Developing Concepts and Methods for Module and Integration Tests for Models of Reactive Systems

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik

von

Yusufu Kyeyune

Dortmund

2000

Tag der mündlichen Prüfung: 03. Juli 2000

Dekan: Professor Dr. Bernd Reusch

Gutachter:
Professor Dr. Bernd Reusch
Professor Dr. Peter Marwedel

Acknowledgements

I would like to express my gratitude to Prof. Dr. Bernd Reusch for supervising my *thesis* as well as to Prof. Dr. Peter Marwedel. I would like to thank Prof. Dr.-Ing. Claudio Moraga for his valuable advice in various areas. I would also like to thank Dr. Hubert Wagner for the valuable discussions about temporal logic. In particular, I would like to thank Dr. Eike Riedemann and Dr. Karl-Heinz Temme, whose friendship has been a constant source of intellectual and moral encouragement.

My special thanks go to Mr. Dieter Ellerkmann for his continuous parental advice and financial support.

Last but not least, I would have been unable to complete my dissertation without the help and support of my wife Amina Betty Kyeyune Nalongo and the understanding which she and my seven children gave me.

I would like to dedicate this work to both my grandmothers the late Mwamiini and princess Namaalwa, who gave me so much love and insipiration. They helped me to become the person I am.

Preface

In his analysis of control-system problems, Neumann[1] states that *human errors* of *users* of computers play a smaller role than *system problems* including hardware and software due to design and implementation faults. The system-design process, he notes, is typically the source of many system flaws. Particularly annoying are those flaws that remain undetected because they are difficult and expensive to fix — *if they are detected in later phases of the development.*

Studies have shown that errors introduced in the specification stage of the system development are often more costly to correct (cf. [143], chapter 2). Boehm states that errors introduced during the requirements phase can cost up to 200 times more to correct than errors introduced later in the cycle (cf. [14]) and can have an impact on safety. Yet Gibbs [42] shows that 88% of all projects were totally redesigned. Today safety control systems are in operation. Increasingly more reactive systems (e. g. control systems for processes) including software and hardware will be developed in the future. Especially the software — if it is incomplete and faulty — would cause not only large-scale economic damage but could also endager human life (cf. [94]). Therefore techniques to provide the intended specifications and to find errors early are of great importance.

The aim of this thesis is to develop practical methods which can aid the system designer/tester by the detection and/or elimination of flaws of reactive systems.

Software reliability poses a remarkable problem for the development of large computer systems. Various methods have been developed to increase software reliability. Those which are based on visual specification gained a broad application due to their success. Pictures, diagrams, graphs and similar graphical representations are used to express control information, data structures, computer organisations and system behaviour in ways which are understandable and (more or less) precise.

State-transition-diagrams, a form of finite state machines, are a recognised and a very wide spread model for the description of system behaviour. Today, many of the present CASE-tools support formal specification with finite state machines. In the area of software verification, automata theory plays a big role due to its formal clarity (cf. [22, 37, 45]). In this research area, a lot of effort is being devoted to finding efficient algorithms or heuristics which, for instance, satisfy certain test criteria.

In this thesis, therefore, the language of *statecharts* which allows the specification of finite automata and their extension serves as a framework for the development of practical

[1] Peter Neumann, moderator of the internet risks forum, presents a comprehensive coverage of many different types of computer-related risks in [112].

methods. In particular, statecharts [50, 56] extend the conventional *finite state machines* (FSMs) with the notions of *hierarchy*, *orthogonality*, *broadcasting* and *history*. *Hierarchy*[2] is achieved by decomposing a state of a statechart into another statechart. *Orthogonality*[3] is supported by a synchronous operation of "statechart composition" permitting a broadcast-style of communication among the constituent statecharts. *History*[4] is a mechanism for allowing to 'remember' a previous visit to a state.

The language of statecharts has been used in the specification of a number of real-world systems. As such, it is implemented by various CASE-tools, e. g. *STATEMATE*[5], a graphical working environment for engineers involved in specifying, analysing and designing large and complex reactive systems (cf. [55]). STATEMATE has sold more than 2000 licenses worldwide which are used with roughly equal parity in the fields of aerospace, electronics, automobile and telecommunications. More recently, a number of CASE-tools supporting the object-oriented development employ statecharts for specifying the behaviour of object classes as state machines (cf. [52]).

Statecharts enable the description to be viewed at different levels of detail[6], and make even very large and complex specifications manageable and comprehensible. However, the *test problem/verification problem* remains difficult: (1) because the interaction between the components, (2) because testing a *complete* system, in many cases, leads to "space explosion" problem. An alternative is to divide the system test into two phases: (*i*) Perform detailed tests for individual components (*module test*). (*ii*) Then test the proper integration of an increasing number of components, adding one by one until all components are integrated. This, nevertheless, leads to *integration test* problems.

After discussions with various outstanding companies (in particular in the automobile industry) and software houses including institutes for software quality and reliability, it is fair to say that there is still a great problem to be solved in finding and applying software methods to overcome/eliminate space explosion problems. In particular, there are hardly any practical methods for *integration* tests. The methods developed here can be applied to large and complex models. In fact, the space explosion problems are reduced by generating test cases on different levels of the system (cf. [88]). This is established following one of the most fundamental notions of the software engineering techniques, namely *abstraction* (cf. [41]). Though the methods in [88] are developed on models of statecharts, they should however, with a little modification be applicable to other graph-based specifications. In other words, the aim here is to establish methods which are not restricted to statecharts. In fact, their application should be easily extendable to different specification languages.

In the following, we[7] give a detailed structure of this thesis:

1. In Chapter 1 we present an introduction to reactive systems and discuss the problems

[2] hierarchical models (modularity)

[3] corresponds to product automata (simultaneity)

[4] automata with memory

[5] trademark of i-logix, Inc. Burlington, MA.

[6] because hierarchical statecharts allow decomposition of components/states into subcomponents/substates

[7] In this thesis 'we' refers to I (the author).

in the specification and design of such large and complex systems.

2. In Chapter 2 we describe the basic mathematical notions that will be needed throughout the rest of this thesis.

3. The main objective of this thesis as well as the general procedure of system development and testing will be presented in Chapter 3.

4. We devote Chapter 4 on one hand to present an overview of the STATEMATE tool (cf. section 4.1), and on the other hand to give an informal introduction to statecharts (cf. Section 4.2).

5. In Chapter 5 we develop a formal account of the syntax and semantics of statecharts. The establishment of such a formal statechart specification guideline contributes to the simplifications of the tests to be designed for the statechart model (design for testability).

6. Generally, the statechart behaviour is described in terms of *steps*, therefore, in Chapter 6 we deal with defining the behaviour precisely in terms of transitions "which execute a step", i. e., describing a step formally, with all its ramifications and side effects. We describe two basic models of the execution of a step, the *qualitative time model* (cf. Section 6.2) and the *qualitative time model* (cf. Section 6.3).

7. In Chapter 7 we address the test problems and establish practical methods which can aid the system designer/tester to detect and/or eliminate reactive system flaws. In particular, we provide a *module test approach* which can be applied to reduce the space explosion problems that have been discussed in the Preface[8].

8. The purpose of Chapter 8 is to establish concepts and methods which can aid the testing of the interacting processes (modules) of the whole system.

9. Chapter 9 summarises the main results as well as research contributions obtained in this thesis, and indicates open problems and further work that remains to be done.

10. Appendix A provides an overview of the subject of temporal logics. Our main purposes in doing so are to enable the reader

    (a) to become acquainted with the basic ideas of this formal language, and

    (b) evaluate the advantages of applying temporal logics to program reasoning, in particular to concurrent programs

    Moreover, the contribution of this Appendix can be seen as a preparation for further work dealing with analysis techniques for reactive programs, in particular, for the development of methods for the analysis of *time dependent* behaviour as well as *internal* variables (cf. Section 8.1.3). Two major classes of techniques, *proof-theoretic reasoning* and *model-theoretic reasoning* can be used for formal reasoning about concurrent systems.

[8] of this thesis

11. The main purpose of Appendix B is to establish the full theoretical background of reactive models, and especially to classify statecharts in terms of *nondeterminism* and *pure parallelism* and *bounded cooperative concurrency*, and above all, to remain as close as possible to classical finite automata. Furthermore, this decision provides us with background information for the comparisons of our concepts developed in Subsection 7.7.3 with other related techniques discussed in Subsection 7.6.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 What are reactive systems?

Generally, systems can be divided conceptually into transformational and reactive systems. Figure 1.1 shows a transformational and a reactive system. Transformational systems are those invoked when the inputs are ready. The outputs are produced after a computation period. Figure 1.1 illustrates, on the other hand, that reactive systems are systems in which inputs are not ready at a specific point of time. A typical reactive system is a traffic-light controller — the inputs arrive in endless and perhaps unexpected sequences.

Many industrial systems can be identified as “reactive”. In a landmark paper [125] Pnueli characterized reactive systems as being to a large extent event-driven, continuously having to react to external and internal stimuli. Reactive systems thus subsume many programs labelled as concurrent, parallel, or distributed, as well as control programs. Typical examples of reactive systems include control and communication systems, computer operating systems, networks, telephones, automobiles, VLSI circuits, industrial robots, interactive software and *real-time,* embedded systems. Real-time systems are usually used for critical operations, e.g., patient monitoring systems, aircraft, process control, and have very strict requirements. Real-time systems are also referred to as mission critical (see [41]).

Figure 1.2 shows the classification of real systems. Since reactive systems are viewed as real systems, it is not possible to give a formal definition.

In [6] Bergè modifies Pnueli’s definition:

Definition 1.1 *A* **reactive system** *is one that is in continual interaction with its environment and executes at a pace determined by that environment.*

Because of the property of “continual interaction”, reactive systems have no distinguished “end state” but rather an “idle state” or “rest state” for every “loop” or *run.* Due to the property of “pace determined by that environment”, the correct operation of the reactive system requires a high performance in order to satisfy the conditions of the environment.

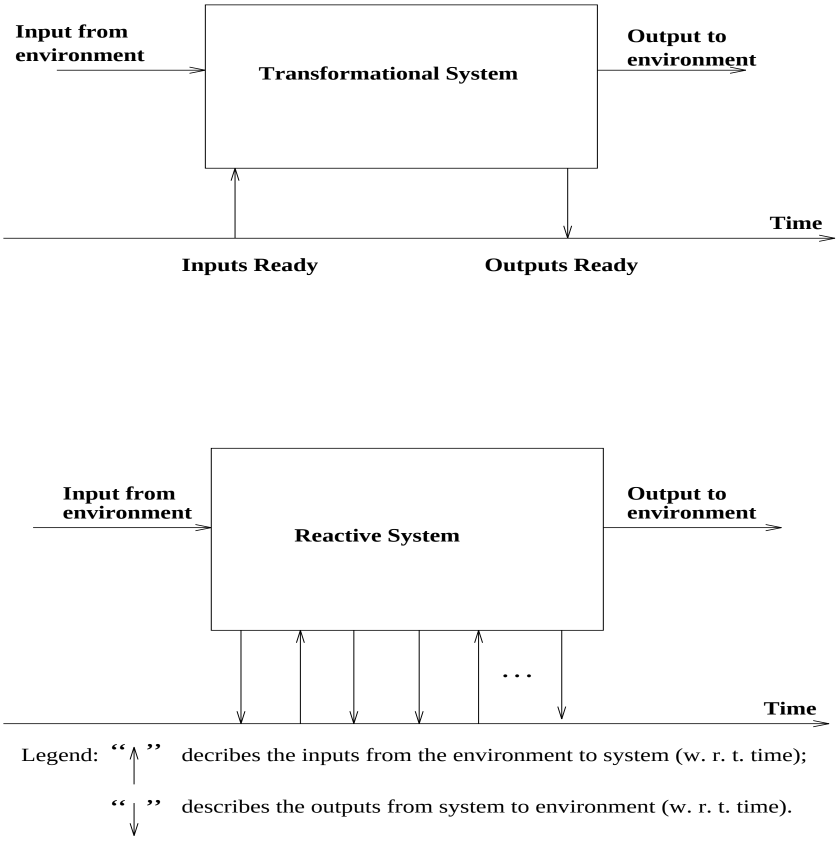


Figure 1.1: A transformational versus a reactive system.

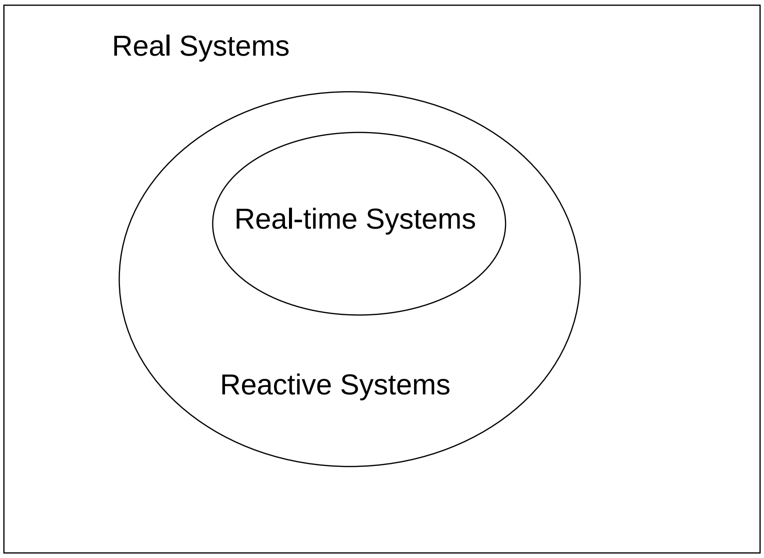


Figure 1.2: Classification of real systems

However, this issue is a very complex and wide ranging subject and therefore it cannot be fully handled in the scope of this thesis.

1.2 Motivation and aim of the approach

Clearly, for the development of large and complex reactive systems, very strict requirements have to be met.

1.2.1 The major problem of reactive systems

Over the last two decades, both industry and academic communities have carried out a lot of research dealing with finding good methods to aid in the development of reactive systems.

One of the major problems in the specification and design of large and complex reactive systems lies in the difficulty of describing the reactive *behaviour*[1] in ways which are realistic to model all relevant aspects of the real behaviour, and clear enough to enable a computerized analysis.

1.2.2 Proposed solutions

Several formal methods have been proposed to address the problem of specifying and modelling behaviour of reactive systems. Notable among the solutions are *Statecharts* [50, 56], *Lustre* [19], *Signal* [47], *ESTEREL* [2], *Petri nets* [139], *communicating sequential processing* [63], the *calculus of communicating systems* [105] and *Temporal logic* [126]. Most of these models provide automata based graphical languages. They are implemented in tools (e. g., logic proof systems, decision procedures and simulators) for verifying different aspects, such as safety properties of specified requirements. Due to the fact that reactive systems are usually large-scale and complex, the specification languages should provide useful features such as hierarchy, broadcasting communication, and ability to incorporate variables. Furthermore, as the verification of safety properties is very critical and complex, especially for real-time systems, the verification task should be performed decisively.

Generally, these formal models can be divided into three major classes:

(1) models based on state machines,

(2) models based on logic, and

(3) models that extend process algebra.

1.2.2.1 Models based on state machines

Finite state machines (FSMs) and their corresponding state transition diagrams (or in short ‘state diagrams’) are the most popular model for describing the dynamic behaviour

[1] The behaviour of a reactive system can be considered the set of input and output events, conditions, actions, perhaps with some additional information such as timing constraints.

of reactive systems since the temporal behaviour of such systems is most naturally represented in form of states and transitions between states (cf. [64, 66]). State diagrams are simply directed graphs with nodes denoting states and arrows (labelled with triggering events and guarding conditions) denoting transitions. From any state, an input language token (event or condition) initiates a transition labelled with that token. A transition may be associated with an output token which provides output to the user or a processing action that is performed by the system during that transition (this is the formalism of the Mealy machine).

However, it should be pointed out that conventional state diagrams are inappropriate for the behavioural description of complex control due to the following problems:

(i) State diagrams do not explicitly support concurrency. Without explicit support for concurrency, a complex system will precipitate an explosion in the number of states.

(ii) State diagrams are "flat" and unstructured. The lack of hierarchy can cause an extreme increase in the number of arcs.

Statecharts

Statecharts were proposed by D. Harel [50] to overcome the limitations of state-transition diagrams that are flat and unstructured while preserving their visual nature. In particular, statecharts [50, 56] extend the conventional FSMs with the notions of *hierarchy*, *orthogonality*, *broadcasting* and *history*. *Hierarchy*[2] is achieved by decomposing a state of a statechart into another statechart. *Orthogonality*[3] is supported by a synchronous operation of "statechart composition" permitting a broadcast-style of communication among the constituent statecharts. *History*[4] is a mechanism to 'remember' a previous visit to a substate of a state. These features make statecharts a very powerful language to specify the behaviour of complex reactive systems. Statecharts are well accepted in industrial applications. They are implemented in commercial tools, e. g., STATEMATE and *RHAPSODY*[5].

Following the STATEMATE specification, the behaviour is described as a set of possible *runs*, where a run consists of a series of detailed *snapshots* of a system's situation and *steps* which lead from one *status* to the next *status*; such a snapshot is known as a *status*.

The first in the sequence is the *initial status*, and each following one is determined from its predecessor by executing a *step*. Figure 1.3 shows a run of a System Under Development (SUD).

Informally speaking, a **step** is a unit of dynamic behaviour, at the beginning and end of which a SUD is in some 'legal' *status*.

[2] hierarchical models (modularity)

[3] corresponds to product automata (simultaneity)

[4] automata with memory

[5] both STATEMATE and RHAPSODY are trademarks of i-logix, Inc.

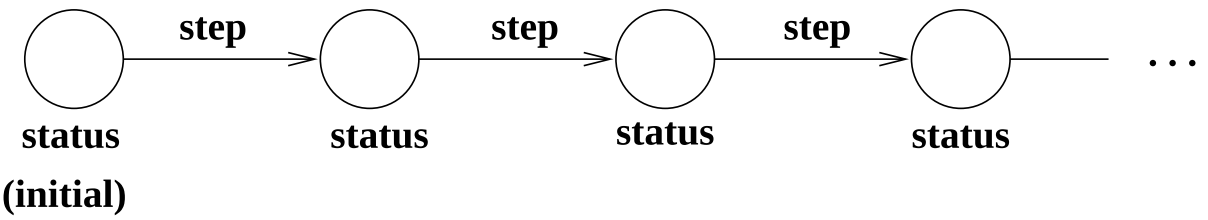


Figure 1.3: A run composed of a sequence of steps and statuses

A **status** consists of the information about active states and activities, values of data items and conditions, generated events and scheduled actions, and some information regarding the system's history (its past behaviour).

During a step (cf. Subsection 6.2.1, Definition 6.6), the environment activities can generate external events, change truth values of conditions, update variables and other data items. Such changes will affect the status (cf. Subsection 6.2.1, Definition 6.1) of the system; they trigger state changes in the controllers (i.e., trigger transitions in statecharts) activate and deactivate activities, modify conditions and variables or other data items, etc.

According to [54], the execution of a step must always lead to a 'legal configuration' (cf. Subsection 5.4.1, Definition 5.25). Generally, it is easy to define the effect of a step involving a single statechart transition, described syntactically by an arc, or several 'non-conflicting' transitions in separate orthogonal components. However, in the design of real life complex systems there are a variety of nontrivial situations with 'conflicting' transitions.

Execution of a statechart model proceeds in steps. The semantics of an execution step is based on the *synchrony hypothesis.* The **perfect synchrony hypothesis** states that the output occurs simultaneously with the input that caused it[6]. The concept of perfect synchrony was defined by the developers of *Esterel* [2, 11]. In [71], this concept is described as a property of *responsiveness.* For some applications this hypothesis can cause paradoxes since in reality no reaction occurs simultaneously with its own cause. Therefore, this notion must be applied with care. In Esterel, e. g., these paradoxes are circumvented by *syntactically* forbidding situations in which they can arise (by a semantical (static) check[7] upon paradoxes). Nevertheless, this hypothesis is justified, because it only represents an abstract[8] notion of time. Indeed, there are many real-time systems whose response times are much shorter than the time intervals between the occurences of two successive input events. Above all, this hypothesis is important for high-level specification where one is not — yet — interested in dealing with the implementation details on one hand, but on the other hand would like to specify in an accurate, non-fuzzy way. During the implementation phase, a more realistic modelling of the actual implementation can be

[6] This hypothesis assumes that the system is faster than the environment and, hence, the response to an external stimulus (event) is always generated in the same step that the stimulus is introduced.

[7] of the compiler

[8] The perfect synchrony hypothesis is an abstraction that limits the interference that may occur in the time period separating the stimulus from the response and, thus, provides a guaranteed response which can be modelled as a discrete process.

done by introducing explicit delay elements if necessary.

The *operational semantics* of statecharts (as described in STATEMATE) is a maximal parallelism semantics inspired by the perfect synchrony hypothesis. Based on the report of Harel and Naamad in [54], section 9, there are two communication modes:

(1) The **synchronous time model** in which case the communication with the environment is performed after each basic step.

Generally, in the synchronous time model, the system reacts whenever it senses a unique event. In this way, it is possible that non-zero time passes between the time point in which an external event occurs and the time the system responds to it. In this particular case, assuming that the response requires several steps, each step is executed only when the unique event is sensed. As a result, the time required to respond depends on the number of steps needed to complete the system's response (but not when implemented in parallel), and as such, the hypothesis does not hold.

(2) The **asynchronous time model** where the communication is achieved by allowing several basic steps to take place within a single point in time.

Under this time model the perfect synchrony hypothesis holds.

In spite of the feature of orthogonality, it may be useful to retain the distinction between *cause* and *effect*. For this purpose the principle of *causality* is introduced. **Causality** means that for every event generated at a particular instant of time, there must be a (causal) chain of events leading backwards to the action that generated this event (see also [131]).

Another important property, which leads to a large discussion in the course of execution of a statechart is the concept of *instantaneous state*. This principle expresses that an **instantaneous state** can be entered and exited at the same instant of time. For example, when an action generated upon entrance also causes the state to be exited. Figure 1.4 shows an example of a statechart containing a (possibly) instantaneous state $S2$. Readers who are unfamiliar with statecharts descriptions are referred to Chapters 4 and 5. In Figure 1.4 and Figure 1.5, we give names to transition arcs, e. g., $t1$ and $t2$ in Figure 1.4. Note that these are not part of the syntax but are added solely to allow us a better exposition. If — due to event $e1$ — transition $t1$ is taken the internal event (action) $e2$ is generated and state $S2$ will become active. This issue concerns the question of whether transition $t2$ (by event $e2$) will be executed at the same time. In various published papers providing a formal statecharts semantics (cf. [50, 54]) instantaneous states are prohibited. In addition to addressing the problem of instantaneous states, the step semantic supports the asynchronous time model. With the asynchronous time model, external events and time progress can trigger an execution step only when no events generated internally can trigger the execution step.

The hypothetical statecharts variant presented in the survey of von der Beek [158], however, discusses two possibilities which appear reasonable if instantaneous states are allowed (as an example, consider Figure 1.5.):

(1) An event is not treated as an entity to be 'consumed' after triggering.

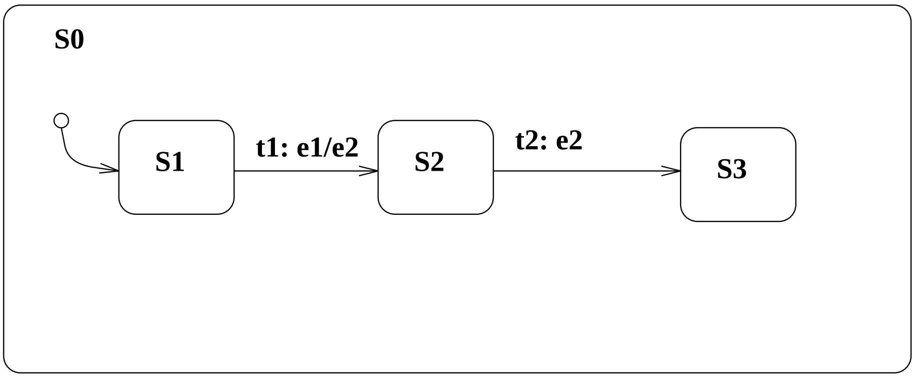


Figure 1.4: A statechart containing a (possibly) instantaneous state $S2$

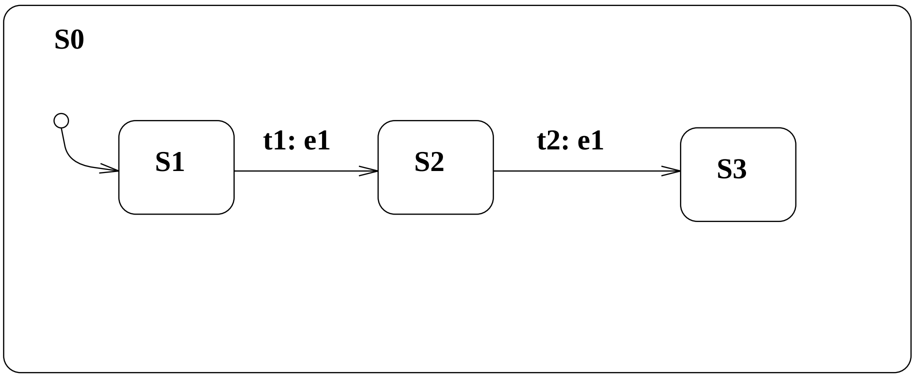


Figure 1.5: How many occurrences of event $e1$ are necessary to change from state $S1$ to state $S3$?

