counter k and index $i \in \texttt{DECS}_\tau$ with $\texttt{act}_i = \texttt{dec}_k$, there is no ℓ with $m(i) < \ell < i$ and $\texttt{act}_\ell = \texttt{ifzero}_k$.

We encode sequences of $\mathcal{M}$-transitions by traces where the transitions are represented by message symbols. More precisely, the traces are defined over the proposition set $\texttt{Prop}^A_{\texttt{act}}$ and the attribute set $\texttt{Attr}^A_{\texttt{act}}$ where A contains for every $\mathcal{M}$-transition $(s, \texttt{act}, s')$, the message symbol $(s, \texttt{act}, s')$ of arity 0. Given a trace w over $\texttt{Prop}^A_{\texttt{act}}$ and $\texttt{Attr}^A_{\texttt{act}}$, we call w' a *restriction of* w *to send positions* if it results from w by eliminating all positions representing create actions. Let $\tau = (s_1, \texttt{act}_1, s'_1) \dots (s_n, \texttt{act}_n, s'_n)$ be a sequence of $\mathcal{M}$-transitions, w a tace over $\texttt{Prop}^A_{\texttt{act}}$ and $\texttt{Attr}^A_{\texttt{act}}$ and w' its restriction to send positions. We call w an *encoding* of τ if w' is of length n and every position i of w' carries the propositions $\texttt{snd}$ and $(s_i, \texttt{act}_i, s'_i)$. Figure 12.1 presents for a sequence τ, a trace encoding and its restriction to send positions.

$$\tau = (s_0, \texttt{inc}_1, s_1)(s_1, \texttt{inc}_2, s_2)(s_2, \texttt{dec}_2, s_3)(s_3, \texttt{ifzero}_2, s_4)(s_4, \texttt{dec}_1, \texttt{target})$$

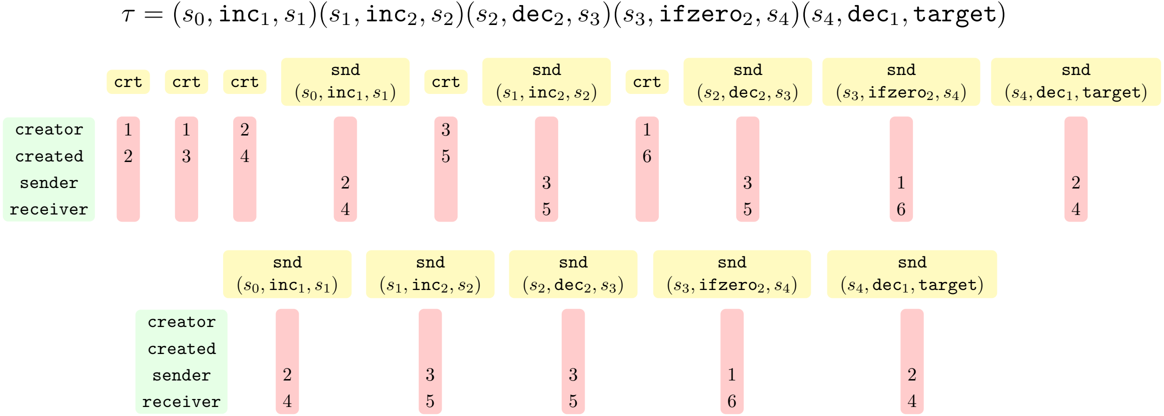


Figure 12.1: An encoding of τ and its restriction

We now describe the construction of the selective DCA $\mathcal{A}$ which has only one register r and whose traces are encodings of sequences of $\mathcal{M}$-transitions. Let $\{t_1^{\texttt{inc}_1}, \dots, t_{n_1}^{\texttt{inc}_1}\}$, $\{t_1^{\texttt{dec}_1}, \dots, t_{n_2}^{\texttt{dec}_1}\}$, $\{t_1^{\texttt{ifzero}_1}, \dots, t_{n_3}^{\texttt{ifzero}_1}\}$, $\{t_1^{\texttt{inc}_2}, \dots, t_{n_4}^{\texttt{inc}_2}\}$, $\{t_1^{\texttt{dec}_2}, \dots, t_{n_5}^{\texttt{dec}_2}\}$, $\{t_1^{\texttt{ifzero}_2}, \dots, t_{n_6}^{\texttt{ifzero}_2}\}$ be, respectively, the sets of all $\texttt{inc}_1$-, $\texttt{dec}_1$-, $\texttt{ifzero}_1$-, $\texttt{inc}_2$-, $\texttt{dec}_2$- and $\texttt{ifzero}_2$-transitions in δ. The automaton $\mathcal{A}$ is depicted in Figure 12.2. In the following, whenever we say that a process *sends* (or *receives*) *a transitions* t, we mean that it sends (or receives) the message symbol standing for t. The automaton $\mathcal{A}$ consists basically of four parts: two parts dealing with the sending and receiving of $\texttt{inc}$- and $\texttt{dec}$-transitions and two parts dealing with the sending and receiving of $\texttt{ifzero}$-transitions of $\mathcal{M}$. We give a high-level description of the behaviour of the processes induced by $\mathcal{A}$. First, the initial process spawns arbitrarily many new processes starting in state x_1 whose task is to deal with $\texttt{inc}$- and $\texttt{dec}$-transitions. Then, it spawns a process starting in state z and sends him arbitrarily many $\texttt{ifzero}$-transitions. The process starting in z only serves for receiving $\texttt{ifzero}$-transitions. Each process starting in x_1 first spawns a new process starting in y and builds a couple with this new

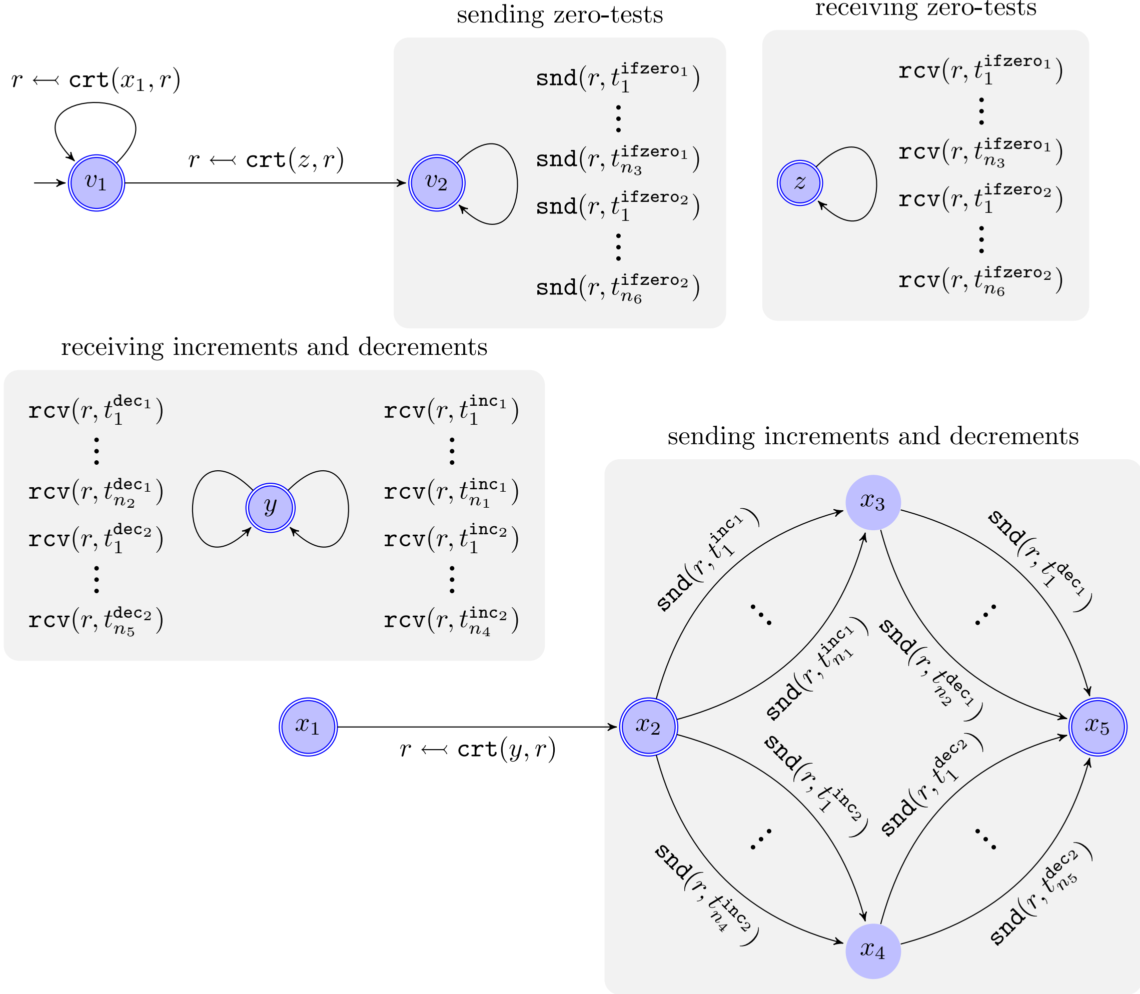


Figure 12.2: The selective 1-DCA $\mathcal{A}$ whose traces encode transition sequences of $\mathcal{M}$

process. Subsequently, it sends an $\texttt{inc}_k$-transition for some counter k and sends then a $\texttt{dec}_k$-transition for the same counter. Thus, processes starting in y only serve for receiving one $\texttt{inc}$- and one subsequent $\texttt{dec}$-transition for the same counter. Observe that in each trace of $\mathcal{A}$, it holds that for every $\texttt{inc}_k$-position for some counter k, there is exactly one subsequent $\texttt{dec}_k$-position with the same process ID and vice-versa. From this, it directly follows that for every trace of $\mathcal{A}$ there is a bijection m guaranteeing consistency with respect to counters.

Now, we construct the formula φ using the single freeze variable x. As the traces of $\mathcal{A}$ are already consistent with respect to counters, it remains to express the other two consistency properties. The property requiring consistency with respect to states does not refer to data values, wherefore its formulation is an easy task. We only give the formula for the third property requiring consistency with respect to zero-tests. Notice that it follows from the construction of $\mathcal{A}$ that every $\texttt{inc}$-position carries the same process ID as its corresponding $\texttt{dec}$-position. Moreover, these IDs do not occur at any other $\texttt{inc}$- or $\texttt{dec}$-position. Thus, it suffices to express that for every $k \in \{1, 2\}$ and every $\texttt{inc}_k$-position, there is no $\texttt{ifzero}_k$-position until the corresponding $\texttt{dec}_k$-position with the same

ID:

$$\bigwedge_{k\in\{1,2\}} \mathbf{G}\Big[\bigvee_{s,s'\in S}(s,\mathtt{inc}_k,s') \rightarrow \Downarrow^x_{\mathtt{@sender}}.\Big(\big(\neg\bigvee_{s,s'\in S}(s,\mathtt{ifzero}_k,s')\big)\mathbf{U}\big(\Uparrow^x_{\mathtt{@sender}} \wedge \bigvee_{s,s'\in S}(s,\mathtt{dec}_k,s')\big)\Big)\Big].$$

□

12.2 Model Checking of Process Register Automata

In this section, we investigate the model checking of PRA against formulas from the logics $\mathrm{LTL}^{\Downarrow}$ and $\mathrm{HTL}^{\sim}$.

In case of $\mathrm{LTL}^{\Downarrow}$, we restrict to the future fragment, i.e., $\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$. We show that model checking of PRA with $\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$ with unboundedly many freeze registers is decidable in exponential space. Moreover, we prove that for every $k \geq 1$, model checking with $\mathrm{LTL}^{\Downarrow}_k(\mathbf{X},\mathbf{U})$, i.e., $\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$ with at most k freeze registers, is complete for PSpace. However, we cannot answer the question whether the upper bound for full $\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$ is tight, and we also have not figured out yet how our proof techniques may be extended to past operators.

In case of $\mathrm{HTL}^{\sim}$, we can show that model checking of PRA with the 1-variable fragment of $\mathrm{HTL}^{\sim}$ is ExpSpace-complete. If one more variable is added to the logic, the problem remains decidable, but has non-elementary complexity. Observe that all mentioned logics, besides $\mathrm{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$, have an undecidable satisfiability problem, even on data words with a single data value per position.

12.2.1 Model Checking with Freeze LTL

We will first show that the model checking problem for PRA and $\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$ with unboundedly many freeze registers is in ExpSpace. Similar to our decidability procedure for the non-emptiness problem of PRA in Section 11.3.1, our proof will be carried out using symbolic runs. It will easily follow from our construction that for every $k \geq 1$, the complexity of model checking with $\mathrm{LTL}^{\Downarrow}_k(\mathbf{X},\mathbf{U})$ drops down to PSpace. Moreover, by a reduction from the satisfiability of LTL, we will follow a PSpace lower bound for all mentioned logics. From these results, we will derive that for every $k \geq 1$, model checking of PRA with $\mathrm{LTL}^{\Downarrow}_k(\mathbf{X},\mathbf{U})$ is PSpace-complete. However, the precise complexities in the cases of $\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$ and $\mathrm{LTL}^{\Downarrow}$ remain open questions. At appropriate point, we will mention the complications we have to deal with if we extend our construction in the decision procedure to past operators.

Solving model checking with $\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$ on symbolic runs

We will prove that the existential model checking problem EModCheck(PRA, $\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$) is in ExpSpace. Since ExpSpace is closed under complementation and $\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$ is closed under negation, it will follow that ModCheck(PRA, $\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$) is also in ExpSpace. Thanks to Observation 11, we will work with symbolic configurations and runs of PRA. The procedure we are going to describe checks whether a given PRA has an accepting symbolic run such that a given formula is satisfied by the traces of corresponding concrete runs.

To facilitate the distinction between the registers of $\mathrm{LTL}^{\Downarrow}$ and those of PRA, we will notate $\mathrm{LTL}^{\Downarrow}$-registers by $x, x_1, x_2, \ldots$ and call them *freeze variables*. Mappings from these variables will be called *freeze assignments*. The registers of a PRA will be notated, as usual, by $r, r_1, r_2, \ldots$ and associated mappings will be called *register assignments*.

Before diving into the technical details of our procedure, we roughly describe the underlying idea. Given a PRA $\mathcal{A}$ and a formula φ, our procedure checks whether there is a trace of $\mathcal{A}$ satisfying

φ. Observe that during the evaluation of an LTL$^{\Downarrow}$(**X**, **U**)-formula, a $\Downarrow^{x}_{\texttt{@a}}$-operation at a trace position resulting from a transition $c \xrightarrow{\nu(\texttt{act})}_{\mathcal{A}} c'$ from some configuration c to some configuration c', stores a process $\nu(r)$ assigned to some $\mathcal{A}$-register r into the freeze variable x. Likewise, $\Uparrow^{x}_{\texttt{@a}}$ tests whether the process stored in x equals to some process $\nu(r')$ assigned to some $\mathcal{A}$-register r'. Due to the definition of runs, we know that a process which is stored in some freeze variable x, but not in any $\mathcal{A}$-register of a current configuration, cannot occur at any following trace position. Thus, since our logic does not allow past operators, equality tests with x will evaluate to false in the rest of the evaluation process (unless x is overwritten by the input of some current $\mathcal{A}$-register). Hence, as soon as the input of a freeze variable is deleted from a current register assignment, it has not to be "remembered" anymore. Therefore, a freeze assignment at a trace position can be adequately simulated by a (symbolic) mapping from freeze variables to $\mathcal{A}$-registers which paves the way for working with symbolic runs. Due to these observations, instead of searching for a concrete trace satisfying φ, our procedure tries to construct an (extended) symbolic run. Every transition $t = sc \xrightarrow{\texttt{act}}{}^{s}_{\mathcal{A}} sc'$ in this run is equipped with a set C of pairs (ψ, σ) where ψ is a sub-formula of φ and $\sigma \in [X \rightharpoonup R]$ is a freeze assignment mapping freeze variables to $\mathcal{A}$-registers. The intuition is that for every concrete trace position resulting from a concrete instantiation $c \xrightarrow{\nu(\texttt{act})}_{\mathcal{A}} c'$ of t and for every pair $(\psi, \sigma) \in C$, the formula ψ is true on that position under a freeze assignment $\lambda \in [X \rightharpoonup \mathbb{P}]$ where λ results from σ by replacing every $\mathcal{A}$-register r by $\nu(r)$. A crucial point in the construction is the preservation of consistency between sets C_1 and C_2 assigned to two consecutive transitions $sc_1 \xrightarrow{\texttt{act}_1}{}^{s}_{\mathcal{A}} sc'_1 \xrightarrow{\texttt{act}_2}{}^{s}_{\mathcal{A}} sc_2$. Here, we have to make sure that for every pair $(\mathbf{X}\psi, \sigma_1) \in C_1$, there is some $(\psi, \sigma_2) \in C_2$ where σ_2 is an update of σ_1 based on the following observations: If $\texttt{act}_2$ is a send action, then, due to the definition of $\mathcal{A}$-runs, the $\mathcal{A}$-registers do not change their inputs in concrete instantiations when going from sc'_1 to sc_2. Therefore, there is no need to update σ_1 and we set $\sigma_2 = \sigma_1$. If, conversely, $\texttt{act}$ is a create action, it means that the input of $r = \texttt{created}(\texttt{act}_2)$ is updated by some fresh process in concrete instantiations of sc_2. Observe that the "old" process in r cannot occur in the rest of the current run anymore. In this case, we obtain σ_2 from σ_1 by mapping every freeze variable x with $\sigma_1(x) = r$ to a pseudo-register $r_\perp$. In doing so, we symbolically express that every x with $\sigma_2(x) = r_\perp$ points to a process which cannot belong to the current or following configurations. The main goal of our procedure is to find an accepting extended symbolic $\mathcal{A}$-run where the first transition is equipped with a set C containing $(\varphi, \sigma[X \mapsto \perp])$.

We now explain the technical details. Similar to the negation normal form for RB-DLTL, given in Section 12.1.1, we introduce a negation normal form for LTL$^{\Downarrow}$(**X**, **U**)-formulas. Like in the case for RB-DLTL, an LTL$^{\Downarrow}$(**X**, **U**)-formula in negation normal form can contain, in addition to the usual atomic formulas and operators, the release operator **R**, the atomic formula `end` and the atomic formula $\perp_{\texttt{a}}$ for all attributes `a`. We briefly recall their semantics. The formula $\texttt{end} \equiv \neg\mathbf{X}\texttt{true}$ only holds at the last positions of data words, $\perp_{\texttt{a}} \equiv \neg \Downarrow^{x}_{\texttt{@a}}. \Uparrow^{x}_{\texttt{@a}}$ expresses that the value of attribute `a` at the current position is not defined and the operator **R** is the dual of **U**, i.e., for all formulas φ_1 and φ_2, we have $\varphi_1\mathbf{U}\varphi_2 \equiv \neg(\neg\varphi_1\mathbf{R}\neg\varphi_2)$ and $\varphi_1\mathbf{R}\varphi_2 \equiv \neg(\neg\varphi_1\mathbf{U}\neg\varphi_2)$. An LTL$^{\Downarrow}$(**X**, **U**)-formula φ is in negation normal form if all negation symbols are immediately in front of propositions or the $\Uparrow$-operator, i.e., if it results from the following formation rules:

$$\varphi := p \mid \neg p \mid \Uparrow^{x}_{\texttt{@a}} \mid \neg \Uparrow^{x}_{\texttt{@a}} \mid \texttt{end} \mid \perp_{\texttt{@a}} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \Downarrow^{x}_{\texttt{@a}}.\varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi \mid \varphi\mathbf{R}\varphi$$

where x is a freeze variable, p a proposition and `a` an attribute.
Using De Morgan's rules and the equivalences $\neg\mathbf{X}\varphi \equiv \texttt{end} \vee \mathbf{X}\neg\varphi$, $\neg \Downarrow^{x}_{\texttt{@a}}.\psi \equiv \perp_{\texttt{@a}} \vee \Downarrow^{x}_{\texttt{@a}}.\neg\psi$ and $\neg(\varphi_1\mathbf{U}\varphi_2) \equiv (\neg\varphi_1\mathbf{R}\neg\varphi_2)$, every LTL$^{\Downarrow}$(**X**, **U**)-formula can be converted into an equivalent formula in negation normal form with length linear with respect to the size of the original formula.

Let $\mathcal{A} = (A, R, r_0, S, s_0, \delta, F)$ be a PRA and φ an LTL$^{\Downarrow}$(**X**, **U**)-formula in negation normal form which uses propositions from $\texttt{Prop}^{A}_{\texttt{act}}$, attributes from $\texttt{Attr}^{A}_{\texttt{act}}$ and freeze variables from some set X. The *closure set* $\texttt{Closure}(\varphi)$ of φ is the smallest set such that

- every sub-formula of φ is contained in $\texttt{Closure}(\varphi)$,
- for every sub-formula $\psi_1 \mathbf{U} \psi_2$ of φ, the formula $\mathbf{X}(\psi_1 \mathbf{U} \psi_2)$ is contained in $\texttt{Closure}(\varphi)$, and
- for every sub-formula $\psi_1 \mathbf{R} \psi_2$ of φ, the formula $\mathbf{X}(\psi_1 \mathbf{R} \psi_2)$ is contained in $\texttt{Closure}(\varphi)$.

Let $R' = R \cup \{r_\perp\}$. We now define *consistent tuples* $[C, \texttt{act}]$ for φ and $\mathcal{A}$ with $C \subseteq \texttt{Closure}(\varphi) \times [X \rightharpoonup R']$ and $\texttt{act} \in \texttt{Actions}(A, R)$. Intuitively, a tuple $[C, \texttt{act}]$ is called consistent if for a possible trace position that is resulting from $\texttt{act}$, C describes a possible set of pairs (ψ, σ) such that ψ holds at that position under the freeze assignment σ. The registers in σ which belong to R represent corresponding processes in the current configuration and the special register $r_\perp$ stands for "old" processes which cannot occur in the suffix of the underlying trace. Formally, a tuple $[C, \texttt{act}]$ is consistent if it fulfills the following rules for every freeze assignment $\sigma \in [X \rightharpoonup R']$, attribute $\texttt{a}$, freeze variable x and formulas ψ, ψ_1 and ψ_2:

`rules for propositions`:
- If $(\texttt{crt}, \sigma) \in C$ or $(\neg\texttt{snd}, \sigma) \in C$, then, $\texttt{act}$ is a create action.
- If $(\texttt{snd}, \sigma) \in C$ or $(\neg\texttt{crt}, \sigma) \in C$, then, $\texttt{act}$ is a send action.
- If $(m, \sigma) \in C$ for some message symbol m, then, $\texttt{act}$ is a send action with message symbol m.
- If $(\neg m, \sigma) \in C$ for some message symbol m, then, $\texttt{act}$ is not a send action with message symbol m.

$\Uparrow$`-rules`:
- If $(\Uparrow^x_{\texttt{@a}}, \sigma) \in C$, then, parameter $\texttt{a}(\texttt{act})$ is defined for $\texttt{act}$ and $\sigma(x) = \texttt{a}(\texttt{act})$.
- If $(\neg \Uparrow^x_{\texttt{@a}}, \sigma) \in C$, then, parameter $\texttt{a}(\texttt{act})$ is not defined or $\sigma(x) \neq \texttt{a}(\texttt{act})$.

$\perp$`-rule`: If $(\perp_{\texttt{@a}}, \sigma) \in C$, then, parameter $\texttt{a}$ is not defined for $\texttt{act}$.

$\Downarrow$`-rule`: If $(\Downarrow^x_{\texttt{@a}}.\psi, \sigma) \in C$, then, parameter $\texttt{a}(\texttt{act})$ is defined and $(\psi, \sigma[x \mapsto \texttt{a}(\texttt{act})]) \in C$.

$\wedge$`-rule`: If $(\psi_1 \wedge \psi_2, \sigma) \in C$, then, $(\psi_1, \sigma) \in C$ and $(\psi_2, \sigma) \in C$.

$\vee$`-rule`: If $(\psi_1 \vee \psi_2, \sigma) \in C$, then, $(\psi_1, \sigma) \in C$ or $(\psi_2, \sigma) \in C$.

$\mathbf{U}$`-rule`: If $(\psi_1 \mathbf{U} \psi_2, \sigma) \in C$, then,

- $(\psi_2, \sigma) \in C$ or
- $(\psi_1, \sigma) \in C$ and $(\mathbf{X}(\psi_1 \mathbf{U} \psi_2), \sigma) \in C$.

$\mathbf{R}$`-rule`: If $(\psi_1 \mathbf{R} \psi_2, \sigma) \in C$, then,

- $(\psi_2, \sigma) \in C$ and
- $(\texttt{end}, \sigma) \in C$ or $(\psi_1, \sigma) \in C$ or $(\mathbf{X}(\psi_1 \mathbf{R} \psi_2), \sigma) \in C$.

A consistent tuple $[C', \texttt{act}']$ is called a *successor* of some consistent tuple $[C, \texttt{act}]$ (notated as $[C, \texttt{act}] \rightarrow [C', \texttt{act}']$) if for every $(\mathbf{X}\psi, \sigma) \in C$, it holds:

- if $\texttt{act}'$ is some create action with $\texttt{created}(\texttt{act}') = r$ for some register $r \in R$, then, $(\psi, \sigma[X' \mapsto r_\perp]) \in C'$ where $X' = \{x \mid \sigma(x) = r\}$,
- if $\texttt{act}'$ is some send action, then, $(\psi, \sigma) \in C'$.

Observe that every σ' in C' maps all freeze variables, belonging to processes which cannot occur in subsequent trace positions, to $r_\perp$. A consistent tuple is called *final* if it does not contain any formula of the form $\mathbf{X}\psi$.