In this case one occurrence of event $e1$ (in Figure 1.5) is sufficient. The drawback of this possibility is that it allows infinite sequences.

(2) An event is treated as an entity to be 'consumed' after triggering.

In this case two occurrences of event $e1$ (in Figure 1.5) are necessary. This choice simplifies the modelling of a counter which has to distiguish between occurrences of the same event.

Nonetheless, von der Beek [158] proposes a third possibility of allowing a combination of the two possibilities. The idea is to determine for each event in the trigger whether it will be 'consumed' or not if it triggers the transition.

Lastly, it is important to discuss the **durability of events**. In many variants of statecharts an event has an instantaneous occurrence. This issue concerns the question of whether an event is durable for more than "an instant of time". Such a kind of event is often referred to as *discrete event*. In the semantics of [54], an event is sensed only in the step following the one in which it was generated. Even in case of asynchronous models when multiple steps of a so-called "super-step" execute at the same instant of time, an event generated in step x is sensed only in step $x + 1$.

Timed and Hybrid Statecharts [78] also use the discrete event approach. In addition, instantaneous states are allowed. Events persist as long as only untimed transitions are executed. That is to say, as long as time does not progress. Events are only consumed if a timed transition is executed.

Petri nets

The *Petri net* model [123, 140] is another type of state-based model, specifically proposed to model systems that comprise interacting concurrent tasks. A Petri net consists of a set of *places*, a set of *transitions*, and set of *tokens*. Single arrows connect places to transitions and transitions to places. A place is called an *input* of a transition (respectively, a transition called an *input* of a place) if there is an arrow leading from the place to the transition (respectively, from the transition to the place). Petri nets are an operational formalism. They support the notion of a state and its evolution. The states of a Petri net are represented graphically as a *marking* of its places, an assignment of a nonnegative number of tokens to each place. The behaviour of a Petri net is specified by the following rules:

(i) A transition is *enabled* in a given marking if all its input places are marked, i. e., each of its input places has at least one token.

(ii) An enabled transition may *fire*. If a transition fires one token is removed from each of its input places, and one token is deposited into each output place.

So far, we have described simple Petri nets, called *place/transition nets.* For a comprehensive study of Petri nets, see [123].

Petri nets are useful because they can effectively model a variety of characteristics. Petri nets can describe both asynchronous and nondeterministic behaviour. Petri nets can as well be used to check and validate certain useful system properties such as *safeness* and *liveness*. *Safeness*, e. g., is the property of Petri nets that guarantees that the number of tokens in the net will not grow indefinitely. *Liveness*, is on the other hand, the property of Petri nets that guarantees a dead-lock free operation, by ensuring that there is always at least one transition that can fire.

However, before Petri nets can be effectively used in the design of real-systems, some problems ought to be addressed, e. g.:

(i) Since Petri nets are designed to describe concurrency rather than the passage of time, they cannot express timeouts and durations.

(ii) Petri nets have limitations that are similar to those of an FSM; they can become incomprehensible with any increase in the number of places and transitions.

(iii) Petri nets only model a system's control features, not its data dependencies. This is due to the fact that tokens are "anonymous", and thus dependencies between data and control cannot be modelled.

To address these problems, a number of approaches, some with tool support, have been proposed. Some of the notable solutions are given in the following:

(1) Adding time to Petri nets: Among the most general extensions is the one which associates a minimum and a maximum firing time with each transition (cf. [123]). In this mechanism, when a transition is enabled, it cannot fire before the minimum time and it must fire at least after the maximum time has expired, unless it was previously disabled by the firing of a 'conflicting' transition.

(2) For modelling real systems, a number of abstraction and modularization techniques have been suggested, e. g., mechanisms that exploit the object-oriented paradigm (cf. [18]). See also special lecture slides "Structured Petri nets" from summer semester 1998 in [30].

(3) Associating tokens with values. In this approach the firing of transitions may depend on such values (see for example [40]).

It is fair to note that most significant properties of systems modelled with Petri nets are either intractable or undecidable (cf. [123]). This means that the analysis of most properties is carried out by applying simulation. This, however, can be time-consuming and unreliable. In [10], an algorithm known as the Berthomieu-Diaz algorithm, is given for analysing a timed Petri net for reachability. The idea behind the reachability problem is to determine whether a given marking m' can be reached from another marking m through a 'suitable' firing sequence. This problem has important applications to the analysis of Petri net properties since many other problems can be reduced to it.

1.2.2.2 Models based on logic

Most logics which are applied to reason about reactive systems, especially about concurrent programs, are either *first-order predicate logics* or *temporal logics*. Mathematical logic has been used for several years to specify and to prove properties about computer programs (cf. [62]). Using logic, one can describe and reason about the behaviour of a system without building the system first. Such reasoning is based on correctness proofs rather than experimental testing like simulation. In first-order predicate logic, as in other logics, formulas are constructed by combining variables, functions, predicates, and logical connectives according to the given rules. In order to meet the requirements of a critical system component, a variable t is usually included to represent time. The logical approach can be used to formalise basic system properties as *axioms* and to derive additional properties as *theorems* (consequences that follow from the basic assumptions).

Nevertheless, since first-order predicate theories are basic mathematical formalisms, they are not well suited for describing real-world complexities. Therefore, in real systems, logic is most useful for proving properties about critical system components rather than the whole system. Another problem of practical application of first-order predicate theories is that a number of general theories are undecidable, i. e., no algorithm exists that can determine whether a given property can be proven as a theorem. Even though the incorporated variable t can be used to represent time, Gabbay [38] illustrates drawbacks of adding an extra time variable to the language of predicate logic (cf. [38] pp. 20ff.). Often the constructed propositions of predicate logic are not readable. The main reason is that the described 'sentence' contains infinite time structures (for more details, see

"Introduction to temporal logic", in appendix Section A.1). Therefore, for the case of non-terminating or continuously operating concurrent programs such as operating systems and network protocols, we need *temporal logical* operators. The difference between the models of first-order predicate logics and temporal logics is that temporal logics uses special symbols to provide a simple and natural, but precise, way of describing the order in which interactions occur, without the adaption of absolute time measures. *Temporal logic* [126] is a formal specification language proposed by Pnueli for the description and analysis of time-dependent and behavioural aspects of reactive systems (cf. Appendix A).

1.2.2.3 Models that extend process algebra

Process algebras include languages like *Calculus of Communicating Systems CCS* [105, 106] and *Communicating sequential processing CSP* [63]. They were proposed to specify and analyse properties and constraints of concurrent systems without the notion of time.

In the last decade, several *timed* process algebras have been proposed. Typical examples are: ACSR, which adds time to CCS [12], and *Timed* CSP, a timed version of CSP [145].

1.2.3 Observations

Despite the great achievements made in the research area of reactive systems, it is fair to say that the problem of developing reactive systems has not yet been completely solved due to the complexity of the systems. For example, as far as real-time systems are concerned, tools such as STATEMATE and RHAPSODY support only simulation-based checking of safety properties, i. e., verification is performed heuristically and no decisive conclusion can be made. Thus, more efforts must be invested in developing practical methods, which can aid the system designer/tester by the detection and/or elimination of reactive system flaws.

1.2.4 Analysis

Many of the modern Computer-Aided Software Engineering (CASE) tools have the ability to check model "consistency" and "completeness". This could be seen as a form of syntax checking in a conventional program. But a mere test of syntactic integrity of the model has very little to do with a model's conceptual and logical aspects. Checking that a model is consistent and complete cannot prevent errors. Since reactive systems have a continuous interaction with their environment and often rely on a specific environment behaviour as a premise for their correct operation, it is necessary to carry out *real* testing and analysis.

1.2.5 Testing

Fortunately, a number of CASE tools, e.g., *STATEMATE* or *ProMod*[9] and *Stateflow,*[10] offer users to codify the specification for the design of a system early in the development process. Such tools often use graphical formalisms, simulations, and prototyping to enable the user to express ideas concisely and unambiguously. STATEMATE offers, in addition, a number of so called "static" tests, such as *reachability test*, *detection of nondeterminism*, *deadlock*, and *usage of transitions.* All of these tests are useful for checking general properties of the system.

Nevertheless, testing is usually carried out using the common, rather inefficient test methods. The specification of test cases based on the requirements specification is carried out on a largely intuitive and unsystematic basis. In other words, the system tester/designer has no information or even hints from the tool how to make a systematic test. Tool support is available for routine activities such as test execution, monitoring and test documentation (e. g. [46, 68]).

In fact, the CASE tools are not in a position to carry out sufficient tests for system models. Tests are offered under particular scenarios. This is often not enough for detection of all system flaws. The statecharts' analysis capability (e. g. offered by STATEMATE) is confined to using reachability analysis to check a small set of properties which is quite limited. It would be better if the analysis tool were in a position to simulate all possible combinations of events, conditions, and values of variables in order to make a global test e. g. for nondeterminism. Unfortunately, even for small statecharts models this leads to an unimaginable number of variations so that a practical application is not possible[11]. As an example, consider a behavioural model that contains about 40 concurrent components, each with about 10 states. This has 10^{40} state configurations in worst case. Therefore, a *test strategy* has to be established which allows to test a smaller number of executions than needed for exhaustive testing and above all, that can be justified to analyse the most *crucial* dynamic properties in the model. The term "crucial" here depends on the abstraction degree of observation and on the required test cases. This involves determining *necessary test criteria* like the ones defined in Section 7.3.

Finding efficient procedures for generating test paths of a specified system, modelled as a statechart, is an important software test problem. Through this process, errors are discovered and corrected, and we increase confidence in the result system. However, the fact is that there are hardly general heuristics to aid the system designer in carrying out efficient and systematic tests.

[9] an environment developed by a division of DaimlerChrysler in Germany for open, heterogeneous, networked systems

[10] an interactive design tool which has been recently developed for modelling and simulating complex reactive systems. With Stateflow, users can develop models of event-driven systems using finite state machines theory, statechart formalisms, and flow diagram notation.

[11] The number rises exponentially with the number of the input and output state variables.

1.2.6 Aim of the approach

The first main aim of this thesis is to address the problems mentioned above and develop a *module test* (cf. Chapter 7) which allows the system designer/tester to (automatically) test the partially designed and implemented system.

Since an automata equivalence test (cf. criterion $C3$, Definition 7.4 on page 121) requires at least a cubic number of tests (in the number of states or events), the **aim** of the *module test approach* is, therefore, to establish an *acceptable small set* of test paths that reaches all used states and carries out all used transitions in a component.

By applying this test method, we thus illustrate that the module test concepts can provide a more convincing approach to the problem testing on a systematic basis.

Yet another main problem when testing *hierarchical*, structured system models is the analysis of interaction between different components (*modules*). The most widespread methodologies isolate interacting concurrent processes one from another and concentrate on the testing of the individual modules. In [147], Ryant argues, that the danger here is that synchronization errors will hardly be detected. For example, system flaws that are due to time dependent behaviours of processes cannot be detected. In order to 'guarantee' a correct analysis, the whole system must be considered including all its interactions. However, the main drawback of most widespread methodologies such as *structured analysis* and *object oriented analysis* is that the errors which remain undetected are mainly those which are not analysed and tested during the interaction of the whole system. Since appropriate test concepts are not offered, the designer might get a wrong impression, i.e., to believe that the system works correctly.

Therefore, the second main aim of this approach is the development of concepts and methods which aid the testing of the *interacting* processes (modules) of the whole system.

Chapter 2

Mathematical Terms

In this chapter the basic mathematical notions that will be needed throughout the rest of this work is given. Obviously, the reader is expected to be familiar with most of the concepts mentioned here, and therefore a comprehensive treatment of them is not provided. Instead, the intention is to indicate briefly those ideas which will be required in some chapters, and to describe the notation that will be used in discussing them. Nonetheless to enable a more complete treatment of the notions presented here, some books are recommended (cf. [48, 49, 93]).

2.1 Sets, Relations

Let B and C be sets.
If $f : B \to C$ and $B' \subset B$ then

${}^{f}|_{B'}$ denotes the restriction of f to B', i.e.,
the function $f' : B' \to C$ where $\forall b \in B' : f'(b) = f(b)$.

$I\!N$ denotes the set of all natural numbers excluding 0.

$I\!N_0$ denotes the set of all natural numbers including 0.

$\sum\limits_{i \in I} b_i$ denotes the sum of all b_i, where $i \in I \subset I\!N_0$.

$\prod\limits_{i \in I} b_i$ denotes the product of all b_i, where $i \in I \subset I\!N_0$.

$|X|$ denotes the cardinality of a finite set X.

Let $\wp(X)$ denote the *power set* of a set X also written as 2^X, i.e., the collection of all subsets of X. It is well known that $|\wp(X)| = 2^{|X|}$.
A *binary relation* on X is defined to be a function $R : X \to \wp(X)$ from X to the power set of X (also $R \subset X \times X$).

2.2 Graphs

Definition 2.1 *A* **directed graph** *(a* **digraph** *for short) is a triple $G = (V, E, inc)$, where*

(1) V is a finite set of so called **vertices** *(Nodes),*

(2) E is a finite set of so called **edges** *and*

(3) $inc : E \rightarrow V \times V$.

Notation 2.1 *The vertex pair (v, w) of a digraph G associated with edge e is called an* **ordered** *pair (not necessarily distinct vertices). Edge e is then said to be* **directed** *from vertex v to vertex w, and the direction is indicated by an arrowhead on the edge.*

Definition 2.2 *A* **digraph** *$G = (V, E, inc)$ is said to have* **simple** *edges only if $inc : E \rightarrow V \times V$ is injective, i.e., $\forall e_i, e_j \in E : e_i \neq e_j \Rightarrow inc(e_i) \neq inc(e_j)$. Such a digraph is called a* **simple digraph.**
Note: *For simple digraphs the set of edges E may be represented by $inc(E) \subset V \times V$ since there is a one-to-one correspondence between edges and the corresponding pairs of vertices.*

Definition 2.3 *Let $X = \{x_1, \ldots, x_n\}$ be a finite set, then $BAG(X)$ denotes the set of all* **bags** *(***multisets**[1]*) over X. Bags are generalizations of sets, similar to multiple sets, i.e., an element of X may occur in a bag over X* **more than once**. *If b is a bag, i.e., $b \in BAG(X)$, then the function μ represents these occurrences, i.e., $\mu(x_i, b) = k \in \mathbb{N}_0$ iff there are exactly k occurrences of x_i in b.*

The following operations and relations are defined for bags b and c over X and $x_i \in X$ (cf. [119]).

Relations:
membership $x_i \in b$ **iff** x_i *appears in* b **iff** $\mu(x_i, b) > 0$**,**
bag equality $b = c$ **iff** $\forall_x \in X : \mu(x_i, b) = \mu(x_i, c)$**,** otherwise $b \neq c$
bag inclusion $b \leq c$ **iff** $\forall_x \in X : \mu(x_i, b) \leq \mu(x_i, c)$**,**
strict bag inclusion $b < c$ **iff** $b < c \wedge b \neq c$**.**

Operations:
bag intersection $b \cap c : \forall_x \in X : \mu(x_i, b \cap c) = min(\mu(x_i, b), \mu(x_i, c))$**,**
bag union $b \cup c : \forall_x \in X : \mu(x_i, b \cup c) = max(\mu(x_i, b), \mu(x_i, c))$**,**
bag sum $b + c : \forall_x \in X : \mu(x_i, b + c) = \mu(x_i, b) + \mu(x_i, c)$**,**
bag difference $b - c : \forall_x \in X : \mu(x_i, b - c) = \mu(x_i, b) - \mu(x_i, b \cap c)$**,**
empty bag $0 : \forall_x \in X : \mu(x_i, 0) = 0$**,**
where $0 \in \mathbb{N}_0$.

[1] Lengauer [93] gives the basic definition of **multisets**, see pp.47-48.

Definition 2.4 *A digraph $G = (V, E, inc)$ is said to have* **multiple** *edges when inc is not an injective mapping, i.e., there exist edges $e_i \neq e_j$ with $inc(e_i) = inc(e_j)$. Such a digraph is called a* **complex digraph.**
Note: *For a complex digraph the set of edges E may be represented by bags over $V \times V$, i.e., a set $\gamma \subset BAG(V \times V)$. For the bag γ and a pair $(v_1, v_2) \in V \times V$ we have $\mu((v_1, v_2), \gamma) = k$, i.e., k occurrences of (v_1, v_2) in γ, iff there are exactly k edges $e_1, \dots, e_k$ which are mapped onto (v_1, v_2) by inc.*

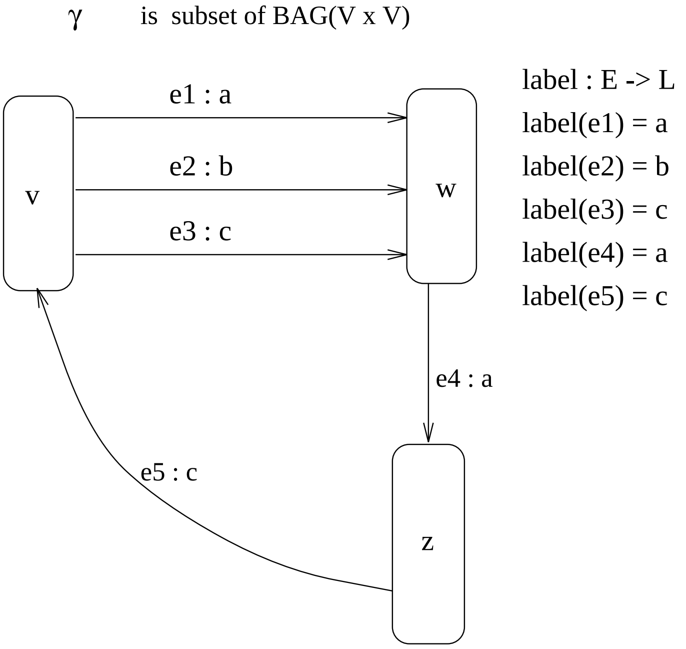


Figure 2.1: A graph with multiple edges.

Figure 2.1 illustrates a graph with multiple edges:

$$inc(e1) = (v, w)$$

$$inc(e2) = (v, w)$$

$$inc(e3) = (v, w)$$

Note that in the Figure 2.1 above, $\gamma \subset BAG(V \times V)$ leads to the following observation. The multiple edges between v and w are represented by the bag $3 \cdot (v, w)$, where $\mu((v, w), 3 \cdot (v, w)) = 3$.

A standard notation of a bag b over X will be $b = \sum_{x_i \in X} \mu(x_i, b) \cdot x_i$.

The bag for all edges of Figure 2.1 is as follows:

$$b = 3(v, w) + 1 \cdot (w, z) + 1 \cdot (z, v) = \sum_{x_i \in X} \mu(x_i, b) \cdot x_i, \text{ where}$$

$$X = \{(v, w), (w, z), (z, v)\} \subseteq V \times V.$$

A *labelled digraph* is a digraph in which each edge and/or each vertex has an associated label. A label can be a name, a cost, or a value of any given data type, including the empty label.

Definition 2.5 *A* **labelled digraph** $G = (V, E, inc, label)$ *is a digraph with a labelling function* $label : E \rightarrow L$, *where* L *is a finite set (of so called labels), with the following restriction:*

For all $e_i, e_j \in E$ *such that* $inc(e_i) = inc(e_j)$ *and* $e_i \neq e_j \Rightarrow label(e_i) \neq label(e_j)$.

In Figure 2.1, $label(e1) = a \neq label(e2) = b \neq label(e3) = c \neq label(e1)$. Indeed label$(e_i) = label(e_j)$ is only possible iff $inc(e_i) \neq inc(e_j)$ with $e_i, e_j \in E$.
That's why in Figure 2.1 $label(e4) = a$ and $label(e5) = c$ are legal labellings even though they are already used for edges $e1$ and $e3$, respectively.

Chapter 3

Tasks and Solutions for the Testing of Hierarchical Statechart Models

Almost all the methods currently used for testing large and complex reactive systems are experience based rather than theoretically founded methods. In particular, very few methods allow us to make an objective statement about the problem of generating sufficient test paths, e. g., for all used transitions and/or all used states in a statechart. We recall that STATEMATE offers a number of static tests, such as reachability test, detection of nondeterminism, deadlock, and usage of transitions. All of these tests are useful for checking general properties of the system.

However, as already observed in the Preface on page vi, there is still a great problem to be solved in finding and applying software methods to overcome/eliminate space explosion problems. In addition, *exhaustive testing*, i.e., testing every execution to prove the system's correctness, often cannot be performed even for models of moderate complexity.

It is important to keep in mind that even if we limit the values of variables to finite sets, the number of scenarios that have to be tested in an exhaustive execution rapidly becomes unmanageable (see example given in Subsection 1.2.5).

The second main problem in testing *hierarchical*, structured system models is the analysis of interaction between different components (*modules*). By perfoming a *module test*, it is namely not possible to detect the invisible interaction flaws between the modules. The development of concepts and methods which aid the testing of the interacting processes (modules) of the whole system (also called *integration test*) has therefore a high priority although it is a very complicated task due to the noted dependencies between modules.

3.1 The main task of testing hierarchical statechart models

In this section the main task of this thesis is given. The task of this thesis can be divided into three main parts:

(1) *Establish some kind of statechart standard specification guideline which formally defines and restricts syntax and semantics of variants of statecharts, and specification transformations which transform complex statecharts into suitable form for a profound testing* (*design for testability*).

(2) *To clarify the process of test execution and simulation of statecharts, the so called* **step semantics** *of statecharts has to be investigated and formalised, where a* **step** *is a unit of dynamic behaviour, at the beginning and end of which a statechart is in some 'legal'* status.

(3) *Development of test concepts and test methods for hierarchical, structured statecharts*

   (a) *Module tests*, a module is a component on some 'level' of the state hierarchy of the statechart.

   (b) *Integration tests* which are based on one of the well known heuristic approaches of practical testing which has two test phases:

      i. Perform detailed tests for individual components (cf. (3)(a) above).

      ii. Test the proper integration of an increasing number of components, adding one by one, until all components are integrated. This test will be called the *incremental integration test* (see Chapter 8).

3.2 General procedure of system development and testing

3.2.1 Prerequisites

A prerequisite to executing complex system models is the availability of a formal semantic for those models, particularly, for the medium that captures the *behavioural view* (cf. Subsubsection 4.1.1.3). Statecharts will be used to describe the behaviour of the 'executable' specification of reactive systems. Throughout this thesis, it will be assumed that the main functions[1] are known. In other words, the *functional view* (cf. Subsubsection 4.1.1.2) is considered to be trivial. In this case, liberal terminology, such as 'battery weakens' will be used to denote certain events of obvious meaning with respect to the environment of the system. Of course, to have a complete specification one would have to specify these aspects elsewhere. These aspects of a system would then be incorporated in the overall specification together with the statecharts ([cf. Chapter 4, Section 4.1.1]). In the next section a brief description of the process which is often followed when developing a complex system is given.

[1] In this context, the activities (actions) of the reactive system. In this thesis, we will often refer to a reactive system as a software system, but it could as well be referred to as a hardware system depending on the described characteristics of the system under development.

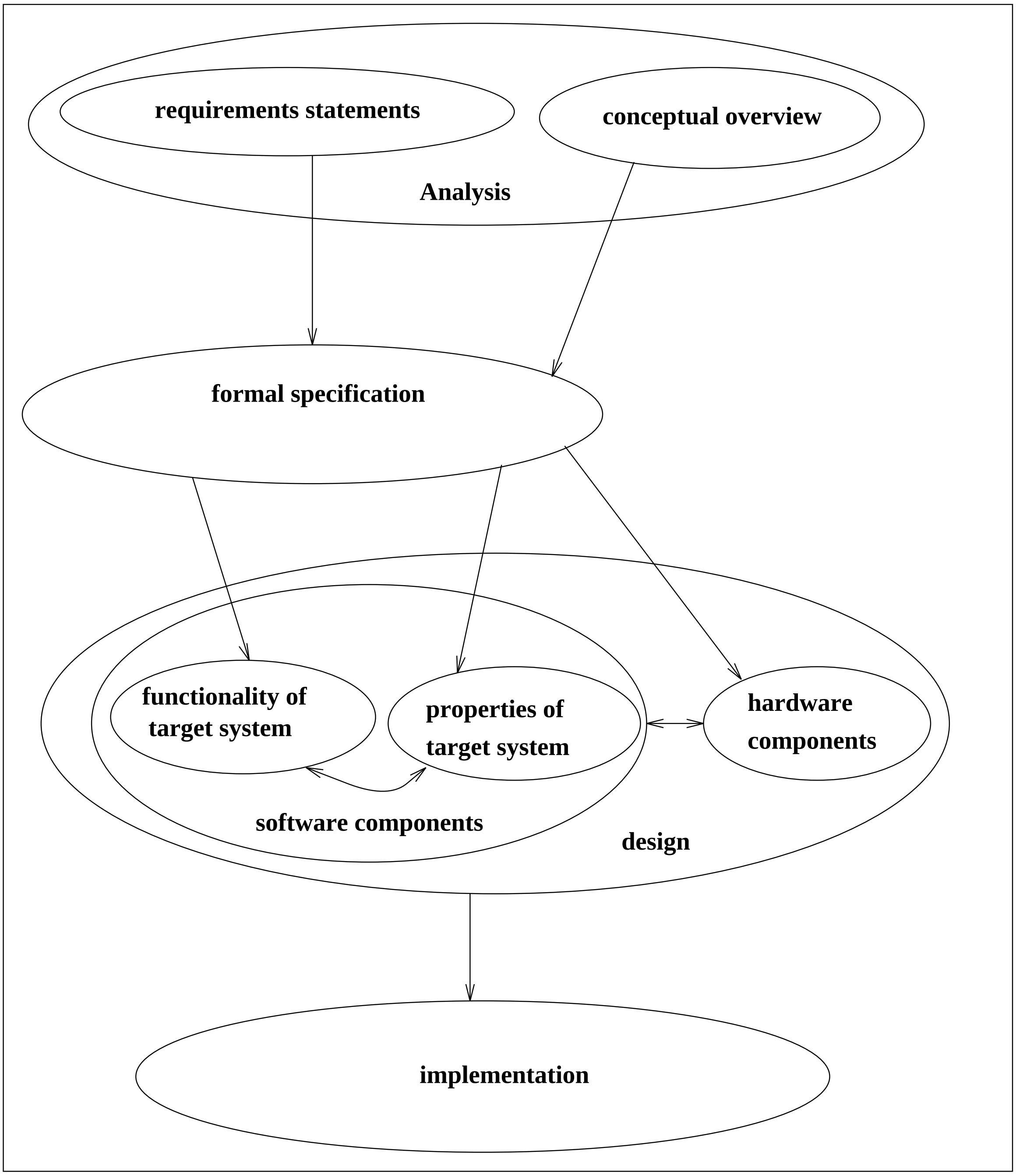


Figure 3.1: An overview of system development

3.2.2 The steps of reactive system development

Figure 3.1 illustrates the steps which may be undertaken within a system development.