Let $\theta' = sc_0 \xrightarrow{\mathtt{act}_1}{}^s_{\mathcal{A}} sc_1 \ldots sc_{n-1} \xrightarrow{\mathtt{act}_n}{}^s_{\mathcal{A}} sc_n$ be a symbolic run of $\mathcal{A}$. An extension of θ' to $\theta = sc_0 \xrightarrow{[C_1,\mathtt{act}_1]}{}^s_{\mathcal{A}} sc_1 \ldots sc_{n-1} \xrightarrow{[C_n,\mathtt{act}_n]}{}^s_{\mathcal{A}} sc_n$ by consistent tuples is called a *symbolic φ-run* of $\mathcal{A}$ if (i) $(\varphi, \sigma[X \mapsto \perp]) \in C_1$, (ii) $[C_i, \mathtt{act}_i] \rightarrow [C_{i+1}, \mathtt{act}_{i+1}]$ for every i with $1 \leq i < n$, and (iii) if there is some i and some freeze assignment σ such that $(\mathtt{end}, \sigma) \in C_i$, then, $i = n$. The sequence θ is called an *accepting* symbolic φ-run, if it is a symbolic φ-run, θ' is accepting and C_n is final.

Before establishing the link between symbolic φ-runs and concrete traces satisfying φ, we formulate an observation on the relationship between freeze assignments which coincide with respect to data values occurring in some suffix of a data word. Given a data word w, let us denote the set of all data values occurring in w by $\mathtt{Val}(w)$. Two freeze assignments $\lambda, \lambda' \in [X \rightharpoonup \mathcal{D}]$ are called *equivalent on a data word $w = w_1 \ldots w_n$ from position i on* (written as $\lambda \overset{w^i}{\equiv} \lambda'$) if for all x with $\lambda(x) \in \mathtt{Val}(w[i, \ldots])$ (recall that $w[i, \ldots]$ denotes the suffix of w starting at i), it holds $\lambda'(x) = \lambda(x)$, and for all x with $\lambda(x) \notin \mathtt{Val}(w[i, \ldots])$, it holds $\lambda'(x) \notin \mathtt{Val}(w[i, \ldots])$.

Observation 15. Let w be a data word, i some position in w and $\lambda, \lambda' \in [X \mapsto \mathcal{D}]$ two freeze assignments such that $\lambda \overset{w^i}{\equiv} \lambda'$. Then, for every $\psi \in \mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$ in negation normal form, it holds $(w, i, \lambda) \models \psi$ if and only if $(w, i, \lambda') \models \psi$.

The correctness of the model checking procedure, we are going to describe, relies on the following lemma.

Lemma 19. *Given a PRA $\mathcal{A}$ and an $\mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$-formula φ in negation normal form, the automaton $\mathcal{A}$ has a trace satisfying φ if and only if there is an accepting symbolic φ-run of $\mathcal{A}$.*

Proof. Let $\mathcal{A} = (A, R, r_0, S, s_0, \delta, F)$ be a PRA and φ an $\mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$-formula in negation normal form using freeze variables from some set X. We first deal with the proof of the "only if"-direction. To this end, assume that φ satisfies a concrete trace $w = w_1 \ldots w_n$ of $\mathcal{A}$ resulting from an accepting concrete run

$$\tau' = (s_0, \nu_0, E_0) \xrightarrow{\nu_1(\mathtt{act}_1)}{}_{\mathcal{A}} (s_1, \nu_1, E_1) \ldots (s_{n-1}, \nu_{n-1}, E_{n-1}) \xrightarrow{\nu_n(\mathtt{act}_n)}{}_{\mathcal{A}} (s_n, \nu_n, E_n).$$

Recall that $w_i = \mathtt{dwrep}(\nu_i(\mathtt{act}_i))$ for every $i \in \{1, \ldots, n\}$. Let

$$\tau = (s_0, \nu_0, E_0) \xrightarrow{[C_1,\nu_1(\mathtt{act}_1)]}{}_{\mathcal{A}} (s_1, \nu_1, E_1) \ldots (s_{n-1}, \nu_{n-1}, E_{n-1}) \xrightarrow{[C_n,\nu_n(\mathtt{act}_n)]}{}_{\mathcal{A}} (s_n, \nu_n, E_n)$$

be an extension of τ where for every formula $\psi \in \mathtt{Closure}(\varphi)$, freeze assignment $\lambda \in [X \rightharpoonup \mathbb{P}]$ and $i \in \{1, \ldots, n\}$, we have $(\psi, \lambda) \in C_i$ if and only $(w, i, \lambda) \models \psi$. Based on τ, we will construct a symbolic accepting φ-run θ for $\mathcal{A}$. Before doing this, we define for freeze assignments $\lambda \in [X \rightharpoonup \mathbb{P}]$ and register assignments $\nu \in [R \rightharpoonup \mathbb{P}]$, functions which "extract" from λ a corresponding freeze assignment from $[X \rightharpoonup R \cup \{r_\perp\}]$. That is, for every freeze assignment $\lambda \in [X \rightharpoonup \mathbb{P}]$ and register assignment $\nu \in [R \rightharpoonup \mathbb{P}]$, we define $\mathtt{extr}_\nu(\lambda) = \{x \mapsto r \mid \lambda(x) = \nu(r)\} \cup \{x \mapsto r_\perp \mid \lambda(x)$ is defined and $\lambda(x) \notin \nu(R)\}$. Then,

$$\theta = sc_0 \xrightarrow{[C'_1,\mathtt{act}_1]}{}^s_{\mathcal{A}} sc_1 \ldots sc_{n-1} \xrightarrow{[C'_n,\mathtt{act}_n]}{}^s_{\mathcal{A}} sc_n$$

is the sequence obtained from τ such that $sc_0 \xrightarrow{\mathtt{act}_1}{}^s_{\mathcal{A}} sc_1 \ldots sc_{n-1} \xrightarrow{\mathtt{act}_n}{}^s_{\mathcal{A}} sc_n$ is $\mathtt{symb}(\tau)$ (whose existence is guaranteed by Observation 11) and for every $i \in \{1, \ldots, n\}$, $C'_i = \{(\psi, \sigma) \mid$ there is $(\psi, \lambda) \in C_i$ with $\sigma = \mathtt{extr}_{\nu_i}(\lambda)\}$. We now show that θ is indeed an accepting symbolic φ-run for $\mathcal{A}$. First of all, notice that it follows from the definition of τ and the construction of θ that $(\varphi, \sigma[X \mapsto \perp]) \in C'_1$. It remains to prove that every $[C'_i, \mathtt{act}_i]$ with $1 \leq i \leq n$ is consistent, every

$[C'_{i+1}, \texttt{act}_{i+1}]$ with $1 < i \leq n$ is a successor of $[C'_i, \texttt{act}_i]$, θ is accepting and $\texttt{end}$ can only occur in C'_n.

We start with the proof the consistency of all $[C'_i, \texttt{act}_i]$ with $1 \leq i \leq n$. We consider an arbitrary $[C'_i, \texttt{act}_i]$. We have to show that all rules for consistent tuples, defined above, hold for $[C'_i, \texttt{act}_i]$. We focus on some interesting cases. For the other cases the argumentation is similar:

- rules for propositions:
  - Let $(\texttt{crt}, \sigma) \in C'_i$ for some σ. It follows from construction that $(\texttt{crt}, \lambda) \in C_i$ with $\sigma = \texttt{extr}_{\nu_i}(\lambda)$. By definition of C_i, $(w, i, \lambda) \models \texttt{crt}$. By the definition of traces, $\nu_i(\texttt{act}_i)$ and, thus, $\texttt{act}_i$ must be a create action.
  - Assume $(\neg m, \sigma) \in C'_i$ for some message symbol m and freeze assignment σ. Again, by construction, $(\neg m, \lambda) \in C_i$ with $\sigma = \texttt{extr}_{\nu_i}(\lambda)$. By definition of C_i, $(w, i, \lambda) \models \neg m$. Consequently, $\nu_i(\texttt{act}_i)$ and $\texttt{act}_i$ cannot represent send actions with message symbol m.
- $\Uparrow$-rules:
  - Assume that $(\Uparrow^x_{\texttt{@a}}, \sigma) \in C'_i$ for some freeze variable x, attribute $\texttt{a}$ and freeze assignment σ. It follows that there must be a pair $(\Uparrow^x_{\texttt{@a}}, \lambda) \in C_i$ with $\sigma = \texttt{extr}_{\nu_i}(\lambda)$. Thus, $(w, i, \lambda) \models \Uparrow^x_{\texttt{@a}}$. Consequently, $\texttt{val}(w, i, \texttt{@a})$ must be defined and $\texttt{val}(w, i, \texttt{@a}) = \lambda(x)$. This means that, $\texttt{a}(\nu_i(\texttt{act}_i))$, i.e., the parameter $\texttt{a}$ of $\nu_i(\texttt{act}_i)$ must be defined and $\lambda(x) = \texttt{a}(\nu_i(\texttt{act}_i))$. We first observe that $\texttt{a}(\texttt{act}_i)$ must also be defined. Moreover, let $r \in R$ be the register such that $\texttt{a}(\nu_i(\texttt{act}_i)) = \nu_i(r) = \lambda(x)$. By the definition of $\nu_i(\texttt{act}_i)$ and the equality $\sigma = \texttt{extr}_{\nu_i}(\lambda)$, it follows that $\texttt{a}(\texttt{act}) = r = \sigma(x)$. Thus, $\sigma(x) = \texttt{a}(\texttt{act})$.
  - Now, let $(\neg \Uparrow^x_{\texttt{@a}}, \sigma) \in C'_i$. Hence, there must be some $(\neg \Uparrow^x_{\texttt{@a}}, \lambda) \in C_i$ with $\sigma = \texttt{extr}_{\nu_i}(\lambda)$. By the definition of C_i, it must hold $(w, i, \lambda) \models \neg \Uparrow^x_{\texttt{@a}}$. By the semantics of the $\Uparrow$-operator, either $\texttt{val}(w, i, \texttt{@a})$ is not defined or $\lambda(x) \neq \texttt{val}(w, i, \texttt{@a})$. By the definition of traces, either $\texttt{a}(\nu_i(\texttt{act}_i))$ is not defined or $\lambda(x) \neq \texttt{a}(\nu_i(\texttt{act}_i))$. By the definition of σ and the construction of symbolic runs, this means that either $\texttt{a}(\texttt{act}_i)$ is not defined or $\sigma(x) \neq \texttt{a}(\texttt{act}_i)$.
- $\Downarrow$-rule: Let $(\Downarrow^x_{\texttt{@a}}.\psi, \sigma) \in C'_i$ for some freeze variable x, attribute $\texttt{a}$, formula ψ and freeze assignment σ. By construction of θ, there is $(\Downarrow^x_{\texttt{@a}}.\psi, \lambda) \in C_i$ with $\sigma = \texttt{extr}_{\nu_i}(\lambda)$. By the definition of C_i, $(w, i, \lambda) \models \Downarrow^x_{\texttt{@a}}.\psi$. Due to the semantics of the $\Downarrow$-operator, $\texttt{val}(w, i, \texttt{@a})$ must be defined and $(w, i, \lambda[x \mapsto \texttt{val}(w, i, \texttt{@a})]) \models \psi$. By the relationship between concrete traces and runs, $\texttt{a}(\nu_i(\texttt{act}_i))$ must be defined and $(w, i, \lambda[x \mapsto \texttt{a}(\nu_i(\texttt{act}_i))]) \models \psi$. Hence, by definition of C_i, $(\psi, \lambda[x \mapsto \texttt{a}(\nu_i(\texttt{act}_i))]) \in C_i$. Let r be the $\mathcal{A}$-register such that $\nu_i(r) = \texttt{a}(\nu_i(\texttt{act}_i))$. Note that due to the construction of runs, $r = \texttt{a}(\texttt{act}_i)$. By the construction of θ and the relationship between λ and σ, the parameter $\texttt{a}(\texttt{act}_i) = r$ is defined and $(\psi, \sigma[x \mapsto r]) \in C'_i$.
- $\mathbf{R}$-rule: Let $(\psi_1 \mathbf{R} \psi_2, \sigma) \in C'_i$ for some freeze assignment σ and formulas ψ_1 and ψ_2. By construction of θ, there is $(\psi_1 \mathbf{R} \psi_2, \lambda) \in C_i$ with $\sigma = \texttt{extr}_{\nu_i}(\lambda)$. Hence, $(w, i, \lambda) \models \psi_1 \mathbf{R} \psi_2$. It follows from the semantics of the $\mathbf{R}$-operator that (i) $(w, i, \lambda) \models \psi_2$, and (ii) $(w, i, \lambda) \models \texttt{end}$ or $(w, i, \lambda) \models \psi_1$ or $(w, i, \lambda) \models \mathbf{X}(\psi_1 \mathbf{R} \psi_2)$. By the construction of C_i, we have (i) $(\psi_2, \lambda) \in C_i$, and (ii) $(\texttt{end}, \lambda) \in C_i$ or $(\psi_1, \lambda) \in C_i$ or $(\psi_1 \mathbf{R} \psi_2, \lambda) \in C_i$. By the construction of θ, we get (i) $(\psi_2, \sigma) \in C'_i$, and (ii) $(\texttt{end}, \sigma) \in C'_i$ or $(\psi_1, \sigma) \in C'_i$ or $(\psi_1 \mathbf{R} \psi_2, \sigma) \in C'_i$.

We now show that for every i with $1 \leq i < n$, the tuple $[C'_{i+1}, \texttt{act}_{i+1}]$ is a successor of $[C'_i, \texttt{act}_i]$, i.e., $[C'_i, \texttt{act}_i] \rightarrow [C'_{i+1}, \texttt{act}_{i+1}]$. Thus, with regard to the definition of successive tuples, for every $(\mathbf{X}\psi, \sigma) \in C'_i$ with $1 \leq i < n$, we have to assure:

- If $\mathtt{act}_{i+1}$ is some create action with $\mathtt{created}(\mathtt{act}_{i+1}) = r$ for some register $r \in R$, then, $(\psi, \sigma[X' \mapsto r_\bot]) \in C'_{i+1}$ where $X' = \{x \mid \sigma(x) = r\}$, and
- if $\mathtt{act}_{i+1}$ is some send action, then, $(\psi, \sigma) \in C'_{i+1}$.

To this end, let for some i with $1 \le i < n$, $(\mathbf{X}\psi, \sigma) \in C'_i$. By definition of θ, there must be some $(\mathbf{X}\psi, \lambda) \in C_i$ with $\mathtt{extr}_{\nu_i}(\lambda) = \sigma$. By the construction of C_i, it must hold $(w, i, \lambda) \models \mathbf{X}\psi$. It follows from the semantics of the $\mathbf{X}$-operator that $(w, i+1, \lambda) \models \psi$, thus $(\psi, \lambda) \in C_{i+1}$. We first consider the easy case that $\nu_{i+1}(\mathtt{act}_{i+1})$ is a send action. Note that in this case, $\nu_i = \nu_{i+1}$. Thus, $\sigma = \mathtt{extr}_{\nu_i}(\lambda) = \mathtt{extr}_{\nu_{i+1}}(\lambda)$. By construction of θ, we get $(\psi, \sigma) \in C'_{i+1}$. Now, we consider the case where $\nu_{i+1}(\mathtt{act}_{i+1})$ is a create action. Let r be an $\mathcal{A}$-register such that $\mathtt{creator}(\nu_{i+1}(\mathtt{act}_{i+1})) = \nu_{i+1}(r)$. Due to the behaviour of PRA, register r is refreshed at step $i+1$ and if $\nu_i(r)$ is defined, it does not occur in any $\nu_j(R)$ with $j \ge i+1$. Thus, by the definition of $\mathtt{extr}$, $\mathtt{extr}_{\nu_{i+1}}(\lambda) = \sigma[X' \mapsto r_\bot]$ where $X' = \{x \in X \mid \sigma(x) = r\}$. Together with $(\psi, \lambda) \in C_{i+1}$, we get $(\psi, \sigma[X' \mapsto r_\bot]) \in C'_{i+1}$.

Finally, we explain that θ must be accepting and the atomic formula $\mathtt{end}$ can only occur in C'_n. First note that it follows from Observation 11 that the underlying symbolic run of θ (devoid of consistent tuples) is accepting. Moreover, as on the last position of trace w, there cannot hold any formula of the form $\mathbf{X}\psi$, such a formula can neither be contained in C_n, nor in C'_n. Hence, C'_n is final. Furthermore, observe that there cannot be any trace position $i \in \{1, \ldots, n-1\}$ where $\mathtt{end}$ holds, because $\mathtt{end}$ can only hold on the last position of the trace. This means that there cannot be any set C_i, and by construction, any set C'_i, with $i \in \{1, \ldots, n-1\}$ containing $\mathtt{end}$. This completes the proof of the "only if"-part of the lemma.

We now turn towards the proof of the "if"-direction. Assume that there is an accepting symbolic φ-run

$$\theta = sc_0 \xrightarrow{[C_1, \mathtt{act}_1]}{}^{s}_{\mathcal{A}} sc_1 \ldots sc_{n-1} \xrightarrow{[C_n, \mathtt{act}_n]}{}^{s}_{\mathcal{A}} sc_n$$

of $\mathcal{A}$ obtained from the underlying symbolic run

$$\theta' = sc_0 \xrightarrow{\mathtt{act}_1}{}^{s}_{\mathcal{A}} sc_1 \ldots sc_{n-1} \xrightarrow{\mathtt{act}_n}{}^{s}_{\mathcal{A}} sc_n.$$

It has to be shown that there is a trace of $\mathcal{A}$ satisfying φ. Before that, we introduce functions which, in contrast to extractions used above, "expand" freeze assignments which map to registers to freeze assignments which map to processes. Let $p_\bot \notin \mathbb{P}$ be a designated process not contained in $\mathbb{P}$. For every register assignment $\nu \in [R \rightharpoonup \mathbb{P}]$ and freeze assignment $\sigma \in [X \rightharpoonup R \cup \{r_\bot\}]$, we define $\mathtt{exp}_\nu(\sigma) = \{x \mapsto \nu(r) \mid \sigma(x) = r \in R\} \cup \{x \mapsto p_\bot \mid \sigma(x) = r_\bot\}$. Let

$$\tau' = (s_0, \nu_0, E_0) \xrightarrow{\nu_1(\mathtt{act}_1)}{}_{\mathcal{A}} (s_1, \nu_1, E_1) \ldots (s_{n-1}, \nu_{n-1}, E_{n-1}) \xrightarrow{\nu_n(\mathtt{act}_n)}{}_{\mathcal{A}} (s_n, \nu_n, E_n)$$

be a concrete accepting run of $\mathcal{A}$ with $\mathtt{symb}(\tau') = \theta'$ as guaranteed by Observation 11 and let $w = w_1 \ldots w_n$ be the trace of τ'. We remind the reader that $w_i = \mathtt{dwrep}(\nu_i(\mathtt{act}_i))$, for every $i \in \{1, \ldots, n\}$. We will show that φ holds on w. To this end, we construct

$$\tau = (s_0, \nu_0, E_0) \xrightarrow{[C'_1, \nu_1(\mathtt{act}_1)]}{}_{\mathcal{A}} (s_1, \nu_1, E_1) \ldots (s_{n-1}, \nu_{n-1}, E_{n-1}) \xrightarrow{[C'_n, \nu_n(\mathtt{act}_n)]}{}_{\mathcal{A}} (s_n, \nu_n, E_n)$$

as an extension of τ' where for every i with $1 \le i \le n$, C'_i results from C_i by replacing every $(\psi, \sigma) \in C_i$ by a pair (ψ, λ) with $\lambda = \mathtt{exp}_{\nu_i}(\sigma)$. We will prove that for every i and every $(\psi, \lambda) \in C'_i$, it holds $(w, i, \lambda) \models \psi$. As by construction, $(\varphi, \lambda[X \mapsto \bot]) \in C'_1$, it will follow that φ holds on w.

Our argumentation is by induction on the structure of formulas ψ. We do not consider all types of formulas and leave those out which can be handled in analogy to the considered cases. Now, let $(\psi, \lambda) \in C'_i$ for some i with $1 \le i \le n$ and some freeze assignment $\lambda \in [X \rightharpoonup \mathbb{P} \cup \{p_\bot\}]$. We first consider the cases where ψ is an atomic formula:

- $\psi = \texttt{crt}$: By construction of C'_i, there must be some $(\texttt{crt}, \sigma) \in C_i$. As θ is a symbolic φ-run, we can derive from the rules concerning propositions that $\texttt{act}_i$ must be a create action. Thus, as $w_i = \texttt{dwrep}(\nu_i(\texttt{act}_i))$, position i of w must contain proposition $\texttt{crt}$. Therefore, $(w, i, \lambda) \models \texttt{crt}$.

- $\psi = \neg m$ for some message symbol m: It follows from construction that there is some σ such that $(\neg m, \sigma) \in C_i$. By the rules for propositions in symbolic φ-runs, $\texttt{act}_i$ is not a send action with message symbol m. Consequently, position i of w cannot contain proposition m. It follows $(w, i, \lambda) \models \neg m$.

- $\psi = \Uparrow^x_{\texttt{@a}}$ for some freeze variable x and attribute $\texttt{a}$: By construction of τ, there must be $(\Uparrow^x_{\texttt{@a}}, \sigma) \in C_i$ with $\lambda = \texttt{exp}_{\nu_i}(\sigma)$. As θ must fulfill the $\Uparrow$-rules, there must be some r such that (i) $\texttt{a}(\texttt{act}_i) = r$, and (ii) $\sigma(x) = r$. From (i) it follows that attribute $\texttt{a}(\nu_i(\texttt{act}_i)) = p$ for some process p. From $\lambda = \texttt{exp}_{\nu_i}(\sigma)$ and (ii) it follows that $\lambda(x) = p = \texttt{a}(\nu_i(\texttt{act}_i))$. As $w_i = \texttt{dwrep}(\nu_i(\texttt{act}_i))$ and, therefore, $\texttt{val}(w, i, \texttt{@a}) = \lambda(x)$, we get $(w, i, \lambda) \models \Uparrow^x_{\texttt{@a}}$.

- $\psi = \neg \Uparrow^x_{\texttt{@a}}$ for some freeze variable x and attribute $\texttt{a}$: By construction of τ, C_i contains the pair $(\neg \Uparrow^x_{\texttt{@a}}, \sigma)$ with $\lambda = \texttt{exp}_{\nu_i}(\sigma)$. As τ follows the $\Uparrow$-rules, this means that (i) $\texttt{a}(\texttt{act}_i)$ does not exist, or (ii) $\sigma(x) \neq \texttt{a}(\texttt{act}_i)$. From (i) it follows that parameter $\texttt{a}$ is not defined for $\nu_i(\texttt{act}_i)$. From (ii) and the definition of λ, we derive that $\lambda(x) \neq \texttt{a}(\nu_i(\texttt{act}_i))$ and, thus, $\lambda(x) \neq \texttt{val}(w, i, \texttt{@a})$. By the disjunction of these two conclusions, we get $(w, i, \lambda) \models \neg \Uparrow^x_{\texttt{@a}}$.

- $\psi = \bot_{\texttt{@a}}$ for some attribute $\texttt{a}$: Due to the construction of τ, $(\bot_{\texttt{@a}}, \sigma)$ must be contained in C_i for some σ. By the $\bot$-rule for φ-runs, it follows that parameter $\texttt{a}$ is not defined for $\texttt{act}_i$. Hence, it cannot be defined for $\nu_i(\texttt{act}_i)$ from which it follows that the value of $\texttt{a}$ is not defined in $\texttt{dwrep}(\nu_i(\texttt{act}_i))$. We get $(w, i, \lambda) \models \bot_{\texttt{@a}}$.

We now consider more complex formulas ψ.

- $\psi = \Downarrow^x_{\texttt{@a}}.\chi$ for some freeze variable x, attribute $\texttt{a}$ and formula χ: By construction, $(\Downarrow^x_{\texttt{@a}}.\chi, \sigma) \in C_i$ with $\texttt{exp}_{\nu_i}(\sigma) = \lambda$. Due to the $\Downarrow$-rule, (i) $\texttt{a}(\texttt{act}_i) = r$ for some register $r \in R$ and (ii) $(\chi, \sigma[x \mapsto r]) \in C_i$. It follows from construction and (ii) that $(\chi, \lambda') \in C'_i$ where $\lambda' = \texttt{exp}_{\nu_i}(\sigma[x \mapsto r])$. We get by induction that $(w, i, \lambda') \models \chi$. Furthermore, from (i), it follows that the value of attribute $\texttt{a}$ at position i of w must be defined with $\texttt{val}(w, i, \texttt{@a}) = \nu_i(r)$. Thus, by the definitions of λ' and λ, we get $\lambda' = \lambda[x \mapsto \texttt{val}(w, i, \texttt{@a})]$. Due to $(w, i, \lambda') \models \chi$, it follows $(w, i, \lambda[x \mapsto \texttt{val}(w, i, \texttt{@a})]) \models \chi$. By the semantics of the $\Downarrow$-operator, we conclude $(w, i, \lambda) \models \Downarrow^x_{\texttt{@a}}\chi$.

- $\psi = \chi_1 \mathbf{U} \chi_2$ for formulas χ_1 and χ_2: By construction of τ, it holds $(\chi_1 \mathbf{U} \chi_2, \sigma) \in C_i$ with $\lambda = \texttt{exp}_{\nu_i}(\sigma)$. As τ obeys the $\mathbf{U}$-rule, it follows (i) $(\chi_2, \sigma) \in C_i$ or (ii) $(\chi_1, \sigma), (\mathbf{X}(\chi_1 \mathbf{U} \chi_2), \sigma) \in C_i$. According to the conditions required from successive tuples, it follows that there must be a $j \geq i$ such that

$$(\chi_1, \sigma_i) \in C_i, \ldots, (\chi_1, \sigma_{j-1}) \in C_{j-1} \text{ and } (\chi_2, \sigma_j) \in C_j$$

  where $\sigma_i = \sigma$ and for every k with $i < k \leq j$, σ_k is obtained from σ_{k-1} as follows:

  - If $\texttt{act}_k$ is some spawn action with $\texttt{created}(\texttt{act}) = r \in R$, then, $\sigma_k = \sigma_{k-1}[X' \mapsto r_\bot]$ where $X' = \{x \mid \sigma_{k-1}(x) = r\}$, and
  - if $\texttt{act}_k$ is some send action, then, $\sigma_k = \sigma_{k-1}$.

  Please keep in mind that the register $r_\bot$ in some σ_k represents a process which cannot occur in the suffix $w[k, \ldots]$ of w starting at k. Due to the structural properties of τ, it must hold

$$(\chi_1, \lambda_i) \in C'_i, \ldots, (\chi_1, \lambda_{j-1}) \in C'_{j-1} \text{ and } (\chi_2, \lambda_j) \in C'_j$$

such that $\lambda = \lambda_i$ and $\mathtt{exp}_{\nu_k}(\sigma_k) = \lambda_k$ for all k with $i \leq k \leq j$. By induction, we have

$$(w, i, \lambda_i) \models \chi_1, \ldots, (w, j-1, \lambda_{j-1}) \models \chi_1 \text{ and } (w, j, \lambda_j) \models \chi_2.$$

Note that for every k with $i < k \leq j$, the register assignment λ_k differs from λ_{k-1} at most with respect to processes represented by $r_\perp$, i.e., with respect to those which never occur in $w[k, \ldots]$. Hence,

$$\lambda \overset{w^i}{\equiv} \lambda_i \overset{w^{i+1}}{\equiv} \lambda_{i+1} \overset{w^{i+2}}{\equiv} \lambda_{i+2} \ldots \lambda_{j-1} \overset{w^j}{\equiv} \lambda_j.$$

Due to Observation 15, we deduce

$$(w, i, \lambda) \models \chi_1, \ldots, (w, j-1, \lambda) \models \chi_1 \text{ and } (w, j, \lambda) \models \chi_2.$$

Finally, by the semantics of the $\mathbf{U}$-operator, it follows $(w, i, \lambda) \models \chi_1 \mathbf{U} \chi_2$.

This concludes the proof of the "if"-part of the lemma. □

Using Lemma 19, we can reduce the existential model checking problem for PRA and $\mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$ to the problem of constructing a symbolic run for the PRA extended by consistent tuples. In the proof of the following theorem we show that such a construction can be done in exponential space. This leads to the result that the general model checking problem for PRA and $\mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$ is in ExpSpace.

Theorem 30. *The problem* ModCheck(*PRA*, *$\mathit{LTL}^{\Downarrow}$*$(\mathbf{X}, \mathbf{U})$) *is in* ExpSpace.

Proof. We will show that the existential model checking problem for PRA and $\mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$ can be solved by a non-deterministic algorithm using exponential space. As $\mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$ is closed under negation and space complexity classes are closed under complementation and determinization [172], the result follows.

Now, we describe the non-deterministic algorithm which, given a PRA $\mathcal{A} = (A, R, r_0, S, s_0, \delta, F)$ and an $\mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$-formula ψ, decides whether $\mathcal{A}$ has a concrete trace satisfying ψ. We let X be the set of freeze variables used in ψ. The algorithm first converts ψ into an equivalent $\mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$-formula φ in negation normal form. Note that according to our explanations concerning this normal form, the transformation to φ does not need more than polynomial space. By Lemma 19, $\mathcal{A}$ has a trace satisfying φ if and only if there is an accepting symbolic φ-run for $\mathcal{A}$. Our algorithm guesses a symbolic accepting φ-run

$$(s_0, D_0) \xrightarrow{[C_1, \mathtt{act}_1]}{}^{s}_{\mathcal{A}} (s_1, D_1) \xrightarrow{[C_2, \mathtt{act}_2]}{}^{s}_{\mathcal{A}} (s_2, D_2) \ldots (s_{n-1}, D_{n-1}) \xrightarrow{[C_n, \mathtt{act}_n]}{}^{s}_{\mathcal{A}} (s_n, D_n)$$

encoded as a *witness sequence*

$$[C_1, \mathtt{act}_1, s_1, D_1][C_2, \mathtt{act}_2, s_2, D_2] \ldots [C_n, \mathtt{act}_n, s_n, D_n]$$

and checks that its guess is correct. Remember that each D_i is a subset of R and each C_i is a subset of $\mathtt{Closure}(\varphi) \times [X \rightharpoonup R \cup \{r_\perp\}]$. While the size of $\mathtt{Closure}(\varphi)$ is at most polynomial in the length of φ, the size of the set $[X \rightharpoonup R \cup \{r_\perp\}]$ is polynomial in the size of R and exponential in the size of X. Thus, a tuple of the form $[C_i, \mathtt{act}_i, s_i, D_i]$ requires a space of at most exponential size and there are at most doubly exponentially many different tuples. It follows that if there is a witness sequence inducing an accepting symbolic φ-run, then, there is one of at most doubly exponential length. The algorithm first guesses the length n (which is stored in binary encoding) of the witness sequence and then constructs this sequence tuple by tuple. The important point is that in each step it does not keep more than two consecutive tuples in its memory why a space of exponential size suffices for the entire algorithm.

The detailed description of the algorithm is as follows. The algorithm first constructs a tuple $[C_1, \mathtt{act}_1, s_1, D_1]$ and ensures that it holds $(s_0, \{r_0\}) \xrightarrow{\mathtt{act}_1}{}^{s}_{\mathcal{A}} (s_1, D_1)$ and that $[C_1, \mathtt{act}]$ constitutes a consistent tuple with $(\varphi, \sigma[X \mapsto \bot]) \in C_1$. If $n = 1$, it further ensures that C_1 is final by checking that it does not contain any pair of the form $(\mathbf{X}\chi, \sigma)$. If the test succeeds, the computation stops by outputting a positive answer. If $n > 1$, the procedure assures that $(\mathtt{end}, \sigma) \notin C_1$ and, while keeping the first tuple in its memory, it guesses a second tuple $[C_2, \mathtt{act}_2, s_2, D_2]$. Like before, it tests that $[C_2, \mathtt{act}_2]$ is consistent, $[C_2, \mathtt{act}_2]$ is a successor of $[C_1, \mathtt{act}_1]$ and $(s_1, D_1) \xrightarrow{\mathtt{act}_2}{}^{s}_{\mathcal{A}} (s_2, D_2)$. If $n = 2$, it additionally checks that C_2 is final and stops with a positive answer if the test succeeds. However, if $n > 2$, it assures that $\mathtt{end}$ is not contained in C_2, deletes the first tuple $[C_1, \mathtt{act}_1, s_1, D_1]$ from its memory, creates a third tuple $[C_3, \mathtt{act}_3, s_3, D_3]$ and carries out the same test between the second and the third tuple as between the first and the second one. This procedure continues until n successive consistent tuples are constructed. ☐

A closer inspection of the complexity of the algorithm in the proof of the last theorem leads to the insight that the exponential blowup of the size of the memory of the algorithm is caused by the exponential number of possible freeze assignments in $[X \rightharpoonup R \cup \{r_\bot\}]$. If we allow only a constant number of freeze variables, the size of this set becomes polynomial and the number of possible tuples $[C_n, \mathtt{act}_n, s_n, D_n]$ exponential. We conclude that the model checking problem for PRA and $\mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$ with a constant number of freeze variables is solvable in polynomial space.

Corollary 9. *For every $k \geq 1$, the problem* ModCheck(*PRA*, *$\mathrm{LTL}^{\Downarrow}_k(\mathbf{X}, \mathbf{U})$*) *is in* PSpace.

An extension of our construction of symbolic runs labelled by consistent tuples $[C, \mathtt{act}]$ to formulas with past operators entails some complications which we were not able to solve yet. Recall that each set C within such a run consists of pairs (ψ, σ) where σ is a symbolic freeze assignment mapping freeze variables to the registers of the automaton and the pseudo register $r_\bot$ symbolizing "old" processes which will not occur in the rest of the run. Note that our construction uses the same register $r_\bot$ for different old processes. Suppose that at some point of a symbolic run, we have a tuple $[C, \mathtt{act}]$ such that C contains a pair $(\mathbf{X}^{\leftarrow}, \sigma)$ where σ maps different freeze registers x and y to $r_\bot$. To assure consistency of $[C, \mathtt{act}]$ with its predecessor $[C', \mathtt{act}']$, we have to determine for which of the variables x and y, the register $r_\bot$ has to be replaced by a register of the automaton when going from $[C, \mathtt{act}]$ to $[C', \mathtt{act}']$. In our current construction we do not see yet how this can be done.

From satisfiability of LTL to model checking with $\mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$

We turn towards the lower bound of the model checking of PRA with $\mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$. We will show that the problem is PSpace-hard, even in the case with plain LTL. The proof is carried out by a polynomial reduction from the satisfiability problem for LTL to the existential model checking problem with LTL. From this, the PSpace-hardness of LTL [188] and Corollay 9 we get that for every $k \geq 1$, the problem ModCheck(PRA, $\mathrm{LTL}^{\Downarrow}_k(\mathbf{X}, \mathbf{U})$) is PSpace-complete.

Lemma 20. *The problem* ModCheck(*PRA*, *LTL*) *is* PSpace*-hard.*

Proof. We give a polynomial reduction from the satisfiability problem for LTL to to the existential model checking of PRA with LTL. As satisfiability for LTL is PSpace-hard [188] and LTL is closed under negation, the result follows. The reduction consists of two steps. First, we reduce the satisfiability problem for LTL on words (with multiple propositions at each position) to the satisfiability problem of LTL on strings (with a single proposition per position). Then, we reduce the latter problem to EModCheck(PRA, LTL).

We start by describing a representation for finite words over some proposition set $\mathtt{Prop}$ by finite strings over $\mathtt{Prop} \cup \{p_\#\}$ with a fresh proposition $p_\#$ which is not contained in $\mathtt{Prop}$. In a string over

$\mathtt{Prop} \cup \{p_\#\}$, we call a maximal sub sequence where only the first position is a $p_\#$-position, a *#-block*. A word w over $\mathtt{Prop}$ is represented by strings w' where every #-block encodes exactly one position in w. More precisely, for a word $w = P_1 \dots P_n$ with $P_1, \dots, P_n \subseteq \mathtt{Prop}$, every string w' of the form $p_\# p_1^1 \dots p_1^{k_1} p_\# p_2^1 \dots p_2^{k_2} p_\# \dots p_\# p_n^1 \dots p_n^{k_n}$ such that for every $i \in \{1, \dots, n\}$, $\{p_i^1, \dots, p_i^{k_i}\} = P_i$, is a representation for w. Observe that a proposition p at a position in w can be encoded by a #-block with multiple occurrences of p. Hence, there are infinitely many string-representations for the same word. Figure 12.3 presents two different string-encodings for a word w.

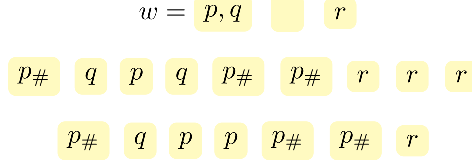


Figure 12.3: Two different string-encodings for the word w

Next, we define a translation t from LTL-formulas φ on words to LTL-formulas $t(\varphi)$ on string-representation such that $t(\varphi)$ simulates the “behaviour” of φ. The main idea in the translation is that for every proposition p for which it is assumed that it holds at some position of a word w, the translated formula ensures that p occurs in the corresponding #-block of string-representations w' of w. Moreover, one step to the right in w corresponds to a navigation to the next #-block in w'. We define t inductively, but omit the Boolean cases. For a better understanding of the translation, it is worth mentioning that the simulation of each sub-formula of an input formula starts at the first position of some #-block.

- $t(p) = \mathbf{X}(\neg p_\# \mathbf{U} p)$ for every proposition $p \in \mathtt{Prop}$
- $t(\mathbf{X}\psi) = \mathbf{X}\Big(\neg p_\# \mathbf{U}\big(p_\# \wedge t(\psi)\big)\Big)$
- $t(\psi_1 \mathbf{U} \psi_2) = \Big(p_\# \rightarrow t(\psi_1)\Big)\mathbf{U}\Big(p_\# \wedge t(\psi_2)\Big)$

Now, let φ be an LTL-formula over $\mathtt{Prop}$. Obviously, $p_\# \wedge t(\varphi)$ is an LTL-formula of polynomial length which is satisfiable on strings if and only if φ is satisfiable on words with propositions. This concludes the first step in our reduction.

We now reduce the satisfiability problem for LTL on strings to EModCheck(PRA, LTL). For this purpose, let ψ be an LTL-formula with propositions from $\mathtt{Prop} = \{p_1, \dots, p_k\}$. The idea is to construct a PRA $\mathcal{A}_{\mathtt{uni}}$ over a message set which contains for every proposition, a message symbol representing this proposition. While the PRA $\mathcal{A}_{\mathtt{uni}}$ produces traces containing all possible sequences of message symbols, we design a formula ψ' testing whether there is a trace of $\mathcal{A}_{\mathtt{uni}}$ satisfying the property expressed by ψ.

We explain the details of the construction. The PRA $\mathcal{A}_{\mathtt{uni}} = (A, R, r_0, S, s_0, \delta, F)$ has two registers $r_0, r_1 \in R$ and is defined over the message set $A = \{m_1, \dots, m_k\}$ where all message symbols have arity 0. The automaton is depicted in Figure 12.4. First, the initial process in register r_0 creates a process and stores it in register r_1. Then, arbitrary messages are sent arbitrarily often from the process in r_0 to the process in r_1. Furthermore, we translate ψ into an LTL-formula ψ' simulating ψ on the traces of $\mathcal{A}_{uni}$. The formula ψ' ignores the first positions of the traces (as they do not contain any message symbol) and interprets the message symbols as propositions from $\mathtt{Prop}$. That is, $\psi' = \mathbf{X}\psi''$ where ψ'' results from ψ by replacing every p_i by m_i.

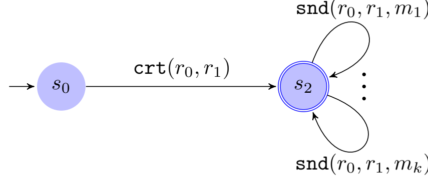


Figure 12.4: The PRA $\mathcal{A}_{\texttt{uni}}$

It is easy to see that ψ is satisfiable if and only if there is a trace of $\mathcal{A}_{uni}$ satisfying ψ'. Moreover, note that the sizes of $\mathcal{A}_{uni}$ and ψ' are polynomial in the size of ψ. This completes the second part of the reduction. ☐

The combination of the results in Corollary 9 and Lemma 20 delivers:

Theorem 31. *For every $k \geq 1$, the problem* ModCheck(*PRA*, *LTL*$^{\Downarrow}_k(\mathbf{X}, \mathbf{U})$) *is* PSpace*-complete.*

12.2.2 Model Checking with Hybrid Temporal Logic

In this section, we consider the model checking of PRA with HTL$^{\sim}$. We will prove that for every $k \geq 1$, model checking with HTL$^{\sim}_k$ is as hard as the satisfiability problem for HTL$_k$. Then, we will derive from known results that ModCheck(PRA, HTL$^{\sim}_1$) is ExpSpace-complete and for every $k \geq 2$, ModCheck(PRA, HTL$^{\sim}_k$) is decidable with non-elementary complexity. We first deal with the upper bound complexities.

From model checking with HTL$^{\sim}$ to satisfiability of HTL

We will show that for every $k \geq 1$, the existential model checking problem for PRA and HTL$^{\sim}_k$ is polynomially reducible to the satisfiability problem for HTL$_k$. Similar to the case with LTL$^{\Downarrow}(\mathbf{X}, \mathbf{U})$, the proof makes use of symbolic runs and traces of PRA.

Proposition 15. For every $k \geq 1$, the problem EModCheck(PRA, HTL$^{\sim}_k$) can be polynomially reduced to the satisfiability problem for HTL$_k$.

Proof. Recall from Section 11.3 that concrete traces of a PRA $\mathcal{A} = (A, R, r_0, S, s_0, \delta, F)$ are defined over the proposition set $\texttt{Prop}^{A}_{\texttt{act}} = \{\texttt{crt},\texttt{snd}\} \cup A$ and the attribute set $\texttt{Attr}^{A}_{\texttt{act}} = \{\texttt{creator}, \texttt{created}, \texttt{sender}, \texttt{receiver}\} \cup \{\texttt{mpar}_1, \ldots, \texttt{mpar}_a\}$ where a is the maximal arity of the message symbols in A. Symbolic traces are defined over $\texttt{Prop}^{A,R}_{\texttt{sact}} = \{\texttt{snd},\texttt{crt}\} \cup \{[\texttt{par},\texttt{par}(\texttt{act})] \mid \texttt{act}$ is an action in $\texttt{Actions}(A, R)$ with parameter $\texttt{par}\}$. We introduce the *existential symbolic model checking problem* for PRA which searches for a symbolic trace satisfying a formula. To be precise, for a class $\mathcal{C}$ of PRA and a logic $\mathcal{L}$, the existential symbolic model checking problem ESMCh($\mathcal{C}$,$\mathcal{L}$) asks the following question: Given a PRA $\mathcal{A}$ from $\mathcal{C}$ and a formula φ from $\mathcal{L}$, is there a symbolic trace of $\mathcal{A}$ satisfying φ? For a $k \geq 1$, the reduction from EModCheck(PRA, HTL$^{\sim}_k$) to the satisfiability of HTL$_k$ consists of two main steps. We first reduce EModCheck(PRA, HTL$^{\sim}_k$) to ESMCh(PRA,HTL$_k$), then we show that the latter problem can be encoded into the satisfiability problem for HTL$_k$.

We describe the main idea of the first reduction. By definition, each concrete trace of a PRA results from one of its concrete runs. Likewise, each symbolic trace belongs to a symbolic run of the PRA. Moreover, by Observation 11, we know that for every concrete run τ, there is a

corresponding symbolic run $\mathtt{symb}(\tau)$ which is obtained from τ, basically by replacing processes by corresponding registers. Furthermore, the same observation tells us that for every symbolic run, there is a corresponding concrete run. Thus, for the first reduction, it suffices to show how an $\mathrm{HTL}_k^{\sim}$-formula on concrete traces of a PRA can be simulated by an HTL_k-formula on symbolic traces of the automaton. Obviously, the main challenge is to recover data equality on symbolic traces. With regard to this, we first recall that it follows from the definition of PRA that two different registers in the same run can never contain the same process. Moreover, whether at two different positions $i < j$ of a run, the same register r points to the same process or not can be checked as follows: r points at positions i and j to the same process if and only if r has an input at position i and this input is not overwritten by a create action between position i and j.

We now dive into the details of the first reduction. Let $\mathcal{A} = (A, S, s_0, R, r_0, \delta, F)$ be a PRA and φ an $\mathrm{HTL}_k^{\sim}$-formula for some $k \geq 1$. We translate φ into an HTL_k-formula $t(\varphi)$ of polynomial length such that φ is satisfied by a concrete trace of $\mathcal{A}$ if and only if $t(\varphi)$ is satisfied by the corresponding symbolic trace. The transformation t is defined inductively. We omit the Boolean cases:

- $t(\mathtt{crt}) = \mathtt{crt}$
- $t(\mathtt{snd}) = \mathtt{snd}$
- $t(m) = [\mathtt{msym}, m]$ for all message symbols $m \in A$
- $t(\mathbf{X}\psi) = \mathbf{X}t(\psi)$
- $t(\psi_1 \mathbf{U} \psi_2) = t(\psi_1)\mathbf{U}t(\psi_2)$
- $t(\mathbf{X}^{\leftarrow}\psi) = \mathbf{X}^{\leftarrow}t(\psi)$
- $t(\psi_1 \mathbf{U}^{\leftarrow} \psi_2) = t(\psi_1)\mathbf{U}^{\leftarrow}t(\psi_2)$
- $t(\mathtt{on}(x).\psi) = \mathtt{on}(x).t(\psi)$ for every variable x
- $t(x) = x$ for every variable x
- $t(\downarrow^x.\psi) = \downarrow^x.t(\psi)$ for every variable x
- In the translation of an atomic formula $@\mathtt{a}{\sim}x.@\mathtt{b}$ for attributes $\mathtt{a}, \mathtt{b} \in \mathtt{Attr}^A_{\mathtt{act}}$ and a variable x, we first ensure that there is some register r such that $[\mathtt{a}, r]$ holds at the current position i. Note that this corresponds to the fact that there is some action performed at position i which uses the process of register r. This means that the process of this register is defined at position i of the corresponding concrete run. Then, we assure that parameter $\mathtt{b}$ of the action at the x-position j is also defined as r. Additionally, in order to guarantee that register r represents the same process at both positions, the following properties are tested: If $j \leq i$, then, in the sub sequence $j+1, \ldots, i$, there is no create operation overwriting r. In the other case, namely if $j > i$, there must not be any create action overwriting r on any position in the sequence $i+1, \ldots, j$. Note that the update of r at position j in the first case or at i in the second case, does not violate the semantics of the original $\mathrm{HTL}^{\sim}$-formula. Thus, we have:

$$t(@\mathtt{a}{\sim}x.@\mathtt{b}) = \bigvee_{r \in R} \Bigg([\mathtt{a}, r] \wedge \bigg(\Big[\neg[\mathtt{created}, r]\mathbf{U}^{\leftarrow}\big(x \wedge [\mathtt{b}, r]\big) \Big] \vee \Big[\mathbf{X}\big(\neg[\mathtt{created}, r]\mathbf{U}(\neg[\mathtt{created}, r] \wedge x \wedge [\mathtt{b}, r]) \big) \Big] \bigg) \Bigg)$$

Observe that compared to φ, the formula $t(\varphi)$ does not use any additional variable. Moreover, the blow-up caused by sub-formulas of the form $\texttt{@a}{\sim}x.\texttt{@b}$ is linear in $|R|$ and, thus, linear in the size of $\mathcal{A}$. Altogether, we get that for every $k \geq 1$, EModCheck(PRA, $\text{HTL}_{\tilde{k}}$) is polynomially reducible to ESMCh(PRA,HTL_k).

Next, we show that for every $k \geq 1$, there is a polynomial reduction from ESMCh(PRA,HTL_k) to the satisfiability of HTL_k. Given a PRA $\mathcal{A} = (A, R, r_0, S, s_0, \delta, F)$ and a formula $\varphi \in \text{HTL}_k$, we define a PLTL-formula $\varphi_{\mathcal{A}}$ whose models are encodings of symbolic traces of $\mathcal{A}$ and ask whether there is a word satisfying $\varphi_{\mathcal{A}}$ and the property defined by φ.

The idea of encoding paths in Kripke-structures by LTL-formulas is well-known (see, e.g., in [182]). For the encoding of symbolic traces, we additionally have to take into account that the execution of actions depends on the set of defined registers. We encode a symbolic trace of $\mathcal{A}$ by a word which not only describes the trace, but also the symbolic run the trace results from. To be more precise, *word encodings of symbolic traces* of $\mathcal{A}$ are words over the proposition set $\texttt{Prop}^{A,R}_{\texttt{sact}} \cup \texttt{Prop}_S \cup \texttt{Prop}_R$ where $\texttt{Prop}_S = \{p_s \mid s \in S\}$ and $\texttt{Prop}_R = \{p_r \mid r \in R\}$. Each position of the encoding models a symbolic configuration along with the action leading to it. A proposition from $\texttt{Prop}_S$ represents the state of the configuration, those from $\texttt{Prop}_R$ symbolize the set D of defined registers and those from $\texttt{Prop}^{A,R}_{\texttt{sact}}$ the executed actions. As each symbolic run starts at the same initial configuration, the latter is not represented in the encoding. Figure 12.5 presents the word encoding of a symbolic trace containing 3 symbolic actions.

$$\theta = (s_0, \{r_0\}) \xrightarrow{\texttt{crt}(r_0,r_1)}{}^{s}_{\mathcal{A}} (s_1, \{r_0, r_1\}) \xrightarrow{\texttt{crt}(r_0,r_2)}{}^{s}_{\mathcal{A}} (s_2, \{r_0, r_1, r_2\}) \xrightarrow{\texttt{snd}(r_0,r_2,m(r_1))}{}^{s}_{\mathcal{A}} (s_3, \{r_0, r_1, r_2\})$$

$p_{s_1}, p_{r_0}, p_{r_1}$ $\texttt{crt}, [\texttt{creator}, r_0], [\texttt{created}, r_1]$	$p_{s_2}, p_{r_0}, p_{r_1}, p_{r_2}$ $\texttt{crt}, [\texttt{creator}, r_0], [\texttt{created}, r_2]$	$p_{s_3}, p_{r_0}, p_{r_1}, p_{r_2}$ $\texttt{snd}, [\texttt{sender}, r_0], [\texttt{receiver}, r_2], [\texttt{msym}, m], [\texttt{mpar}_1, r_1]$

Figure 12.5: The word encoding of a symbolic trace θ of $\mathcal{A}$.

In the following, we describe the precise properties of the encoding and show that the conjunction of all of them can be expressed in some PLTL-formula $\varphi_{\mathcal{A}}$.

- Every position carries exactly one proposition from $\texttt{Prop}_S$, some propositions from $\texttt{Prop}_R$ and a set $P \subseteq \texttt{Prop}^{A,R}_{\texttt{sact}}$ such that $P = \texttt{wrep}(\texttt{act})$ for some action $\texttt{act}$. This can easily be described by LTL.