Legend:

"⟶" describes the flow of information within system development;

"A ⟷ B" describes A depends on B and vice versa.

The first step in the system development is finding the requirements posed on the intended system. These are normally formulated in a natural language. During this early software life cycle, an intensified study of the requirements of the ultimate customer as well as the relationship between the product and the environment of its use is undertaken. Often misunderstandings between partners involved occur. To minimize such problems, graphical formalisms are employed. These offer both the customer and developer a visual and clear description and contribute to a better communication thus enabling a precise formulation of the problem.

The transition from an *analysis* of requirements to the specification of a system to meet them is the most crucial stage in the whole software development; the discovery of only a single error or a single simplification would fully repay all the effort expended (see Preface[2]). *Inspections*, *walkthroughs* and *reviews* are required to check important transformations between requirement and specification phases.

Formal description techniques are used to describe the behaviour and the structure of systems. Their basic idea is to produce correct and unambiguous descriptions, that should ultimately produce efficient implementations. As depicted in Figure 3.1, the input to the *formal specification* is a set of *requirement statements* and a *conceptual overview* of the proposed system. The output is a formal representation that captures the significant properties and functionality of the proposed system. Generally, a formal specification should abstract from the implemention details in order to give an overview of the system, to postpone the implementation decisions, and to allow all valid implementations. In that way, it makes use of neither design nor implementation concepts; it defines rather a *model* that represents the significant properties and functionality of the system which can be controlled through *validation, verification* and *conformance testing.* In the approach, the concepts and methods developed are based on such models.

3.2.3 Validation of the model using tests

Validation of the model using tests requires *test cases.* A test case generator will be responsible for the creation of test cases from the formal specifications (models). The main purpose of the generator is to produce a set of *suitable* test cases for the given model. Generally, the test cases are developed on the following test principles:

- *Test principle A* — the formal model is validated against its informal requirements;
- *Test principle B* — the formal model is verified against its implementation.

In this thesis, the test cases are *suitable*[3] for the *test principle A*, if for each error in the statechart model violating the informally specified requirements a test case will be created which is capable of detecting this error. In this particular case, it will be assumed that the informally specified requirements are correct.

[2] of this thesis

[3] Note that in this sense *suitable* describes the "ideal" case (cf. [143], chapter 2).

Note that in the case of reactive systems a test case is not simply a sequence of inputs and expected outputs. Due to the fact that the intended system may behave undeterministically on the interface level, the same sequence of inputs may simulate different responses of the intended system, or the system may even fail during repeated executions. This is often the case for so called *time dependent behaviour*. Therefore, it should be advisable to additionally define a test case as a description of *execution rules* specifying the full set of possible sequences of input and output events, together with *real-time assertions* which should be met when experimenting the test cases on the model. It is important to emphasize that the behaviour of the reactive models is continuous and hence without end state but rather stopping in a so called "idle state" or "rest state" for every "loop" or *run*.

Chapter 4

Introduction to the STATEMATE Tool and Statecharts

The purpose of this chapter is on one hand to present an overview of the STATEMATE tool (cf. section 4.1), and on the other hand to give an informal introduction to statecharts (cf. Section 4.2).

4.1 The CASE-Tool STATEMATE

In this section a brief introduction to the STATEMATE tool is given. Details about STATEMATE can be found in [54, 55, 72, 73, 74]. STATEMATE[1] is a computerized working environment for system designers. STATEMATE was intended for the specification, analysis, design, and documentation of large and complex reactive systems, such as real-time embedded systems, control and communication systems, and interactive software. Examples are digital watches, television sets, chips, airport activities, etc.

4.1.1 How does STATEMATE deal with the problems of reactive systems?

STATEMATE offers a set of three distinct but interrelated viewpoints to model a real-world process. The points of view are: *structural, functional and behavioural*. Taken as a whole, these three perspectives cover the traditional questions "*who, what, where, when, how, and why*" of a process description.

- **Structural** - *Who* does it and *Where* is it done; represented by "*Module Charts*".

- **Functional** - *What is done*; represented by "*Activity Charts*".

- **Behavioural** - *When, How, and Why* it is done; represented by "*Statecharts*".

[1] trademark of i-logix, Inc. Burlington, MA.

Figure 4.1 shows the correspondences of the STATEMATE views of a "System Under Development" (SUD). There is a tight relationship between the functional and behavioural views. Here, activities and data-flow of a system require *dynamic control* in order to terminate or to be activated (see **Behavioural view**, Subsubsection 4.1.1.3). On the other hand, the behavioural aspects are all but worthless if activities have nothing to control. Usually, each activity chart is related to at least one statechart, whose role is to control the activity's internal parts, i. e., its subactivities and their flow of information (see **Functional view**, Subsubsection 4.1.1.2). The structural view describes subsystems, modules or objects constituting the real system, and the communication between them (see **Structural view** Subsubsection 4.1.1.1).

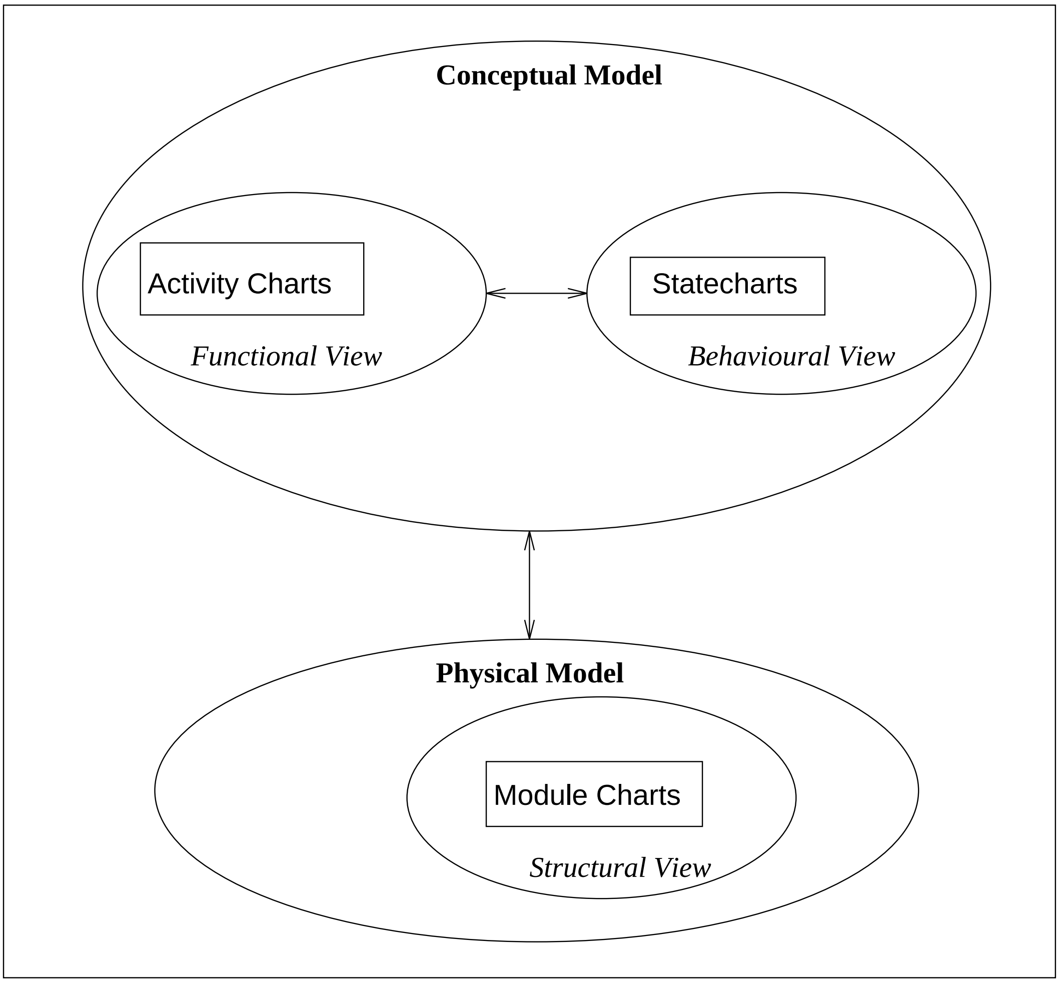


Figure 4.1: Different views of a System Under Development (SUD) in STATEMATE Legend: "A ⟷ B" describes A depends on B and vice versa.

While the functional and behavioural views build the *conceptual model* of the system, the structural view is considered to be its *physical model* since it is concerned with the various aspects of the system's implementation. As such, the conceptual model usually involves terms and notions from the problem domain, whereas the physical model deals more with the solution domain.

Using these views the developer can specify and analyse the SUD in terms of its functional capabilities and its behaviour over time, and determine the main subsystems that implement these capabilities. The specification process uses particular languages and techniques to describe the system. The resulting specification is called the **model** of the system.

4.1.1.1 Structural view

The structural view provides a hierarchical decomposition of the SUD into its physical components, called modules, and identifies the information that flows between them, i.e., the "chunks" of data and control signals that flow through physical links between the modules. The structural view is represented by *Module Charts*. The word "physical" should be interpreted in a general sense, with modules meaning anything from actual pieces of hardware to subroutines or packages of software.

4.1.1.2 Functional view

The backbone of the system model is the hierarchy of *activities*, completed with the details of data items and control signals that flow between them. This is essentially what is called the *functional decomposition* of the SUD. The functional capabilities of the system are captured here. However, in the functional view one does not specify dynamics; it is not stated *when* the activities will be activated, *whether* or not they terminate on their own, and whether they can be carried out in parallel. The same is true for data flow; in the functional view one specifies only that data *can* flow, and not whether and when it will. The functional view is represented by *Activity Charts*.

4.1.1.3 Behavioural view

It is the behavioural view that is responsible for specifying control. Control activities serve as the "central nervous system" of the model. They are meant to sense and control the dynamics of that portion of the functional description on their level. This is achieved by allowing a control to be present on each level of the activity hierarchy, controlling that particular level. These controllers are responsible for specifying *when, how, and why* things happen as the SUD reacts over time. The controllers are represented by *Statecharts*.

Among other things, a controller can start and stop activities, can generate new events, and can change the values of variables. It can also sense whether activities are active or data have been transferred and it can respond to the values of variables as well as test them. These connections between activities and control involve a rather elaborate set of events, conditions and actions, whereas the relationship between modules and activities is far simpler, and consists essentially of specifying which modules implement which activities.

4.1.2 The overall structure of STATEMATE

For each of these three views, the structural, functional, and behavioural, STATEMATE provides a graphical, diagrammatic language with a rule based graphics editor that checks for syntactic validity as the appropriate specifications are developed.

Figure 4.2 illustrates the overall structure of STATEMATE. The database is central, and obtains much of its input from the three graphics editors, *Statecharts*, *Activity-Charts* and *Module-Charts* and also from a textual *Forms* editor for informal narrative descriptions in which the non-graphical information is specified. The most interesting parts of STATEMATE, however, are the analysis capabilities. These include testing and simulation (execution) packages which will play a central role in this approach. In addition, STATEMATE provides a number of static tests, such as *reachability test, detection of nondeterminism* and *deadlock*, as well as *usage of transitions* (see page 95). Other interesting parts of STATEMATE are the documentation package and the 'ADA' and 'C' code-generation and prototype capability.

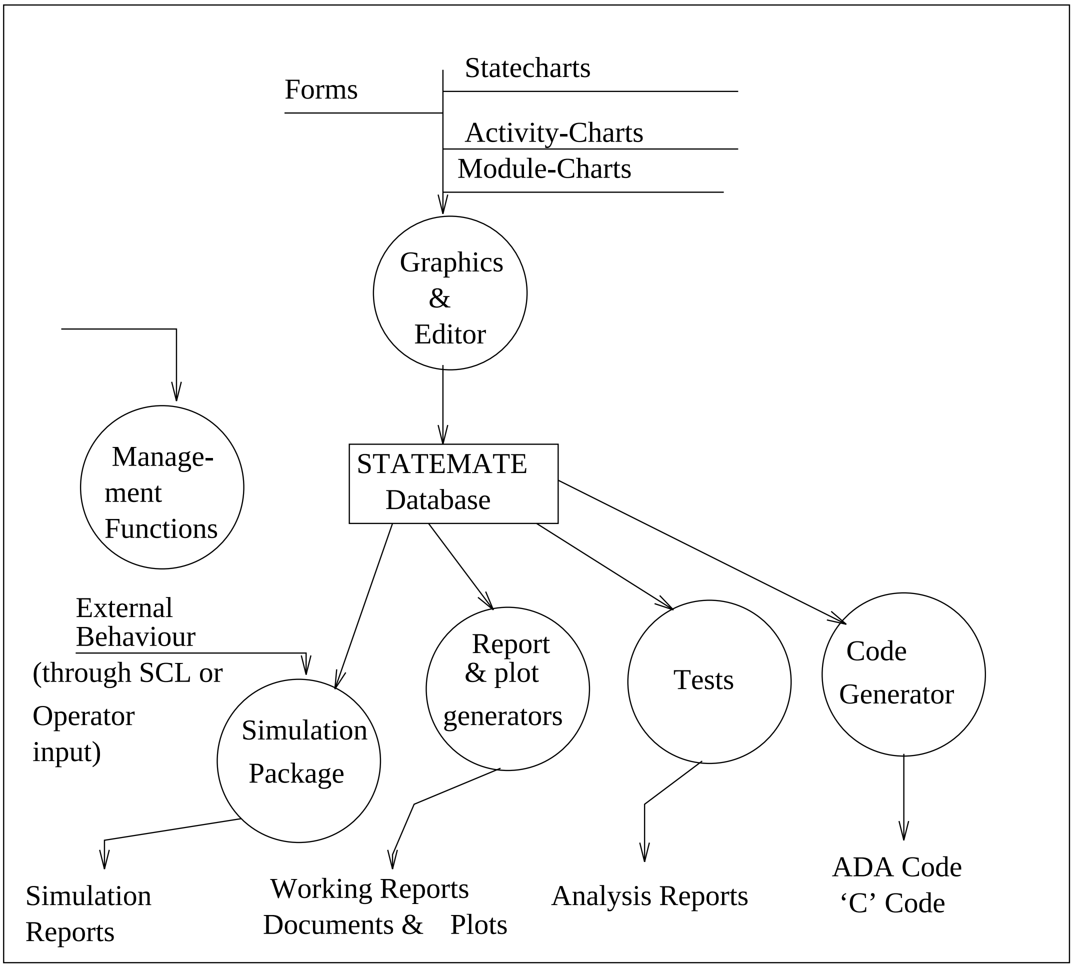


Figure 4.2: Overall structure of STATEMATE.

4.2 Informal introduction to statecharts

In this section an informal introduction to statecharts is given. In this thesis, due to space limitations, not all details of statecharts can be considered. Therefore the concentration is devoted to the basic features of statecharts.

Statecharts [50, 56] have been proposed to overcome the limitations of state-transition diagrams that are flat and unstructured while preserving their visual nature. Statecharts extend the conventional *finite state machines* (FSMs) with the notions of *hierarchy*, *orthogonality*, *broadcasting* and *history*. The features allow very succinct representations and naturally support stepwise development.

Before the features of statecharts are introduced, the general syntax of an expression labelling a transition in a statechart is given. Transitions are marked by *labels* "$e[c]/a$", where

(1) e is the *event* that triggers the transition

(2) c is a *condition* that guards the transition from being taken unless it is true when e occurs, and

(3) a is an *action* (or a sequence of actions) that is carried out if and only if the transition is taken.

All these are optional. Events and conditions are closed under Boolean operations *or*, *and* and *not*. The expression $e[c]$ is interpreted as "do transition if e occurs and c is true". Actions can be concatenated by ';' which means that they are carried out simultaneously.

First of all, consider Figure 4.3 which depicts an example of a statechart for a simple coffee vending machine (*SCVM*). *SCVM* is composed of three states: *On*, *Off*, and *Empty* with *Off* as the *initial*[2] state. *SCVM* changes from *Off* to *On* when the event *power-on* occurs. *On* is composed of four substates *Cup, Coffee, Controller* and *Light*. When the *Controller* of *SCVM* is in *Standby*, coins may be inserted or returned. If a *button* event occurs and the condition $money > 0$ is satisfied, *SCVM* turns the light on and begins to supply a cup of coffee. This is accomplished by action *cup-start* which is an *internal* event[3] of *Controller* and an input event of component (state) *Cup* as well as *Light*. Immediately when the action *cup-start* is generated in *Controller*, the transitions with label *cup-start* in components *Cup* and *Light* are triggered simultaneously. After a cup of coffee has been dispensed, *SCVM* goes back to *Standby* and the light is turned off. This is accomplished by the event *coffee-done* which is generated in *Coffee* and at the same time is input event of *Controller* as well as *Light*. Thus, as soon as the action *coffee-done* is generated in *Coffee*, the transitions with label *coffee-done* in components *Controller* and *Light* are triggered simultaneously. *SCVM* changes to *Empty* when either an event *cup-empty* from state *Cup-idle* or event *coffee-empty* from state *Coffee-idle*, respectively, occurs. In other words, *SCVM* moves from *On* to *Empty* when *SCVM* runs out of cups or coffee.

[2] which is represented by the small arrow with an indicating point at its start position.

[3] also called output event

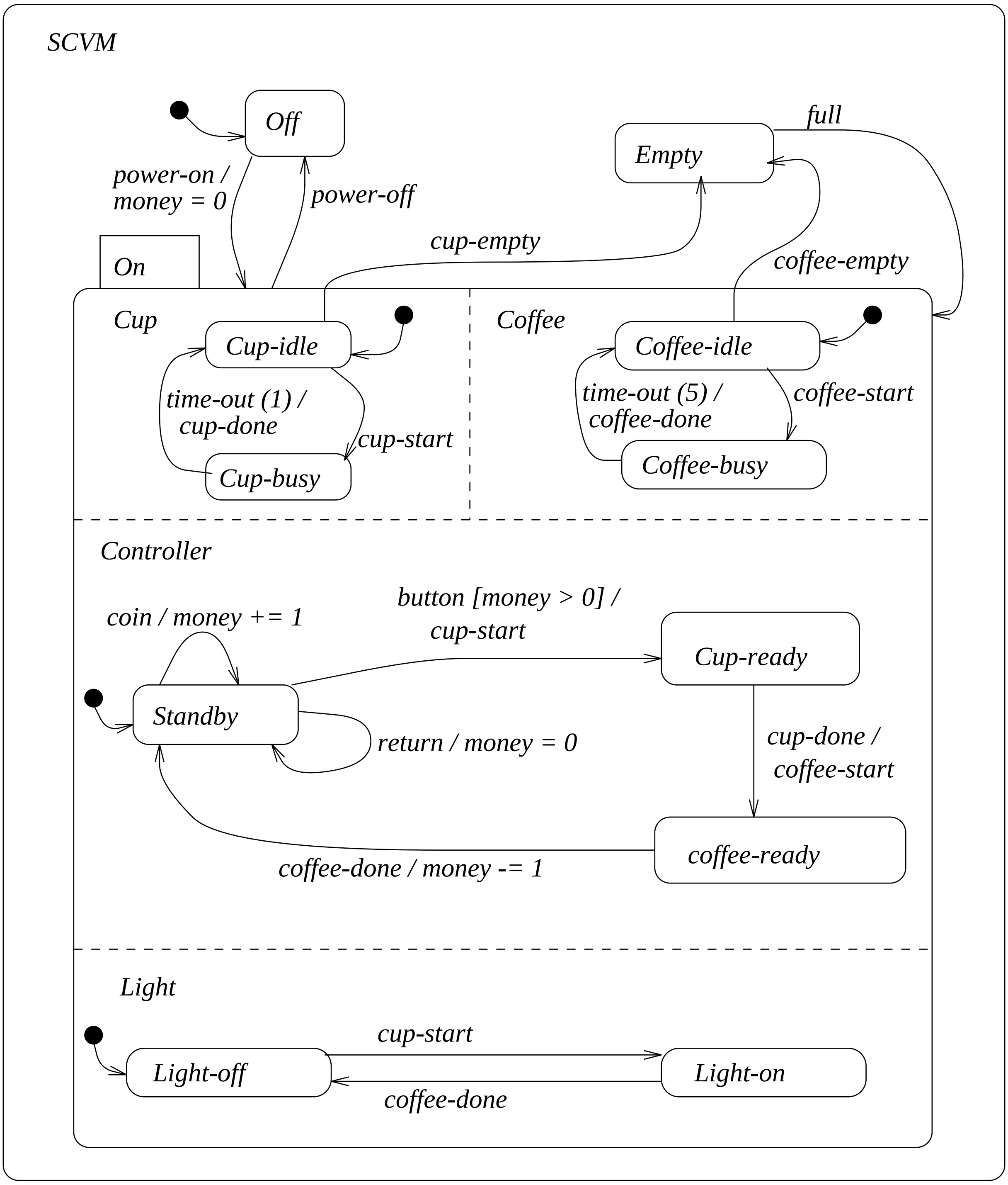


Figure 4.3: A statechart for a simple coffee vending machine *SCVM*

4.2.1 The basic features of statecharts

4.2.1.1 Concept of hierarchy

The concept of hierarchy is achieved by decomposing a state of a statechart into another statechart. *OR*-states have substates that are related to each other by “exclusive-or”.

In Figure 4.3, *Light* consists of mutually exclusive states, *Light-off* and *Light-on*. Thus when *SCVM* is in *Light*, it is either in *Light-off* or *Light-on*, but not in both.

The states at the bottom of the state hierarchy, i. e., those that have no substates, are called *BASIC* states. The state at the top level, i. e., the one with no parent state, is called the *root*.

4.2.1.2 Concept of orthogonality

Orthogonality is the dual of hierarchy. It is achieved by the *AND*-decomposition of an FSM. In Figure 4.3, *On* is the *AND*-composition of the orthogonal components *Cup, Coffee, Controller* and *Light*. Thus when *SCVM* is in *On*, this implies being in all of these components.

4.2.1.3 Concept of broadcasting

Next the idea behind the concept of broadcasting is given. When orthogonal components are not totally independent, communications between these components should be specified. This communication is achieved by associating an action with a transition. This is assumed to broadcast so that every system component in the *AND*-state recognizes it. In Figure 4.3, the transition (in *Controller*) from *Standby* to *Cup-ready* is associated with an action *cup-start* which triggers the transition from *Cup-idle* to *Cup-busy* (in *Cup*) as well as the transition from *Light-off* to *Light-on* (in *Light*). The disadvantage of this representation, however, is that the communication will not be represented graphically, i. e., lightly visible. The advantage of this representation is that a lot of transition arcs which would cross one another will be avoided. Therefore, a Test-tool is required to check whether the communication is modelled correctly. Another alternative can be to establish an extra representation for the "cross"-transitions. In the scope of this thesis, however, due to time limitations, we will not be able to deal with this aspect.