- The action and the state at the first position represent an enabled transition at the initial configuration. Note that due to the definition of PRA, the first action must be a create action. Moreover, for every position $i > 1$, the state of position $i-1$, the action at position i and the state at position i represent a transition enabled at the configuration represented by position $i-1$:

$$\bigvee_{(s_0,\texttt{crt}(r_0,r),s)\in\delta} \Big(p_s \wedge \texttt{crt} \wedge [\texttt{creator}, r_0] \wedge [\texttt{created}, r]\Big)$$

$$\wedge$$

$$\mathbf{X}\top \rightarrow \mathbf{XG}\Big[\bigvee_{(s,\texttt{crt}(r,r'),s')\in\delta} \Big(\mathbf{X}^{\leftarrow}(p_s \wedge p_r) \wedge p_{s'} \wedge \texttt{crt} \wedge [\texttt{creator}, r] \wedge [\texttt{created}, r']\Big)$$

$$\vee$$

$$\bigvee_{(s,\mathtt{snd}(r,r',m(r_1,\ldots r_{\mathtt{ar}(m)})),s')\in\delta} \Big(\mathbf{X}^{\leftarrow}(p_s \wedge p_r \wedge p_{r'} \wedge p_{r_1} \wedge \ldots \wedge p_{r_{\mathtt{ar}(m)}}) \wedge p_{s'} \wedge \mathtt{snd} \wedge [\mathtt{sender}, r] \wedge$$

$$[\mathtt{receiver}, r'] \wedge [\mathtt{msym}, m] \wedge [\mathtt{mpar}_1, r_1] \wedge \ldots \wedge [\mathtt{mpar}_{\mathtt{ar}(m)}, r_{\mathtt{ar}(m)}]\Big)\Big].$$

- The register set of the first position consists of register r_0 and the parameter `created` of the first create action. Furthermore, for every position $i > 1$, we have: If the action at position i is a create action, then, the register set at position i is the union of the register set at $i-1$ and the parameter `created` of the action at i. Otherwise, the register set at i is the same as the one at $i-1$:

$$p_{r_0} \wedge \bigwedge_{r\in R-\{r_0\}} (p_r \leftrightarrow [\mathtt{created}, r])$$

$$\wedge$$

$$\mathbf{X}\top \rightarrow \mathbf{XG}\Big[\Big(\mathtt{crt} \rightarrow \bigwedge_{r\in R} \big(p_r \leftrightarrow (\mathbf{X}^{\leftarrow} p_r \vee [\mathtt{created}, r])\big)\Big) \wedge \Big(\mathtt{snd} \rightarrow \bigwedge_{r\in R} \big(p_r \leftrightarrow \mathbf{X}^{\leftarrow} p_r\big)\Big)\Big].$$

- The run is accepting:

$$\mathbf{F}(\neg\mathbf{X}\top \wedge \bigvee_{s\in F} p_s).$$

Observe that every symbolic trace of $\mathcal{A}$ has a corresponding encoding and every encoding represents a symbolic trace. Thus, by construction, $\mathcal{A}$ has a symbolic trace if and only if $\varphi_{\mathcal{A}}$ is satisfiable. Moreover, $\mathcal{A}$ has a symbolic trace satisfying φ if and only if $\varphi_{\mathcal{A}} \wedge \varphi$ is satisfiable. Finally, observe that the length of $\varphi_{\mathcal{A}} \wedge \varphi$ is at most polynomial in the size of $\mathcal{A}$ and φ. This completes the second polynomial reduction in the proof. □

From Proposition 15 and the fact that for every $k \geq 1$, $\mathrm{HTL}^{\sim}_k$ is closed under negation and HTL_k is decidable, we obtain:

Lemma 21. *For every $k \geq 1$, the problem* ModCheck*(PRA,$\mathrm{HTL}^{\sim}_k$) is decidable.*

For $k = 1$, we can even strengthen this result. Since satisfiability for HTL_1 is ExpSpace-complete [53] and the class ExpSpace is closed under complementation, we formulate:

Lemma 22. *The problem* ModCheck*(PRA,$\mathrm{HTL}^{\sim}_1$) is in* ExpSpace.

From satisfiability of HTL to model checking with HTL$^{\sim}$

We now show that for every $k \geq 1$, there is also a polynomial reduction from the satisfiability problem for HTL_k to EModCheck(PRA, HTL_k). The reduction is a simple extension of the reduction from LTL to EModCheck(PRA, LTL) in the proof of Lemma 20. From this result, we will follow lower bounds for the general model checking problem for PRA and fragments of HTL$^{\sim}$.

Proposition 16. For every $k \geq 1$, the satisfiability problem for HTL_k can be polynomially reduced to EModCheck(PRA, HTL_k).

Proof. Let $k \geq 1$. Like in the proof of Lemma 20 where the satisfiability of LTL is reduced to model checking of PRA with LTL, we first reduce from the satisfiability of HTL_k on words with propositions to the satisfiability of HTL_k on strings. Then, we reduce from the latter problem to $\mathrm{EModCheck}(\mathrm{PRA}, \mathrm{HTL}_k)$. In this proof, we only emphasize on the additional cases which have to be taken into account when going from LTL to HTL. Recall that HTL extends LTL by variables and past operators.

In the first reduction, we take the same string encoding as in the proof of Lemma 20 as a basis. In the definition of the translation t converting formulas on words with propositions to formulas on string representation, we add the cases for past operators and the operations on variables. Note that since the evaluation of each sub-formula starts at the $p_\#$-position of a current #-block, variables x are always assigned to such positions:

- $t(\mathtt{on}(x).\psi) = \mathtt{on}(x).t(\psi)$
- $t(\downarrow^x.\psi) = \downarrow^x.t(\psi)$
- $t(x) = x$
- $t(\mathbf{X}^{\leftarrow}\psi) = \mathbf{X}^{\leftarrow}\Big(\neg p_\# \mathbf{U}^{\leftarrow}\big(p_\# \wedge t(\psi)\big)\Big)$
- $t(\psi_1 \mathbf{U}^{\leftarrow}\psi_2) = \Big(p_\# \rightarrow t(\psi_1)\Big)\mathbf{U}^{\leftarrow}\Big(p_\# \wedge t(\psi_2)\Big)$

In the second reduction, namely from satisfiability of HTL_k on strings to the existential model checking of PRA with HTL_k, we use the same PRA $\mathcal{A}_{\mathtt{uni}}$ with message set A constructed in the proof of Lemma 20 which generates all possible sequences of message symbols. However, when translating an HTL_k-formula ψ into a corresponding HTL_k-formula ψ' on traces of $\mathcal{A}_{\mathtt{uni}}$, we additionally have to make sure that the range of past operators do not reach the first position of traces, because they do not represent any string position. That is, $\psi' = \mathbf{X}\psi''$ where ψ'' results from ψ by

- replacing every p_i by m_i,
- every sub-formula $\mathbf{X}^{\leftarrow}\chi$ by $\mathbf{X}^{\leftarrow}(\bigvee_{m \in A} m \wedge \chi)$, and
- every sub-formula $\chi_1 \mathbf{U}^{\leftarrow}\chi_2$ by $\chi_1 \mathbf{U}^{\leftarrow}(\bigvee_{m \in A} m \wedge \chi_2)$.

□

From the last proposition and the fact that satisfiability for HTL_1 is ExpSpace-hard [53], we conclude:

Lemma 23. *The problem* ModCheck(*PRA*, $\widetilde{HTL_1}$) *is* ExpSpace-*hard.*

Combined with Lemma 22, we get the main result of this section:

Theorem 32. *The problem* ModCheck(*PRA*, $\widetilde{HTL_1}$) *is* ExpSpace-*complete.*

From [184] it follows that HTL with only two variables has already non-elementary complexity. Thus, together with Lemma 21, we derive:

Theorem 33. *For every $k \geq 2$, the problem* ModCheck(*PRA*, $\widetilde{HTL_k}$) *is decidable with non-elementary complexity.*

12.3 Model Checking of Branching High-Level MSCs

As announced in the introductory part of this chapter, we introduce in this section *MSC Navigation Logic (MNL)* which allows existential and universal quantification over paths of events and navigation on these paths via temporal operators. The logic is inspired by Temporal Logic of Causalities (TLC) defined in [24] on partially ordered structures and used in [173] for the model checking of HMSCs with finitely many processes. We will show that model checking of BHMSCs with MNL is ExpTime-complete, thus, as hard as non-emptiness and executability for this model. Similar to the decision procedures for the mentioned problems, our model checking algorithm works with symbolic runs of BHMSCs.

12.3.1 MSC Navigation Logic

Let A be a message alphabet. Formulas φ of MSC Navigation Logic (MNL) over A are constructed according to the following formation rules:

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}^{\downarrow}\varphi \mid \mathbf{X}^{\rightarrow}\varphi \mid \varphi\mathbf{U}^{\downarrow}\varphi \mid \mathbf{E}(\varphi\mathbf{U}\varphi) \mid \mathbf{A}(\varphi\mathbf{U}\varphi)$$

where $p \in \{\texttt{start}, \texttt{crt}\} \cup \{\texttt{snd}(m), \texttt{rec}(m) \mid m \in A\}$.

Formulas of MNL are evaluated on events of MSCs. The atomic formulas $\texttt{start}$, $\texttt{crt}$, $\texttt{snd}(m)$ and $\texttt{rec}(m)$ are true at an event if the event is from the corresponding type and, in case of $\texttt{snd}(m)$ and $\texttt{rec}(m)$, the sent or received message symbol is m. Intuitively, the operator $\mathbf{X}^{\downarrow}$ allows to make a step forward along a process edge and $\mathbf{X}^{\rightarrow}$ describes a step forward along a create or message edge. The operator $\mathbf{U}^{\downarrow}$ allows until-navigation along process edges starting at the current event. The quantifiers $\mathbf{E}$ and $\mathbf{A}$ express existential and universal quantification over paths which can contain process, message and create edges. We say that an MNL-formula *holds* on an MSC M if it is satisfied at the initial event $\texttt{init}(M)$ of M.

We illustrate the semantics of MNL by an example and leave the formal definition of the semantics to the Appendix (Section A.9):

Example 24. Let us go back to our initial client-and-server example in Chapter 2 and consider Property CS3:

Every client sending a request gets an acknowledgement after some time.

This can be expressed by the MNL-formula

$$\mathbf{AG}(\texttt{snd}(\texttt{req}) \rightarrow \mathbf{F}^{\downarrow}\texttt{rec}(\texttt{ack}))$$

containing the usual abbreviations $\mathbf{AG}\varphi \equiv \neg\mathbf{E}(\top\mathbf{U}\neg\varphi)$ and $\mathbf{F}^{\downarrow}\varphi \equiv \top\mathbf{U}^{\downarrow}\varphi$. □

Observe that in extended scenarios with two servers, properties like CS9 which demand that every client gets its acknowledgment from the server to which it sent the request, cannot be formulated in MNL. To express such a property, MNL would need some kind of freeze mechanism memorizing processes. Moreover, the logic does not allow to access processes in messages. We leave such extensions to future work.

12.3.2 Model Checking with MSC Navigation Logic

We will prove that model checking of BHMSCs with the logic MNL is ExpTime-complete. The lower bound is obtained from the lower bound of the non-emptiness problem for BHMSCs. For the upper bound, we give an ExpTime-procedure which, similar to the executability procedure in

Section 11.4.2, constructs symbolic runs where at each location, each register r is assigned to a set of formulas holding at the last event of the process represented by r. We start by explaining how the validity of an MNL-formula on an MSC can be checked based on the syntactical material of the formula.

From semantical to syntactical validity

We first define the *closure set* $\texttt{Closure}(\varphi)$ of an MNL-formula φ over some message alphabet A. For an MNL-formula φ, the set $\texttt{Closure}(\varphi)$ is the smallest set such that

- $\top \in \texttt{Closure}(\varphi)$,
- $\mathbf{X}^{\downarrow}\top \in \texttt{Closure}(\varphi)$,
- $\mathbf{X}^{\rightarrow}\top \in \texttt{Closure}(\varphi)$,
- all sub-formulas of φ are contained in $\texttt{Closure}(\varphi)$,
- for every formula in $\texttt{Closure}(\varphi)$, its negation is in $\texttt{Closure}(\varphi)$ (identifying $\neg\neg\psi$ with ψ),
- for every formula $\psi\mathbf{U}^{\downarrow}\chi \in \texttt{Closure}(\varphi)$, the formula $\mathbf{X}^{\downarrow}(\psi\mathbf{U}^{\downarrow}\chi)$ is in $\texttt{Closure}(\varphi)$,
- for every formula $\mathbf{E}(\psi\mathbf{U}\chi) \in \texttt{Closure}(\varphi)$, we have that $\mathbf{X}^{\downarrow}\mathbf{E}(\psi\mathbf{U}\chi)$ and $\mathbf{X}^{\rightarrow}\mathbf{E}(\psi\mathbf{U}\chi)$ are contained in $\texttt{Closure}(\varphi)$, and
- for every formula $\mathbf{A}(\psi\mathbf{U}\chi) \in \texttt{Closure}(\varphi)$, it holds that $\mathbf{X}^{\downarrow}\mathbf{A}(\psi\mathbf{U}\chi)$ and $\mathbf{X}^{\rightarrow}\mathbf{A}(\psi\mathbf{U}\chi)$ are contained in $\texttt{Closure}(\varphi)$.

A set $C \subseteq \texttt{Closure}(\varphi)$ is a *consistent set* of φ if

- $\top$ is included in C,
- exactly one atomic formula from $\{\texttt{start}, \texttt{crt}\} \cup \{\texttt{snd}(m), \texttt{rec}(m) \mid m \in A\}$ is included in C,
- for every formula $\psi \in \texttt{Closure}(\varphi)$, ψ is contained in C if and only if $\neg\psi$ is not contained in C (identifying $\neg\neg\psi$ with ψ),
- for every $\psi_1 \wedge \psi_2 \in \texttt{Closure}(\varphi)$, $\psi_1 \wedge \psi_2$ is contained in C if and only if ψ_1 and ψ_2 are contained in C,
- for every formula $\psi_1\mathbf{U}^{\downarrow}\psi_2 \in \texttt{Closure}(\varphi)$, $\psi_1\mathbf{U}^{\downarrow}\psi_2$ is contained in C if and only if (i) ψ_2 is contained in C, or (ii) ψ_1 and $\mathbf{X}^{\downarrow}(\psi_1\mathbf{U}^{\downarrow}\psi_2)$ are contained in C,
- for every formula $\mathbf{E}(\psi_1\mathbf{U}\psi_2) \in \texttt{Closure}(\varphi)$, the formula $\mathbf{E}(\psi_1\mathbf{U}\psi_2)$ is in C if and only if (i) ψ_2 is in C, or (ii) ψ_1 and $\mathbf{X}^{\downarrow}\mathbf{E}(\psi_1\mathbf{U}\psi_2)$ are in C, or (iii) ψ_1 and $\mathbf{X}^{\rightarrow}\mathbf{E}(\psi_1\mathbf{U}\psi_2)$ are in C, and
- for every formula $\mathbf{A}(\psi_1\mathbf{U}\psi_2) \in \texttt{Closure}(\varphi)$, the formula $\mathbf{A}(\psi_1\mathbf{U}\psi_2)$ is in C if and only if (i) ψ_2 is in C, or (ii) ψ_1, $\neg\mathbf{X}^{\rightarrow}\top$ and $\mathbf{X}^{\downarrow}\mathbf{A}(\psi_1\mathbf{U}\psi_2)$ are in C, or (iii) ψ_1, $\neg\mathbf{X}^{\downarrow}\top$ and $\mathbf{X}^{\rightarrow}\mathbf{A}(\psi_1\mathbf{U}\psi_2)$ are in C, or (iv) ψ_1, $\mathbf{X}^{\downarrow}\mathbf{A}(\psi_1\mathbf{U}\psi_2)$, and $\mathbf{X}^{\rightarrow}\mathbf{A}(\psi_1\mathbf{U}\psi_2)$ are in C.

A consistent set is called *final* if it does not contain any formula of the form $\mathbf{X}^{\downarrow}\psi$. The set of all consistent sets of φ is denoted by $\texttt{ConSets}(\varphi)$.

In the sequel, we show how the question whether a formula holds on an MSC can be solved by labelling the events of the MSC by consistent sets of φ and checking some conditions between the labels of neighbouring events. To this end, let $M = (E, \lhd, \lambda, \mu)$ be an MSC with $\lhd = \lhd_{\texttt{proc}} \uplus \lhd_{\texttt{crt}} \uplus \lhd_{\texttt{msg}}$. A labelling $h \in [E \to \texttt{ConSets}(\varphi)]$ is *valid* for M if it obeys the following rules:

- For every event $e \in E$ and atomic formula $p \in \{\texttt{start}, \texttt{crt}\} \cup \{\texttt{snd}(m), \texttt{rec}(m) \mid m \in A\}$, the formula p is contained in $h(e)$ if λ maps e to the type corresponding to p and, in case of $p = \texttt{snd}(m)$ or $p = \texttt{rec}(m)$, the sent or received symbol is m.
- *Vertical necessity condition*: For every event e, it holds that if a formula $\mathbf{X}^{\downarrow}\psi \in \texttt{Closure}(\varphi)$ is contained in $h(e)$, then, there is an event e' with $e \lessdot_{\texttt{proc}} e'$ such that ψ is contained in $h(e')$.
- *Horizontal necessity condition*: For every event e, it holds that if a formula $\mathbf{X}^{\rightarrow}\psi \in \texttt{Closure}(\varphi)$ is contained in $h(e)$, then, there is an event e' with $e \lessdot_{\texttt{crt}} \cup \lessdot_{\texttt{msg}} e'$ such that ψ is contained in $h(e')$.
- *Vertical consistency condition*: For every two neighbouring events e_1 and e_2 with $e_1 \lessdot_{\texttt{proc}} e_2$ and every formula $\mathbf{X}^{\downarrow}\psi \in \texttt{Closure}(\varphi)$, it holds that $\mathbf{X}^{\downarrow}\psi$ is contained in $h(e_1)$ if and only if ψ is contained in $h(e_2)$. If for two events e_1 and e_2, the vertical consistency condition holds, then, e_2 is called *vertically consistent* to e_1.
- *Horizontal consistency condition*: For every two neighbouring events e_1 and e_2 with $e_1 \lessdot_{\texttt{crt}} \cup \lessdot_{\texttt{msg}} e_2$ and every formula $\mathbf{X}^{\rightarrow}\psi \in \texttt{Closure}(\varphi)$, it holds that $\mathbf{X}^{\rightarrow}\psi$ is contained in $h(e_1)$ if and only if ψ is contained in $h(e_2)$. Similar to above, if for two events e_1 and e_2 the horizontal consistency condition holds, we call e_2 *horizontally consistent* to e_1.

Note that it follows from the conditions above that a valid labelling cannot assign formulas of the form $\mathbf{X}^{\downarrow}\psi$ to the last events e of processes in M, i.e., to those for which there is no e' with $e \lessdot_{\texttt{proc}} e'$. A labelling h on a partial MSC M is called *valid up to a set $P \subseteq \mathbb{P}$ of processes* if h is valid for M except that for the last events of processes in M which are contained in P, the vertical necessity condition does not have to hold. We say that a (full) MSC M can be φ-labelled if there is a valid labelling h for M such that $\varphi \in h(\texttt{init}(M))$.

Proposition 17. For every MSC M and MNL-formula φ, it holds that φ holds on M if and only if M can be φ-labelled.

Proof. Let $M = (E, \lessdot, \lambda, \mu)$ be an MSC with $\lessdot = \lessdot_{\texttt{proc}} \uplus \lessdot_{\texttt{crt}} \uplus \lessdot_{\texttt{msg}}$ and φ an MNL-formula. We first proof the"only if"-direction. Let $M \models \varphi$ and h be a labelling function mapping events in E to subsets of $\texttt{Closure}(\varphi)$ such that for every $e \in E$ and $\psi \in \texttt{Closure}(\varphi)$, the formula ψ is contained in $h(e)$ if and only if $(M, e) \models \psi$. The labelling h obviously constitutes a valid φ-labelling. To be convinced, first observe that $\varphi \in h(\texttt{init}(M))$ and for every $e \in E$, the set $h(e)$ must be a consistent set. Furthermore, by construction of h, for every $e \in E$ and every atomic formula $p \in \{\texttt{start}, \texttt{crt}\} \cup \{\texttt{snd}(m), \texttt{rec}(m) \mid m \in A\}$, the formula p is contained in $h(e)$ if and only if the type and possible message symbol at e correspond to p. Finally, the necessity and consistency conditions hold. We exemplarily show that the horizontal consistency condition is not violated. Let $e_1 \lessdot_{\texttt{crt}} \cup \lessdot_{\texttt{msg}} e_2$ be two neighbouring events in M and $\mathbf{X}^{\rightarrow}\psi \in \texttt{Closure}(\varphi)$. By construction of $\texttt{Closure}(\varphi)$, we have $\psi \in \texttt{Closure}(\varphi)$. As h labels every event with those formulas from $\texttt{Closure}(\varphi)$ which are true at that event, it must hold $\mathbf{X}^{\rightarrow}\psi \in h(e_1) \Leftrightarrow \psi \in h(e_2)$.

We now deal with the "if"-direction of the statement of the proposition. Let h be a valid φ-labelling for M. We will show that for every event e and every $\psi \in \texttt{Closure}(\varphi)$, it holds $\psi \in h(e) \Leftrightarrow M, e \models \psi$. Then, since $\varphi \in h(\texttt{init}(M))$, the result follows. The proof is by induction on the structure of ψ. We exemplarily consider the induction steps in the cases $\psi = \mathbf{X}^{\downarrow}\chi$ and $\psi = A(\psi_1 \mathbf{U} \psi_2)$.

- $\psi = \mathbf{X}^{\downarrow}\chi$: Let $\mathbf{X}^{\downarrow}\chi \in h(e)$ for some event e. By the necessity and consistency conditions, this holds if and only if there is a successor e' of e with $e \lessdot_{\texttt{proc}} e'$ and $\chi \in h(e')$. By induction hypothesis, the latter is equivalent to $(M, e') \models \chi$. This in turn is equivalent to $(M, e) \models \mathbf{X}^{\downarrow}\chi$.

- $\psi = \mathbf{A}(\psi_1 \mathbf{U} \psi_2)$: Let $\mathbf{A}(\psi_1 \mathbf{U} \psi_2) \in h(e)$. By the definition of consistent sets and valid labellings, this holds if and only if for all sequences $e_1 \lhd \ldots \lhd e_n$ with $e_1 = e$ and $n \geq 1$, we have $\psi_2 \in h(e_n)$ and $\psi_1 \in h(e_i)$ for all i with $1 \leq i < n$. By induction hypothesis, this is true if and only if for all such sequences $e_1 \lhd \ldots \lhd e_n$, it holds $(M, e_n) \models \psi_2$ and $(M, e_i) \models \psi_2$ for all i with $1 \leq i < n$. Obviously, this is equivalent to $(M, e) \models \mathbf{A}(\psi_1 \mathbf{U} \psi_2)$.

□

Solving model checking with MNL on symbolic BHMSC-runs

As usual, we will reduce the model checking problem for BHMSCs and MNL to the existential model checking problem for these formalisms. The decidability proof of the latter problem consists of two main steps: First, we will show that the problem can be reduced to some reachability problem on symbolic runs. Then, we will give a decision procedure for this reachability problem. In the sequel, we introduce some notions and preliminary propositions that will be helpful for the decidability proof. For the rest of this section, we fix a BHMSC $\mathcal{H} = (A, L, L_{\mathtt{init}}, L_{\mathtt{acc}}, R, r_0, \delta)$ and a formula φ.

Process validity mappings. A *process validity mapping* $PV \in [\mathbb{P} \rightharpoonup \mathtt{ConSets}(\varphi)]$ is a partial function which maps processes in $\mathbb{P}$ to consistent sets of φ. Next, we define transitions of the form $\xrightarrow[h \mid P]{M}$ between process validity mappings where M is a partial MSC, P a set of processes and h a labelling function on M which is valid up to P. Given two process validity mappings PV and PV', the intuitive meaning of $PV \xrightarrow[h \mid P]{M} PV'$ is as follows: Imagine that there is some MSC $\hat{M}$ labelled via some mapping $\hat{h}$. Assumed that PV represents the labels of the last events of the processes in $\hat{M}$, the MSC M can be appended to $\hat{M}$ such that $\hat{h}$ can be continued on M using h. Moreover, the process validity mapping PV' represents the labels of the last events of some processes occurring in $\hat{M} \circ M$. The only reason why h is valid up to P is that we keep the option open that the processes in P can be continued in some further MSC appended to M. More formally, for a partial MSC M with event set E, two process validity mappings PV and PV', a labelling function $h \in [E \to \mathtt{ConSets}(\varphi)]$ and a set $P \subseteq \mathbb{P}$, we write $PV \xrightarrow[h \mid P]{M} PV'$ if the following conditions hold:

1. $\mathtt{Free}(M) \subseteq \mathtt{dom}(PV)$.
2. h is up to P a valid labelling for M such that for every process $p \in \mathtt{Free}(M)$, it holds: if e is the first event of p in M, then, $h(e)$ is vertically consistent to $PV(p)$.
3. PV and PV' have the following properties:
   (a) $\mathtt{dom}(PV') \subseteq \mathtt{dom}(PV) \cup \mathtt{Bound}(M)$,
   (b) for all $p \in \mathtt{dom}(PV) \backslash (\mathtt{Pids}(M) \cup \mathtt{dom}(PV'))$ (recall that $\mathtt{Pids}(M)$ denotes the set of all processes in M), the set $PV(p)$ is final,
   (c) for every process $p \in \mathtt{dom}(PV')$, it holds:
      - if p occurs in M with last event e, then, $PV'(p) = h(e)$,
      - otherwise, $PV'(p) = PV(p)$.

The intuition behind condition (3.b) is that the set $\mathtt{dom}(PV) \backslash (\mathtt{Pids}(M) \cup \mathtt{dom}(PV'))$ contains processes which will never occur in MSCs appended to M.

Register validity mappings. *Register validity mappings* can be seen as symbolic counterparts of process validity mappings and build the key elements for our procedure solving the model checking

problem for BHMSCs on symbolic runs. A register validity mapping $RV \in [R \rightharpoonup \texttt{ConSets}(\varphi)]$ is a partial function mapping registers in R to consistent sets of φ. The process validity mapping induced by a register assignment $\nu \in [R \rightharpoonup \mathbb{P}]$ and a register validity mapping RV is denoted by $PV_{\{\nu,RV\}}$ and is defined as follows: For every process $p \in \mathbb{P}$, it holds (i) if for some register r, $\nu(r)$ is defined by p and $RV(r)$ is defined, then, $PV_{\{\nu,RV\}}(p) = RV(r)$, and (ii) otherwise, $PV_{\{\nu,RV\}}(p)$ is undefined. Conversely, we define the register validity mapping $RV_{\{\nu,PV\}}$ induced by a register assignment ν and a process validity mapping PV by: For every register $r \in R$, we have (i) if for some process p, $\nu(r)$ is defined by p and $PV(p)$ is defined, then, $RV_{\{\nu,PV\}}(r) = PV(p)$, and (ii) otherwise, $RV_{\{\nu,PV\}}(r)$ is undefined. Let G be a run with input register assignment ν and output register assignment ν', let RV be a register validity mapping with $\texttt{dom}(RV) = \texttt{dom}(\nu)$ and let $R' \subseteq R$. We write $RV \xrightarrow[R']{G} RV'$, if there exists some labelling h (on the event set of $M(G)$) such that $PV_{\{\nu,RV\}} \xrightarrow[h \,\mid\, \nu'(R')]{M(G)} PV_{\{\nu',RV'\}}$.

The following two propositions describe how register validity mappings for complex runs can be obtained from the register validity mappings of sub runs. As usual, for a mapping $f \in [A \rightharpoonup B]$ and a subset $A' \subseteq A$, we mean by $f_{\restriction A'}$ the mapping which results from f by restricting its domain to A'. Recall that for two mappings $f_1 \in [A_1 \rightharpoonup B]$ and $f_2 \in [A_2 \rightharpoonup B]$ with $A_1 \cap A_2 = \emptyset$, the mapping $f_1 \cup f_2 \in [A_1 \cup A_2 \rightharpoonup B]$ is defined by: for every $a \in A_1 \cup A_2$, it holds that (i) $f_1 \cup f_2(a) = f_1(a)$ if $a \in A_1$ and $f_1(a)$ is defined, (ii) $f_1 \cup f_2(a) = f_2(a)$ if $a \in A_2$ and $f_2(a)$ is defined and (iii) $f_1 \cup f_2(a)$ is undefined, otherwise.

Proposition 18. Let G be a run resulting from the concatenation of two runs G_1 and G_2 and let $R' \subseteq R$. Then, $RV \xrightarrow[R']{G} RV'$ if and only if there is an RV_1 such that $RV \xrightarrow[R]{G_1} RV_1 \xrightarrow[R']{G_2} RV'$.

Proof. Let G be a run resulting from the concatenation of two runs G_1 and G_2 and let $R' \subseteq R$. Furthermore, let ν and ν' be the input and output register assignments of G and ν_1 the output register assignment of G_1. Let $M(G) = (E, \lhd, \lambda, \mu)$ and $M(G_i) = (E_i, \lhd_i, \lambda_i, \mu_i)$ for $i \in \{1, 2\}$.

We start with the proof of the "only if"-direction. From $RV \xrightarrow[R']{G} RV'$ it follows by definition that there exists some labelling h on E such that $PV_{\{\nu,RV\}} \xrightarrow[h \,\mid\, \nu'(R')]{M(G)} PV_{\{\nu',RV'\}}$. We can show that there exists some PV_1 with $\texttt{dom}(PV_1) = \nu_1(R)$ such that (i) $PV_{\{\nu,RV\}} \xrightarrow[h_{\restriction E_1} \,\mid\, \nu_1(R)]{M(G_1)} PV_1$ and (ii) $PV_1 \xrightarrow[h_{\restriction E_2} \,\mid\, \nu'(R')]{M(G_2)} PV_{\{\nu',RV'\}}$. Then, it follows by definition that $RV \xrightarrow[R]{G_1} RV_{\{\nu,PV_1\}} \xrightarrow[R']{G_2} RV'$. For the sake of the correctness of (i) and (ii), it can easily be checked that all conditions in the definition of transitions on process validity mappings are satisfied. We just explain why for case (i) condition (3.b) must hold. Let $p \in PV_{\{\nu,RV\}} \backslash (\texttt{Pids}(M(G_1)) \cup \texttt{dom}(PV_1))$. Observe that it follows $p \in PV_{\{\nu,RV\}} \backslash (\texttt{Pids}(M(G)) \cup \texttt{dom}(PV_{\{\nu',RV'\}}))$. Since $PV_{\{\nu,RV\}} \xrightarrow[h \,\mid\, \nu'(R')]{M(G)} PV_{\{\nu',RV'\}}$, by definition, $PV_{\{\nu,RV\}}(p)$ must be final and, thus, condition (3.b) is satisfied.

We now turn towards the "if"-direction. From $RV \xrightarrow[R]{G_1} RV_1 \xrightarrow[R']{G_2} RV'$ it follows by definition that there exist some labellings h_1 and h_2 on E_1 and E_2, respectively, such that $PV_{\{\nu,RV\}} \xrightarrow[h_1 \,\mid\, \nu_1(R)]{M(G_1)} PV_{\{\nu_1,RV_1\}} \xrightarrow[h_2 \,\mid\, \nu'(R')]{M(G_2)} PV_{\{\nu',RV'\}}$. We can easily show that we get (i) $PV_{\{\nu,RV\}} \xrightarrow[h_1 \cup h_2 \,\mid\, \nu'(R')]{M(G)} PV_{\{\nu',RV'\}}$. From the latter it follows by definition that that $RV \xrightarrow[R']{G} RV'$. For the correctness of (i), we again only consider condition (3.b). Let $p \in PV_{\{\nu,RV\}} \backslash (\texttt{Pids}(M(G)) \cup \texttt{dom}(PV_{\{\nu',RV'\}}))$. As it follows $p \in PV_{\{\nu,RV\}} \backslash (\texttt{Pids}(M(G_1)) \cup \texttt{dom}(PV_{\{\nu_1,RV_1\}}))$, the set $PV_{\{\nu,RV\}}(p)$ must be final. $\square$

Proposition 19. Let G be a run resulting from some runs $G_1, \ldots, G_n$ via a fork-and-join transition $\ell \to \{(\ell_1, R_1, \ell'_1), \ldots, (\ell_n, R_n, \ell'_n)\} \to \ell$ and let $R' \subseteq R$. Then,

$$RV \xrightarrow[R']{G} RV' \text{ if and only if } RV_{\upharpoonright R_i} \xrightarrow[R_i \cap R']{G_i} RV'_i \text{ for every } i \in \{1 \ldots, n\}$$

where $RV' = RV_{\upharpoonright R_0} \cup \bigcup_{1 \leq i \leq n} RV'_{i \upharpoonright R_i}$.

Proof. Assume that G is a run resulting from subruns $G_1, \ldots, G_n$ via a fork-and-join transition $\ell \to \{(\ell_1, R_1, \ell'_1), \ldots, (\ell_n, R_n, \ell'_n)\} \to \ell$. Let $M(G) = (E, \lhd, \lambda, \mu)$ and $M(G_i) = (E_i, \lhd_i, \lambda_i, \mu_i)$ for every i with $1 \leq i \leq n$. Moreover, let ν and ν' be the input and output register assignments of G and ν_i and ν'_i the input and output register assignments of each G_i. Finally, let $R' \subseteq R$.

We first deal with the "only if"-direction of the proposition. From the definition of $RV \xrightarrow[R']{G} RV'$ it follows $PV_{\{\nu, RV\}} \xrightarrow[h \ | \ \nu'(R')]{M(G)} PV_{\{\nu', RV'\}}$ for some labelling function h. It can easily be checked that from this it follows that for every $i \in \{1, \ldots, n\}$, it holds $PV_{\{\nu_i, RV_{\upharpoonright R_i}\}} \xrightarrow[h_{\upharpoonright E_i} \ | \ \nu'_i(R_i \cap R')]{M(G_i)} PV_{\{\nu'_i, RV'_i\}}$ such that $RV' = RV_{\upharpoonright R_0} \cup \bigcup_{1 \leq i \leq n} RV'_{i \upharpoonright R_i}$. Thus, we get $RV_{\upharpoonright R_i \cap R'} \xrightarrow{G_i} RV'_i$ for every $i \in \{1, \ldots, n\}$.

The "if"-direction can be shown as follows. Let $RV_{\upharpoonright R_i} \xrightarrow[R_i \cap R']{G_i} RV'_i$ for every $i \in \{1, \ldots, n\}$. By definition, it holds that for every $i \in \{1, \ldots, n\}$, there is some mapping h_i on E_i such that $PV_{\{\nu_i, RV_{\upharpoonright R_i}\}} \xrightarrow[h_i \ | \ \nu'_i(R_i \cap R')]{M(G_i)} PV_{\{\nu'_i, RV'_i\}}$. It follows $PV_{\{\nu, RV\}} \xrightarrow[h \ | \ \nu'(R')]{M(G)} PV_{\{\nu', RV'\}}$ where $h = \bigcup_{i \in 1, \ldots, n} h_i$ and $RV' = RV_{\upharpoonright R_0} \cup \bigcup_{1 \leq i \leq n} RV'_{i \upharpoonright R_i}$. By definition, it follows $RV \xrightarrow[R']{G} RV'$. □

Register validity mappings on symbolic runs. We now define a transition relation on register validity mappings which does not refer to concrete runs, but to symbolic ones. This definition and the following lemma provide the basis for our model checking procedure on symbolic runs.

Let S be a symbolic run with input register set D and output register set D', let RV and RV' be register validity mappings with $\mathtt{dom}(RV) = D$ and $\mathtt{dom}(RV') = D'$ and let $R' \subseteq D'$. We inductively define what $RV \xrightarrow[R']{S} RV'$ means:

- Assume that S is a symbolic run resulting from a sequential transition $(\ell, M, \ell') \in \delta$. Then, $RV \xrightarrow[R']{S} RV'$ if for every $r \in \mathtt{Bound}(M) \cap D$, the consistent set $RV(r)$ is final and there is some labelling h such that $RV \xrightarrow[h \ | \ R']{M} RV'$. Note that in the last expression, the registers are interpreted as processes.
- Assume that S is a symbolic run resulting from the concatenation of S_1 and S_2. Then, $RV \xrightarrow[R']{S} RV'$ if $RV \xrightarrow[R]{S_1} RV_1 \xrightarrow[R']{S_2} RV'$ for some RV_1.
- Now, assume that S is a symbolic run resulting from subruns $S_1, \ldots, S_n$ by a fork-and-join transition $\ell \to \{(\ell_1, R_1, \ell'_1), \ldots, (\ell_n, R_n, \ell'_n)\} \to \ell'$. Then, $RV \xrightarrow[R']{S} RV'$ if for every $i \in \{1, \ldots, n\}$, it holds $RV_{\upharpoonright R_i} \xrightarrow[R_i \cap R']{S_i} RV'_i$ with $RV' = RV_{\upharpoonright R_0} \cup \bigcup_i RV'_{i \upharpoonright R_i}$.

Recall that in Section 11.4, we defined a mapping `symb` from concrete BHMSC-runs to symbolic ones and proved in Lemma 12 that the mapping is surjective. We use this mapping in the following lemma to build a bridge between register validity mappings on concrete and symbolic runs.

Lemma 24. *Let G be a run, $R' \subseteq R$ and RV and RV' two register validity mappings. Then,*

$$RV \xrightarrow[R']{G} RV' \text{ if and only if } RV \xrightarrow[R']{\mathit{symb}(G)} RV'.$$

Proof.

- If G is a run resulting from a sequential transition $(\ell, M, \ell') \in \delta$, then, by definition, it holds $RV \xrightarrow[R']{G} RV'$ if and only if there is some labelling function h such that $PV_{\{\nu,RV\}} \xrightarrow[h \mid \nu'(R')]{M(G)} PV_{\{\nu',RV'\}}$. Note that this is equivalent to $RV \xrightarrow[h \mid R']{M} RV'$ where the registers are interpreted as processes and for every $r \in \mathtt{Bound}(M) \cap D$ the consistent set $RV(r)$ is final. The latter condition is needed to meet condition (3.b) from the definition of the transitions on process validity mappings. Finally, by definition, $RV \xrightarrow[h \mid R']{M} RV'$ is equivalent to $RV \xrightarrow[R']{\mathtt{symb}(G)} RV'$.

- Assume that G is a run resulting from the concatenation of some runs G_1 and G_2. By Proposition 18, we have $RV \xrightarrow[R']{G} RV'$ if and only if there is some RV_1 such that $RV \xrightarrow[R]{G_1} RV_1 \xrightarrow[R']{G_2} RV'$. By induction hypothesis, this holds if and only if $RV \xrightarrow[R]{\mathtt{symb}(G_1)} RV_1 \xrightarrow[R']{\mathtt{symb}(G_2)} RV'$. By definition, this is equivalent to $RV \xrightarrow[R']{\mathtt{symb}(G)} RV'$.

- Let G be a symbolic run resulting from runs $G_1, \ldots, G_n$ via a fork-and-join transition $\ell \to \{(\ell_1, R_1, \ell'_1), \ldots, (\ell_n, R_n, \ell'_n)\} \to \ell'$. By Proposition 19, it holds $RV \xrightarrow[R']{G} RV'$ if and only if $RV_{\restriction R_i} \xrightarrow[R_i \cap R']{G_i} RV'_i$ for every $i \in \{1, \ldots, n\}$ with $RV' = RV_{\restriction R_0} \cup \bigcup_{1 \le i \le n} RV'_{i\,\restriction R_i}$. By induction hypothesis, this is equivalent to $RV_{\restriction R_i} \xrightarrow[R_i \cap R']{\mathtt{symb}(G_i)} RV'_i$ for $1 \le i \le n$. By definition, the latter is equivalent to $RV \xrightarrow[R']{\mathtt{symb}(G)} RV'$.

□

Now, we are ready to state and prove the main theorem of this section.

Theorem 34. *The problem* ModCheck*(BHMSC,MNL) is* ExpTime*-complete.*

Proof. The lower bound follows from the ExpTime-hardness of the non-emptiness problem for BHMSCs [46] and the fact that MNL is closed under negating formulas. For the upper bound, we will show that the existential model checking problem EModCheck(BHMSC,MNL) is in ExpTime. Then, the result easily follows. For the proof that EModCheck(BHMSC,MNL) is in ExpTime, we first observe the following equivalences. Let $\mathcal{H} = (A, L, L_{\mathtt{init}}, L_{\mathtt{acc}}, R, r_0, \delta)$ be a BHMSC and φ an MNL-formula.

There is an MSC $M \in \mathcal{L}(\mathcal{H})$ with $M \models \varphi$
$\Leftrightarrow$ there is an MSC $M \in \mathcal{L}(\mathcal{H})$ which can be φ-labelled (Proposition 17)
$\Leftrightarrow$ there is a an accepting run G of $\mathcal{H}$ with some initial process p, a set $C \in \texttt{ConSets}$ containing φ and a labelling h such that $\{p \mapsto C\} \xrightarrow[h \mid \emptyset]{M(G)} PV'$ for some process validity mapping PV' (by the definition of the transitions on process validity mappings)
$\Leftrightarrow$ there is a an accepting run G of $\mathcal{H}$ and a set $C \in \texttt{ConSets}$ containing φ such that $\{r_0 \mapsto C\} \xrightarrow[\emptyset]{G} RV'$ for some register validity mapping RV' (by the definition of the transitions on register validity mappings)
$\Leftrightarrow$ there is a an accepting symbolic run S of $\mathcal{H}$ and a set $C \in \texttt{ConSets}$ containing φ such that $\{r_0 \mapsto C\} \xrightarrow[\emptyset]{S} RV'$ for some RV' (by Lemma 24)

Thus, the question whether a given BHMSC $\mathcal{H}$ with initial register r_0 generates an MSC satisfying a given formula φ can be answered by searching for a symbolic run S for $\mathcal{H}$ such that $\{r_0 \mapsto C\} \xrightarrow[\emptyset]{S} RV'$ for some consistent set C containing φ and some register validity mapping RV'. We design an algorithm which, similar to the non-emptiness procedure in the proof of Lemma 13 and the executability procedure in the proof of Lemma 17, computes for a given BHMSC $\mathcal{H} = (A, L, L_{\texttt{init}}, L_{\texttt{acc}}, R, r_0, \delta)$ and a formula φ, the set $\mathcal{T}$ of all tuples $(\ell, RV, \ell', RV', R')$ such that there is some symbolic run S, the input location of S is ℓ, its output location is ℓ' and it holds $RV \xrightarrow[R']{S} RV'$. If $\mathcal{T}$ contains a tuple $(\ell, \{r_0 \mapsto C\}, \ell', RV', \emptyset)$ with $\ell \in L_{\texttt{init}}$, $\ell' \in L_{\texttt{acc}}$ and C is a consistent set containing φ, the procedure returns yes, otherwise no. Observe that there are at most $(2^{\mathcal{O}(|\varphi|)})^{|R|}$ different register validity mappings (and exponentially many different subsets of R). Hence, the cardinaity of $\mathcal{T}$ is at most exponential. In each iteration, our procedure checks for every tuple $t = (\ell, RV, \ell', RV', R')$ whether it can be obtained via a symbolic run resulting from a sequential transition, a concatenation of symbolic runs or a fork-and-join transition. In the first case, the algorithm tests in time at most exponential in the length of φ and the size of δ whether there is a sequential transition in δ containing an MSC whose events can be labelled by consistent sets such that the labels of the first events fit to RV and those of the last ones fit to RV'. In the second case, it checks whether among the tuples computed so far, there is a pair of tuples which can be concatenated such that t is obtained. The number of choices to decide this test is at most polynomial in the number of computed tuples. In the last case, it tests whether there is a fork-and-join transition and a set of tuples which can be combined such that t results. The number of choices for the latter task is polynomial in the number of computed tuples and exponential in $|\delta|$. Since the computation of $\mathcal{T}$ requires at most exponential iterations and each iteration requires at most exponential time, the overall computation time is at most exponential. $\square$

12.4 Discussion

We considered the model checking of DCA, PRA and BHMSCs by data logics. Our results should be seen as first steps towards the investigation of the verification of systems with unboundedly many processes by the use of data logics.

In case of DCA, we restricted our considerations to selective 1-register DCA, since this is the biggest DCA-fragment for which we were able to show decidable non-emptiness (and in our setting, model checking is at least as hard as non-emptiness). For the model checking of this fragment, we used RB-DLTL, a restriction of B-DLTL where shift values are skipped, and $\text{LTL}^{\Downarrow}$.

In case of PRA, our original motivation was driven by a result from [45] which says that model checking of Data Multi-Pushdown Automata (DMPA) with full $\text{MSO}^{\sim}$ is decidable. Our aim was to find interesting fragments of $\text{MSO}^{\sim}$ which can deliver moderate, at least elementary, model checking complexity for DMPA. To obtain a full understanding of the inherent complexity of the system model, we restricted our investigations to PRA which is a fragment of DMPA where pushdown stacks are skipped. On the logic side, we considered subsets of $\text{LTL}^{\Downarrow}$ and $\text{HTL}^{\sim}$.

As explained in Chapter 10, there are two popular approaches in the verification of high-level MSC-descriptions. In the first approach, formulas are evaluated on all linearizations of events of MSCs. This approach mostly results in high model checking complexities (see, e.g., in [25]). The second approach, which yields better complexity results, uses structural logics whose formulas can navigate on partially ordered sets, but cannot distinguish between different linearizations of the same MSC. Following the second approach and inspired by structural logics on MSCs considered in [48, 173, 153, 154], we designed the logic MNL and used it for the model checking of BHMSCs.

Our complexity results are summarized in Figure 12.6. A "c" behind a complexity class means

	selective 1-DCA	PRA	BHMSCs
RB-DLTL	dec. (12.1.1)	in ExpSpace	-
$\text{LTL}^{\Downarrow}_k(\mathbf{X},\mathbf{U})$ for $k \geq 1$	undec. (12.1.2)	PSpace-c (12.2.1)	-
$\text{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$	undec.	in ExpSpace (12.2.1)	-
$\text{HTL}^{\sim}_1$	undec.	ExpSpace-c (12.2.2)	-
$\text{HTL}^{\sim}_k$ for $k \geq 2$	undec.	dec., non-el. (12.2.2)	-
MNL	-	-	ExpTime-c (12.3.2)

Figure 12.6: Our model checking results in this chapter

that the problem is complete for this class. A "-" indicates that the corresponding system model and the logic are not compatible with each other. The numbers in brackets indicate the sections in which the results are proven. In the sequel, we first discuss some questions left open and then state some conclusions.

We assume that our technique in the decidability proof for the model checking of selective 1-DCA with RB-DLTL can be extended to B-DLTL. Remember that we showed in Proposition 1 of Section 6.2 that every B-DNL-formula can be translated into an equivalent formula in normal form where in all sub-formulas of the form $\mathcal{C}^{\ell}_{\texttt{@a}}\psi$ with $\ell \neq 0$, it holds $\psi = \sim\texttt{@b}$ or $\psi = \neg \sim\texttt{@b}$. The latter formulas express (un-)equality conditions on data values of positions of bounded distance. It can easily be observed that the translation given in Section 6.2 also works for B-DLTL-formulas. Hence, in order to extend our model checking procedure with RB-DLTL to B-DLTL, it would suffice to take such assertions into account. Moreover, recall that the decidability of ModCheck(selective 1-DCA,RB-DLTL) relies on the fact that configurations of this DCA-fragment can be separated into isolated sub configurations consisting of single processes and pairs of processes (Observations 8 and 9 in Section 11.2.1). It seems that our decision procedure can be extended to all DCA-fragments where configurations can be separated into finitely many sets of isolated sub configurations such that elements in the same set are of the same shape. Yet, the most interesting question with regard to selective 1-DCA is whether model checking with full Data Navigation Logic (DNL) is decidable. The undecidability result in case of $\text{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$ does not give any hint here, because it follows from Theorem 6 in Section 6.4 that the property expressed by $\text{LTL}^{\Downarrow}_1(\mathbf{X},\mathbf{U})$ in Theorem 29 cannot be formulated in DNL. Finally, we assume that, similar to the lower bound of the satisfiability of $\text{FO}^{\sim}_2(\texttt{Suc},<)$ [41], it can be shown that ModCheck(selective 1-DCA,RB-DLTL) is as hard as non-emptiness of Multicounter Automata for which no elementary upper bound is known.

The upper bound of ModCheck(PRA,RB-DLTL) is derived from the result for ModCheck

(PRA, $\mathrm{HTL}_1^{\sim}$). The precise complexities of ModCheck(PRA,RB-DLTL) and ModCheck(PRA, $\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$) remain open. Due to the complications explained in Section 12.2.1, the extension of our model checking algorithm for ModCheck(PRA,$\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$) to past operators remains an interesting challenge.

Observe that MNL is quite restrictive as it does not contain past operators. Furtermore, it does not provide mechanisms in the style of $\mathrm{LTL}^{\Downarrow}$ to "freeze" processes at events in order to compare them with processes at other events. Due to this shortcoming, it is not clear how to express properties like CS9 from Chapter 2. We think that, just like automata constructions for PLTL, our strategy of labelling events by consistent sets in the upper bound proof for ModCheck(BHMSC,MNL) in Section 12.3.2 can easily be extended to past operators. The insertion of freeze mechanisms could be handled by consistent sets of pairs of formulas and freeze assignments, like in the model checking algorithm for ModCheck(PRA,$\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$).

We conclude from our results that model checking of DCA which describe the behaviour of single processes is much more difficult than the model checking of PRA and BHMSCs which describe systems from a more global point of view. Particularly in case of PRA, we surprisingly recognize that model checking with undecidable logics like $\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$ and $\mathrm{HTL}_1^{\sim}$ deliver elementary complexities. However, concerning our initial motivation with respect to DMPA, we unfortunately must notice that already for the DMPA-fragment PRA and the $\mathrm{MSO}^{\sim}$-fragment $\mathrm{HTL}_2^{\sim}$, the complexity of model checking becomes non-elementary. Nevertheless, an interesting question for future work is whether the complexity of model checking with logics like $\mathrm{LTL}^{\Downarrow}(\mathbf{X},\mathbf{U})$ and $\mathrm{HTL}_1^{\sim}$ remains elementary if PRA are extended by a stack, resulting in a model which is closer to DMPA. Moreover, the symbolic representation of runs of PRA (and BHMSCs) turned out to be useful method which might be employed in further investigations.

It should be noted that the system models considered in this work generate finite traces and MSCs. In classical system verification it is usually assumed that systems run ad infinitum (see, e.g., in [30]). We think that further investigations on model checking with data logics should take this into account and consider system models with infinite traces and MSCs.

▮ The results presented in this chapter are developed by myself and not published in any paper.

Chapter 13

The Journey of Data Logics - A Glance into the Future

In this work, we studied logics and automata on data words. We first investigated their expressivity and their complexity with respect to satisfiability and non-emptiness. Then, we took first steps towards the study of the computational properties of data logics in the area of model checking of concurrent systems with unboundedly many processes. To this end, we searched for models which are suitable for the description of such systems. In particular, we were interested in models whose traces can be represented by data words. Then, we decided for the three models Dynamic Communicating Automata, Process Register Automata and Branching High-Level Message Sequence Charts. Before the investigation of their model checking with respect to data logics, we analyzed different problems like non-emptiness, reachability and executability for these models. As explained in previous chapters, in our setting, the complexity of non-emptiness for these models sets a lower bound for their model checking complexity with data logics. Summaries of obtained results and discussions of questions left open were already given in the final sections of the corresponding chapters. In this concluding chapter, we would like to formulate some general thoughts, based on our insights in this work, about future research on data logics.

As stated in the introduction, even though the verification of systems with unboundedly many processes was one of the main motivations for the investigation of logics and automata on data words, the most considered problems in previous works were satisfiability and non-emptiness until now. We explained this fact by the lack of standardized system models and the intention to find expressive, but decidable logics and automata which can constitute a foundation for model checking. Nevertheless, we observe that in recent years the research on formalisms on data words approaches more and more the field of verification and model checking. For instance, in [19, 108] it is discovered that Data and Register Automata form convenient tools for the verification of programs that are accessing arrays and lists with data. Moreover, the designers of newly introduced logics and automata on data words put particular emphasis on the usefulness of their formalisms for the verification of concurrent systems [75, 71]. We also find recent publications which use data logics for the model checking of parameterized systems [110, 111]. These works raise the expectation that future works on data logics will mainly focus on verification and model checking. Our results in this work, in particular those in Part C, should be seen as a contribution in this respect.