4.2.1.4 Concept of history

Another feature of statecharts is the ability to 'remember' a previous visit to a state. Statecharts use two kinds of *history connectors* (also called history-entries), H and H^*. The history mechanism realizes a sort of *suspend/resume* formalism. Figure 4.4 shows a *history entrance* into $S11$. The H enclosed in a circle is the simplest kind of history which means: The system enters the state most recently visited. The simple symbol H means that history is applied only on the level in which it appears. In Figure 4.4, if $S11$ is entered via the $ev2$-arrow then H chooses only between $S7$ and $S6$, i.e.: the system enters initial state $S2$ if it was in $S1$ or $S2$ when it most recently left $S11$; the system enters initial state $S3$ if it was in $S3$, $S4$ or $S5$ on that last visit. The choice here is by default arrows. On the other hand, when the entrance is via the $ev1$-arrow then $S7$, the

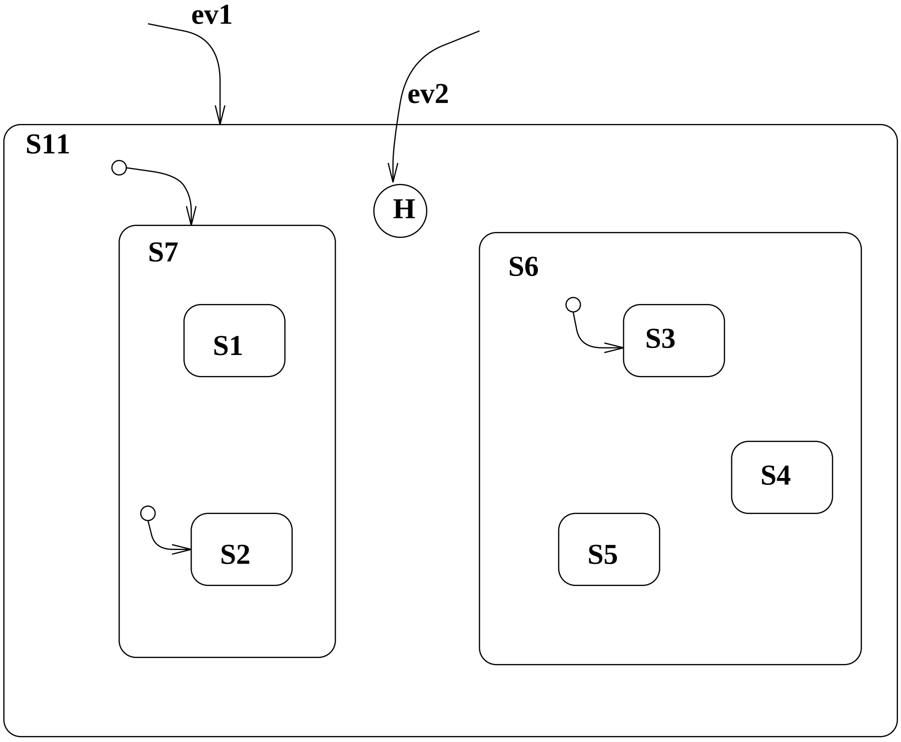


Figure 4.4: Enter-by-history

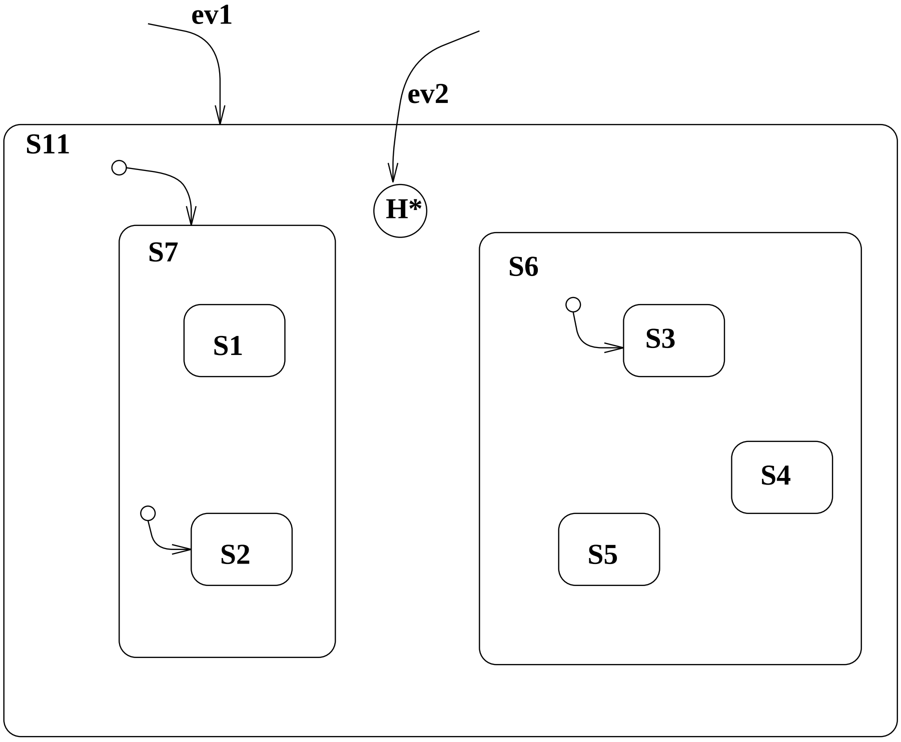


Figure 4.5: Enter-by-deep-history.

default of $S11$ is taken. Note that when the system enters $S11$ for the first time via the $ev2$-arrow, still $S7$ is taken. This is due to the fact that there is no history state yet.

When the history mechanism is applied to the lower levels, this is denoted by a star which is attached to the H-entry, i. e., H^* is used. In Figure 4.5 the system will enter the most recently visited state among $S1$ to $S5$, overriding the mechanism of default(initial)-states $S2$ and $S3$. This kind of history is referred to as *deep history*.

4.2.1.5 Static reactions

Statecharts are always complete: if there is no transition enabled from a given state, a *static reaction* (SR) in that state is executed. In other words, each state can be associated with static reactions which can be defined using the same format transition label $e[c]/a$ (see page 27). The main difference between the two is that a static reaction does not leave or enter any state. Thus each SR in a state $s \in S$ can be regarded as a transition in a virtual substate of s that is orthogonal to its ordinary substates and to the other SRs of state s. For example, Figures 4.6(a) and 4.6(b) depict the same behaviour.

If no transition or static reaction is *enabled*, nothing happens. In this particular case, it can be assumed that there is an implicit SR which does not do anything. In case of the simple Coffee Vending Machine on page 28, ‘do nothing’ SRs are in all states.

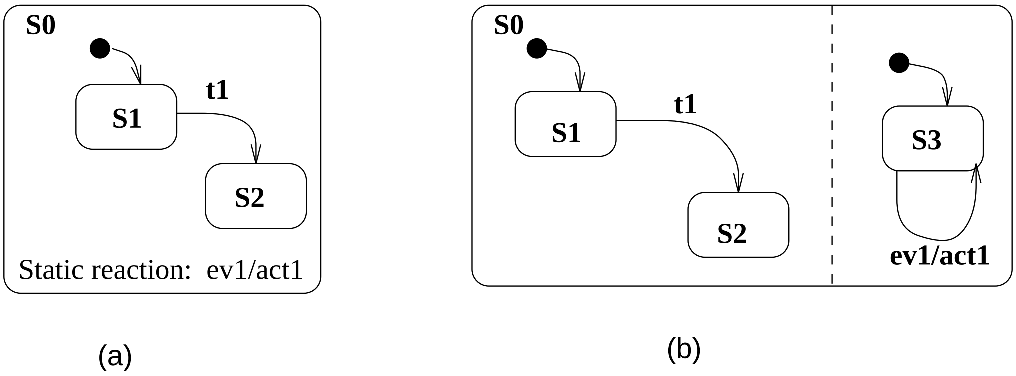


Figure 4.6: Representing static reactions

4.2.2 Other basics of statecharts

At the beginning of the Section 4.2, the general syntax of an expression labelling a transition in a statechart was given. There are several special events, conditions and actions that relate to STATEMATE model’s other entities, such as activities, data items or other states. In the following, only a few of them are mentioned. For further details see [51]. The following abbreviations are used, where $x \equiv y$ means, that y is the abbreviation of x. Action a can be the special action

(1) $start(p) \equiv st!(p)$ that causes the activity p to start;

(2) $stop(p)$ that causes the activity p to stop.

Similarly, primitive event e can be the special event

(1) $entered(s) \equiv en(s)$ that occurs when the state s is entered;

(2) $exit(s) \equiv ex(s)$ that occurs when the state s is exited.

Moreover, an action a can be scheduled for d time-units later on by carrying out a new action of the form $scheduled(a, d) \equiv sc!(a, d)$.

Similarly, the special event $timeout(e, d) \equiv tm(e, d)$ occurs d time-units after the most recent occurence of the event e.

Chapter 5

The Formal Syntax and Semantics of Statecharts

In Chapter 4, an informal description of statecharts was presented. The informal methods are widely used because they are intuitive, but they suffer from incomplete and imprecisely defined syntax and semantics. Therefore, the aim of this chapter is to develop the formal syntax and semantics of statecharts. These formal definitions are prerequisites to the test concepts and test methods which are to be developed in this thesis.

Although Harel, Pnueli, Schmidt and Sherman presented a formal syntax and semantics of statecharts in their paper from 1987 (cf. [51]), a number of issues, e. g., transitions in parallel states, hierarchy in states, etc. are not accurately defined. Other features such as *inter-level transitions* and conflictness between transitions are hardly defined. Fortunately, many of these problems are discussed by Harel and Naamad in their report of the recent STATEMATE semantics of statecharts (cf. [54]). Nevertheless, this report is *informal* and thus does not provide precise definitions. Therefore, this Chapter 5 is intended to establish the *formal* syntax and semantics of statecharts which can be adopted to solve various problems, e. g., problems with respect to

(1) *transitions in parallel states* (*nodes*),

(2) *compound transitions* (*arcs*),

(3) *inter-level transitions*,

(4) *conflict between transitions* (*nondeterminism*), etc.

With regard to its importance, this chapter builds one of the most fundamental parts of this thesis. The establishment of such formal descriptions contributes to the simplifications of the tests to be designed for the statechart model. For example, the tester can assume that the transitions in the statechart model are 'legal'. Consequently, the number of tests performed by the system designer/tester to examine his/her "System Under Development" (SUD) specification is reduced. Futhermore, the tests may not be as complicated as would be the case if one had to consider 'illegal' transitions as well.

5.1 Overview of statechart specification guideline

Figure 5.1 presents an overview of the approach which is developed in this chapter to establish formal syntax and semantics of statecharts. As we will see in Section 5.3, there are many variants of syntax or semantics of statecharts proposed in the literature, e. g., *Argos* [101, 102], *SpecCharts* [111], *Modecharts* [75], *RSML* [95], as well as those in [69, 70, 78, 158]. The goal of the formal syntax and semantics of statecharts described in this chapter is to establish some kind of standard specification guideline over which the test concepts and test methods to be developed in this thesis are based.

Following Figure 5.1, the syntax of statecharts is viewed in terms of STATEMATE's graphical representation of a statechart. In other words, the visual descriptions of a statechart are considered. Statecharts extend the conventional *finite state machines* (FSMs) and their corresponding state transition diagrams (*state diagrams* for short) with the notions of *hierarchy*, *orthogonality*, *broadcasting* and *history* (cf. Section 4.2). These state diagrams may be described as *labelled directed graphs* (*digraphs* for short) (cf. Section 2.2) with nodes denoting states and arrows (labelled with triggering events and guarding conditions) denoting transitions. The STATEMATE-Version approach, therefore, can be seen as the extension of state diagrams by *AND*/*OR*-decomposition of states together with 'inter-level' transitions and a broadcast communication between concurrent components. Based on these observations, we develop formal definitions which can be used to describe the syntax and behaviour of a given statechart. The resulting formal description is referred to as *Canonical Normal Form Version* (cf. Figure 5.1).

Considering the visual descriptions above, the STATEMATE-approach is described solely by its diagrammatic terms, and thus allows various interpretations of the behaviour of a given statechart. The formal description should, therefore, in any case be language-theoretic or at least algebraic in order to allow a precise description as well as interpretation of the behaviour of a given statechart. Our approach begins by partitioning the elements of the statechart into two classes: (*a*) *nodes/states* and (*b*) *arcs/transitions* as depicted by Figure 5.1.

5.1.1 Nodes/States

Nodes: Rounded rectangles or boxes are used to represent the nodes (states) at any level of a statechart. These can be clustered into new nodes (states). This means that a natural notion of hierarchy or modularity is provided, and thus a stepwise top-down development is supported. The nodes of a statechart are represented by V (cf. Def. 2.1). In *Canonical Normal Form Version* nodes are defined as *states* and represented by S (cf. Subsection 5.2.1). The "hierarchy" of states, where a state may consist of substates, e. g., *SCVM* of Figure 4.3 page 28, is formally defined by the hierarchy functions *children* and $children^*$ (cf. Definition 5.3). There is a one-to-one correspondence between nodes and states.

Syntax of Statecharts (STATEMATE-Version)		Statecharts ($CNF^\star$ Version)
graphical representation of statechart is in form of		**Definitions**
(structured) *nodes* (boxes) V (cf. Def. 2.1)	$\xleftrightarrow{1:1}$	(hierarchical) *states* S (cf. Subsection 5.2.1)
+ *edges* (*arrows or arcs*) E (cf. Def. 2.1)	$\xleftrightarrow{n:m}$	*transitions* $\mathcal{T}$ (cf. Def. 5.11)
complicated cases: E. g.,		
♣ *'transitions' which consist of linked transition segments* (*i. e., labelled arrows that connect states and connectors of various kinds*)	$\longleftrightarrow$	*Syntax* with *transformations* based on notion of full compound transitions (**full** *CT*)
		$\Downarrow$
		Canonical Normal Form CNF (cf. Subsection 5.2.5)
		(**Transitions**) **Semantic**
♣ *transitions in parallel states* (*nodes*)	$\longrightarrow$	*Complex statechart* defined by concept of *legal* configurations (cf. Def. 5.25 and Def. 5.28)
		$\Downarrow$
		♠ Def. 5.32 (*Legal transitions*) ♠ *modified automata product* (cf. Definitions 5.36 and 5.37) $\hat{=}$ *simple OR-graph product* $G0$ (cf. Def. 5.35)
♣ *inter-level transitions* (*arcs*)	$\longrightarrow$	Notion of scope of transitions (cf. Subsection 5.4.4)
		$\Downarrow$
		Def. Sets of entered/exited states (cf. Definitions 5.39, 5.40, 5.41, 5.42 and 5.43)
♣ *conflicts between transitions*	$\longrightarrow$	Notion of *priority* of transitions (cf. Subsections 5.4.5 and 5.4.6)
$\downarrow$ **simple case**		
a transition $t1 = (S1, e1, S2)$ corresponds to an arrow with label $e1$ from a simple state $S1$ to simple state $S2$	$\xleftrightarrow{1:1}$	*transition t* (*cf. Def. 5.11* (*Simple statechart*))

$\star$: *CNF* stands for "Canonical Normal Form"

Figure 5.1: Overview of statechart specification guideline

5.1.2 Arcs/Transitions

Arcs (**arrows**): An arrow is labelled with triggering events and guarding conditions. The arcs of a statechart are represented by E (cf. Def. 2.1). In *Canonical Normal Form Version* arcs are defined as *transitions* and represented by $\mathcal{T}$ (cf. Def. 5.11). Generally, there is an n-to-m correspondence between arcs and transitions (cf. Figure 5.1). In case of a simple statechart, however, there is a one-to-one correspondence between arcs and transitions. In other words, a transition may be described syntactically by an *arc*, e. g., $t1 = (S1, e1, S2)$ corresponds to an arrow with label $e1$ from single state $S1$ to single state $S2$.

Nevertheless, in the graphical language supported by STATEMATE a transition may consist of a number of linked transition segments which are the labelled arrows that connect states and connectors of various kinds. Therefore, a *transformation* of such representations into a special statechart called **Canonical Normal Form** (CNF) is necessary. This is based on the notion of 'full compound transitions'.

There is yet a number of other complicated cases such as transitions in parallel states (nodes), conflicts between transitions (nondeterminism) and inter-level transitions (arcs). In particular, if we consider the case of a direct exit of an AND-state, the source of a transition is a set and contains *more than one* state. According to the definition of a legal transition in [51], the only condition which must be satisfied is that every two states in the source set of t must be mutually orthogonal. Nevertheless, this condition delivers only 'weakly legal' transitions (cf. Subsection 5.4.2). To be in a position to define a legal transition t, the 'maximality' of the source set of t must also be defined. The concept of 'weakly legal' transition must thus be extended to 'legal' transitions which have legal configurations (cf. Definition 5.32). To deal with the conflicts between transitions (nondeterminism), we formalise a concept — the notion of 'scope' of transitions.

The formal syntax and semantics of statecharts described in this thesis is mainly based on the *informal* report of [54]. In addition, the work of [124] will be consulted. It should be underlined, however, that the statechart model formalised in this thesis differs from that of Harel (cf. [50], [54]) in a way that the root state of a statechart must be of OR-type and the substates of an AND-state are always of OR-type. Assuming that the statechart model is represented as a digraph[1], the tester can apply efficient algorithms to the digraph to generate test paths that satisfy given criteria.

In Section 5.2 we present the formal syntax of statecharts. After that we glance at the variants of statecharts in Section 5.3. Finally, in Section 5.4 we deal with the formal aspects which will be required to describe and formalise the *step semantics* in Chapter 6.

[1] The transformation of a statechart which consists of OR-states only into the associated *digraph* (also called associated OR*-graph*) has a remarkable characteristic with respect to the states (vertices) and transitions (edges): it forms a one-to-one relationship.

5.2 The formal syntax of statecharts

In the following, the formal syntax of statecharts and basic definitions that are necessary to describe the main features described above are given. To begin with, some basic definitions over which the syntax of hierarchical statecharts will be established are given.

5.2.1 State hierarchy

In order to avoid an immense number of states in one statechart diagram, it is allowed to start with a manageable number of states on the "first level" and then to refine some states into substates represented by different diagrams. This "hierarchy" of states where a state may consist of substates, e. g., *SCVM* of Figure 4.3 page 28, is formally defined by the hierarchy functions *children* and *children**.

In the following S is a nonempty *finite* set of so called **states**. At the beginning of this chapter we pointed out that a number of issues are not accurately defined in [51]. For example, the *hierarchy function* $\rho : S \rightarrow 2^S$ defined in [51] delivers for $\rho^i(s)$ a set of states for $0 \leq i \leq 1$ and a set of subsets of states for $i > 1$. This definition is thus not quite applicable. To overcome such shortcomings, we introduce a hierarchy-relation ϱ before the definitions of the hierarchy functions *children* and *children** can be given.

Definition 5.1 *Let $\varrho \subseteq S \times S$, then ϱ^i, $i \geq 0$, and ϱ^* will be defined as follows:*

(1) $\varrho^0 := \{(s,s) \mid s \in S\}$

(2) $\varrho^1 := \varrho$

(3) $\varrho^{n+1} := \varrho \circ \varrho^n$, *for* $n \geq 1$ [2]

(4) $\varrho^* := \bigcup\limits_{n \geq 0} \varrho^n$

A *hierarchy-relation* ϱ over S will be given as follows:

Definition 5.2 *A relation ϱ is called a* **hierarchy-relation** *over S with root $r \in S$* $:\Leftrightarrow$

(1) $\varrho \subseteq S \times S$

(2) $\forall s \in S : (s,r) \notin \varrho$

(3) $\forall s \in S \backslash \{r\} : (r,s) \in \varrho^*$ *and* $(\overset{=1}{\exists}\, n \in I\!N) :$ $(\overset{=1}{\exists}\, s_1, s_2, \ldots, s_n \in S)$ $s_1 = r$ *and* $s_n = s$, *and* $(s_i, s_{i+1}) \in \varrho$ *for* $i = 1, \ldots, n-1$.

Now a hierarchy-relation will be represented by a hierarchy function "*children*" which again can be represented as a directed tree with root r. In the next definition, *children*$^i(s)$ delivers for a state s its *descendants* on *level* i with respect to root r.

[2] For two arbitrary relations ξ and ζ over a given set (domain) S: $\xi \circ \zeta := \{(s,t) \in S \times S \mid (\exists u \in S) : (s,u) \in \xi \text{ and } (u,t) \in \zeta\}$.

Definition 5.3 *The* **hierarchy functions** *children, children^i, children* : $S \to 2^S$ are defined as follows (with respect to a hierarchy-relation ϱ over S) :*

(1) $children^0(s) := \{s\}$

(2) $children^i(s) := \{t \in S \mid (s,t) \in \varrho^i\}$

(3) $children(s) := children^1(s)$

(4) $children^*(s) := \{t \in S \mid (s,t) \in \varrho^*\}$*, called the set of* **descendants** *of state* s[3]

Remark 5.1 *There exists a* **unique** *state $r \in S$, called* **root state***, such that $\forall s \in S$: $r \notin children(s)$.*

Remark 5.2 The hierarchy function *children* describes for each state $s \in S$ its substates. Note that due to the properties of Definition 5.2 the definition of *children* allows no loops in the construction since ϱ is a hierarchy-relation. $children^i(s)$ describes for a state s its *descendants* on *level* i with respect to root r. $children^* : S \to 2^S$, which for a state $s \in S$ computes its descendants[4], is a function satisfying the following:

(i) $s \in children^*(s)$,

(ii) $s_2 \in children^*(s_1) \Rightarrow children(s_2) \subseteq children^*(s_1)$,

(iii) $s_2 \in children^*(s_1) \Rightarrow children^*(s_2) \subseteq children^*(s_1)$.

Definition 5.4 *The generalization of the definition $children^*(s)$ for a set of states $X \subseteq S$ is given as follows:* $children^*(X) = \bigcup_{s \in X} children^*(s)$.

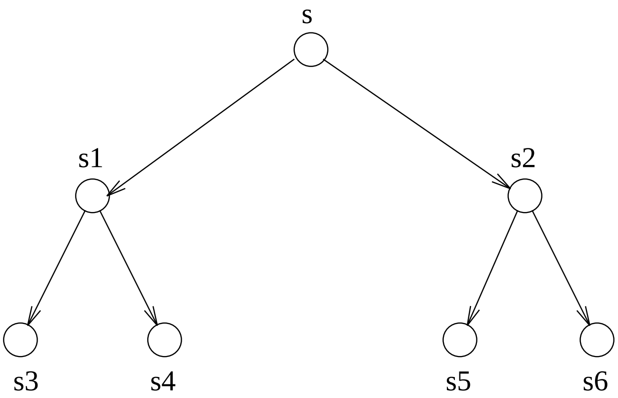


Figure 5.2: Illustrating the *children* function

Figure 5.2 is used to illustrate the hierarchy function *children*:

$children^0(s) = \{s\}$

[3] Note that the state s is considered to be a descendant of itself for reasons of simplicity.

[4] which consists of children, children of children, and so on

$children^1(s) = \{s1, s2\} = children(s)$

$children^2(s) = children(s1) \ \cup \ children(s2) = \{s3, s4, s5, s6\}$

$children^3(s) = \{\emptyset\}$

The following definitions of *parent*, $parent^i$ and $parents^*$ are the reverse terms of *children*, $children^i$ and $children^*$.

Definition 5.5 *The* **reverse hierarchy functions** *parent*, $parent^i$ *and* $parents^*$ *are defined as follows (with respect to a hierarchy-relation* ϱ *over* S*):*

(1) $parent^0(s) := s$

(2) $parent^i(s) := \begin{cases} t & \textit{if there exists some } t \textit{ with } (t, s) \in \varrho^i \\ \textit{undefined} & \textit{otherwise} \end{cases}$

(3) $parent(s) := parent^1(s)$*, called the* **direct ancestor**[5] *of state* s *(if defined).*

(4) $parents^*(s) := \{t \in S \mid (t, s) \in \varrho^*\}$*, called the set of* **ancestors** *of state* s*.*

Remark 5.3 Restricted to its defined domain, $parent : (S \backslash \{r\}) \to S$ uniquely defines for each state $s \in S\backslash\{r\}$ its direct ancestor. $parent(s) = u \Leftrightarrow s \in children(u)$. The function $parents^* : S \to 2^S$ which for a state computes $s \in S$ its ancestors[6], is a function satisfying the following:

(1) $s \in parents^*(s)$,

(2) $\forall s_1 \in (S\backslash\{r\}) : (s_1 \in parents^*(s_2) \Rightarrow parent(s_1) \in parents^*(s_2))$.

(3) $\forall s_1 \in (S\backslash\{r\}) : (s_1 \in parents^*(s_2) \Rightarrow parents^*(s_1) \subseteq parents^*(s_2))$.

Definition 5.6 *The generalization of the definition* $parents^*(s)$ *for a set of states* $X \subseteq S$ *is given as follows:* $parents^*(X) = \bigcup_{s \in X} parents^*(s)$.

For the hierarchy function illustrated by Figure 5.2, the following is true:

$parent^0(s4) = s4$,

$parent(s4) = parent^1(s4) = s1$,

$parent^2(s4) = s$,

$parent^i(s4)$ undefined for $i \geq 3$,

$parents^*(s4) = \{s, s1, s4\}$,

$parent(s1) = s \in parents^*(s4)$, and

$parents^*(s1) = \{s, s1\} \subseteq parents^*(s4)$, since $s1 \in parents^*(s4)$.

[5] Note that the state s is considered to be an ancestor of itself for reasons of simplicity.

[6] which consists of ancestor, ancestor of ancestor, and so on

5.2.2 Labels of transitions

In Subsection 4.1.1.3 (*behavioural view*) it was noted that the main task of a statechart is to capture the reactive behaviour of a reactive system. This behaviour (of a reactive system) is described as the set of input and output events, conditions and actions etc. (cf. Subsection 1.2.1). The general syntax form of an expression labelling a transition in a statechart is given as $e[c]/a$ (see page 27). In the following, the general form of *labels* of the transitions[7], i. e., expressions for *events*, *conditions* and *actions* are formally defined. Note that primitive events, primitive conditions and primitive actions are given as atomic elements.