One of our main conclusions is that the bad properties of data logics with respect to satisfiability should not prevent from the investigation of these logics in the verification of concurrent systems with unboundedly many processes. We admit that one of our results provides a bad example: It turned out that the combination of selective 1-DCA and $\text{LTL}^{\Downarrow}{}_1(\mathbf{X}, \mathbf{U})$, which are de-

cidable with regard to non-emptiness and satisfiability, respectively, leads to an undecidable model checking problem. On the contrary, our decidability results on PRA and BHMSCs are quite motivating. In particular in case of PRA, we found out that model checking with undecidable logics like $\mathrm{LTL}^{\Downarrow}(\mathbf{X}, \mathbf{U})$ and $\mathrm{HTL}_1^{\sim}$ is decidable with elementary complexity. An obvious reason for this is that the traces which are generated by PRA are represented by data words of restricted nature. In [45], it is mentioned that the decidability of the model checking problem for DMPA, a generalization of PRA, with full $\mathrm{MSO}^{\sim}$ relies on the *bounded tree-width* of the traces of this model. As a possible future work, it should be investigated whether there is a more specific characterization for PRA-traces which can be formalized independently from the system model. Such a characterization can offer the opportunity to design more powerful models with traces which are subject to similar restrictions. In the area of XML and XPath, we observe that similar strategies have led to fruitful results. Several characterizations like *bounded guidance width* [42], *bounded braid width* [32] and *bounded match width* [32] on classes of data trees have formed the basis for new expressivity and decidability results in the area of XML and XPath. We think that in the future, such strategies should be also pursued in the area of model checking with data logics.

Acronyms

CA Communicating Automaton. 117–120, 122, 159
  DCA Dynamic Communicating Automaton. 118–122, 124–126, 128–136, 141, 142, 144, 148, 159, 167–169, 171–173, 176, 177, 179, 180, 182–184, 192, 207, 208, 211
    bDCA Buffered Dynamic Communicating Automaton. 141, 142
CLTL$^{\mathsf{XF}}$ Constraint Logic. 35–37, 43, 44, 73, 234, 235
CMA Class Memory Automaton. 27, 29, 30
CTL Computational Tree Logic. 115, 122
DA Data Automaton. 25–31, 38, 45, 49, 52, 53, 65–67, 69, 74, 96–99, 101, 106, 109, 172, 173
  BDA Büchi Data Automaton. 26, 27, 31, 38, 57, 69, 70, 109
  CDA Commutative Data Automaton. 109
  EDA Extended Data Automaton. 28–30
  NDA Nested Data Automaton. 74
    PNDA Locally Prefix-Closed Nested Data Automaton. 74
    SNDA Locally Suffix-Closed Nested Data Automaton. 74
  PDA Locally Prefix-closed Data Automaton. 74
  SDA Locally Suffix-closed Data Automaton. 74
  TDA Transparent Data Automaton. 27–30
  WBDA Weak Büchi Data Automaton. 46, 108, 109
  WDA Weak Data Automaton. 45, 46, 96–99, 101, 106–109
DMPA Data Multi-Pushdown Automaton. 118, 121, 122, 207, 208, 212
DNL Data Navigation Logic. 65, 69, 71–74, 113, 171, 208
  B-DLTL Basic Data LTL. 48, 74, 121, 172, 207, 208
    B-DLTL^{+} Class Future Basic Data LTL. 74
    B-DLTL^{-} Class Past Basic Data LTL. 74
    N-DLTL Nested Data LTL. 74
    N-DLTL^{+} Class Future Nested Data LTL. 74
    N-DLTL^{-} Class Past Nested Data LTL. 74
    RB-DLTL Restricted Basic Data LTL. 172–174, 176–179, 186, 207, 208
  B-DNL Basic Data Navigation Logic. 48–54, 57–59, 61–66, 69, 72–74, 121, 172, 207, 235
  X-DNL Extended Data Navigation Logic. 65, 66, 69, 73, 74
DWA Data Walking Automaton. 28–30
HTL Hybrid Temporal Logic. 122, 195, 196, 198, 199
HTL$^{\sim}$ Hybrid Temporal Logic on Data Words. 45, 75–80, 82–93, 121, 122, 185, 195, 196, 198, 199, 207, 208, 212, 237
LLT Letter-to-Letter Transducer. 18, 25, 53, 96
  BLLT Büchi Letter-To-Letter Transducer. 18, 26, 58
LRV Logic of Repeating Values. 36, 37, 235
  PLRV Logic of Repeating Values with Past. 36, 37, 43, 44, 47, 71, 73, 235

- **LTL** Linear-Time Temporal Logic. 1, 29, 32, 43–45, 47, 48, 74–76, 86, 87, 89, 115, 116, 120, 122, 185, 193, 194, 197, 198
  - **PLTL** Linear Temporal Logic with Past Operators. 29, 71, 82, 86, 88, 89, 197, 208
- **LTL**$^{\Downarrow}$ Linear-Time Temporal Freeze Logic. 32, 33, 35–38, 43–45, 61, 64, 71, 72, 75–79, 82–93, 116, 121, 122, 172, 182, 185, 186, 188, 192, 193, 195, 207, 208, 211, 212, 232, 234, 237
- **MCM** Minsky Counter Machine. 16, 17, 59, 61, 64, 116, 136, 182
  - **MCA** Multicounter Automaton. 17, 31, 172, 173, 179
- **MNL** MSC Navigation Logic. 122, 171, 199, 200, 202, 206–208, 237, 238
- **MSC** Message Sequence Chart. 116–118, 120, 122–124, 148–155, 158–163, 171, 200–203, 206, 207, 209, 238
  - **BHMSC** Branching High-Level Message Sequence Chart. 122, 148, 150–152, 154–156, 158–164, 167–169, 171, 199, 200, 202, 203, 205–208, 212
  - **HMSC** High-Level Message Sequence Chart. 116–119, 122, 159, 199
- **MSO** Monadic Second Order Logic. 31, 99, 118, 121
  - **EMSO** Existential Monadic Second Order Logic. 31, 99, 106
  - **FO** First Order Logic. 29, 71, 76, 89–93, 101
- **MSO**$^{\sim}$ Monadic Second Order Logic on Data Words. 31, 118, 121, 207, 208, 212, 231
  - **E**$_{\infty}$**MSO**$^{\sim}$ Existential Monadic Second Order Logic on Data ω-Words. 108
  - **EMSO**$^{\sim}$ Existential Monadic Second Order Logic on Data Words. 31, 33, 37, 38, 45, 46, 95, 97, 99, 101, 106, 108, 109, 231, 232
  - **FO**$^{\sim}$ First Order Logic on Data Words. 30–33, 36–38, 44–47, 61, 70–73, 78, 79, 88, 95, 102, 108, 109, 116, 208, 231
- **NFA** Non-Deterministic Finite Automaton. 14, 15, 18, 25, 27, 28, 49, 69, 106, 107, 129, 134, 135, 147
  - **AFA** Alternating Finite Automaton. 15, 53
  - **AFA**$^{\leftrightarrow}$ Two-Way Alternating Finite Automaton. 15, 53
  - **BAFA** Alternating Finite Büchi Automaton. 15, 58
  - **BAFA**$^{\leftrightarrow}$ Two-Way Alternating Finite Büchi Automaton. 15, 58
  - **DFA** Deterministic Finite Automaton. 14, 51, 53
- **PA** Pebble Automaton. 28–31, 38
  - **TWPA** Top-View Weak Pebble Automaton. 28–30, 33, 38
  - **WPA** Weak Pebble Automaton. 28–30
- **PCP** Post's Correspondence Problem. 18, 19, 62
- **PDL** Propositional Dynamic Logic. 117
- **PRA** Process Register Automaton. 121, 122, 142–148, 151, 167–169, 171, 185, 186, 188, 190, 192–199, 207, 208, 212
- **PathLog** Two-Way Path Logic. 34–37, 43, 44, 72, 234
- **RA** Register Automaton. 23–25, 27–30, 35, 36, 38, 49, 51–53, 65, 69, 72, 95, 97–99, 108, 109
  - **ARA** Alternating Register Automaton. 28–30, 33, 38
  - **ARA**$^{\leftrightarrow}$ Two-Way Alternating Register Automaton. 33, 36, 38, 79, 80
  - **BARA** Alternating Büchi Register Automaton. 35, 38
  - **BARA**$^{\leftrightarrow}$ Two-Way Alternating Büchi Register Automaton. 25, 38
  - **BRA** Büchi Register Automaton. 57, 69, 70, 108, 109
  - **FRA** Fresh-Register Automaton. 27, 29, 30
  - **GRA** Guessing Register Automaton. 27–30
  - **HRA** History-Register Automaton. 27, 29, 30
  - **RA**$^{\leftarrow}$ Backward Register Automaton. 65–67, 69
  - **RA**$^{\leftrightarrow}$ Two-Way Register Automaton. 28–30
- **REM** Regular Expressions with Memory. 33–38, 44, 72, 233
  - **REME** Regular Expressions with Equality. 35–37

TLC Temporal Logic of Causalities. 117, 171, 199
TransProb Transduction Problem. 18, 129, 134, 135
VFA Variable Finite Automaton. 28–30
WSTS Well-Structured Transition System. 17, 18, 117, 136

Bibliography

[1] *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, IEEE Computer Society, 2006.

[2] P. A. Abdula, M. F. Atig, and O. Rezine, *Verification of directed acyclic ad hoc networks*, in FMOODS/FORTE, 2013, pp. 193–208.

[3] P. A. Abdulla, M. F. Atig, A. Kara, and O. Rezine, *Verification of dynamic register automata*, in 34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India, V. Raman and S. P. Suresh, eds., vol. 29 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014, pp. 653–665.

[4] P. A. Abdulla, M. F. Atig, A. Kara, and O. Rezine, *Verification of buffered dynamic register automata*, in NETYS 2015, May 11–13, Agadir, Morocco, Springer Berlin/Heidelberg, 2015.

[5] P. A. Abdulla, K. Cerans, B. Jonsson, and Y. Tsay, *General decidability theorems for infinite-state systems*, in Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996, IEEE Computer Society, 1996, pp. 313–321.

[6] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine, *Regular model checking without transducers (on efficient verification of parameterized systems)*, in Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings, O. Grumberg and M. Huth, eds., vol. 4424 of Lecture Notes in Computer Science, Springer, 2007, pp. 721–736.

[7] P. A. Abdulla and B. Jonsson, *Verifying programs with unreliable channels*, in Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993, IEEE Computer Society, 1993, pp. 160–170.

[8] P. A. Abdulla and B. Jonsson, *Model checking of systems with many identical timed processes*, Theor. Comput. Sci., 290 (2003), pp. 241–264.

[9] P. A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso, *Regular tree model checking*, in Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings, E. Brinksma and K. G. Larsen, eds., vol. 2404 of Lecture Notes in Computer Science, Springer, 2002, pp. 555–568.

[10] P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso, *Regular model checking made simple and efficient*, in Brim et al. [55], pp. 116–130.

[11] P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso, *Algorithmic improvements in regular model checking*, in Computer Aided Verification, 15th International Conference,

CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings, W. A. H. Jr. and F. Somenzi, eds., vol. 2725 of Lecture Notes in Computer Science, Springer, 2003, pp. 236–248.

[12] P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena, *A survey of regular model checking*, in CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings, P. Gardner and N. Yoshida, eds., vol. 3170 of Lecture Notes in Computer Science, Springer, 2004, pp. 35–48.

[13] P. A. Abdulla and M. Kindahl, *Decidability of simulation and bisimulation between lossy channel systems and finite state systems (extended abstract)*, in Lee and Smolka [144], pp. 333–347.

[14] P. A. Abdulla, A. Legay, J. d'Orso, and A. Rezine, *Tree regular model checking: A simulation-based approach*, J. Log. Algebr. Program., 69 (2006), pp. 93–121.

[15] B. Adsul, M. Mukund, K. N. Kumar, and V. Narayanan, *Causal closure for MSC languages*, in FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference, Hyderabad, India, December 15-18, 2005, Proceedings, R. Ramanujam and S. Sen, eds., vol. 3821 of Lecture Notes in Computer Science, Springer, 2005, pp. 335–347.

[16] S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. E. Nikoletseas, and W. Thomas, eds., *Automata, Languages and Programming, 36th Internatilonal Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part II*, vol. 5556 of Lecture Notes in Computer Science, Springer, 2009.

[17] R. Alur and P. Cerný, *Algorithmic verification of single-pass list processing programs*, CoRR, abs/1007.4958 (2010).

[18] R. Alur and P. Cerný, *Streaming transducers for algorithmic verification of single-pass list-processing programs*, in Ball and Sagiv [31], pp. 599–610.

[19] R. Alur, P. Cerný, and S. Weinstein, *Algorithmic analysis of array-accessing programs*, in Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings, E. Grädel and R. Kahle, eds., vol. 5771 of Lecture Notes in Computer Science, Springer, 2009, pp. 86–101.

[20] R. Alur, C. Courcoubetis, and D. L. Dill, *Model-checking for real-time systems*, in Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990, IEEE Computer Society, 1990, pp. 414–425.

[21] R. Alur and D. L. Dill, *Automata for modeling real-time systems*, in Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, July 16-20, 1990, Proceedings, M. Paterson, ed., vol. 443 of Lecture Notes in Computer Science, Springer, 1990, pp. 322–335.

[22] R. Alur, K. Etessami, and M. Yannakakis, *Realizability and verification of MSC graphs*, in Orejas et al. [171], pp. 797–808.

[23] R. Alur and T. A. Henzinger, *A really temporal logic*, in 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989, IEEE Computer Society, 1989, pp. 164–169.

[24] R. Alur, D. Peled, and W. Penczek, *Model-checking of causality properties*, in LICS, IEEE Computer Society, 1995, pp. 90–100.

[25] R. Alur and M. Yannakakis, *Model checking of message sequence charts*, in CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands,

August 24-27, 1999, Proceedings, J. C. M. Baeten and S. Mauw, eds., vol. 1664 of Lecture Notes in Computer Science, Springer, 1999, pp. 114–129.

[26] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta, *Computation in networks of passively mobile finite-state sensors*, in Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004, S. Chaudhuri and S. Kutten, eds., ACM, 2004, pp. 290–299.

[27] K. R. Apt and D. Kozen, *Limits for automatic verification of finite-state concurrent systems*, Inf. Process. Lett., 22 (1986), pp. 307–309.

[28] C. Areces, P. Blackburn, and M. Marx, *The computational complexity of hybrid temporal logics*, Logic Journal of the IGPL, 8 (2000), pp. 653–679.

[29] J. Aspnes and E. Ruppert, *An introduction to population protocols*, Bulletin of the EATCS, 93 (2007), pp. 98–117.

[30] C. Baier and J.-P. Katoen, *Principles of model checking*, MIT Press, 2008.

[31] T. Ball and M. Sagiv, eds., *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, ACM, 2011.

[32] V. Bárány, M. Bojańczyk, D. Figueira, and P. Parys, *Decidable classes of documents for xpath*, in D'Souza et al. [89], pp. 99–111.

[33] K. Baukus, Y. Lakhnech, and K. Stahl, *Verification of parameterized protocols*, J. UCS, 7 (2001), pp. 141–158.

[34] J. Beauquier, J. Clément, S. Messika, L. Rosaz, and B. Rozoy, *Self-stabilizing counting in mobile sensor networks*, in Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007, I. Gupta and R. Wattenhofer, eds., ACM, 2007, pp. 396–397.

[35] M. Benedikt, C. Ley, and G. Puppis, *Automata vs. logics on data words*, in CSL, A. Dawar and H. Veith, eds., vol. 6247 of Lecture Notes in Computer Science, Springer, 2010, pp. 110–124.

[36] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen, *Systems and software verification: model-checking techniques and tools*, Springer Science & Business Media, 2013.

[37] O. Bernholtz, M. Y. Vardi, and P. Wolper, *An automata-theoretic approach to branching-time model checking (extended abstract)*, in Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings, D. L. Dill, ed., vol. 818 of Lecture Notes in Computer Science, Springer, 1994, pp. 142–155.

[38] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, *Symbolic model checking without bdds*, in Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings, R. Cleaveland, ed., vol. 1579 of Lecture Notes in Computer Science, Springer, 1999, pp. 193–207.

[39] H. Björklund and T. Schwentick, *On notions of regularity for data languages*, Theor. Comput. Sci., 411 (2010), pp. 702–715.

[40] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin, *Two-variable logic on data trees and XML reasoning*, in PODS, S. Vansummeren, ed., ACM, 2006, pp. 10–19.

[41] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin, *Two-variable logic on data words*, ACM Trans. Comput. Log., 12 (2011), p. 27.

[42] M. Bojanczyk and S. Lasota, *An extension of data automata that captures XPath*, in LICS, IEEE Computer Society, 2010, pp. 243–252.

[43] M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David, *Two-variable logic on words with data*, in LICS [1], pp. 7–16.

[44] B. Bollig, *An automaton over data words that captures EMSO logic*, in CONCUR, J.-P. Katoen and B. König, eds., vol. 6901 of Lecture Notes in Computer Science, Springer, 2011, pp. 171–186.

[45] B. Bollig, A. Cyriac, P. Gastin, and K. N. Kumar, *Model checking languages of data words*, in FoSSaCS, L. Birkedal, ed., vol. 7213 of Lecture Notes in Computer Science, Springer, 2012, pp. 391–405.

[46] B. Bollig, A. Cyriac, L. Hélouët, A. Kara, and T. Schwentick, *Dynamic communicating automata and branching high-level MSCs*, in Language and Automata Theory and Applications - 7th International Conference, LATA 2013, Bilbao, Spain, April 2-5, 2013. Proceedings, A. H. Dediu, C. Martín-Vide, and B. Truthe, eds., vol. 7810 of Lecture Notes in Computer Science, Springer, 2013, pp. 177–189.

[47] B. Bollig and L. Hélouët, *Realizability of dynamic MSC languages*, in CSR, F. M. Ablayev and E. W. Mayr, eds., vol. 6072 of Lecture Notes in Computer Science, Springer, 2010, pp. 48–59.

[48] B. Bollig, D. Kuske, and I. Meinecke, *Propositional dynamic logic for message-passing systems*, in FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science, 27th International Conference, New Delhi, India, December 12-14, 2007, Proceedings, V. Arvind and S. Prasad, eds., vol. 4855 of Lecture Notes in Computer Science, Springer, 2007, pp. 303–315.

[49] A. Bouajjani, J. Esparza, and O. Maler, *Reachability analysis of pushdown automata: Application to model-checking*, in Mazurkiewicz and Winkowski [161], pp. 135–150.

[50] A. Bouajjani, P. Habermehl, Y. Jurski, and M. Sighireanu, *Rewriting systems with data*, in Fundamentals of Computation Theory, 16th International Symposium, FCT 2007, Budapest, Hungary, August 27-30, 2007, Proceedings, E. Csuhaj-Varjú and Z. Ésik, eds., vol. 4639 of Lecture Notes in Computer Science, Springer, 2007, pp. 1–22.

[51] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar, *Abstract regular tree model checking*, Electr. Notes Theor. Comput. Sci., 149 (2006), pp. 37–48.

[52] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili, *Regular model checking*, in Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings, E. A. Emerson and A. P. Sistla, eds., vol. 1855 of Lecture Notes in Computer Science, Springer, 2000, pp. 403–418.

[53] L. Bozzelli and R. Lanotte, *Complexity and succinctness issues for linear-time hybrid logics*, Theor. Comput. Sci., 411 (2010), pp. 454–469.

[54] D. Brand and P. Zafiropulo, *On communicating finite-state machines*, J. ACM, 30 (1983), pp. 323–342.

[55] L. Brim, P. Jancar, M. Kretínský, and A. Kucera, eds., *CONCUR 2002 - Concurrency Theory, 13th International Conference, Brno, Czech Republic, August 20-23, 2002, Proceedings*, vol. 2421 of Lecture Notes in Computer Science, Springer, 2002.

[56] J. A. Brzozowski and E. L. Leiss, *On equations for regular languages, finite automata, and sequential networks*, Theor. Comput. Sci., 10 (1980), pp. 19–35.

[57] J. R. Büchi, *Weak second-order arithmetic and finite automata*, Z. Math. Logik Grundl. Math., 6 (1960), pp. 66–92.

[58] O. Burkart and B. Steffen, *Composition, decomposition and model checking of pushdown processes*, Nord. J. Comput., 2 (1995), pp. 89–125.

[59] S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen, *A succinct canonical register automaton model*, in ATVA, T. Bultan and P.-A. Hsiung, eds., vol. 6996 of Lecture Notes in Computer Science, Springer, 2011, pp. 366–380.

[60] K. Cerans, *Feasibility of finite and infinite paths in data dependent programs*, in Logical Foundations of Computer Science - Tver '92, Second International Symposium, Tver, Russia, July 20-24, 1992, Proceedings, A. Nerode and M. A. Taitslin, eds., vol. 620 of Lecture Notes in Computer Science, Springer, 1992, pp. 69–80.

[61] K. Cerans, *Deciding properties of integral relational automata*, in Automata, Languages and Programming, 21st International Colloquium, ICALP94, Jerusalem, Israel, July 11-14, 1994, Proceedings, S. Abiteboul and E. Shamir, eds., vol. 820 of Lecture Notes in Computer Science, Springer, 1994, pp. 35–46.

[62] A. K. Chandra, D. Kozen, and L. J. Stockmeyer, *Alternation*, J. ACM, 28 (1981), pp. 114–133.

[63] I. Chatzigiannakis, O. Michail, and P. G. Spirakis, *Mediated population protocols*, in Albers et al. [16], pp. 363–374.

[64] I. Chatzigiannakis, O. Michail, and P. G. Spirakis, *Recent advances in population protocols*, in Královic and Niwinski [137], pp. 56–76.

[65] E. Y. C. Cheng and M. Kaminski, *Context-free languages over infinite alphabets*, Acta Inf., 35 (1998), pp. 245–267.

[66] J. Clark, S. DeRose, et al., *XML path language (XPath) version 1.0*, 1999.

[67] E. M. Clarke and E. A. Emerson, *Design and synthesis of synchronization skeletons using branching-time temporal logic*, in Kozen [136], pp. 52–71.

[68] E. M. Clarke, E. A. Emerson, and A. P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Trans. Program. Lang. Syst., 8 (1986), pp. 244–263.

[69] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*, MIT Press, 2001.

[70] C. Cotton-Barratt, D. Hopkins, A. S. Murawski, and C. L. Ong, *Fragments of ML decidable by nested data class memory automata*, in Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, A. M. Pitts, ed., vol. 9034 of Lecture Notes in Computer Science, Springer, 2015, pp. 249–263.

[71] C. Cotton-Barratt, A. S. Murawski, and C. L. Ong, *Weak and nested class memory automata*, in Language and Automata Theory and Applications - 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings, A. H. Dediu, E. Formenti,

C. Martín-Vide, and B. Truthe, eds., vol. 8977 of Lecture Notes in Computer Science, Springer, 2015, pp. 188–199.

[72] C. David, *Mots et données infinies*, Master's thesis, LIAFA, (2004).

[73] C. David, L. Libkin, and T. Tan, *On the satisfiability of two-variable logic over data words*, in LPAR (Yogyakarta), 2010, pp. 248–262.

[74] N. Decker, P. Habermehl, M. Leucker, and D. Thoma, *Learning transparent data automata*, in Application and Theory of Petri Nets and Concurrency - 35th International Conference, PETRI NETS 2014, Tunis, Tunisia, June 23-27, 2014. Proceedings, G. Ciardo and E. Kindler, eds., vol. 8489 of Lecture Notes in Computer Science, Springer, 2014, pp. 130–149.

[75] N. Decker, P. Habermehl, M. Leucker, and D. Thoma, *Ordered navigation on multi-attributed data words*, in CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings, P. Baldan and D. Gorla, eds., vol. 8704 of Lecture Notes in Computer Science, Springer, 2014, pp. 497–511.

[76] G. Delzanno, A. Sangnier, R. Traverso, and G. Zavattaro, *On the complexity of parameterized reachability in reconfigurable broadcast networks*, in D'Souza et al. [89], pp. 289–300.

[77] G. Delzanno, A. Sangnier, and G. Zavattaro, *Parameterized verification of ad hoc networks*, in CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings, P. Gastin and F. Laroussinie, eds., vol. 6269 of Lecture Notes in Computer Science, Springer, 2010, pp. 313–327.

[78] G. Delzanno, A. Sangnier, and G. Zavattaro, *On the power of cliques in the parameterized verification of ad hoc networks*, in Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings, M. Hofmann, ed., vol. 6604 of Lecture Notes in Computer Science, Springer, 2011, pp. 441–455.

[79] S. Demri, D. D'Souza, and R. Gascon, *A decidable temporal logic of repeating values*, in LFCS, S. N. Artëmov and A. Nerode, eds., vol. 4514 of Lecture Notes in Computer Science, Springer, 2007, pp. 180–194.

[80] S. Demri, D. D'Souza, and R. Gascon, *Temporal logics of repeating values*, J. Log. Comput., 22 (2012), pp. 1059–1096.

[81] S. Demri, D. Figueira, and M. Praveen, *Reasoning about data repetitions with counter systems*, in LICS, IEEE Computer Society, 2013, pp. 33–42.

[82] S. Demri and R. Lazic, *LTL with the freeze quantifier and register automata*, in LICS [1], pp. 17–26.

[83] S. Demri and R. Lazic, *LTL with the freeze quantifier and register automata*, ACM Trans. Comput. Log., 10 (2009).

[84] S. Demri, R. Lazic, and D. Nowak, *On the freeze quantifier in constraint ltl: Decidability and complexity*, Inf. Comput., 205 (2007), pp. 2–24.

[85] S. Demri, R. Lazic, and A. Sangnier, *Model checking freeze LTL over one-counter automata*, in FoSSaCS, R. M. Amadio, ed., vol. 4962 of Lecture Notes in Computer Science, Springer, 2008, pp. 490–504.

[86] S. Demri, R. Lazic, and A. Sangnier, *Model checking memoryful linear-time logics over one-counter automata*, Theor. Comput. Sci., 411 (2010), pp. 2298–2316.

[87] S. Demri and A. Sangnier, *When model-checking freeze LTL over counter machines becomes decidable*, in FOSSACS, C.-H. L. Ong, ed., vol. 6014 of Lecture Notes in Computer Science, Springer, 2010, pp. 176–190.

[88] G. Ding, *Subgraphs and well-quasi-ordering*, Journal of Graph Theory, 16 (1992), pp. 489–502.

[89] D. D'Souza, T. Kavitha, and J. Radhakrishnan, eds., *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India*, vol. 18 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[90] H.-D. Ebbinghaus and J. Flum, *Finite Model Theory*, Springer, Heidelberg, 2005.