Definition 5.7

(1) The finite set of **primitive events** *is denoted by SE_p.*

(2) **Event expressions** *SE_{expr} are defined by the grammar:*

SE_{expr} ::= ε | SE, *where ε is the empty expression*

SE ::= e | $\neg(SE)$ | $(SE \vee SE)$ | $(SE \wedge SE)$, *where e can be replaced by each element of SE_p.*

The set of all event expressions is denoted by $\mathcal{L}_e$.

(3) The primitive events e which are taken in a special situation SIT are those for which $Val_{SIT}(e) = true$. Val_{SIT} is a mapping, called assignment which expresses whether a primitive event is true (or false) in a special situation SIT:

$Val_{SIT} : SE_p \rightarrow \{true,\ false\}$.

Remark 5.4 The exact content of the term *special situation SIT* will be discussed and defined later as 'system status' (cf. Definition 6.1, page 97).

Definition 5.8 *(1) The finite set of* **primitive conditions** *is denoted by SV_p.*

(2) Let $R = \{=, >, <, \neq, \leq, \geq\}$ be the set of relational operators[8]; **condition expressions** *SV_{expr} are defined by the grammar:*

SV_{expr} ::= T | F | SV,

where the logical values $true$ and $false$ are denoted by T and F.

SV ::= c | $(A\ REL\ A)$ | $\neg(SV)$ | $(SV \vee SV)$ | $(\ SV \wedge SV)$, *where c can be replaced by each element of SV_p, and REL by each element of R. Term 'A' can be replaced by an arithmetic expression, a variable or a constant (data-item), but the expressions replacing term 'A' on both sides of the comparison ($A\ REL\ A$) must be of the same type, e. g., both numeric (integer, real, bit and bit-array), or both strings or structured strings[9].*

[7] The formal definition of a transition will be given in Definition 5.11.

[8] It is often necessary to compare data-items in one of several ways. For this comparison conditions *expr*1 *REL expr*2 where *expr*1 and *expr*2 are data-items and $REL \in R$ will be allowed in this thesis.

[9] A more precise definition of all expressions which may replace term '*A*' does not seem necessary since the following concepts do not depend on the special choice of expressions.

The set of all condition expressions is denoted by $\mathcal{L}_c$.

(3) The primitive conditions c which are taken in a special situation SIT *are those for which* $Val_{SIT}(c) = true$. Val_{SIT} *is a mapping, called assignment, which expresses whether a primitive condition is true (or false) in a special situation* SIT:

$Val_{SIT} : SV_p \rightarrow \{true,\ false\}$.

Definition 5.9 *Let* $SV_{cond} = \{fs!(c) |\ c \in SV_p\} \ \cup\ \{tr!(c) \mid c \in SV_p\}$ *be the set of all assignments to condition variables;* **action expressions** SA_{expr} *are then defined by the grammar:*

SA_{expr} ::= $\varepsilon \mid SA$,

SA ::= $e \mid ca \mid (SA; SA)$, *where* e *can be replaced by a primitive event (each element of* SE_p*) and* ca *can be replaced by each element of* SV_{cond}.

The set of all action expressions is denoted by $\mathcal{L}_a$.

Note *The action expression* $(SA_1; SA_2)$ *represents a simultaneous, non-deterministic execution of* SA_1 *and* SA_2.

Definition 5.10 *Let* $\mathcal{L}_e, \mathcal{L}_c, \mathcal{L}_a$ *be the sets of event, condition and action expressions, respectively, as defined above (cf. Definition 5.7, Definition 5.8 and Definition 5.9). The set of* **labels** L *is the cartesian product of the set of events, the set of conditions and the set of actions. A label* $l \in L$ *where* $l = (e, c, a) \in \mathcal{L}_e \times \mathcal{L}_c \times \mathcal{L}_a$ *is also written as* $e[c]/a$.

Note *(Informal semantics) If* $e[c]/a$ *is the label of a transition* t*, then* t *is triggered by event* e *and condition* c *and action* a *is executed when* t *is taken.*

5.2.3 Statechart definitions

In this subsection we describe various definitions of a statechart. We begin with the definition of a *simple statechart* in Subsection 5.2.3.1. This is followed by the definition of an *orthogonal statechart/orthogonal component* in Subsection 5.2.3.2. In Subsection 5.2.3.3 we then describe a *substatechart* which is also called *OR-statechart*. Finally, in Subsection 5.2.3.4 we define a *very simple OR-statechart* in which all states with the exception of the root are basic.

5.2.3.1 Definition of a simple statechart

In the following, the definition of a simple statechart is given.

Definition 5.11
A **simple statechart** *is a 8-tuple* $Sct = (S, r, children, type, \mathcal{H}, hist, default, \mathcal{T})$ *where:*

(1) S *is a nonempty finite set of so called* **states**.

(2) $r \in S$ *is a unique state, called* **root state**.

(3) children is the **hierarchy function** *of Definition 5.3 (with respect to root r and state set S).*

(4) $type : S \to \{OR, AND, BASIC\}$ *defines for each state its type, such that:*

(a) $type(r) \; = \; OR,$

(b) $\forall s \in S : \; children(s) \; \neq \; \emptyset \; \Leftrightarrow \; type(s) \in \{OR, AND\},$

(c) $\forall s_1, s_2 \in S : \; (type(s_1) = AND \; \wedge \; s_2 \in children(s_1) \; \wedge \; children(s_2) \; \neq \; \emptyset \; \Rightarrow type(s_2) = OR).$

A state $s \in S$ *is called an* **OR-state** *[or an* **AND-state** *or a* **basic state**, *respectively] iff*
$type(s) = OR$ *[or* $type(s) = AND$ *or* $type(s) = BASIC$*, respectively].*

Note: *A statechart root r must have children due to requirements* (4)(*a*) *and* (*b*) *above.*

(5) $\mathcal{H} = \{H, H^*\}$ *is the set of so called* **history symbols** (*also called* **history connectors**).

Note: *There are two kinds of history connectors, H and* H^**, where H represents the simplest kind of history, and* H^* *stands for the deep history*[10].

(6) $hist : S \overset{\subseteq}{\to} \mathcal{H}$ *is a* **partial function** *which associates history symbols (history-entries)*[11] *to OR-states only, i. e.,* $hist(s)$ *is undefined if s is not an OR-state, but even for an OR-state* s_i $hist(s_i)$ *may be undefined.*

(7) $default : S \overset{\subseteq}{\to} S$*, called the* **default function**, *defines for each OR-state s the default state* $default(s)$ *(a "child" of s which is entered when state s is entered).*
$default(s)$ *is defined iff s is an OR-state, and – if defined –* $default(s) \in children(s)$*.*
$default(s)$ *is called* **start state** *or* **initial state** *or even* **default state** *of state s.*

Note: *The syntactic default function is used to compute the state (or states) a statechart will enter when transitions occur. When an OR-state s is not entered by the history-symbol, the specified state* $default(s)$ *is indeed entered. When an AND-state s is entered, the parallel states that compose s are entered (which again may lead to the entrance of the default states of these parallel states). In the example of the simple coffee vending machine SCVM on page 28, the default states are "marked" by the small arrows with indicating points at their start positions (cf. Section 4.2, page 27).*

$default(SCVM) = Off$ *for the OR-state SCVM.*

(8) $\mathcal{T} \subset S \times L \times (S \cup \mathcal{H})$ *is the set of* **transition arcs**, *where L is a finite set of* **labels** *(cf. Definition 5.10).*
If $ta = (s, l, t) \in \mathcal{T}$*, i. e., ta is a transition arc, then*

[10] In the following, however, only the semantics of H will be formalised, H^* can be easily implemented by allowing the simple history symbol H to be applied on each level of the statechart.

[11] (see Subsubsection 4.2.1.4)

(a) s is called the **source state** *of ta,*

(b) l is called the label of ta,

(c) t is called the **target symbol** *of ta (not the target state, since t can also be a history symbol).*

(*End of Definition 5.11*)

Remark: Note that in comparison to Harel, Pnueli, Schmidt and Sherman's definition of statechart (cf. [51]), the statechart definition above requires that the root state of a statechart must be of OR-type and also that the substates of an AND-state are always of OR-type. This distinguishing feature will play a very important role in the establishment of relations between a statechart and a digraph. Above all, it is easy to show that this special characteristic does not impose any restrictions on the expressive power of a statechart.

5.2.3.2 Definition of an orthogonal statechart/orthogonal component

Using Definition 5.11 of a simple statechart, in this Subsection 5.2.3.2 we introduce the terms *orthogonal statechart/orthogonal component* which will be required for the development of methods and concepts for transforming AND-states into OR-states (cf. Subsection 5.4.3). Informally, an *orthogonal statechart* is a substate s of a simple statechart (cf. Definition 5.11) which satisfies the following requirements:

(i) the substate s is of type AND.

(ii) the children of the substate s are either of type $BASIC$ or of type OR.

The 'direct' descendants (i. e., children) of the substate s are called the *orthogonal components* of s.

Figure 5.3 depicts a simple statechart *S00*. The substate *S0* in Figure 5.3 represents an orthogonal statechart. $S1$ and $S2$ are the orthogonal components of $S0$.

We formalise the terms *orthogonal statechart* and *orthogonal components* as follows:

Definition 5.12 *Assume $Sct = (S, r, children, type, \mathcal{H}, hist, default, \mathcal{T})$ is a simple statechart and $s \in S$.*

(1) Then $Sct(s) = (S', r, children', type', \mathcal{H}, hist', default', \mathcal{T}')$ is an **orthogonal statechart** *iff*

(a) $s \in children(r) \wedge type(s) = AND$

(b) $\forall s' \in children(s) : type(s') \in \{BASIC, OR\}$

The components of $Sct(s)$ are the following:

i. $S' = \{s, r\} \cup children(s)$

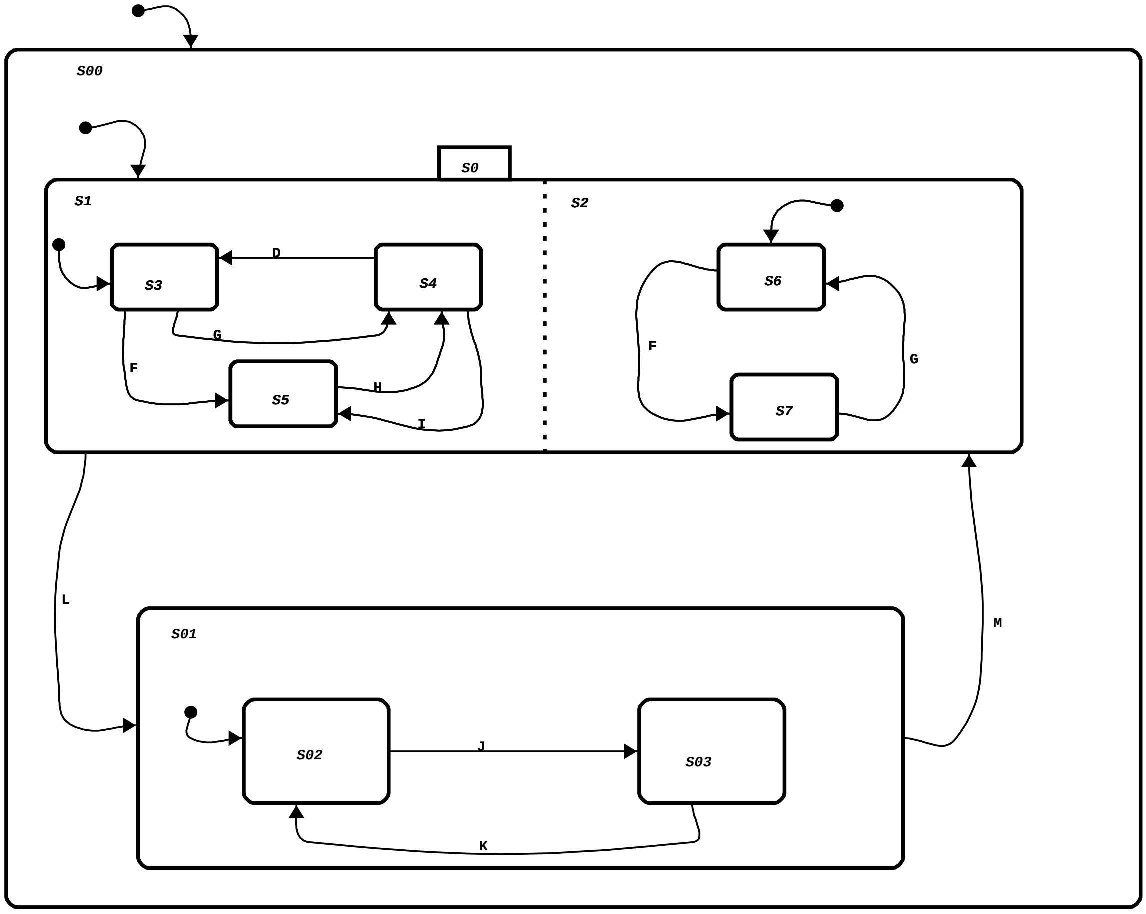


Figure 5.3: A simple statechart *S00*.

ii. *children′, type′, hist′ and default′ are restrictions of Sct to S′ instead of S*

iii. *$\mathcal{T}'$ is the restriction of $\mathcal{T}$ to $S' \times L \times (S' \cup \mathcal{H})$.*

(2) *Each state $s' \in children(s)$ is called an* **orthogonal component** *of the orthogonal statechart. The* **set of transition arcs** *of an orthogonal statechart will be denoted by $\mathcal{T}_{orth_{arcs(s)}} \subseteq \mathcal{T}$.*

As an example, consider Figure 5.3 again. According to Definition 5.12, the substate *S0* represents an orthogonal statechart.
S00 is the root of the simple statechart *S00*

(a) $type(S00) = OR$

(b) $S0 \in children(S00) \wedge type(S0) = AND$

(c) $children(S0) = \{S1, S2\}$ and $type(S1) = type(S2) = OR$.

$S1$ and $S2$ are the orthogonal components of $S0$.

5.2.3.3 Definition of a sequential statechart (OR-statechart)

In this Subsection 5.2.3.3 we give the definition of a *sequential statechart* (also called *OR-statechart*). The idea behind this definition is to construct a statechart model such that it consists of *OR*-states only, i. e., *AND*-states are not allowed. Such an *OR-statechart* has the advantage that it can easily be represented as a graph (called 'OR-graph', cf. Subsection 5.2.4). Above all, the transformation of a statechart which consists of *OR*-states only into an 'OR-graph' has a remarkable characteristic with respect to the states[12] (vertices) and transitions (edges): it is a one-to-one relationship. Informally, an *OR-graph* is a *labelled directed graph* whose nodes are type of *OR*. The derived *OR*-graph will be required in the phase of testing statechart models, e. g., when developing efficient algorithms which can be used to generate test paths in a statechart model. Furthermore, such *OR*-graphs can be used in the process of computing 'legal configurations' of a statechart (cf. Definition 5.27, page 67).

The definition of an *OR-statechart* is stated as follows:

Definition 5.13 *A simple statechart* $Sct = (S, r, children, type, \mathcal{H}, hist, default, \mathcal{T})$ *is called an* **OR-statechart** *iff* $\forall s \in S : children(s) \neq \emptyset \Leftrightarrow type(s) = OR$*, i. e., AND-states are not allowed.*

Example of OR-statechart

Example 5.1 An example of an OR-statechart is depicted by Figure 5.4. Figure 5.4 shows an OR-statechart *S0*. The example illustrates that some states of an OR-statechart are decomposed into further states. Due to lack of space these are indicated by the symbol @ before the state name, e. g., the decomposition (refinement) of states @S2, @S3 and @S6 is given later in Figure 5.5, Figure 5.6 and Figure 5.7, respectively.

Indeed, Definition 5.13 imposes certain restrictions on the statechart Definition 5.11. In particular, it does not allow *AND*-states. However, this restriction to the statechart language is not a restriction of the modelling power of statecharts because every *AND*-state can be represented as an *OR*-state (as will be demonstrated in Section 5.4.3).

The following terms are necessary in the establishment of relationships between a statechart and a digraph. In addition, they are required for the description of a *very simple OR-statechart* which follows in Subsection 5.2.3.4. Recall that a state $s \in S$ is basic iff $type(s) = BASIC$, i. e., $children(s) = \emptyset$ (cf. Definition 5.11, (4)(*c*)).

Definition 5.14 *A statechart state* $s \in S$ *which is not a basic state is called a* **super-state (hierarchy state)***.*

Notation 5.1 *The set of basic states of a statechart Sct will be denoted by* $S_{bas}(Sct)$ *or simply* S_{bas} *if the statechart Sct is determined by other circumstances. Similarly, the set of all super-states will be denoted by* $S_{sup}(Sct)$ *or simply* S_{sup}*.*

[12] without the state r

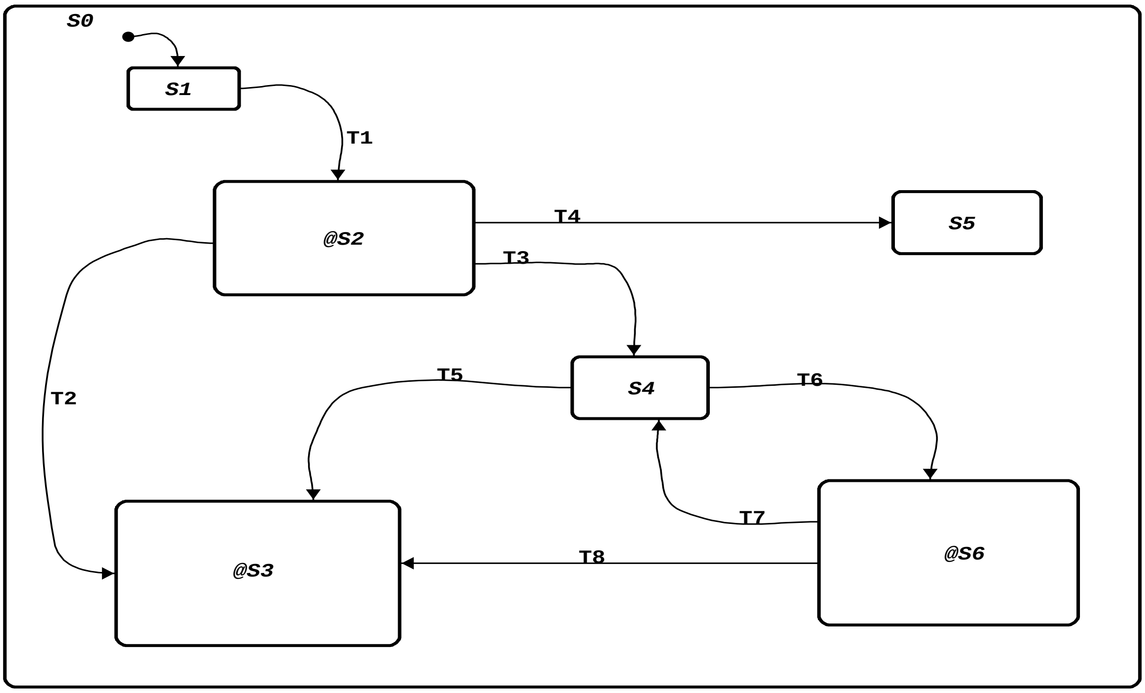


Figure 5.4: Statechart *S0*

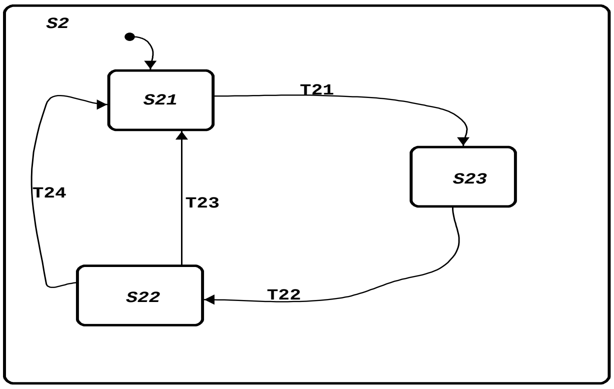


Figure 5.5: The decomposition of *S2*

As an example, consider Figure 5.4 again. States $S1, S4$ and $S5$ are basic states of $S0$, whereas states @S2, @S3 and @S6 are super-states of $S0$.

5.2.3.4 Definition of a very simple *OR*-statechart

The goal of this Subsection 5.2.3.4 is to define a *very simple OR-statechart* in which all states with the exception of the root are basic.

In the following, only *OR*-statecharts are considered (cf. Definition 5.13). *AND*-statecharts will be handled later in Subsection 5.4.3. For a component (*OR*-state) of a statechart, only the associated *basic* states and the *super-states* in this component will be considered. This will be captured by the definition of an *abstract OR-statechart* as

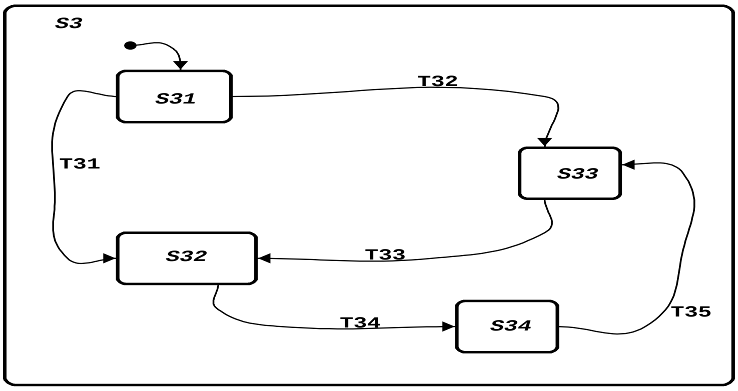


Figure 5.6: The decomposition of *S3*

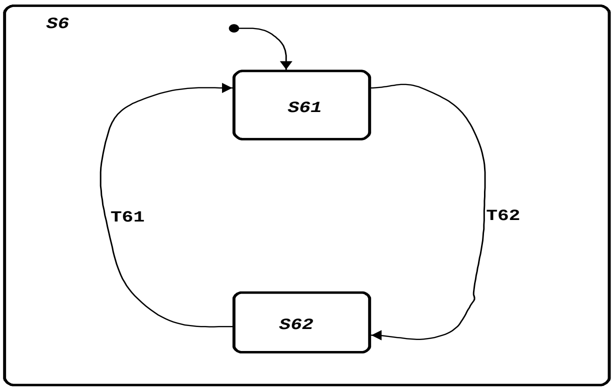


Figure 5.7: The decomposition of *S6*

follows:

Definition 5.15 *Let* $Sct = (S, r, children, type, \mathcal{H}, hist, default, \mathcal{T})$ *be an OR-statechart (cf. Definition 5.13) and* $s \in S$.

(1) The **abstract** *OR***-statechart** *which will be denoted by* $abstr(Sct(s))$ *is defined as follows:* $abstr(Sct(s)) = (S', r', children', type', \mathcal{H}, hist', default', \mathcal{T}')$

(a) $S' = \{s\} \cup children(s)$

(b) $r' = s$

(c) $children', type', hist'$ *and* $default'$ *are restrictions of Sct to* S' *instead of* S

(d) $\mathcal{T}'$ *is the restriction of* $\mathcal{T}$ *to* $S' \times L \times (S' \cup \mathcal{H})$.

(e) $\forall s' \in S' \backslash \{s\} : type'(s) := BASIC$.

(2) The set of all states $S' \backslash \{s\} = children(s)$ *on the corresponding abstraction level is decomposed into the following sets:*

(a) $S'_{bas}(s) = \{z \in S' \mid type(z) = BASIC\}$ *is called the set of all basic states on the corresponding abstraction level, and*

(b) $S'_{sup}(s) = \{z \in S' \mid z$ *is a super-state*$\}$ *(i. e., $type(z) \neq BASIC$) is called the set of all super-states on the corresponding abstraction level.*

Note: $\mathcal{T}' = \{(\{x\}, l, \{z\}) \in \mathcal{T} \mid x \in S', z \in S'\}$ *is the set of all transitions on the corresponding abstraction level excluding the internal transitions of $S_{sup}(s)$ (i.e., transitions inside a super-state are not considered).*

Remark 5.5 Although in the original statechart *Sct* the elements z of S'_{sup} are non-basic ($type(z) \neq BASIC$) and thus consist of substates, in Definition 5.15 of the abstract *OR*-statechart we consider them as 'black boxes' on the corresponding abstraction level, i. e., they are basic states in the abstract *OR*-statechart ($type'(z) = BASIC$). The idea behind this is to demand a 'clean' type of modelling which does not allow any 'interlevel-transitions' (cf. Subsection 5.3.1, page 59). Indeed, constructing an abstract *OR*-statechart is one way of constructing a simpler *OR*-statechart (called "very simple *OR*-statechart") from an *OR*-statechart.

In particular, very simple *OR*-statecharts will be helpful in the process of computing 'basic configurations' of a statechart as well as for the test case derivation. The definition of a *very simple OR-statechart* is given as follows:

Definition 5.16 *A simple statechart Sct* $= (S, r, children, type, \mathcal{H}, hist, default, \mathcal{T})$ *is called a* **very simple *OR*-statechart** *iff*

$\forall s \in children(r) : type(s) = BASIC.$

Theorem 5.1
The abstract OR-statechart $abstr(Sct(s))$ of an OR-statechart Sct is a very simple OR-statechart.

Proof
We have to show that all the states (without the root) of $abstr(Sct(s))$ are considered basic. This is given by Remark 5.5, i. e., $\forall z \in S' \backslash \{s\} : type'(s) := BASIC$, since basic states can have no children by Definition 5.11(4)(*b*). This corresponds to the requirement $\forall s \in children(r) : type(s) = BASIC$ in Definition 5.16. □

Example
Figure 5.8 shows a very simple *OR*-statechart *S00*.

After giving various definitions of a statechart, we now proceed to Subsection 5.2.4 to establish the relationship between a formal statechart and a *labelled directed graph* to provide the necessary background which will be required in Subsections 5.4.3 and 5.4.4.

5.2.4 Relationship between a statechart and a labelled digraph

The purpose of this subsection is to establish the necessary background which will be helpful for the development of methods and concepts to overcome the test problems, e. g., developing algorithms which may be used to compute 'legal configuration' sets

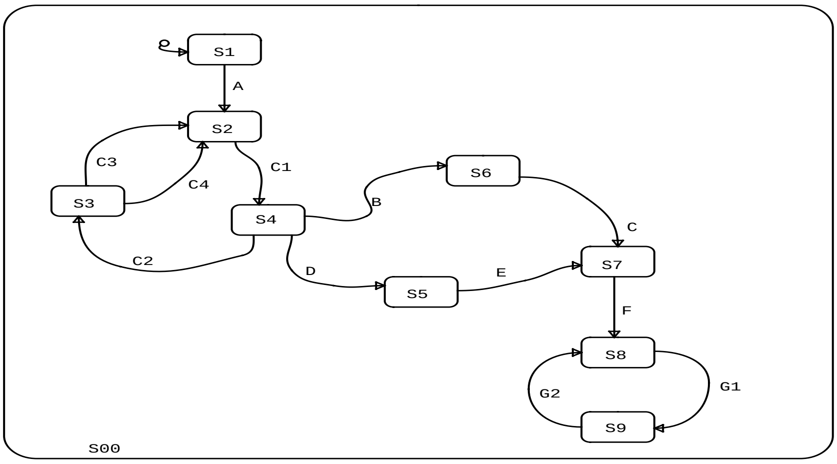


Figure 5.8: Very simple *OR*-statechart *S00*

(cf. Definition 5.32, Subsection 5.4.2). As already stated in the Preface[13], *statecharts* serve as a framework for the development of practical methods. Since the application of the developed concepts and methods should be easily extendable to different specification languages, concepts based on graphs give a more promising solution. Therefore, in this subsection the relationship between a formal statechart[14] and a *labelled directed graph*[15] is investigated. For a better comprehension, Definition 2.5 of a *labelled directed graph* is rewritten as follows:

Definition 5.17
A **rooted labelled directed graph** (*a* **digraph** *for short*) *is a 5-tupel* $G = (V, v_0, E, inc, label)$, *where*

(a) V *is a nonempty finite set of so called* **vertices** (**nodes**),

(b) $v_0 \in V$ *is called the* **root** *of* G,

(c) E *is a nonempty finite set of so called* **edges**,

(d) $inc : E \rightarrow V \times V$ *maps each edge to its source and target node.*
If $inc : E \rightarrow V \times V$ *is not injective, then* G *is said to have* **multiple edges** e_i, e_j, *i. e.,* $inc(e_i) = inc(e_j)$, $e_i \neq e_j$; *an edge* e *is called* **simple edge**, *if* $inc(e_i) \neq inc(e)$ *for all edges* $e_i \neq e$.

(e) $label : E \rightarrow L$, *where* L *is a finite set* (*of so called* **labels**), *is a mapping with the following restriction:* $\forall e_i, e_j \in E$: $inc(e_i) = inc(e_j)$ *and* $e_i \neq e_j \Rightarrow label(e_i) \neq label(e_j)$.
Note: *Definition 5.17 resembles Definition 2.5, but introduces a root* v_0, *too.*

[13] of this thesis
[14] cf. Section 5.2 Definition 5.11
[15] Chapter 2 gives a brief view about concepts and notions of the graph.

As mentioned in Subsection 5.2.3 above, the basic idea behind the concepts to be developed is to consider a statechart which consists of only *OR*-states and to describe it as a digraph. We also remind the reader, that such a restriction to the statechart language is not a restriction of the modelling power of statecharts because every *AND*-state can be represented as an *OR*-state (as will be shown in Subsection 5.4.3).

Definition 5.18 *Let* $Sct = (S, r, children, type, \mathcal{H}, hist, default, \mathcal{T})$ *be an* OR*-statechart (cf. Definition 5.13). Then* $G = (V,\ v_0,\ E,\ inc,\ label)$ *is the* **associated digraph***, also called* **associated OR-graph***, where*

(a) $V := (S \backslash \{r\}) \cup \mathcal{H}$ *(where* S *is the set of states and* $\mathcal{H}$ *is the set of history symbols of* Sct*),*

(b) $v_0 := s_0 = default(r)$ *(where* s_0 *is the default state (start state) of* Sct*),*

(c) $E := \mathcal{T}$ *(where* $\mathcal{T}$ *is the set of transitions of* Sct*),*

(d) $inc : E \rightarrow V \times V$ *and* $label : E \rightarrow L$ *are defined as follows:*
for a transition $t = (x, l, y) \in \mathcal{T} = E$*:* $inc(t) = (x, y)$ *and* $label(t) = l$*.*

Definition 5.19 *Let* $G = (V,\ v_0,\ E,\ inc,\ label)$ *be the* **associated digraph** *of the* OR*-statechart* Sct*.* $G' = (V', v_0', E', inc', label')$ *is called the* **super-graph** *of* G *with respect to* S'*, where*

(a) $V' := S'$*,* $E' := \mathcal{T}'$ *where* S' *and* $\mathcal{T}'$ *are defined as in Definition 5.15,*

(b) $v_0' := s_0' = default(s)$ *where* s_0' *is the default state (start state) of component* s *of the* OR*-statechart* Sct

(c) $inc' : E' \rightarrow V' \times V'$*, the restriction of* inc *to* E'*.*

(d) $label' : E' \rightarrow L$ *is the restriction of* $label$ *to* E'*.*

Remark: *The restriction of* inc *to* E' *is not necessarily injective.*

In Subsection 5.2.3.4 we noted that very simple *OR*-statecharts will be helpful in the process of computing 'basic configurations' of a statechart as well as for the test case derivation. Therefore, the following formal definition:

Definition 5.20
Let an OR*-state* s *of an* OR*-statechart* $Sct = (S, r, children, type, \mathcal{H}, hist, default, \mathcal{T})$ *like that of Definition 5.15 be chosen as a root of a* **very simple** OR**-statechart** $Sct(s)$*, then* $G = (V(s),\ v_0(s),\ E(s),\ inc(s),\ label(s))$ *is its* **associated simple OR-graph** *where*

(a) $V(s) := S' = \{s' \in children(s) \mid type(s') = BASIC\}$ *(where* S' *is the set of states of* $Sct(s)$*),*

(b) $v_0(s) := s_0 = default(r)$ *(where* s_0 *is the default state (start state) of* $Sct(s)$*),*

(c) $E(s) := \mathcal{T}' = \{(x,l,z) \in \mathcal{T}' \mid x \in S', z \in S'\}$ *(where* $\mathcal{T}'$ *is the set of transitions of* $Sct(s)$*),*

(d) $inc(s) : E(s) \rightarrow (V(s) \times V(s))$ *the restriction of inc to* $E(s)$*.*

(e) $label : E(s) \rightarrow L$ *is the restriction of label to* $E(s)$*.*

Remark: *The restriction of inc to* $E(s)$ *is not necessarily injective.*

Remark 5.6 Since all states are of type *OR*, all transitions of a very simple *OR*-statechart are of "type" (x, l, y), i. e., there is only one state in the source set and target set of the transition.

Theorem 5.2 *Let G be an associated OR-graph of a very simple OR-statechart Sct. Then the nodes of G represent all (possible) basic states of the corresponding statechart Sct and the labelled edges of G represent all transitions of Sct.*

Proof

(a) Correspondence of nodes of G and states of Sct: By Definition 5.16 $\forall s \in children(r) : type(s) = BASIC$, i. e., $S\backslash\{r\} = \{s \in S \mid s \in children(r)\{s \in S \mid type(s) = BASIC\}$, since basic states can have no children by Definition 5.11(4)(b). This corresponds to $V = S\backslash\{r\}$ in Definition 5.18.

(b) Correspondence of edges of G and transitions of Sct: The correspondence of edges and transitions is given by Remark 5.6 and Definition 5.18 (c) and (d). □

Nevertheless, since the elements of S'_{sup} are non-basic in the original statechart Sct, a mechanism is necessary which can be used to deliver the 'basic configurations' of the original statechart Sct. In the following, however, we are not — yet — interested in dealing with the low-level specification, but rather with the high-level specification only which is represented by a corresponding abstract OR-statechart Sct.

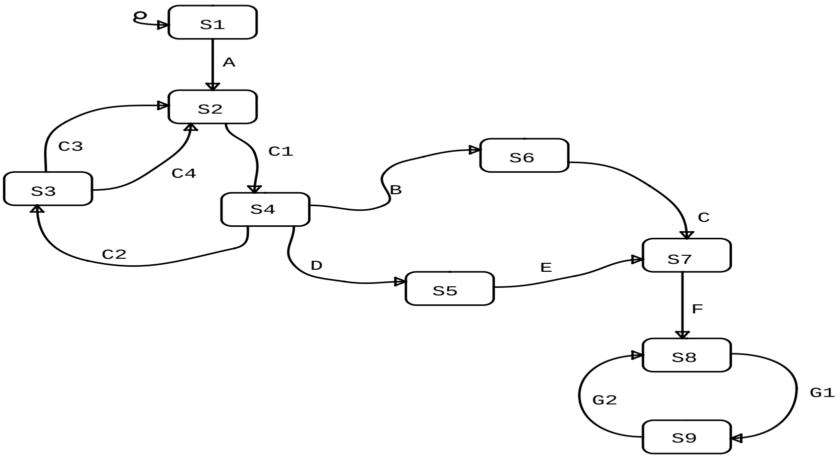


Figure 5.9: The simple OR-graph G of very simple OR-statechart *S00*

Recall Figure 5.8 on page 49 which depicts a very simple *OR*-statechart *S00*. Figure 5.9 shows the graphical representation of the associated OR-graph G of the very simple *OR*-statechart *S00*. Principally, the OR-graph G is the same as the *OR*-statechart *S00*, and this is so because of Theorem 5.2.

5.2.5 Syntax of statecharts with transformations

For a simple statechart a single transition may be described syntactically by an *arc* (cf. Definition 5.11). However, the graphical language as supported by STATEMATE allows a 'transition' to consist of a number of linked transition segments which are the labelled arrows that connect states and connectors of various kinds. Therefore, in the next Subsection 5.2.5.1 an overview of STATEMATE's representation of connectors is given. The technical mechanism of connectors may create problems, e. g., when a transition segment a_i ends at a connector[16] then this is not defined in the sense of Definition 5.11. This is due to the fact that the target state is neither a state nor a history symbol (cf. Definition 5.11, part (8)(c)). Another problem which is rather semantical may arise when the simulation tool STATEMATE tries to carry out a sequence of transition segments. In particular, if only the initial part of transition segments is enabled this may lead to an incomplete transition execution. Consequently, the *transformation* of such representations into a special statechart form called Canonical Normal Form CNF is necessary. The transformation is based on the notion of 'full compound transitions', and will be described in Subsection 5.2.5.2.

5.2.5.1 Connectors

A connector is a technical mechanism described by STATEMATE to economize the number of arrows, to reduce clutter, and to provide a clearer and more intuitive *graphical* representation. Connectors are a new sort of nodes. Arcs in a statechart may lead from states (or default states or history symbols) to connectors, or connectors to connectors, or connectors to states. But each connector must be "reachable" from a state or must reach a state by a sequence of arcs. Default states may be replaced by so called **default connectors**. For example, when statecharts contain transitions with much functionality in common, connectors (or junction connectors) are used to separate common parts. We remind the reader that in the following figures, and in many others elsewhere in this thesis, we give names to transition arcs, e. g., $a1, a2, a3, a4$ and $a5$ in Figure 5.10. Note that these are not part of the syntax but are added solely to allow us a better exposition. Figure 5.10 depicts two connectors as realized by STATEMATE. The first one connects transition segment $a1$ (from state $S1$) with $a2$ (ending at state $S2$). The second one is an example of a default connector and connects transition segment $a5$ (which is taken on entering $S2$) with either transition segment $a3$ or $a4$. In this case, if the system is in state $S1$, and event $ev1$ and event $ev2$ are generated simultaneously and condition $C3$ is true, then the set of transition segments $(a1, a2, a5, a3)$ will be carried out. On the other hand, if the system is in state $S1$, and event $ev1$ and event $ev2$ are generated simultaneously

[16] in this particular case, if it is not a history symbol

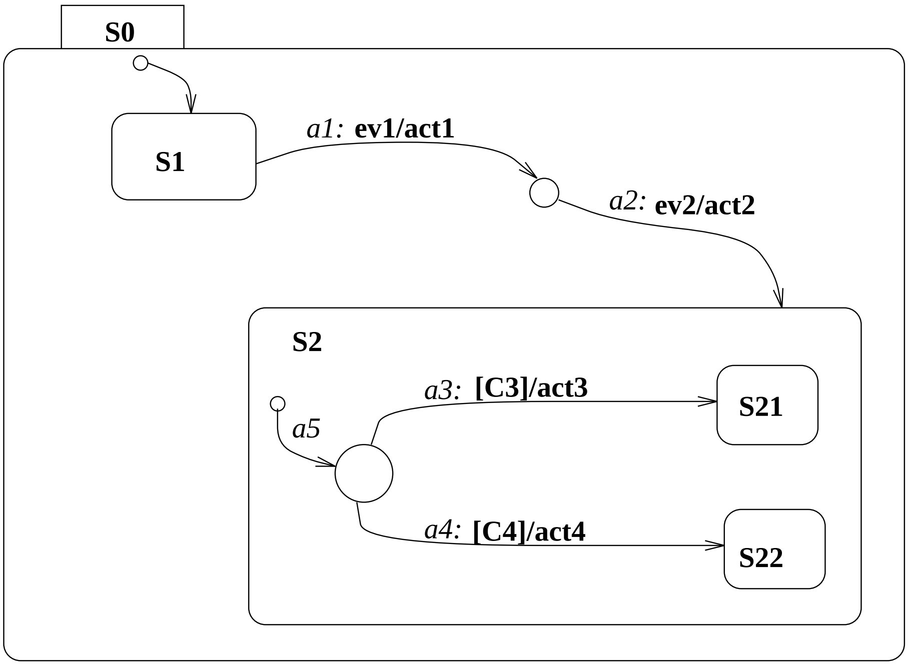


Figure 5.10: Some connectors realized by STATEMATE

and condition $C4$ is true, then the set of transition segments $(a1, a2, a5, a4)$ will be carried out. When the transition segments $(a1, a2, a5, a3)$ are carried out, then the actions $act1; act2; act3$ are also executed simultaneously. Similarly, the actions $act1; act2; act4$ are executed simultaneously when the transition segments $(a1, a2, a5, a4)$ are carried out.

Now consider a case in which the system is in state $S1$, and event $ev1$ and event $ev2$ are generated simultaneously but both conditions $C3$ and $C4$ are false. The simulation tool STATEMATE notifies the user that the basic states $S21$ and $S22$ could not be reached after the 'initial compound transition' $(a1, a2)$ was enabled. But even then, it will execute the initial compound transition $(a1, a2)$, and only after reaching state $S2$ it will discover that it cannot enter the basic states. At this juncture, it will not 'roll back' to $S1$. Problems of this nature require an intricate concept — the notion of 'full compound transitions' — which will be formalised in Subsection 5.2.5.2.

5.2.5.2 Notion of full compound transitions

While in the graphical language as supported by STATEMATE a transition "segment" may consist of several components also called transitions (cf. Figure 5.10), the semantics that will be described in this thesis uses the notion of 'full compound transitions' (cf. Figure 5.11). The behaviour of the statechart of Figure 5.11 is equivalent to that of Figure 5.10. The sequences of transition segments $(a1, a2, a5, a3)$ and $(a1, a2, a5, a4)$ in Figure 5.10 build 'full compound transitions' $CT1$ and $CT2$, respectively as depicted in Figure 5.11.

The representation based on 'full compound transitions' is a **transformation** *of a statechart which consists of linked transition segments* (*the labelled arrows that connect*

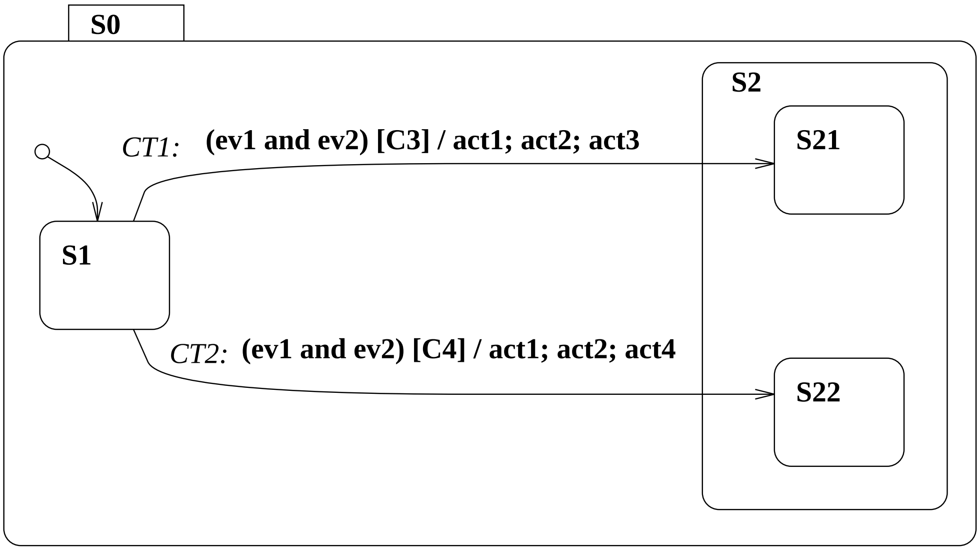

S0
S2
S21
S22
S1
CT1: (ev1 and ev2) [C3] / act1; act2; act3
CT2: (ev1 and ev2) [C4] / act1; act2; act4

Figure 5.11: Representing figure 5.10 in another way

states and connectors of various kinds) into a statechart in CNF. The statechart of Figure 5.11 represents the CNF of that of Figure 5.10.
A statechart in *CNF* has the following advantages:

(1) it leads to a concise description of the state where a full compound transition is enabled;

(2) it allows the definition of scope of the transition (cf. Subsection 5.4.4) which is used to deal with nondeterminism and to associate priorities to transitions.

These advantages create a fundamental basis (a 'good' framework) which allows the tester to assume that the transitions are unambigiuous.

The idea behind the notion of full compound transitions in which the (default) connectors 'vanish' from the original statechart can be summarized as follows:

Definition 5.21 *A* **basic compound transition** *(CT) is the result of the* **transformation** *of a maximal chain of transition segments, linked by connectors, that are replaced by a single transition.*

Furthermore, basic compound transitions can be divided into two types/parts:

Definition 5.22 *An* **initial compound transition** *is a CT whose source (start of the first transition segment) is a state, and a* **continuation compound transition** *is a CT whose source is a default state or a history symbol. The targets of both types of CTs are states or history symbol.*

The main purpose of full compound transitions is to determine the states to be entered when the transition is taken. According to the two definitions above, a **full compound transition** is a sequence of one initial CT and possibly several continuation *CT*s, and may be summarized as follows:

Definition 5.23 *A* **full compound transitions** (**full** *CT for short*) *is sequence of one initial CT ct_1 and possibly several continuation CTs $ct_2, \dots, ct_n$ which satisfies the following condition:*

The start connector of ct_i is a default connector or default state of a state s_i which is the final state of $ct_{i-1}, i = 2 \dots, n$.

Definition 5.24 Algorithm for the transformation of a statechart with connectors into a CNF **statechart***:*
A full compound transition CT is replaced by a single transition t, where the events (*respectively the conditions*) *of the transition t are the conjunction of the events* (*respectively the conditions*) *of the constituent segments of CT, and the action of the transition t is the concatenation of the actions of CT.*

Remark 5.7 Definitions 5.23 and 5.24 describe the syntactical transformations of a statechart which consists of linked transition segments into a statechart in CNF. In Subsection 5.4.4, however, we present a rather semantical definition of full compound transitions (cf. Definition 5.42).

Theorem 5.3 *A statechart with connectors which is transformed by the algorithm of Definition 5.24 is a simple statechart* (*cf. Definition 5.11*).

Proof idea: The proof idea is given using the following example:
Example of initial and full compound transitions

In Figure 5.10, the sequence of transition segments $(a1, a2)$ is an initial CT, and the sequences of transition segments $(a5, a3)$ and $(a5, a4)$ are continuation CTs. Of course, to lead to a full basic configuration, $(a1, a2)$ must be accompanied by either $(a5, a3)$ or $(a5, a4)$ which consequently form two full CTs, $(a1, a2, a5, a3)$ and $(a1, a2, a5, a4)$.

It is easy to show that the statechart of Figure 5.11 represents the CNF of that of Figure 5.10.

Now by Definition 5.23, the start connector of ct_i is a default connector or default state of a state s_i which is the final state of $ct_{i-1}, i = 2 \dots, n$. The sets of transition segments $\{a1, a2, a5, a3\}$ and $\{a1, a2, a5, a4\}$ in Figure 5.10 build full compound transitions $CT1$ and $CT2$, respectively as depicted in Figure 5.11. $CT1$ and $CT2$ can be described according to Definition 5.11 part (8). This is due to the fact that each transition arc of a CNF statechart has at most one source state and one target symbol. We also notice that the transformation algorithm of Definition 5.24 solely replaces a sequence of one initial CT and possibly several continuation CTs and does not affect any other aspects of the statechart. Obviously, the statechart of Figure 5.11 is a simple statechart.
(End of **Proof idea** for Theorem 5.3).

The rest of this thesis concentrates on full CTs.

5.2.6 Summary of Section 5.2

In this subsection we present a summary of Section 5.2. Section 5.2 begins with giving basic definitions over which the syntax of hierarchical statecharts is established. At the beginning of Chapter 5 we pointed out that a number of issues are not accurately defined in [51]. Some of these issues are dealt with in Subsection 5.2.1. For example, the *hierarchy function* $\rho : S \rightarrow 2^S$ defined in [51] delivers for $\rho^i(s)$ a set of states for $0 \leq i \leq 1$ and a set of subsets of states for $i > 1$. This definition is thus not quite applicable. Therefore, in Definition 5.3 we describe the hierarchy function $children^i(s)$ which delivers for each state $s \in S$ its *descendants* on *level* i with respect to root r. The reverse hierarchy function $parent^i(s)$ which computes for a state $s \in S$ its ancestor on *level* i is given by Definition 5.5. In addition, we define the *labels* of transitions, i. e., expressions of *events*, *conditions* and *actions* (cf. Subsection 5.2.2).

In Subsection 5.2.3 we define subclasses of statecharts. Definition 5.11 describes a *simple statechart* which serves as the fundamental definition over which several concepts are established in the course of this thesis. In comparison to Harel, Pnueli, Schmidt and Sherman's definition of statechart (cf. [51]), Definition 5.11 requires that the root state of a statechart must be of *OR*-type and also that the substates of an *AND*-state are always of *OR*-type. In particular, for a simple statechart, we define $\mathcal{T} \subset S \times L \times (S \cup \mathcal{H})$ as the set of **transition arcs**, instead of $\mathcal{T} \subset 2^S \times L \times 2^{S \cup \mathcal{H}}$ the set of **transitions** (cf. Definition 5.28). Thus, in a simple statechart the arcs form a one-to-one relationship (cf. Figure 5.1 on page 35). In other words, a transition may be described syntactically by an *arc* e. g., $t1 = (S1, e1, S2)$ corresponds to an arrow with label $e1$ from a single state $S1$ to single state $S2$.

Using Definition 5.11 of a simple statechart, we constuct the definition of *orthogonal statechart/orthogonal component* (cf. Definition 5.12). This will be required for the development of methods and concepts for transforming *AND*-states into *OR*-states (cf. Subsection 5.4.3). As another important foundation of the test methods and concepts developed in this thesis, we describe a *sequential statechart* (also called the *OR-statechart*) in Definition 5.13. The basic idea behind this definition is to construct a statechart model in such a way that it consists of *OR*-states only, i. e., *AND*-states are not allowed. The OR-statechart given by Definition 5.13 can be represented as an **associated OR-graph** (cf. Definition 5.18). The derived *OR*-graph will be required in the phase of testing statechart models, e. g., when developing efficient algorithms which can be used to generate test paths in a statechart model. Furthermore, such *OR*-graphs can be used in the process of computing 'legal configurations' of a statechart (cf. Subsection 5.2.4).

In the above context, we require a *very simple OR-statechart* in which all states with the exception of the root are basic. This is given by Definition 5.16. Very simple *OR*-statecharts will be helpful in the process of computing 'basic configurations' of a statechart as well as for the test case derivation.