[91] C. Eisner and D. Fisman, *A Practical Introduction to PSL*, Series on Integrated Circuits and Systems, Springer, 2006.

[92] C. C. Elgot, *Decision problems of finite automata design and related arithmetics*, Transactions of The American Mathematical Society, 98 (1961), pp. 21–21.

[93] E. A. Emerson and V. Kahlon, *Reducing model checking of the many to the few*, in Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings, D. A. McAllester, ed., vol. 1831 of Lecture Notes in Computer Science, Springer, 2000, pp. 236–254.

[94] K. Etessami, M. Y. Vardi, and T. Wilke, *First-order logic with two variables and unary temporal logic*, Information and Computation, 179 (2002), pp. 279 – 295.

[95] D. Figueira, *Satisfiability of downward XPath with data equality tests*, in PODS, J. Paredaens and J. Su, eds., ACM, 2009, pp. 197–206.

[96] D. Figueira, *A decidable two-way logic on data words*, in LICS, IEEE Computer Society, 2011, pp. 365–374.

[97] D. Figueira and L. Segoufin, *Future-looking logics on data words and trees*, in Královic and Niwinski [137], pp. 331–343.

[98] A. Finkel and P. Schnoebelen, *Well-structured transition systems everywhere!*, Theor. Comput. Sci., 256 (2001), pp. 63–92.

[99] A. Finkel, B. Willems, and P. Wolper, *A direct symbolic approach to model checking pushdown systems*, Electr. Notes Theor. Comput. Sci., 9 (1997), pp. 27–37.

[100] M. Franceschet, M. de Rijke, and B.-H. Schlingloff, *Hybrid logics on linear structures: Expressivity and complexity*, in TIME, IEEE Computer Society, 2003, pp. 166–173.

[101] D. M. Gabbay, *The declarative past and imperative future: Executable temporal logic for interactive systems*, in Temporal Logic in Specification, 1987, pp. 409–448.

[102] D. M. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, *On the temporal basis of fairness*, in Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980, P. W. Abrahams, R. J. Lipton, and S. R. Bourne, eds., ACM Press, 1980, pp. 163–173.

[103] B. Genest, *Compositional message sequence charts (CMSCs) are better to implement than MSCs*, in Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on

Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings, N. Halbwachs and L. D. Zuck, eds., vol. 3440 of Lecture Notes in Computer Science, Springer, 2005, pp. 429–444.

[104] B. Genest, D. Kuske, and A. Muscholl, *A kleene theorem and model checking algorithms for existentially bounded communicating automata*, Inf. Comput., 204 (2006), pp. 920–956.

[105] B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun, *Infinite-state high-level MSCs: Model-checking and realizability*, in Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings, P. Widmayer, F. T. Ruiz, R. M. Bueno, M. Hennessy, S. Eidenbenz, and R. Conejo, eds., vol. 2380 of Lecture Notes in Computer Science, Springer, 2002, pp. 657–668.

[106] J. L. Gischer, *Shuffle languages, Petri nets, and context-sensitive grammars*, Commun. ACM, 24 (1981), pp. 597–605.

[107] E. Grädel and M. Otto, *On logics with two variables*, Theoretical Computer Science, 224 (1999), pp. 73–113.

[108] R. Grigore, D. Distefano, R. L. Petersen, and N. Tzevelekos, *Runtime verification based on register automata*, in Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, N. Piterman and S. A. Smolka, eds., vol. 7795 of Lecture Notes in Computer Science, Springer, 2013, pp. 260–276.

[109] O. Grumberg, O. Kupferman, and S. Sheinvald, *Variable automata over infinite alphabets*, in LATA, A. H. Dediu, H. Fernau, and C. Martín-Vide, eds., vol. 6031 of Lecture Notes in Computer Science, Springer, 2010, pp. 561–572.

[110] O. Grumberg, O. Kupferman, and S. Sheinvald, *Model checking systems and specifications with parameterized atomic propositions*, in Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings, S. Chakraborty and M. Mukund, eds., vol. 7561 of Lecture Notes in Computer Science, Springer, 2012, pp. 122–136.

[111] O. Grumberg, O. Kupferman, and S. Sheinvald, *An automata-theoretic approach to reasoning about parameterized systems and specifications*, in Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings, D. V. Hung and M. Ogawa, eds., vol. 8172 of Lecture Notes in Computer Science, Springer, 2013, pp. 397–411.

[112] R. Guerraoui and E. Ruppert, *Names trump malice: Tiny mobile agents can tolerate byzantine failures*, in Albers et al. [16], pp. 484–495.

[113] E. L. Gunter, A. Muscholl, and D. Peled, *Compositional message sequence charts*, in Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings, T. Margaria and W. Yi, eds., vol. 2031 of Lecture Notes in Computer Science, Springer, 2001, pp. 496–511.

[114] J. G. Henriksen, M. Mukund, K. N. Kumar, M. A. Sohoni, and P. S. Thiagarajan, *A theory of regular MSC languages*, Inf. Comput., 202 (2005), pp. 1–38.

[115] J. G. Henriksen and P. S. Thiagarajan, *A product version of dynamic linear time temporal logic*, in Mazurkiewicz and Winkowski [161], pp. 45–58.

[116] J. G. Henriksen and P. S. Thiagarajan, *Dynamic linear time temporal logic*, Ann. Pure Appl. Logic, 96 (1999), pp. 187–207.

[117] G. Higman, *Ordering by divisibility in abstract algebras*, Proceedings of the London Mathematical Society, (1952), pp. 326–336.

[118] J. Hromkovic, *Communication complexity and parallel computing*, Texts in theoretical computer science, Springer, 1997.

[119] ITU-TS, *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*, Geneva, 1997.

[120] P. Jančar, *Decidability of a temporal logic problem for Petri nets*, Theoretical Computer Science, 74 (1990), pp. 71–93.

[121] P. Jancar and F. Moller, *Checking regular properties of Petri nets*, in Lee and Smolka [144], pp. 348–362.

[122] B. Jonsson and M. Nilsson, *Transitive closures of regular relations for verifying infinite-state systems*, in Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings, S. Graf and M. I. Schwartzbach, eds., vol. 1785 of Lecture Notes in Computer Science, Springer, 2000, pp. 220–234.

[123] M. Jurdzinski and R. Lazic, *Alternating automata on data trees and XPath satisfiability*, ACM Trans. Comput. Log., 12 (2011), p. 19.

[124] M. Kaminski and N. Francez, *Finite-memory automata*, Theor. Comput. Sci., 134 (1994), pp. 329–363.

[125] M. Kaminski and T. Tan, *Regular expressions for languages over infinite alphabets*, Fundam. Inform., 69 (2006), pp. 301–318.

[126] M. Kaminski and T. Tan, *Tree automata over infinite alphabets*, in Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday, A. Avron, N. Dershowitz, and A. Rabinovich, eds., vol. 4800 of Lecture Notes in Computer Science, Springer, 2008, pp. 386–423.

[127] M. Kaminski and D. Zeitlin, *Extending finite-memory automata with non-deterministic reassignment (extended abstract)*, in Automata and Formal Languages, 12th International Conference, AFL 2008, Balatonfüred, Hungary, May 27-30, 2008, Proceedings., E. Csuhaj-Varjú and Z. Ésik, eds., 2008, pp. 195–207.

[128] H. Kamp, *Tense logic and the theory of linear order*, (1968).

[129] A. Kara and T. Schwentick, *Expressiveness of hybrid temporal logic on data words*, Electr. Notes Theor. Comput. Sci., 278 (2011), pp. 115–128.

[130] A. Kara, T. Schwentick, and T. Tan, *Feasible automata for two-variable logic with successor on data words*, in LATA, A. H. Dediu and C. Martín-Vide, eds., vol. 7183 of Lecture Notes in Computer Science, Springer, 2012, pp. 351–362.

[131] A. Kara, T. Schwentick, and T. Zeume, *Temporal logics on words with multiple data values*, in FSTTCS, K. Lodaya and M. Mahajan, eds., vol. 8 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010, pp. 481–492.

[132] R. M. Karp and R. E. Miller, *Parallel program schemata*, Journal of Computer and system Sciences, 3 (1969), pp. 147–195.

[133] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar, *Symbolic model checking with rich assertional languages*, in Computer Aided Verification, 9th International

Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings, O. Grumberg, ed., vol. 1254 of Lecture Notes in Computer Science, Springer, 1997, pp. 424–435.

[134] S. R. Kosaraju, *Decidability of reachability in vector addition systems*, in Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA, H. R. Lewis, B. B. Simons, W. A. Burkhard, and L. H. Landweber, eds., ACM, 1982, pp. 267–281.

[135] O. Kouchnarenko and P. Schnoebelen, *A model for recursive-parallel programs*, Electr. Notes Theor. Comput. Sci., 5 (1996), p. 30.

[136] D. Kozen, ed., *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, vol. 131 of Lecture Notes in Computer Science, Springer, 1982.

[137] R. Královic and D. Niwinski, eds., *Mathematical Foundations of Computer Science 2009, 34th International Symposium, MFCS 2009, Novy Smokovec, High Tatras, Slovakia, August 24-28, 2009. Proceedings*, vol. 5734 of Lecture Notes in Computer Science, Springer, 2009.

[138] O. Kupferman, N. Piterman, and M. Y. Vardi, *Extended temporal logic revisited*, in CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings, K. G. Larsen and M. Nielsen, eds., vol. 2154 of Lecture Notes in Computer Science, Springer, 2001, pp. 519–535.

[139] S. La Torre, P. Madhusudan, and G. Parlato, *A robust class of context-sensitive languages*, in 22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings, IEEE Computer Society, 2007, pp. 161–170.

[140] R. E. Ladner, R. J. Lipton, and L. J. Stockmeyer, *Alternating pushdown automata (preliminary report)*, in 19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978, IEEE Computer Society, 1978, pp. 92–106.

[141] F. Laroussinie and P. Schnoebelen, *A hierarchy of temporal logics with past*, Theor. Comput. Sci., 148 (1995), pp. 303–324.

[142] R. Lazic, *Safely freezing LTL*, in FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings, S. Arun-Kumar and N. Garg, eds., vol. 4337 of Lecture Notes in Computer Science, Springer, 2006, pp. 381–392.

[143] R. Lazic, *Safety alternating automata on data words*, ACM Trans. Comput. Log., 12 (2011), p. 10.

[144] I. Lee and S. A. Smolka, eds., *CONCUR '95: Concurrency Theory, 6th International Conference, Philadelphia, PA, USA, August 21-24, 1995, Proceedings*, vol. 962 of Lecture Notes in Computer Science, Springer, 1995.

[145] M. Leucker, P. Madhusudan, and S. Mukhopadhyay, *Dynamic message sequence charts*, in FSTTCS, M. Agrawal and A. Seth, eds., vol. 2556 of Lecture Notes in Computer Science, Springer, 2002, pp. 253–264.

[146] M. Leucker and C. Sánchez, *Regular linear temporal logic*, in Theoretical Aspects of Computing - ICTAC 2007, 4th International Colloquium, Macau, China, September 26-28, 2007, Proceedings, C. B. Jones, Z. Liu, and J. Woodcock, eds., vol. 4711 of Lecture Notes in Computer Science, Springer, 2007, pp. 291–305.

[147] L. Libkin and D. Vrgoc, *Regular expressions for data words*, in LPAR, N. Bjørner and A. Voronkov, eds., vol. 7180 of Lecture Notes in Computer Science, Springer, 2012, pp. 274–288.

[148] K. Lodaya and P. Weil, *Series-parallel languages and the bounded-width property*, Theor. Comput. Sci., 237 (2000), pp. 347–380.

[149] K. Lodaya and P. Weil, *Rationality in algebras with a series operation*, Inf. Comput., 171 (2001), pp. 269–293.

[150] M. Lohrey, *Safe realizability of high-level message sequence charts*, in Brim et al. [55], pp. 177–192.

[151] M. Lohrey, *Realizability of high-level message sequence charts: closing the gaps*, Theor. Comput. Sci., 309 (2003), pp. 529–554.

[152] N. A. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996.

[153] P. Madhusudan, *Reasoning about sequential and branching behaviours of message sequence graphs*, in Orejas et al. [171], pp. 809–820.

[154] P. Madhusudan and B. Meenakshi, *Beyond message sequence graphs*, in FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science, 21st Conference, Bangalore, India, December 13-15, 2001, Proceedings, R. Hariharan, M. Mukund, and V. Vinay, eds., vol. 2245 of Lecture Notes in Computer Science, Springer, 2001, pp. 256–267.

[155] Z. Manna and A. Pnueli, *The modal logic of programs*, in Automata, Languages and Programming, 6th Colloquium, Graz, Austria, July 16-20, 1979, Proceedings, H. A. Maurer, ed., vol. 71 of Lecture Notes in Computer Science, Springer, 1979, pp. 385–409.

[156] Z. Manna and A. Pnueli, *Verification of concurrent programs: Temporal proof principles*, in Kozen [136], pp. 200–252.

[157] Z. Manna and A. Pnueli, *The anchored version of the temporal framework*, in Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/-Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings, J. W. de Bakker, W. P. de Roever, and G. Rozenberg, eds., vol. 354 of Lecture Notes in Computer Science, Springer, 1988, pp. 201–284.

[158] A. Manuel, A. Muscholl, and G. Puppis, *Walking on data words*, in Computer Science - Theory and Applications - 8th International Computer Science Symposium in Russia, CSR 2013, Ekaterinburg, Russia, June 25-29, 2013. Proceedings, A. A. Bulatov and A. M. Shur, eds., vol. 7913 of Lecture Notes in Computer Science, Springer, 2013, pp. 64–75.

[159] N. Markey, *Temporal logic with past is exponentially more succinct, concurrency column*, Bulletin of the EATCS, 79 (2003), pp. 122–128.

[160] E. W. Mayr, *An algorithm for the general Petri net reachability problem*, SIAM J. Comput., 13 (1984), pp. 441–460.

[161] A. W. Mazurkiewicz and J. Winkowski, eds., *CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings*, vol. 1243 of Lecture Notes in Computer Science, Springer, 1997.

[162] M. L. Minsky, *Computation: finite and infinite machines*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[163] S. Miyano and T. Hayashi, *Alternating finite automata on omega-words*, Theor. Comput. Sci., 32 (1984), pp. 321–330.

[164] R. Morin, *Recognizable sets of message sequence charts*, in STACS 2002, 19th Annual Symposium on Theoretical Aspects of Computer Science, Antibes - Juan les Pins, France, March 14-16, 2002, Proceedings, H. Alt and A. Ferreira, eds., vol. 2285 of Lecture Notes in Computer Science, Springer, 2002, pp. 523–534.

[165] A. Muscholl, *Matching specifications for message sequence charts*, in Foundations of Software Science and Computation Structure, Second International Conference, FoSSaCS'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings, W. Thomas, ed., vol. 1578 of Lecture Notes in Computer Science, Springer, 1999, pp. 273–287.

[166] A. Muscholl and D. Peled, *Analyzing message sequence charts*, in SAM 2000, 2nd Workshop on SDL and MSC, Col de Porte, Grenoble, France, June 26-28, 2000, E. Sherratt, ed., VERIMAG, IRISA, SDL Forum, 2000, pp. 3–17.

[167] A. Muscholl and D. Peled, *Deciding properties of message sequence charts*, in Scenarios: Models, Transformations and Tools, International Workshop, Dagstuhl Castle, Germany, September 7-12, 2003, Revised Selected Papers, S. Leue and T. Systä, eds., vol. 3466 of Lecture Notes in Computer Science, Springer, 2003, pp. 43–65.

[168] A. Muscholl, D. Peled, and Z. Su, *Deciding properties for message sequence charts*, in Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings, M. Nivat, ed., vol. 1378 of Lecture Notes in Computer Science, Springer, 1998, pp. 226–242.

[169] F. Neven, T. Schwentick, and V. Vianu, *Finite state machines for strings over infinite alphabets*, ACM Trans. Comput. Log., 5 (2004), pp. 403–435.

[170] M. Niewerth and T. Schwentick, *Two-variable logic and key constraints on data words*, in ICDT, T. Milo, ed., ACM, 2011, pp. 138–149.

[171] F. Orejas, P. G. Spirakis, and J. van Leeuwen, eds., *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*, vol. 2076 of Lecture Notes in Computer Science, Springer, 2001.

[172] C. H. Papadimitriou, *Computational complexity*, Addison-Wesley, 1994.

[173] D. Peled, *Specification and verification of message sequence charts*, in Formal Techniques for Distributed System Development, FORTE/PSTV 2000, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX), October 10-13, 2000, Pisa, Italy, T. Bolognesi and D. Latella, eds., vol. 183 of IFIP Conference Proceedings, Kluwer, 2000, pp. 139–154.

[174] A. Pnueli, *The temporal logic of programs*, in Foundations of Computer Science, 1977., 18th Annual Symposium on, IEEE, 1977, pp. 46–57.

[175] E. L. Post, *A variant of a recursively unsolvable problem*, Bulletin of the American Mathematical Society, 52 (1946), pp. 264–269.

[176] A. Prior, *Past, Present, and Future*, Oxford University Press, 1967.

[177] M. O. Rabin and D. Scott, *Finite automata and their decision problems*, IBM journal of research and development, 3 (1959), pp. 114–125.

[178] B. Rossman, *On the constant-depth complexity of k-clique*, in STOC, C. Dwork, ed., ACM, 2008, pp. 721–730.

[179] E. Rudolph, P. Graubmann, and J. Grabowski, *Tutorial on message sequence charts*, Computer Networks and ISDN Systems, 28 (1996), pp. 1629–1641.

[180] H. Sakamoto and D. Ikeda, *Intractability of decision problems for finite-memory automata*, Theor. Comput. Sci., 231 (2000), pp. 297–308.

[181] C. Sánchez and M. Leucker, *Regular linear temporal logic with past*, in Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings, G. Barthe and M. V. Hermenegildo, eds., vol. 5944 of Lecture Notes in Computer Science, Springer, 2010, pp. 295–311.

[182] P. Schnoebelen, *The complexity of temporal logic model checking*, in Advances in Modal Logic 4, papers from the fourth conference on "Advances in Modal logic," held in Toulouse (France) in October 2002, P. Balbiani, N. Suzuki, F. Wolter, and M. Zakharyaschev, eds., King's College Publications, 2002, pp. 393–436.

[183] P. Schnoebelen, *Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets*, in Mathematical Foundations of Computer Science 2010, 35th International Symposium, MFCS 2010, Brno, Czech Republic, August 23-27, 2010. Proceedings, P. Hlinený and A. Kucera, eds., vol. 6281 of Lecture Notes in Computer Science, Springer, 2010, pp. 616–628.

[184] T. Schwentick and V. Weber, *Bounded-variable fragments of hybrid logics*, in STACS, W. Thomas and P. Weil, eds., vol. 4393 of Lecture Notes in Computer Science, Springer, 2007, pp. 561–572.

[185] J. C. Shepherdson, *The reduction of two-way automata to one-way automata*, IBM Journal of Research and Development, 3 (1959), pp. 198–200.

[186] A. Singh, C. R. Ramakrishnan, and S. A. Smolka, *A process calculus for mobile ad hoc networks*, in Coordination Models and Languages, 10th International Conference, COORDINATION 2008, Oslo, Norway, June 4-6, 2008. Proceedings, D. Lea and G. Zavattaro, eds., vol. 5052 of Lecture Notes in Computer Science, Springer, 2008, pp. 296–314.

[187] A. Singh, C. R. Ramakrishnan, and S. A. Smolka, *Query-based model checking of ad hoc network protocols*, in CONCUR 2009 - Concurrency Theory, 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings, M. Bravetti and G. Zavattaro, eds., vol. 5710 of Lecture Notes in Computer Science, Springer, 2009, pp. 603–619.

[188] A. P. Sistla and E. M. Clarke, *The complexity of propositional linear temporal logics*, J. ACM, 32 (1985), pp. 733–749.

[189] L. Stockmeyer, *The complexity of decision problems in automata and logic*, 1974. Ph.D. Thesis, MIT, 1974.

[190] Y. Sudo, J. Nakamura, Y. Yamauchi, F. Ooshita, H. Kakugawa, and T. Masuzawa, *Loosely-stabilizing leader election in population protocol model*, in Structural Information and Communication Complexity, 16th International Colloquium, SIROCCO 2009, Piran, Slovenia, May 25-27, 2009, Revised Selected Papers, S. Kutten and J. Zerovnik, eds., vol. 5869 of Lecture Notes in Computer Science, Springer, 2009, pp. 295–308.

[191] T. Tan, *Graph reachability and pebble automata over infinite alphabets*, in LICS, IEEE Computer Society, 2009, pp. 157–166.

[192] T. Tan, *On pebble automata for data languages with decidable emptiness problem*, J. Comput. Syst. Sci., 76 (2010), pp. 778–791.

[193] W. Thomas, *Automata on infinite objects*, in Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), 1990, pp. 133–192.

[194] W. Thomas, *Languages, automata, and logic*, in Handbook of Formal Languages, Vol. III, G. Rozenberg and A. Salomaa, eds., Springer, New York, 1997, pp. 389–455.

[195] B. Trakhtenbrot, *Finite automata and logic of monadic predicates*, Doklady Akademii Nauk SSSR, 140 (1961), pp. 326–329.

[196] N. Tzevelekos, *Fresh-register automata*, in Ball and Sagiv [31], pp. 295–306.

[197] N. Tzevelekos and R. Grigore, *History-register automata*, in Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, F. Pfenning, ed., vol. 7794 of Lecture Notes in Computer Science, Springer, 2013, pp. 17–33.

[198] M. Y. Vardi, *A temporal fixpoint calculus*, in Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988, J. Ferrante and P. Mager, eds., ACM Press, 1988, pp. 250–259.

[199] M. Y. Vardi and P. Wolper, *An automata-theoretic approach to automatic program verification (preliminary report)*, in LICS, IEEE Computer Society, 1986, pp. 332–344.

[200] V. Weber, *Branching-time logics repeatedly referring to states*, Journal of Logic, Language and Information, 18 (2009), pp. 593–624.

[201] P. Wolper and B. Boigelot, *Verifying systems with infinite but regular state spaces*, in Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings, A. J. Hu and M. Y. Vardi, eds., vol. 1427 of Lecture Notes in Computer Science, Springer, 1998, pp. 88–97.

[202] Z. Wu, *Commutative data automata*, in Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France, P. Cégielski and A. Durand, eds., vol. 16 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012, pp. 528–542.

Appendix A

Full Synax and Semantics of introduced Logics

In this appendix, we give the full syntax and semantics of the main logics mentioned in this work. All logics are defined over some set $\mathtt{Prop}$ of propositions and some $\mathtt{Att}$ of attributes.

A.1 First Order Logic on Data Words ($\mathrm{FO}^{\sim}$)

Syntax

Let $\mathtt{PV}$ be an infinite supply of position variables. The syntax of $\mathrm{FO}^{\sim}$ is defined as follows:

$$\varphi := \forall x\varphi \mid \exists x\varphi \mid \neg\varphi \mid \varphi \wedge \varphi \mid x = y \mid \mathtt{Suc}(x, y) \mid x < y \mid p(x) \mid x.\mathtt{@a}{\sim}y.\mathtt{@b} \mid \mathtt{Suc}_{\sim}(x.\mathtt{@a}, y.\mathtt{@b})$$

with $x \in \mathtt{PV}$, $p \in \mathtt{Prop}$ and $\mathtt{a}, \mathtt{b} \in \mathtt{Att}$.

The logic $\mathrm{MSO}^{\sim}$ extends $\mathrm{FO}^{\sim}$ by universal and existential set quantifications $\forall X$ and $\exists X$ and atomic formulas $X(x)$ where x is a position variable from $\mathtt{PV}$ and X is a set variable from an infinite supply $\mathtt{SV}$. We denote the fragment of $\mathrm{MSO}^{\sim}$ which consists of formulas of the type $\exists X_1 \ldots \exists X_n\varphi$ such that $n \geq 0$, every X_i with $i \in \{1, \ldots, n\}$ is a set variable and φ is an $\mathrm{MSO}^{\sim}$-formula which does not contain any set quantification, by $\mathrm{EMSO}^{\sim}$.

Semantics

We give the semantics of full $\mathrm{MSO}^{\sim}$. The satisfaction of an $\mathrm{MSO}^{\sim}$-formula is defined with respect to a data word, a partial mapping from $\mathtt{PV}$ to the set of positions of the word and a partial mapping from $\mathtt{SV}$ to the power set of the set of positions. Thus, let w be a data word and $\mu \in [\mathtt{PV} \rightharpoonup \mathtt{pos}(w)]$ and $\nu \in [\mathtt{SV} \rightharpoonup 2^{\mathtt{pos}(w)}]$ the respective mappings.

- $(w, \nu, \mu) \models \forall X\varphi$ if for all $S \in 2^{\mathtt{pos}(w)}$ it holds $(w, \nu[X \mapsto S], \mu) \models \varphi$
- $(w, \nu, \mu) \models \exists X\varphi$ if there exists $S \in 2^{\mathtt{pos}(w)}$ with $(w, \nu[X \mapsto S], \mu) \models \varphi$
- $(w, \nu, \mu) \models \forall x\varphi$ if $(w, \nu, \mu[x \mapsto i]) \models \varphi$ for all $i \in \mathtt{pos}(w)$
- $(w, \nu, \mu) \models \exists x\varphi$ if there exists $i \in \mathtt{pos}(w)$ with $(w, \nu, \mu[x \mapsto i]) \models \varphi$
- $(w, \nu, \mu) \models \neg\varphi$ if $(w, \nu, \mu) \not\models \varphi$
- $(w, \nu, \mu) \models \varphi \wedge \psi$ if $(w, \nu, \mu) \models \varphi$ and $(w, \nu, \mu) \models \psi$

- $(w, \nu, \mu) \models x = y$ if $\mu(x) = \mu(y)$
- $(w, \nu, \mu) \models \texttt{Suc}(x, y)$ if $\mu(y) = \mu(x+1)$
- $(w, \nu, \mu) \models x < y$ if $\mu(x) < \mu(y)$
- $(w, \nu, \mu) \models p(x)$ if $p \in \texttt{props}(w, \mu(x))$
- $(w, \nu, \mu) \models x.\texttt{@a} \sim y.\texttt{@b}$ if $\texttt{val}(w, \mu(x), \texttt{@a})$ and $\texttt{val}(w, \mu(y), \texttt{@b})$ are defined and equal
- $(w, \nu, \mu) \models \texttt{Suc}_\sim(x.\texttt{@a}, y.\texttt{@b})$ if $x < y$, $\texttt{val}(w, \mu(x), \texttt{@a})$ and $\texttt{val}(w, \mu(y), \texttt{@b})$ are defined and equal and for every z with $x < z < y$, it does not hold $\texttt{val}(w, \mu(x), \texttt{@a}) = \texttt{val}(w, \mu(z), \texttt{@b})$
- $(w, \nu, \mu) \models X(x)$ if $\mu(x) \in \nu(X)$

We say that a data word w *satisfies* an EMSO$^\sim$-formula φ (written as $w \models \varphi$) if $(w, \texttt{SV}_{\mapsto\bot}, \texttt{PV}_{\mapsto\bot}) \models \varphi$.