The purpose of Subsection 5.2.4 is to investigate the relationship between a formal statechart[17] and a *labelled directed graph*[18]. For a better comprehension, Definition 2.5 of a *labelled directed graph* is rewritten (cf. Definition 5.17). The concept we have summarized so far can be represented diagrammatically as in Figure 5.12.

Finally, in Subsection 5.2.5 we deal with *"non-simple statecharts"*, i. e., statecharts that allow a 'transition' to consist of a number of linked transition segments which are

[17] cf. Subsection 5.2.3 Definition 5.11

[18] Chapter 2

the labelled arrows that connect states and connectors of various kinds. We discuss some problems which may be caused by STATEMATE's representation of connectors. For example, when a transition segment a_i ends at a connector[19] then this is not defined in the sense of Definition 5.11. Another problem is rather semantical and may arise when the simulation tool STATEMATE tries to carry out a sequence of transition segments. In particular, if only the initial part of transition segments is enabled this may lead to an incomplete transition execution. As a possible solution to the noted problems above, we propose the *transformation* of such representations into a special statechart form called Canonical Normal Form CNF. The transformation is based on the notion of 'full compound transitions' and is described in Subsection 5.2.5.2. For this purpose, we develop an algorithm for the transformation of a statechart with connectors into a CNF statechart (cf. Definition 5.24). As a conclusion, we summarise in Theorem 5.3 that "A statechart with connectors which is transformed by the algorithm of Definition 5.24 is a simple statechart (cf. Definition 5.11)".

$$\begin{tikzcd}
\begin{array}{l}\textbf{simple statechart } Sct\\ \text{(Def. 5.11)}\end{array} \arrow[rr, leftrightarrow] \arrow[d] & & \begin{array}{c}\textbf{digraph } G\\ \text{(Def. 5.17)}\end{array} \arrow[d] \\
\begin{array}{l}\textbf{OR-statechart } Sct(s)\\ \text{(Def. 5.13)}\end{array} \arrow[rr, leftrightarrow] \arrow[d] & & \begin{array}{c}\textbf{OR-graph } G\\ \text{(Def. 5.18)}\end{array} \arrow[d] \\
\begin{array}{l}\textbf{abstract OR-statechart } abstr(Sct)\\ \text{(Def. 5.15)}\end{array} \arrow[rr, leftrightarrow] \arrow[d] & & \begin{array}{c}\textbf{super-graph } G\\ \text{(Def. 5.19)}\end{array} \arrow[d] \\
\begin{array}{l}\textbf{very simple OR-statechart } Sct(s)\\ \text{(Def. 5.16)}\end{array} \arrow[rr, leftrightarrow] \arrow[dr] & & \begin{array}{c}\textbf{simple OR-graph } G\\ \text{(Def. 5.20)}\end{array} \arrow[dl] \\
& \textbf{Theorem } 5.2 \text{ (applied on high-level specifications)} &
\end{tikzcd}$$

NB: Orthogonal statechart (Def. 5.12) is omitted here because the basic idea is to construct OR-graphs which represent OR-statecharts. AND-statecharts will be transformed later into OR-statecharts (as demonstrated in Section 5.4.3).

Figure 5.12: Relationship between a statechart and a digraph

5.3 A glance at the variants of statecharts

Immediately after the publication of the statecharts formalism extensive investigations began in order to improve its syntax and semantics (e. g., the step semantics proposed by Pnueli and Shalev in [131]). This has led to many variants of syntax or semantics of statecharts proposed in the literature. Amongst others are *Argos* [101, 102], *SpecCharts* [111], *Modecharts* [75], *RSML* [95], as well as those in [69, 70, 78, 158]. The survey

[19] in this particular case, if it is not a history symbol

of von der Beek [158] e. g., discusses around 20 variants. In it a list of 19 issues which possibly lead to various proposals for the semantics of statecharts is given. In this thesis, however, all those items cannot be discussed. Therefore, in the next Subsection 5.3.1 only those which are relevant to the development of the test concepts in this thesis will be outlined. Furthermore, in Subsection 5.3.2 some statecharts variants are compared.

5.3.1 Discussing syntactic and semantic problems of statecharts

As mentioned above, in this subsection syntactic and semantic problems of statecharts are described and the approaches intended to overcome them are evaluated. In the course of this discussion only particular variants will be picked out for comparison with the original statecharts (cf. [50, 51, 54]). These will include the following formalisms: *Argos* [101, 102], *RSML* [95], *Esterel* [11] and [69, 70, 71, 78, 79, 131, 158].

In Subsection 1.2.2 (see particularly pp. 5ff.) we explained that real-time reactive systems pose specific problems in defining languages to specify and execute them. Among them, we discussed the principle of the **perfect synchrony hypothesis**, and its importance for high-level specification where one is not — yet — interested in dealing with the implementation details on one hand, but on the other hand would like to specify in an accurate, non-fuzzy way. The application of the synchrony hypothesis is very common in the *operational semantics* of statecharts as used in STATEMATE. We also discussed two communication modes: the **synchronous time model** in which the communication with the environment is performed after each basic step, and the **asynchronous time model** where the communication is achieved by allowing several basic steps to take place within a single point in time. Then the principle of *causality* was introduced. **Causality** means that for every event generated at a particular instant of time, there must be a (causal) chain of events leading backwards to the action that generated this event (see also [131]). Another important property which was discussed is the *instantaneous state.* This principle states that an **instantaneous state** can be entered and exited at the same instant of time. Finally, we discussed **durability of events**. This problem concerns the question of whether an event is durable for more than "an instant of time".

Now, we extend the list of problems with the following:

(i) **Modularity**
A *module* is a relatively independent part of a system, in essence a reactive system in its own right. *Within* modules the principle of causality holds, making it possible to develop a smaller subsystem in an intuitive, operational manner. *Between* modules, however, it is the principle of 'modularity' that holds, enabling to keep a global understanding of the system. According to [71], *modularity* means that all parts of the system should be treated symmetrically. The interface between the environment and the system should be the same as the interface between the parts of the system itself. Also, every part of the system should have the same view of events occurring in the total system *at any moment.* As a result, the communication mechanism between the subsystems is the immediate, asynchronous broadcast[20]. This mechanism

[20] If the time for distributing events is relatively high, e. g., in widely distributed systems, then one

is covered by the statecharts semantics described in [54].

(ii) **Inter-Level Transitions**
The term *inter-level transitions* is used in the literature to refer to transitions that cross the borderlines of hierarchy states. Recall the simple Coffee Vending Machine, page 28. The transitions with the labels *coffee-empty* or *cup-empty* from state *Coffee-idle* or *Cup-idle*, respectively, to state *Empty* are examples of inter-level transitions. As supported by STATEMATE, this mechanism can be seen as a 'goto' method which allows an arbitrary movement of control across the state hierarchy.

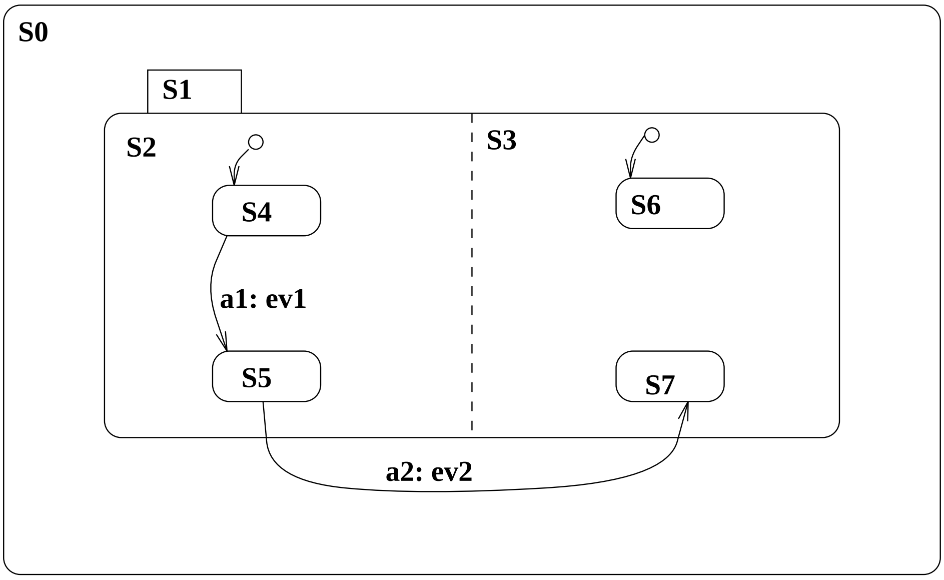


Figure 5.13: Exiting and entering state $S1$?

In particular, when an inter-level transition is dealt with, one has to accurately define the states that have been exited and those that are entered by taking the transition. Mainly, one has to know which *non-basic* states are exited and entered in the process of taking the transition. As will be seen later (cf. Subsection 5.4.4), it is quite important to know, for instance, the set of actions that may be called for when exiting and entering states. Consider Figure 5.13 which is an allowed statechart specification in STATEMATE. Does the system exit and reenter state $S1$ and which substate of $S2$ is entered when transition arc $a2$ is executed? How can the semantics deal with such and similar ambiguities?

Consequently, in this thesis an intricate concept — the notion of 'scope of transitions' associating priorities to transitions — will be formalised in order to deal with interlevel transitions which in general deny a structural operational semantics for statecharts. In [84], chapter 6, an interesting restriction on the modelling power of statecharts has been proposed to overcome this problem of not respecting the hierarchy of states. The idea behind this restriction is to support "good modelling" (see also [88]).

can introduce an explicit delay between the moment that an event is generated and the moment it will actually be sensed by the other subsystems.

Some languages, e. g., Argos [101, 102], SpecCharts [111], Modecharts [75] and RSML [95], forbid the explicit use of interlevel transitions in order to gain elegant structural operational semantics. Therefore, other mechanisms[21] are necessary to simulate the effect of interlevel transitions.

(iii) **Transition Refinement**
Usually, during the refinement of a state transition diagram a transition is replaced by a set of interconnected transitions. Following the synchronous time model, no sequence of transitions of an *OR*-state may be executed during one instant of time. Therefore, the refinement of transitions creates problems because the execution of a transition of "unrefined" statecharts may have transitions whose execution time is positive (non-zero time). The reason for this is that in each non-instantaneous state time must progress before the state can be left.

This refinement problem can be solved by allowing instantaneous states. In this procedure, a transition can be replaced by a sequence of transitions connected by instantaneous states. The complete sequence is then executed in zero time.

In the asynchronous time model this is not a problem since several transitions may be executed at the same instant of time.

(iv) **Multiple Entered or Exited Instantaneous States**
Semantics which allow instantaneous states have the drawback that an infinite loop caused by repeatedly entering and exiting of instantaneous state is possible at one instant of time. In order to avoid this problem, the semantics can have a restriction which requires that an instantaneous state must not be entered twice at one instant of time.

(v) **Preemptive Versus Non-Preemptive Interrupt**
If $s \in S$ is an *OR*-state with substates s_i $(1 \leq i \leq n)$ then the set of transitions $\{t_1, \ldots, t_n\}$ with t_i starting from state s_i $(1 \leq i \leq n)$ and ending in a common state s' can be replaced by only one transition t'. The new transition t' starts from *OR*-state s and ends in state s'. The transition t' represents an *interrupt mechanism* which is either preemptive or non-preemptive. Figure 5.14 is used to illustrate the two mechanisms. The problem now is to determine the behaviour of a statechart using an interrupt mechanism. As an example Figure 5.14 with "interrupt" transition $t2$ is used — in case state $S11$ is active and the events $e1$ and $e3$ occur simultaneously. The interrupt is called *preemptive* if the execution of $t2$ prevents the execution of $t1$. It is called *non-preemptive* if the execution of $t2$ does not prevent the execution of $t1$. In the above context a preemptive mechanism is thus a high-level transition that prevents executing transitions on lower, encapsulated levels. This is defined by the use of priorities of transitions which in turn are determined by the 'scope of the transition'. Other forms of interrupt can be found in [158].

(vi) **Representation of communicating events**
The statechart concept of broadcasting (cf. Subsubsection 4.2.1.3 and (i) Modularity

[21] Argos, e. g., uses an encoding to simulate the effect of interlevel transitions.

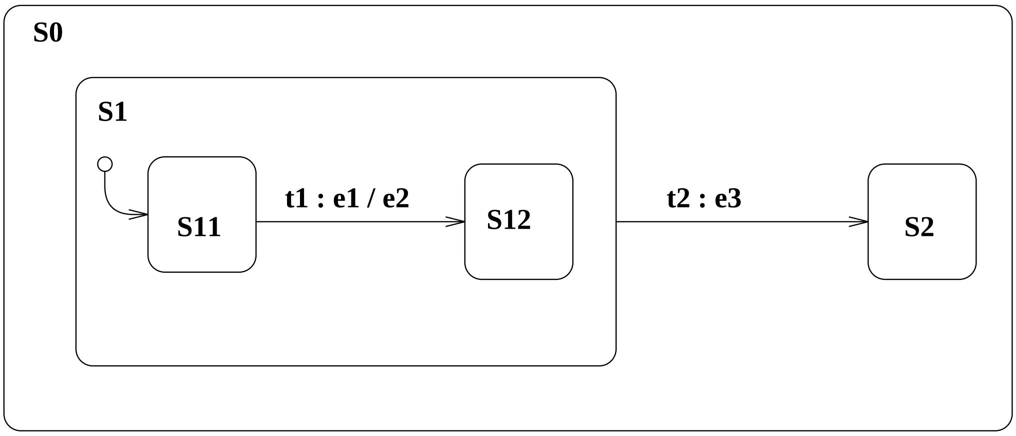


Figure 5.14: A statechart with preemptive or non-preemptive interrupt by transition $t2$

above) has the disadvantage that the communicating events are not represented graphically, i. e., lightly visible. The main advantage of this representation, however, is that a lot of transitions whose graphical representations (arcs) would cross one another will be avoided. Therefore, a test-tool is required to check whether the communication is modelled correctly. Another alternative can be to establish an extra representation for the "cross"-transitions. The limited scope of this thesis precludes an exhaustive treatment of this issue.

5.3.2 Comparison of statecharts variants

A detailed comparison of several statecharts variants was presented in the survey of von der Beek [158]. Even then there are still many other variants which have been developed later. A detailed study of all existing variants is beyond the scope of this thesis. Therefore, the main purpose of this subsection is to provide a brief overview and comparison of selected variants which fall into different classes of specifications. These are given in the following:

- *Argos* [101, 102]: The Argos language is derived from statecharts, but its description is more restricted in many ways. The most typical restriction is that no interlevel transitions are allowed. The advantage of this is that it uses the graphical notion of refinement better. In contrast to the statecharts, a *non-preemptive* mechanism is used. Another important difference between statecharts and Argos is that Argos, similar to the *Esterel* language [2, 11], does not allow any nondeterminism. All specifications that could lead to causal parodoxes are illegal. The compiler is used to check for such behaviours. Nevertheless, the restriction of nondeterminism cannot be completely accepted as an advantage since nondeterminism sometimes exists in the systems to be modelled. For such systems, it is more intuitive and easier to use a formalism which provides nondeterminism, e. g., statecharts.

- *Modecharts* [75]: This variant is based on a *Real-Time Logic* (*RTL*) formalism. The main feature of *Modechart* is that its specification can be translated into *RTL* specifications and safety properties can be verified using the *RTL* proof system (cf. [25]).

The main drawback of the analysis methods, however, is that the main part of the verification task is performed heuristically and in particular the correctness of the verification depends on the ability of the analyst. As with statecharts, Modecharts' graphical notation and its support for hierarchy enable the designer/tester of the system to produce specifications that are relatively easy to understand and to develop. Due to its dual approach, i. e., using the operational language and the descriptive language *RTL*, it is more expressive than a single language.

- *Timed Transition Models* (*TTM*) [120, 121, 122]: The approach of TTM is similar to that of Modechart. It supports the dual language approach. In this variant, *Real Time Temporal Logic* (*RTTL*) can be used to specify (safety) properties of timed transition models. As in Modechart, this method is supported by a set of tools which includes a verifier based on model-theoretic reasoning. In order to check a system model for a given property, the analyst expresses the model as *TTM* specification and the property in *RTTL* and then executes the verifier. Ostroff gave an algorithm for analysing (safety) properties that may be computerized. *TTM*, however, does not support hierarchical decomposition of states which is very important in the specification of complex real-time systems.

- *Requirements State Machine Language* (*RSML*) [95]: The main differences between *RSML* and statecharts are syntactical. Unlike statecharts, *RSML* e. g.,

  - does not support history and event selector connectives;
  - it features directed communication over channels instead of broadcast communication;
  - it uses *AND/OR* tables for entering with condition expressions;
  - it does not support event selector connectives (disjunctions of trigger events), etc.

  Although there appears to be no significant difference between RSML's and statecharts' dynamic semantics, they use different terms. The difference will be given in Subsection 6.2.3 page 102, when **Step semantic** is explained.

- *Timed* and *Hybrid Statecharts*: In [78], structured operational semantics are presented for *Timed* and *Hybrid Statecharts* which are generalisations of statecharts. The language introduces a textual representantion of statecharts to treat real-time and continuous behaviours. The language uses statements for delays, preemption, and timeouts.

  In statecharts those mechanisms have to be modelled by using a *discrete events* approach. The idea behind this approach is to describe a reactive system as a sequence of discrete events that cause abrupt changes (taking no time) in the state of the system, separated by intervals in which the system's state remains unchanged. This approach is particularly effective for describing the behaviour of programs and other digital systems.

- *Clocked Transition Systems* (*CTS*) [79]: The *CTS* model is a modification of the model in [78]. The new model provides a simpler style of temporal specification and verification and requires no extention of the temporal language. The model represents time by a set of clocks (timers) which increase uniformly whenever time progresses but can be set to arbitrary values by the system (program) transitions. The *CTS* can be viewed as a natural first-order extension of 'timed automata' (cf. [1]). Among the notable advantages of this new model are:

  (1) It leads to a more natural style of the specification by explicitly referring to clocks which in this particular case are another kind of system variables. In other words, these variables are used instead of introducing special new constructs such as the 'bounded temporal operators' proposed in *metric temporal logic* (*MTL*) (cf. [81]).

  (2) Many of the methods and tools developed for verifying untimed reactive systems (for example in [130]) intended for verifying real-time systems can be reused under the *CTS* model.

- *Extended Hierarchical Automata* (*EHA*) [107]: Extended hierarchical automata (*EHA*) were proposed in [107] as an intermediate format to facilitate the linking of new tools to the STATEMATE environment. The *EHA* formalism employs single-source/single-target transitions as in usual automata. In particular, interlevel transitions are not permitted in order to achieve a simple priority concept which facilitates computing the next step of an *EHA* when compared to statecharts. The idea behind *EHA* is to devise a simple formalism with a more restricted syntax than statecharts but nonetheless allowing to capture the richer formalism. As the priority concept of statecharts is very simple in case of non-interlevel transitions, investigations are made to reformulate interlevel transitions and hence handle them just as ordinary transitions with respect to the priority concept.

  We use Figure 5.15 and Figure 5.16 to illustrate the idea of *EHA* formalism. In the following, the 'dot notation' is used to select states of the statechart. Figure 5.15 shows a model of a TV-set as described in [69]. The transitions from *TV.SERVICE* to *TV.IS_OFF.DISCONNECTED*, from *TV.SERVICE* to *TV.IS_OFF.STANDBY* and from *TV.IS_OFF.STANDBY* to *TV.SERVICE* are interlevel transitions (cf. Subsection 5.3.1, part (*ii*)). Now assume that one of them is taken, then not only the source state is exited but also the parent of the respective source state. Similarly, when the target state is entered, the corresponding parent state is also entered. The idea of *EHA* formalism is to lift such transitions to the uppermost states that are exited and entered when a transition is taken. This is represented by Figure 5.16. Note, however, that the statechart specifications are more natural when compared to those of *EHA*. To explain this, consider Figure 5.16 again. Its specification is not quite that what is modelled in Figure 5.15. In Figure 5.15, when the TV-set is in *TV.SERVICE* and the event *Out* occurs, then we would like that the TV-set moves straight to the substate *TV.I_OFF.DISCONNECTED*. This information is not reflected by Figure 5.16. In extended hierarchical automata, therefore, a facility must be provided to explicitly specify such target states of transitions in order to

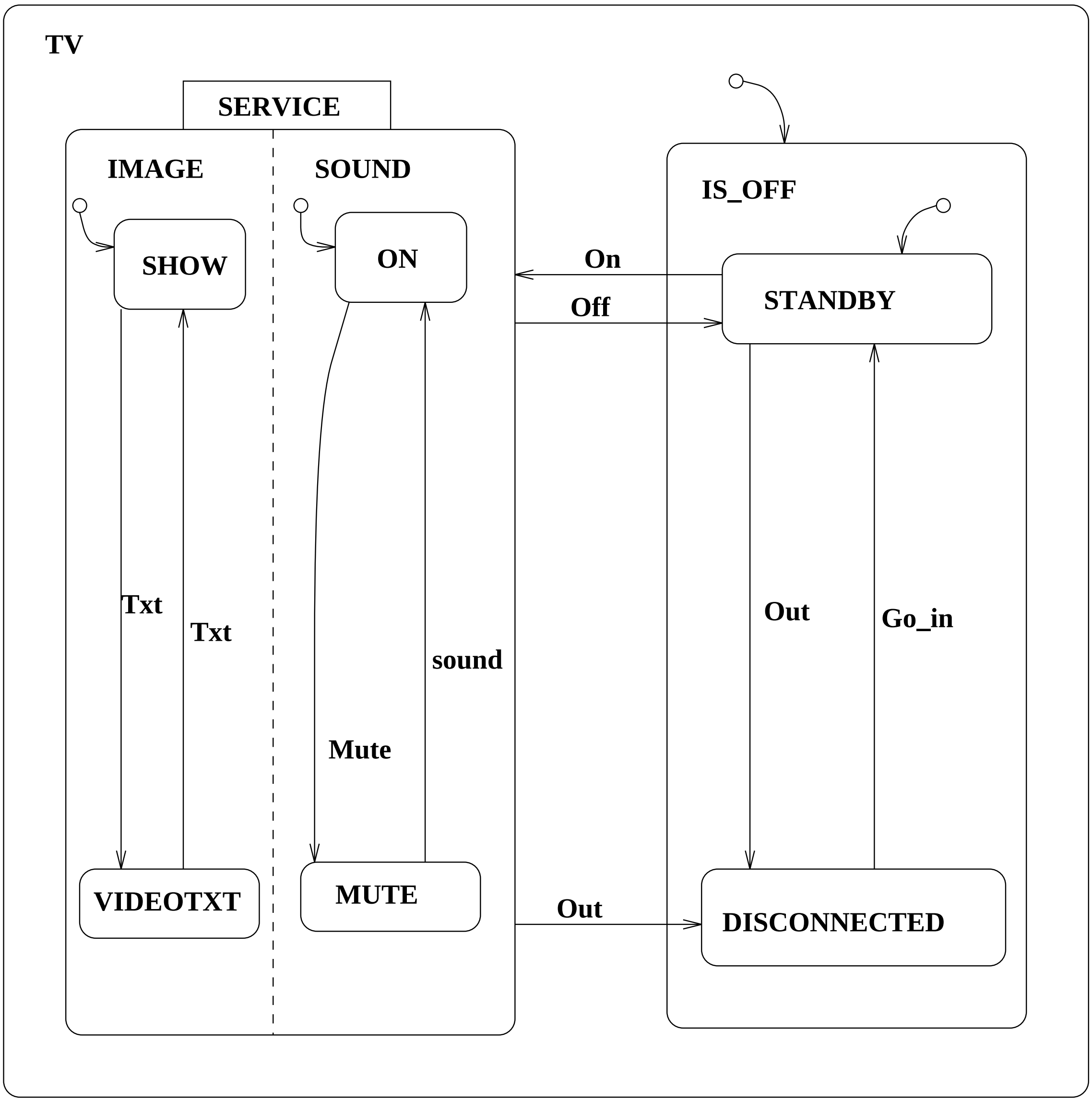


Figure 5.15: Statechart with interlevel transitions

preserve the infomation needed to determine the target states. *EHA*s thus cannot be used for specifying large and complex reactive systems, they can rather be applied to simplify the implementation of tools for statecharts.