A.2 Freeze LTL (LTL$^\Downarrow$)

Syntax

Given an infinite supply R of freeze registers, LTL$^\Downarrow$-formulas are constructed according the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \Downarrow^r_{\texttt{@}a}.\varphi \mid \Uparrow^r_{\texttt{@}a} \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi \mid \mathbf{X}^{\leftarrow}\varphi \mid \varphi\mathbf{U}^{\leftarrow}\varphi$$

with $r \in R$, $p \in \texttt{Prop}$ and $\texttt{a} \in \texttt{Att}$.

Semantics

A register mapping λ is a partial mapping from R to $\mathcal{D}$. An LTL$^\Downarrow$-formula is evaluated with respect to a data word w, a position $i \in \texttt{pos}(w)$ and a register mapping λ.

- $(w, i, \lambda) \models p$ if $p \in \texttt{props}(w, i)$
- $(w, i, \lambda) \models \neg\varphi$ if $(w, i, \lambda) \not\models \varphi$
- $(w, i, \lambda) \models \varphi_1 \wedge \varphi_2$ if $(w, i, \lambda) \models \varphi_1$ and $(w, i, \lambda) \models \varphi_2$
- $(w, i, \lambda) \models \Downarrow^r_{\texttt{@}a}.\varphi$ if $\texttt{val}(w, i, \texttt{@a})$ is defined and $(w, i, \lambda[r \mapsto \texttt{val}(w, i, \texttt{@a})]) \models \varphi$
- $(w, i, \lambda) \models \Uparrow^r_{\texttt{@}a}.\varphi$ if $\texttt{val}(w, i, \texttt{@a})$ is defined and $\lambda(r) = \texttt{val}(w, i, \texttt{@a})$
- $(w, i, \lambda) \models \mathbf{X}\varphi$ if $i + 1 \in \texttt{pos}(w)$ and $(w, i+1, \lambda) \models \varphi$
- $(w, i, \lambda) \models \varphi_1\mathbf{U}\varphi_2$ if there is a position $j \in \texttt{pos}(w)$ with $j \geq i$ such that $(w, j, \lambda) \models \varphi_2$ and $(w, k, \lambda) \models \varphi_1$ for all k with $i \leq k < j$
- $(w, i, \lambda) \models \mathbf{X}^{\leftarrow}\varphi$ if $i - 1 \geq 1$ and $(w, i-1, \lambda) \models \varphi$
- $(w, i, \lambda) \models \varphi_1\mathbf{U}^{\leftarrow}\varphi_2$ if there is a position $j \in \texttt{pos}(w)$ with $j \leq i$ such that $(w, j, \lambda) \models \varphi_2$ and $(w, k, \lambda) \models \varphi_1$ for all k with $j < k \leq i$

A formula φ is *satisfied* by a data word w (written as $w \models \varphi$) if $(w, 1, R_{\mapsto\bot}) \models \varphi$.

A.3 Regular Expressions with Memory (REM)

Syntax

First, we introduce register conditions. A register condition over a register set R is formulated according to the following grammar:

$$c := \top \mid \bot \mid \uparrow^{r}_{\texttt{@a}} \mid \neg c \mid c \wedge c$$

with $\texttt{a} \in \texttt{Att}$ and $r \in R$. Next, we define the grammar of REM-expressions over R:

$$\alpha := \emptyset \mid \varepsilon \mid p[c]\downarrow^{R'}_{\texttt{@a}} \mid \alpha \cdot \alpha \mid \alpha + \alpha \mid \alpha^{*}$$

where p is from $\texttt{Prop}$, c a register condition and R' a subset of R.

Semantics

Register conditions are evaluated with respect to an attribute-value mapping $v \in [\texttt{Att} \rightharpoonup \mathcal{D}]$ and a *register assignment* $\lambda \in [R \rightharpoonup \mathcal{D}]$:

- $(v, \lambda) \models \top$
- $(v, \lambda) \not\models \bot$
- $(v, \lambda) \models \uparrow^{r}_{\texttt{@a}}$ if $\lambda(r)$ and $v(\texttt{a})$ are defined and equal
- $(v, \lambda) \models \neg c$ if $(v, \lambda) \not\models c$
- $(v, \lambda) \models c_1 \wedge c_2$ if $(v, \lambda) \models c_1$ and $(v, \lambda) \models c_2$

A REM-expression α is evaluated with respect to a data word w and a "current" register assignment λ. The evaluation delivers a resulting register assignment λ' (written as $(w, \lambda) \overset{\lambda'}{\models} \alpha$).

- $\emptyset$ is not satisfied by any pair of a data word and a register assignment
- $(w, \lambda) \overset{\lambda'}{\models} \varepsilon$ if $w = \varepsilon$ and $\lambda' = \lambda$
- $(w, \lambda) \overset{\lambda'}{\models} p[c]\downarrow^{R'}_{\texttt{@a}}$ if $w = (P, v)$ consists of a single position with $p \in P$, $(v, \lambda) \models c$, $v(\texttt{a})$ is defined and $\lambda' = \lambda[R' \mapsto v(\texttt{a})]$
- $(w, \lambda) \overset{\lambda'}{\models} \alpha_1 \cdot \alpha_2$ if $w = w_1 w_2$ and there exists an assignment λ_1 such that $(w_1, \lambda) \overset{\lambda_1}{\models} \alpha_1$ and $(w_2, \lambda_1) \overset{\lambda'}{\models} \alpha_2$
- $(w, \lambda) \overset{\lambda'}{\models} \alpha_1 + \alpha_2$ if $(w, \lambda) \overset{\lambda'}{\models} \alpha_1$ or $(w, \lambda) \overset{\lambda'}{\models} \alpha_2$
- $(w, \lambda) \overset{\lambda'}{\models} \alpha^{*}$ if
  - $w = \varepsilon$ and $\lambda' = \lambda$ or
  - $w = w_1 w_2$ and there is some register mapping λ_1 such that $(w_1, \lambda) \overset{\lambda_1}{\models} \alpha$ and $(w_2, \lambda_1) \overset{\lambda'}{\models} \alpha^{*}$

A word w belongs to the language of some expression α if there is some register assignment λ' with $(w, R_{\mapsto \bot}) \overset{\lambda'}{\models} \alpha$.

A.4 Two-Way Path Logic (PathLog)

Syntax

Position formulas φ and path expressions α of PathLog are defined as follows:

$$\varphi := p \mid \texttt{@a}\langle \overleftarrow{\alpha} \sim \overrightarrow{\alpha} \rangle\texttt{@b} \mid \texttt{@a}\langle \overleftarrow{\alpha} \not\sim \overrightarrow{\alpha} \rangle\texttt{@b} \mid \neg\varphi \mid \varphi \wedge \varphi$$
$$\alpha := \varepsilon \mid [\varphi] \cdot \alpha$$

with $p \in \texttt{Prop}$ and $\texttt{a}, \texttt{b} \in \texttt{Att}$. The language PathLog consists of all position formulas.

Semantics

Satisfaction of a path expression α is defined with respect to a data word w and two positions $i, j \in \texttt{pos}(w)$. We distinguish between *future satisfaction* (denoted as $(w, i, j) \models_{\rightarrow} \alpha$) and *past satisfaction* (denoted as $(w, i, j) \models_{\leftarrow} \alpha$).

- $(w, i, j) \models_{\rightarrow} \varepsilon$ and $(w, i, j) \models_{\leftarrow} \varepsilon$ if $i = j$
- $(w, i, j) \models_{\rightarrow} [\varphi] \cdot \alpha$ if there exists k with $i \leq k \leq j$ such that $(w, k) \models \varphi$ and $(w, k, j) \models_{\rightarrow} \alpha$
- $(w, i, j) \models_{\leftarrow} [\varphi] \cdot \alpha$ if there exists k with $i \geq k \geq j$ such that $(w, k) \models \varphi$ and $(w, k, j) \models_{\leftarrow} \alpha$

Position formulas are evaluated with respect to a data word w and a position i.

- $(w, i) \models p$ if $p \in \texttt{props}(w, i)$
- $(w, i) \models \texttt{@a}\langle \overleftarrow{\alpha} \sim \overrightarrow{\beta} \rangle\texttt{@b}$ if there are $j \leq i$ and $k \geq i$ such that $(w, i, j) \models_{\leftarrow} \alpha$, $(w, i, k) \models_{\rightarrow} \beta$ and $\texttt{val}(w, j, \texttt{@a})$ and $\texttt{val}(w, k, \texttt{@b})$ are both defined and equal,
- $(w, i) \models \texttt{@a}\langle \overleftarrow{\alpha} \not\sim \overrightarrow{\beta} \rangle\texttt{@b}$ if there are $j \leq i$ and $k \geq i$ such that $(w, i, j) \models_{\leftarrow} \alpha$, $(w, i, k) \models_{\rightarrow} \beta$ and either one of $\texttt{val}(w, j, \texttt{@a})$ and $\texttt{val}(w, k, \texttt{@b})$ is not defined or they are not equal

Evaluation of formulas of the form $\neg\varphi$ and $\varphi_1 \wedge \varphi_2$ is defined as expected. A data word w satisfies a formula φ (written as $w \models \varphi$) if $(w, 1) \models \varphi$.

A.5 Constraint Logic (CLTL^{XF})

Syntax

$$\varphi := \texttt{@a} \sim \mathbf{X}^{\ell}\texttt{@b} \mid \texttt{@a} \sim \langle\rangle\texttt{@b} \mid \texttt{@a} \sim \langle\rangle^{\leftarrow}\texttt{@b} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi \mid \mathbf{X}^{\leftarrow}\varphi \mid \varphi\mathbf{U}^{\leftarrow}\varphi$$

with $\ell \in \mathbb{N}$ and $\texttt{a}, \texttt{b} \in \texttt{Att}$.

Semantics

A CLTL^{XF}-formula is evaluated with respect to a propositionless $\texttt{Att}$-complete data word w and a position $i \in \texttt{pos}(w)$. We leave out the cases for boolean and temporal operators. The latter are interpreted as in $\text{LTL}^{\Downarrow}$.

- $(w, i) \models \texttt{@a} \sim \mathbf{X}^{\ell}\texttt{@b}$ if $i + \ell \in \texttt{pos}(w)$ and $\texttt{val}(w, i, \texttt{@a}) = \texttt{val}(w, i + \ell, \texttt{@b})$
- $(w, i) \models \texttt{@a} \sim \langle\rangle\texttt{@b}$ if there is some $j \in \texttt{pos}(w)$ with $i < j$ such that $\texttt{val}(w, i, \texttt{@a}) = \texttt{val}(w, j, \texttt{@b})$
- $(w, i) \models \texttt{@a} \sim \langle\rangle^{\leftarrow}\texttt{@b}$ if there is some j with $1 \leq j < i$ such that $\texttt{val}(w, i, \texttt{@a}) = \texttt{val}(w, j, \texttt{@b})$

A data word w satisfies a formula φ (written as $w \models \varphi$) if $(w, 1) \models \varphi$.

A.6 Logic of Repeating Values (LRV)

Syntax

We give the full syntax of PLRV. The fragment LRV results from this logic by skipping sub-formulas of the forms $\mathtt{@a} \sim \langle\varphi\rangle^{\leftarrow}\mathtt{@b}$ and $\mathtt{@a} \not\sim \langle\varphi\rangle^{\leftarrow}\mathtt{@b}$.

$$\varphi := \mathtt{@a} \sim \mathbf{X}^{\ell}\mathtt{@b} \mid \mathtt{@a} \sim \langle\varphi\rangle\mathtt{@b} \mid \mathtt{@a} \not\sim \langle\varphi\rangle\mathtt{@b} \mid \mathtt{@a} \sim \langle\varphi\rangle^{\leftarrow}\mathtt{@b} \mid \mathtt{@a} \not\sim \langle\varphi\rangle^{\leftarrow}\mathtt{@b} \mid$$

$$\neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi \mid \mathbf{X}^{\leftarrow}\varphi \mid \varphi\mathbf{U}^{\leftarrow}\varphi$$

where $\ell \in \mathbb{N}$ and $\mathtt{a}, \mathtt{b} \in \mathtt{Att}$.

Semantics

A PLRV-formula is evaluated with respect to an $\mathtt{Att}$-complete data word w and a position $i \in \mathtt{pos}(w)$. We only give the cases for sub-formulas not contained in $\mathrm{CLTL}^{\mathtt{XF}}$.

- $(w, i) \models \mathtt{@a} \sim \langle\varphi\rangle\mathtt{@b}$ if there is some $j \in \mathtt{pos}(w)$ with $i < j$ such that $\mathtt{val}(w, i, \mathtt{@a}) = \mathtt{val}(w, j, \mathtt{@b})$ and $(w, j) \models \varphi$
- $(w, i) \models \mathtt{@a} \not\sim \langle\varphi\rangle\mathtt{@b}$ if there is some $j \in \mathtt{pos}(w)$ with $i < j$ such that $\mathtt{val}(w, i, \mathtt{@a}) \neq \mathtt{val}(w, j, \mathtt{@b})$ and $(w, j) \models \varphi$
- $(w, i) \models \mathtt{@a} \sim \langle\varphi\rangle^{\leftarrow}\mathtt{@b}$ if there is some j with $1 \leq j < i$ such that $\mathtt{val}(w, i, \mathtt{@a}) = \mathtt{val}(w, j, \mathtt{@b})$ and $(w, j) \models \varphi$
- $(w, i) \models \mathtt{@a} \not\sim \langle\varphi\rangle^{\leftarrow}\mathtt{@b}$ if there is some j with $1 \leq j < i$ such that $\mathtt{val}(w, i, \mathtt{@a}) \neq \mathtt{val}(w, j, \mathtt{@b})$ and $(w, j) \models \varphi$

A data word w satisfies a formula φ (written as $w \models \varphi$) if $(w, 1) \models \varphi$.

A.7 Basic Data Navigation Logic (B-DNL)

Syntax

The syntax of global formulas φ and class formulas ψ is defined as follows:

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\rho\rangle\varphi \mid \langle\rho\rangle^{\leftarrow}\varphi \mid \mathcal{C}^{\ell}_{\mathtt{@a}}\psi$$

$$\psi := \varphi \mid \neg\psi \mid \psi \wedge \psi \mid \langle\theta\rangle_{=}\psi \mid \langle\theta\rangle^{\leftarrow}_{=}\psi \mid \sim\mathtt{@a}$$

where $p \in \mathtt{Prop}$, $\mathtt{a} \in \mathtt{Att}$ and $\ell \in \mathbb{Z}$.
Next we give the syntax of global path expressions ρ and class path expressions θ.

$$\rho := \epsilon \mid \varphi \mid \rho \cdot \rho \mid \rho + \rho \mid \rho^{*}$$

$$\theta := \epsilon \mid \psi \mid \theta \cdot \theta \mid \theta + \theta \mid \theta^{*}$$

where φ and ψ are global and class formulas, respectively.
The logic B-DNL consists of the set of all global formulas.

Semantics

While global formulas are evaluated with respect to a data word w and a single position i, the evaluation of global path expressions depends on w and two positions i and j. For the evaluation of their class versions, in both cases we additional refer to some data value d. Moreover, for path expressions we distinguish between a future and a past satisfaction relation denoted by $\models$ and $\models_{past}$, respectively. Intuitively, $(w,i,j) \models_{past} \rho$ holds if $j \leq i$ and $w[j,\dots,i]$ matches ρ “backwards”. Note that the following semantic definition involves mutual recursion between the different types of formulas and expressions. We omit the boolean cases and start with global formulas.

- $(w,i) \models p$ if $p \in \texttt{props}(w,i)$
- $(w,i) \models \langle\rho\rangle\varphi$ if there is some position $j \in \texttt{pos}(w)$ with $j \geq i$ such that $(w,i,j) \models \rho$ and $(w,j) \models \varphi$
- $(w,i) \models \langle\rho\rangle^{\leftarrow}\varphi$ if there is some position $j \in \texttt{pos}(w)$ with $j \leq i$ such that $(w,i,j) \models_{past} \rho$ and $(w,j) \models \varphi$
- $(w,i) \models \mathcal{C}^{\ell}_{\texttt{@a}}\psi$ if $\texttt{val}(w,i,\texttt{@a}) = d$ for some data value d, $i+\ell \in \texttt{pos}(w)$ and $(w,i+\ell,d) \models \psi$

We proceed with class formulas.

- $(w,i,d) \models \varphi$ if $(w,i) \models \varphi$ for global formulas φ
- $(w,i,d) \models \langle\theta\rangle_{=}\psi$ if there is some position $j \in \texttt{clpos}(w,d)$ with $j \geq i$ such that $(w,i',j,d) \models \theta$ and $(w,j,d) \models \psi$ where i' is the minimal position in $\texttt{clpos}(w,d)$ with $i' \geq i$
- $(w,i,d) \models \langle\theta\rangle^{\leftarrow}_{=}\psi$ if there is some position $j \in \texttt{clpos}(w,d)$ with $j \leq i$ such that $(w,i',j,d) \models_{past} \theta$ and $(w,j,d) \models \psi$ where i' is the maximal position in $\texttt{clpos}(w,d)$ with $i' \leq i$
- $(w,i,d) \models \sim\texttt{@a}$ if $\texttt{val}(w,i,\texttt{@a}) = d$

We now turn towards global path expressions.

- $(w,i,j) \models \epsilon$ if $i = j$
- $(w,i,j) \models \varphi$ if $j = i+1$ and $(w,i) \models \varphi$
- $(w,i,j) \models \rho_1 \cdot \rho_2$ if there is some k with $i \leq k \leq j$, such that $(w,i,k) \models \rho_1$ and $(w,k,j) \models \rho_2$
- $(w,i,j) \models \rho_1 + \rho_2$ if $(w,i,j) \models \rho_1$ or $(w,i,j) \models \rho_2$
- $(w,i,j) \models \rho^*$ if $i = j$ or there is a sequence $i = i_0 \leq i_1 \leq \ldots \leq i_n = j$ such that $(w,i_k,i_{k+1}) \models \rho$ for every k with $0 \leq k < n$
- $(w,i,j) \models_{past} \varphi$ if $j = i-1$ and $(w,i) \models \varphi$
- $(w,i,j) \models_{past} \rho_1 \cdot \rho_2$ if there is some k with $i \geq k \geq j$, such that $(w,i,k) \models_{past} \rho_1$ and $(w,k,j) \models_{past} \rho_2$
- $(w,i,j) \models_{past} \rho^*$ if $i = j$ or there is a sequence $i = i_0 \geq i_1 \geq \ldots \geq i_n = j$ such that $(w,i_k,i_{k+1}) \models_{past} \rho$ for every k with $0 \leq k < n$

We conclude with class path expressions.

- $(w,i,j,d) \models \epsilon$ if $i = j$

- $(w, i, j, d) \models \psi$ if j is the immediate successor of i in $\texttt{clpos}(w, d)$ and $(w, i, d) \models \psi$
- $(w, i, j, d) \models \theta_1 \cdot \theta_2$ if if there is some $k \in \texttt{clpos}(w, d)$ with $i \leq k \leq j$ such that $(w, i, k, d) \models \theta_1$ and $(w, k, j, d) \models \theta_2$
- $(w, i, j, d) \models \theta^*$ if $i = j$ or there is a sequence $i = i_0 \leq i_1 \leq \ldots \leq i_n = j \in \texttt{clpos}(w, d)$ of d-class positions such that $(w, i_k, i_{k+1}, d) \models \theta$ for every k with $0 \leq k < n$
- $(w, i, j, d) \models_{past} \psi$ if j is the immediate predecessor of i in $\texttt{clpos}(w, d)$ and $(w, i, d) \models \psi$
- $(w, i, j, d) \models_{past} \theta_1 \cdot \theta_2$ if there is some $k \in \texttt{clpos}(w, d)$ with $i \geq k \geq j$ such that $(w, i, k, d) \models_{past} \theta_1$ and $(w, k, j, d) \models_{past} \theta_2$
- $(w, i, j, d) \models_{past} \theta^*$ if $i = j$ or there is a sequence $i = i_0 \geq i_1 \geq \ldots \geq i_n = j \in \texttt{clpos}(w, d)$ of d-class positions such that $(w, i_k, i_{k+1}, d) \models_{past} \theta$ for every k with $0 \leq k < n$

A data word w satisfies a formula φ (written as $w \models \varphi$) if $(w, 1) \models \varphi$.

A.8 Hybrid Temporal Logic on Data Words (HTL$^{\sim}$)

Syntax

Let $\texttt{PV}$ be an infinite supply of position variables.

$$\varphi ::= p \mid x \mid \varphi \wedge \varphi \mid \neg\varphi \mid {\downarrow^x}.\varphi \mid \texttt{@a}{\sim}x.\texttt{@b} \mid \texttt{on}(x).\varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi \mid \mathbf{X}^{\leftarrow}\varphi \mid \varphi\mathbf{U}^{\leftarrow}\varphi$$

where $p \in \texttt{Prop}$, $\texttt{a}, \texttt{b} \in \texttt{Att}$ and $x \in \texttt{PV}$.

Semantics

An HTL$^{\sim}$-formula is evaluated with respect to a data word w, a position i on w and a variable assignment $\mu \in [\texttt{PV} \rightharpoonup \texttt{pos}(w)]$. The evaluation of propositions and temporal operators is defined like in LTL$^{\Downarrow}$. We give the semantics for constructs not contained in LTL$^{\Downarrow}$:

- $(w, i, \mu) \models x$ if $\mu(x) = i$
- $(w, i, \mu) \models {\downarrow^x}.\varphi$ if $(w, i, \mu[x \mapsto i]) \models \varphi$
- $(w, i, \mu) \models \texttt{@a} \sim x.\texttt{@b}$ if $\texttt{val}(w, i, \texttt{@a})$ and $\texttt{val}(w, \mu(x), \texttt{@b})$ are defined and $\texttt{val}(w, i, \texttt{@a}) = \texttt{val}(w, \mu(x), \texttt{@b})$
- $(w, i, \mu) \models \texttt{on}(x).\varphi$ if $(w, \mu(x), \mu) \models \varphi$

A data word w satisfies an HTL$^{\sim}$-formula φ (denoted as $w \models \varphi$) if $(w, 1, \texttt{PV}_{\mapsto\perp}) \models \varphi$.

A.9 MSC Navigation Logic (MNL)

Syntax

Formulas of MNL over some message alphabet A are constructed according to the following grammar:

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}^{\downarrow}\varphi \mid \mathbf{X}^{\rightarrow}\varphi \mid \varphi\mathbf{U}^{\downarrow}\varphi \mid \mathbf{E}(\varphi\mathbf{U}\varphi) \mid \mathbf{A}(\varphi\mathbf{U}\varphi)$$

where $p \in \{\texttt{start}, \texttt{crt}\} \cup \{\texttt{snd}(m), \texttt{rec}(m) \mid m \in A\}$.

Semantics

Formulas of MNL are evaluated with respect to an MSC and an event. Thus, let $M = (E, \lhd, \lambda, \mu)$ with $\lhd = \lhd_{\texttt{proc}} \uplus \lhd_{\texttt{crt}} \uplus \lhd_{\texttt{msg}}$ be an MSC and $e \in E$ and event in M.

- $(M, e) \models p$ for some $p \in \{\texttt{start}, \texttt{crt}\} \cup \{\texttt{snd}(m), \texttt{rec}(m) \mid m \in A\}$ if λ maps e to a type corresponding to p and, in case of $p = \texttt{snd}(m)$ or $p = \texttt{rec}(m)$, the sent or received symbol is m
- $(M, e) \models \mathbf{X}^{\downarrow}\varphi$ if there is an event $e' \in E$ with $e \lhd_{\texttt{proc}} e'$ and $(M, e') \models \varphi$
- $(M, e) \models \mathbf{X}^{\rightarrow}\varphi$ if there is an event $e' \in E$ with $e \lhd_{\texttt{crt}} \cup \lhd_{\texttt{msg}} e'$ and $(M, e') \models \varphi$
- $(M, e) \models \varphi_1 \mathbf{U}^{\downarrow} \varphi_2$ if there is a sequence $e_1 \lhd_{\texttt{proc}} \ldots \lhd_{\texttt{proc}} e_n$ of events such that $e_1 = e$, $n \geq 1$, $(M, e_n) \models \varphi_2$ and $(M, e_i) \models \varphi_1$ for all i with $1 \leq i < n$
- $(M, e) \models \mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$ if there is a sequence $e_1 \lhd \ldots \lhd e_n$ of events such that $e_1 = e$, $n \geq 1$, $(M, e_n) \models \varphi_2$ and $(M, e_i) \models \varphi_1$ for all i with $1 \leq i < n$
- $(M, e) \models \mathbf{A}(\varphi_1 \mathbf{U} \varphi_2)$ if for all sequences $e_1 \lhd \ldots \lhd e_n$ of events such that $e_1 = e$ and $n \geq 1$, it holds $(M, e_n) \models \varphi_2$ and $(M, e_i) \models \varphi_1$ for all i with $1 \leq i < n$

For an MSC M and a formula φ we say that φ holds on M (written as $M \models \varphi$) if $(M, \texttt{init}(M)) \models \varphi$, i.e., φ holds at the first event of the initial process in M.