5.3.3 Summary of Section 5.3

In this subsection we present a summary of Section 5.3. The main purpose of Section 5.3 is to glance at the variants of statecharts. This section is divided into two subsections. In the first Subsection 5.3.1 we describe syntactic and semantic problems of statecharts. At the same time, we evaluate the approaches intended to overcome these problems. There are several issues which possibly lead to various proposals for the semantics of statecharts. Amongst others, we discuss *inter-level transitions* the ones that cross the borderlines of

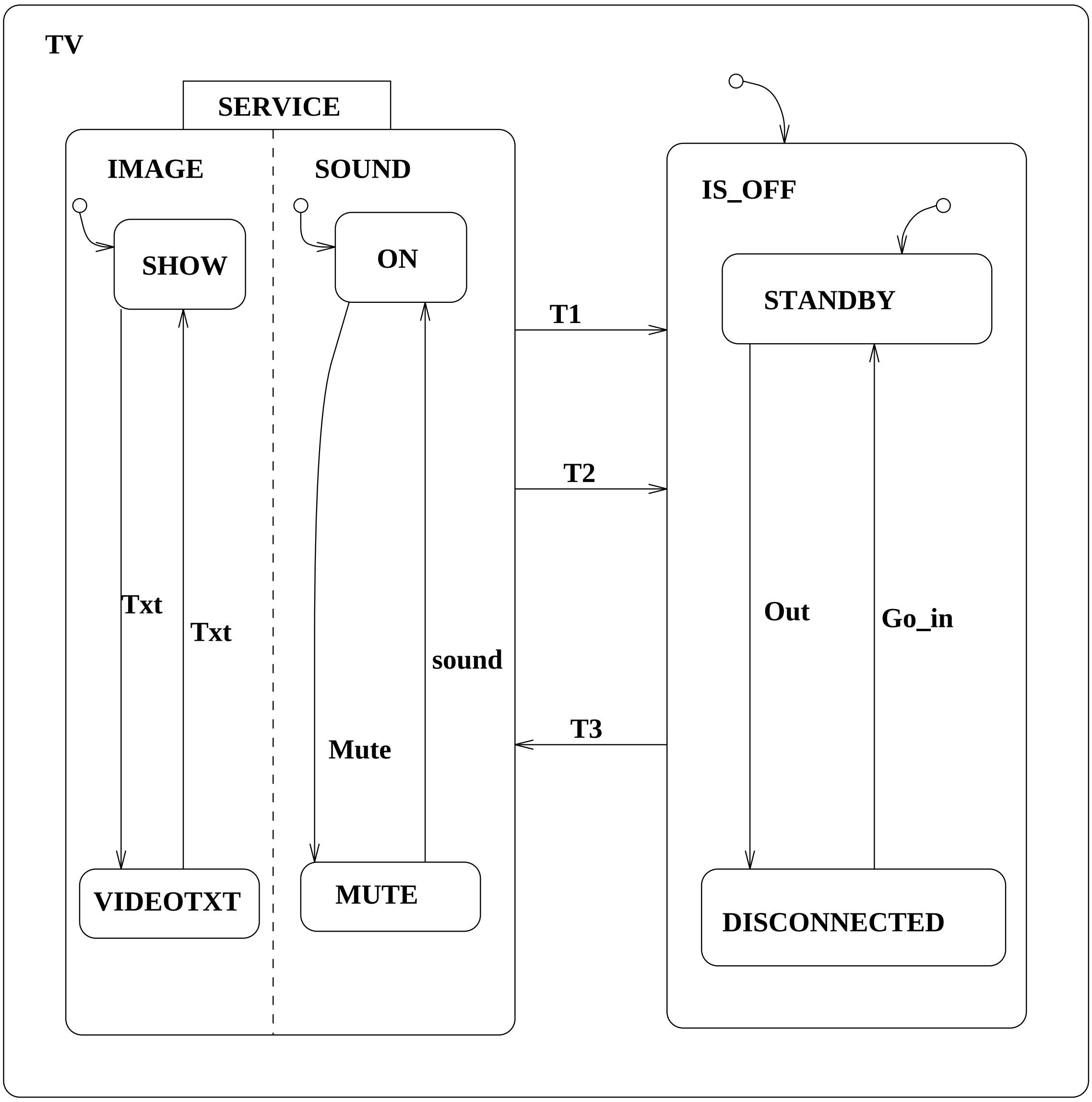


Figure 5.16: Statechart with desired extension

hierarchy states. In particular, we note that when an inter-level transition is dealt with, one has to accurately define the *non-basic* states that have been exited and those that are entered by taking the transition. These problems will be dealt with in Subsection 5.4.4 (see pp. 83ff.). Furthermore, it is essential to know the set of actions that may be called for when exiting and entering states. We will deal with these problems in Chapter 6 (see pp. 97ff.).

The main goal of the second Subsection 5.3.2 is to provide a brief overview and comparison of the original statecharts (cf. [50, 51, 54]) with some selected variants which fall into different classes of specifications. These include the following formalisms: *Argos* [101, 102], *RSML* [95], *Esterel* [11] and [69, 70, 71, 78, 79, 131, 158]. As a conclusion, we can say that there are no standard guideline of the statechart specifications. Above all, there are still a number of issues which have to be dealt with, e. g., inter-level transitions. Therefore, in the course of this thesis, we have to develop some kind of guideline for the statechart specifications over which the test methods and test concepts will be established.

5.4 Semantics of statecharts

In the previous Section 5.3 syntactical and semantical problems of statecharts have been described and the approaches intended to overcome them have been evaluated. Furthermore, several statecharts variants have been compared. As a result of this discussion, we can conclude that there are still series of problems which have to be dealt with.

That is, even though the syntax of statecharts has been defined, there remain behavioural concepts that are not obvious or even ambigiuous. The main purpose of this Section 5.4, therefore, is to develop a formal account of the semantics of statecharts that on one hand can be used to overcome some of the outlined problems in the section before, but also on the other hand will serve as a basis for the test methods and concepts within the rest of this thesis.

This Section 5.4 is organised as follows: In Subsection 5.4.1 we present the concepts of 'legal configuration'. In Subsection 5.4.2 we consider the case of a direct exit of an AND-state and give the formal concept of a 'legal' transition. In Subsection 5.4.3 we present techniques which can be used to transform AND-states into OR-states. In Subsection 5.4.4 we formalise the concept of 'scope' of transitions. In Subsection 5.4.5 we formally describe one of the most delicate aspects in the design of real life complex systems, namely 'conflicts' between two transitions. In this context, we then introduce the notion of *priority* which has been proposed in [54] as a solution to conflicting transitions.

5.4.1 State configurations

As seen in Subsection 1.2.2.1, page 4, a computation (run) of a statechart is a sequence of statuses. One component of the statuses is the set of currently active states that forms a 'configurations'. *Informally speaking, a* **configuration** *is a maximal set of states that the system* (*model*) *can be in simultaneously* (cf. [54]).

In the following, a definition of possible (state) configurations of a statechart will be given. The formal description of a configuration will be required in Subsection 5.4.4 when defining the 'scope' concept of transitions which is used to handle nondeterminism and to associate priorities to transitions. Above all, these definitions will as well be used to determine legal statuses (cf. Subsection 6.2.1).

Definition 5.25 *Given a statechart Sct with root state r, a* (**full**) **configuration** *is a set of states* $\mathcal{C} \in 2^S$ *satisfying the following requirements:*

(i) $r \in \mathcal{C}$, *i. e.,* $\mathcal{C}$ *contains r.*

(ii) $s \in \mathcal{C} \wedge type(s) = OR \;\Rightarrow\; |children(s) \cap \mathcal{C}| = 1$, *i. e., if C contains a state s of type OR, it contains* **exactly one** *of s's descendants.*

(iii) $s \in \mathcal{C} \wedge type(s) = AND \;\Rightarrow\; children(s) \subseteq \mathcal{C}$, *i. e., if* $\mathcal{C}$ *contains a state* s *of type* AND*, it contains* **all** *of* s*'s descendants.*

In the STATEMATE specifications, it follows that configurations are "closed upwards"; i. e., when the system is in any state s, it must also be in the parent state of s. This observation can be formulated as follows:

Remark 5.8 $\forall s \in (\mathcal{C} \backslash \{r\}) : parents^*(s) \subset \mathcal{C}$
In other words, $\forall s \in \mathcal{C}$ there exists a path from r to s (in the graph representing the children- or parent-function, cf. Definition 5.3 and 5.5) and all states of the path are also contained in $\mathcal{C}$.

Denotation 5.1 $\mathcal{C}_{full} : S \rightarrow 2^{2^S}$ *will be used to denote a function which maps a root state* r *to the set of configurations* $\mathcal{C}$ (*relative to* r) *satisfying the requirements of Definition 5.25.*
Note*:* $\mathcal{C}_{full}(r)$ *is abbreviated by* $\mathcal{C}_{full}$*.*

Whilst the above Definition 5.25 refers to a *full* configuration, it is in any case necessary to uniquely determine the *basic* configuration of a (full) configuration as the maximal set of basic states which are part of this configuration.

Definition 5.26 *A* **basic** *configuration relative to* $\mathcal{C}$*, denoted* $\mathcal{C}_{basic}$*, is defined as follows:* $\mathcal{C}_{basic} = \{s \in \mathcal{C} \mid children(s) = \emptyset\}$.

Remark 5.9 $parents^*(\mathcal{C}_{basic}) = \mathcal{C}$.

Hence, a basic configuration is a maximal set of basic states that a system can be in simultaneously.

Example for basic and full configuration

To illustrate the above definitions, recall Figure 4.3, page 28. In it, {*Cup-idle, Coffee-idle, Standby, Light-off*} is a basic configuration, and its full configuration also contains *Cup, Coffee, Controller, Light, On* and *SCVM* (the root state). {*Cup-idle, Coffee-idle, Light-off*} for instance is not maximal and is therefore not a basic configuration. {*Cup-idle, Coffee-idle, Coffee-busy, Standby, Light-off*} is accordingly not a 'legal' configuration since the system cannot simultaneously be in two descendants of the *OR*-state *Coffee*. Similarly, {*Off*} is a basic configuration, and its full configuration is the set {*Off, SCVM*}.

The formal concept of *legal* configurations may be formalised in the following way.

Definition 5.27 *A set of states* S *is a* **legal** *configuration if there exists a* (*full*) *configuration* $\mathcal{C}$ *such that* $S = \mathcal{C}$ *or* $S = \mathcal{C}_{basic}$*.*

5.4.2 Transitions in parallel states (nodes)

After the formalisation of the concept of legal configurations, we can now consider the case of a direct exit of an *AND*-state. Recall Definition 5.11, page 41 which describes for a simple statechart a single transition arc $a = (s, l, t)$ syntactically. Now, since in case of a direct exit of an AND-state the source of a transition is a set, part (8) of Definition 5.11 must be revised in order to compute transitions in parallel states (nodes). $\mathcal{T}$ will now be used to define the set of **transitions** instead of the set of **arcs**.

Definition 5.28 *We define a* **complex statechart** *as a simple statechart* $Sct = (S, r, children, type, \mathcal{H}, hist, default, \mathcal{T})$ *such that:*
$\mathcal{T} \subset 2^{S} \times L \times 2^{S \cup \mathcal{H}}$ *is the set of* **transitions**.
That is, if $t = (X, l, Y)$ *is a transition, then*

(i) X *is called the* **source set**,

(ii) l *is a label,*

(iii) Y *is called the* **target set** *(not the set of target states, since* Y *can also contain history symbols).*

Note*:*

- *The set* X *generally consists of only one state; only in case of a direct exit of an* AND*-state* X *contains more than one state.*
- *The set* Y *which is also called* $target(t)$ *generally contains only one state and — if necessary — one history symbol.*
- *In the case where the sets* X *and* Y *consist of only one state* s_i *and* s_j*, respectively, a transition between* s_i *and* s_j *will be expressed as* $(s_i, e/a, s_j)$ *instead of* $(\{s_i\}, e/a, \{s_j\})$.

(*End of Definition 5.28*)

In case of a direct exit of an AND-state, the source set X contains more than one state. To be able to define a legal transition t, every two states in X (the source set of t) and every two states in Y (the target set of t) must be mutually orthogonal. Before the definition of a *legal* transition of a statechart is given, the following terms are defined:

The *lowest common ancestor* will be defined with the help of the relation $\leq$.

Definition 5.29 *(1) The* **partial order relations** $\leq$ *and* $<$ *on the set of states* S *are defined as follows:*

For each pair $s_1, s_2 \in S$*:* $s_1 \leq s_2$ *iff* $s_1 \in children^{*}(s_2)$,
$s_1 < s_2$ *iff* $s_1 \leq s_2$ *and* $s_1 \neq s_2$.

(2) For a non-empty set of states $S_1 \subseteq S$*, the* **lowest common ancestor** *of* S_1*, denoted* $LCA(S_1)$ *is an element of* S *defined as follows:*

(a) $\forall s \in S_1 : s \leq LCA(S_1)$ *and*

(b) $\forall s' \in S : (\forall s \in S_1 : s \leq s') \Rightarrow (LCA(S_1) \leq s')$.

Remark: Since the *children*-Relation can be represented by a tree and *children** is related to children in a canonical way, $LCA(S_1)$ is uniquely defined.

Definition 5.30 *Two states* $s_1, s_2 \in S$ *are* **orthogonal***, denoted* $s_1 \perp s_2$*, iff the following is true:*

(a) $type(LCA(\{s_1, s_2\})) \;=\; AND$

(b) $\neg(s_1 \leq s_2 \;\vee s_2 \leq s_1)$.

The definition of a *weakly legal* transition of a statechart is given as follows:

Definition 5.31 *A transition* $t \;=\; (X, l, Y)$ *is* **weakly legal** *iff*

(a) $\forall s_1, s_2 \in X : s_1 \neq s_2 \;\Rightarrow\; s_1 \perp s_2$ *and*

(b) $\forall s_1, s_2 \in Y \cap S : s_1 \neq s_2 \;\Rightarrow\; s_1 \perp s_2$

The set of weakly legal transitions will be denoted by $\mathcal{T}_{W_{legal}}$.

Remark Definition 5.31 corresponds to the definition of 'legal' transitions described in [51]. However, it is easy to show that the requirements stated in Definition 5.31 are necessary, but they are not sufficient to define a 'legal' transition. Therefore, we refer to it as the definition of a *weakly legal* transition.

To illustrate the Definitions 5.29, 5.30 and 5.31, consider Figure 4.3, page 28. If $S1 =$ $\{$*Cup-idle, Coffee-idle, Standby, Light-off*$\}$, then $LCA(S_1) \;=\; On$, $type(LCA(S1)) \;=\; AND$.
The transition $t1 = (\{$*Cup-idle, Coffee-idle, Standby, Light-off*$\}$, *power-off*, $\{$*Off*$\})$ is a weakly legal transition of statechart *SCVM*.
Transition $t2 = (\{$*Cup-idle, Coffee-idle, Light-off*$\}$, *power-off*, $\{$*Off*$\})$ is not a legal transition of statechart *SCVM*. The explanation for the transition $t2$ not being 'legal' requires the 'maximality' of X to be defined. In order to deal with such problems, the concept of weakly legal transitions must be extended to 'legal' transitions which have legal configurations (cf. Definition 5.27 in Subsection 5.4.1).

Therefore, the formal concept of *legal* transition will be defined as follows:

Definition 5.32 *A transition* $t \;=\; (X, l, Y)$ *is* **legal** *iff*

(a) t is weakly legal, and

(b) X *and* Y *are legal configurations.*

The set of legal transitions will be denoted by $\mathcal{T}_{legal}$.

In the example above $t2$ is not a legal transition of statechart $SCVM$ because $X = \{$*Cup-idle, Coffee-idle, Light-off*$\}$ is not a legal configuration. That is, $X = \{$*Cup-idle, Coffee-idle, Light-off*$\}$ is not a basic configuration. The idea behind the formal concept of legal transition is that when $SCVM$ is e. g., in AND-state *On*, then exiting *On* implies leaving all component states of the AND-state *On*. The transition $t1 = (\{$*Cup-idle, Coffee-idle, Standby, Light-off*$\}$, *power-off*, *Off*) for instance is hence not only a weakly legal transition of statechart $SCVM$ but it is legal transition as well.

Nevertheless, Definition 5.32 does not give a procedure which can be used to compute the legal configuration sets X and Y. Therefore, the next Subsections 5.4.3 and 5.4.4 will deal with this problem.

5.4.3 Transforming *AND*-states into *OR*-states

In Subsection 5.2.4 we established the necessary background which can be helpful for the development of methods and concepts to overcome the test problems, e. g., developing algorithms which may be used to compute the aforementioned legal configuration sets X and Y (cf. Definition 5.32). However, the main idea behind the methods developed in Subsection 5.2.4 requires that a hierachical statechart consists merely of states of type OR, i. e., AND-states are allowed. This requirement leads to the following observation: Since reactive systems are normally described by hierachical statecharts that may also contain orthogonal features, it is necessary to find some means of transforming states of type AND, i. e., AND-states into OR-states. The main aim of this Subsection 5.4.3, therefore, is to present techniques which can be used to transform AND-states into OR-states.

The transformation of an AND-state into an OR-state corresponds to building the automata product of an orthogonal statechart (cf. Definition 5.12). This subsection, therefore, is devoted to the construction of this automata product. In particular, it is shown how the adjacency-matrix technique can be used for determining the corresponding automata product of an orthogonal statechart. Note that the constructed automata product is different from the conventional automata product. Hence the constructed automata product will be referred to as *modified automata product*.

Our main goal is to achieve an OR-graph which represents the OR-states derived from the AND-states. The OR-graph will be needed when computing all legal configurations of the AND-state. These legal configurations are necessary for the illustration of the concept of 'scope' of transitions (cf. Subsection 5.4.4). Above all, the OR-graph will also be required in the process of test case derivation (cf. Chapter 7).

This Subsection 5.2.4 is organised as follows. In Subsubsection 5.4.3.1 we describe the adjacency-matrix representation of statecharts. In Subsubsection 5.4.3.2 we then construct the (statechart) modified automata product.

5.4.3.1 Matrix representation of statecharts

In the following, the *adjacency-matrix representation* which will be used to express the associated digraph of a very simple OR-statechart (cf. Definition 5.20) is described.

Definition 5.33
Assume that the vertices of a digraph $G = (V, E, inc, label)$ are numbered $1, 2, \ldots, |V|$ in arbitrary manner[22]*. The* **adjacency-matrix representation** *of digraph G denoted GMatrx then consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ with entries in $\{0, 1\}$ such that*

$$a_{ij} = \begin{cases} 1 & \textit{if } \exists e \in E : inc(e) = (i, j), \\ 0 & \textit{otherwise.} \end{cases}$$

Notation 5.2 *Adjacency-matrix representation of the associated digraph of a very simple OR-statechart will be called* **very simple OR-statechart matrix** *and will be abbreviated to VSSctM.*

The adjacency-matrix representation of the associated digraph of a very simple OR-statechart is defined as follows:

Definition 5.34
Assume that the vertices of the associated digraph $G = (V, v_0, E, inc, label)$ of a very simple OR-statechart Sct are numbered $1, 2, \ldots, |V|$ in arbitrary manner[23]*. Let L be the set of labels as described in Definition 5.10. Then the* **adjacency-matrix representation** *VSSctM of digraph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ with entries in 2^L such that*

$$a_{ij} = \{label(e) \mid e \in E \textit{ and } inc(e) = (i, j)\}\,.$$

Remark: *If $a_{ij} = \emptyset$, i. e., there does not exist an edge $e \in E$ with a_{ij}, then $a_{ij} = 0$ in the corresponding adjacency-matrix representation of Definition 5.33.*

5.4.3.2 Building modified automata products

In this Subsubsection 5.4.3.2, we use the adjacency-matrix technique described in Subsubsection 5.4.3.1 above to determine the corresponding automata product of an orthogonal statechart. Because the constructed automata product is different from the conventional automata product, we will refer to it as *modified automata product.*

Motivation of modified automata

In Subsection 5.2.4, we examined the relationship between a formal statechart and a labelled directed graph. Among other things, we defined an OR-graph (cf. Definition 5.18) which was based on the definition of OR-statechart (cf. Definition 5.13). Then we gave the definition of a very simple OR-statechart (i. e[24]., in which all states with the exception of the root are basic) and its associated simple OR-graph (cf. Definition 5.20).

Now on one hand, we already know that reactive systems are normally described by hierachical statecharts that may also contain orthogonal features, on the other hand, we

[22] $V = \{1, 2, \ldots, |V|\}$

[23] $V = \{1, 2, \ldots, |V|\}$

[24] a statechart

argued that *OR*-graphs represent OR-statecharts which basically consist merely of states of type *OR*, i. e., *OR*-states. There are several reasons for insisting on transforming states of type *AND*, i. e., *AND*-states into *OR*-states. For our main purpose, namely the development of methods and concepts to overcome the test problems, the derived *OR*-graphs will aid the development of simple and efficient algorithms which are used to generate test paths in a statechart model (cf. Chapter 7). Therefore, our main goal in this subsubsection is to establish techniques which will allow the transformation of *AND*-states into *OR*-states. The orthogonal statechart will be represented by a 'modified automata product'.

Recall the simple statechart *S00* in Figure 5.3 on page 44. Now the orthogonal statechart for *S0* in Figure 5.3 is represented by Figure 5.17. The corresponding 'modified automata product' for *S0* is depicted by Figure 5.18.

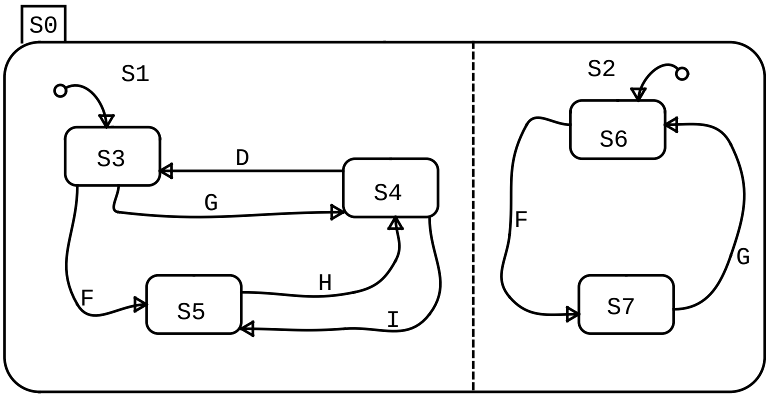


Figure 5.17: An orthogonal Statechart for *S0*.

As a motivation for this modified automata product which will be formally described in Definition 5.35 we refer to the work of Böhling and Schütt in [17]. In [17] (cf. definition 7.1(2) page 63) a *direct product* of automaton M_1 and automaton M_2 (without outputs) is given as follows:

Let $M_i = (S_i, X_i, (\tau_x^i)x \in X_i)$ with X_i input and τ_x^i transition relation ($i \in \{1, 2\}$),

$S = S_1 \times S_2$, and

$X = X_1 = X_2$ be given.

For all $x \in X$:

$((S_1, S_2), (S_1', S_2')) \in \tau_x$ (i. e., $S_1S_2 \xrightarrow{x} S_1'S_2'$)

$:\Longleftrightarrow (S_1, S_1') \in \tau_x^1$ and $(S_2, S_2') \in \tau_x^2$ where

$(S_1, S_1') \in \tau_x^1$ is a transition in automaton 1 with x (from S_1 to S_1') and

$(S_2, S_2') \in \tau_x^2$ is a transition in automaton 2 with x (from S_2 to S_2').

Remark 5.10 Note that for the 'statechart product matrix' it will be assumed that $(s, s) \in \tau_x$ is true if $(s, t) \notin \tau_x$ for all $t \neq s$ (cf. Figure 5.19).

The modified automata product for *S0* in Figure 5.18 represents a 'simple OR-graph product' G_0. In short G_0 is the OR-graph product of the two associated simple OR-graphs

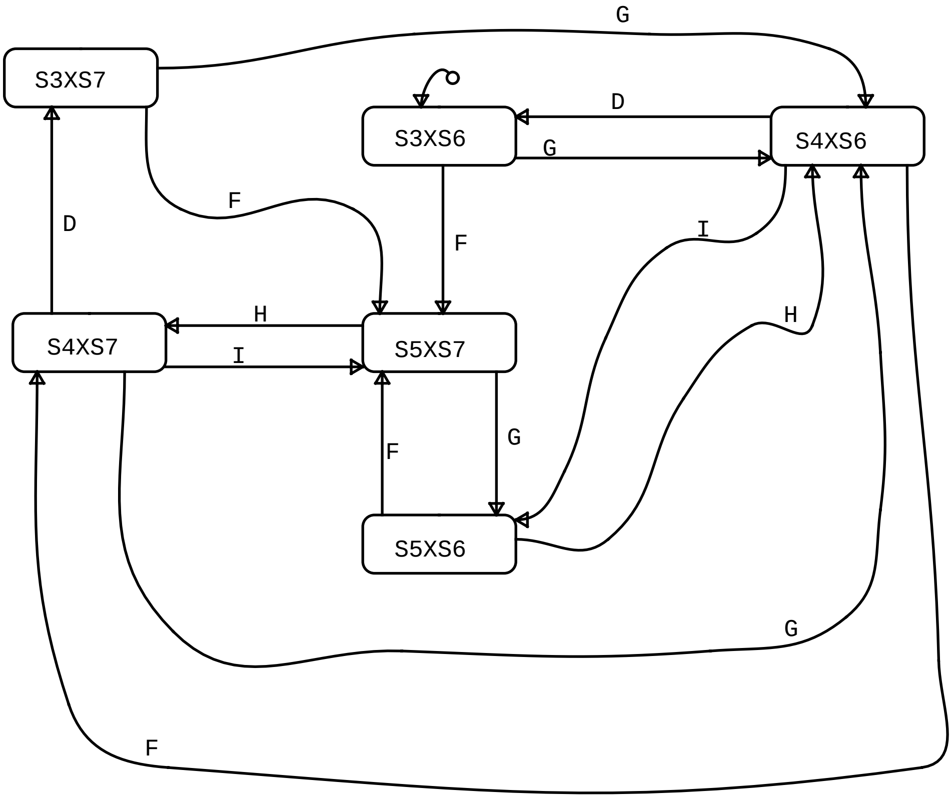


Figure 5.18: The automata product for *S0*

of the corresponding orthogonal components of *S0*. In the following, we give the formal definition of the *simple OR-graph product* G_0.

Definition 5.35 *For an orthogonal statechart S0 consisting of two orthogonal components, S1 and S2 (i. e., children*$(S0) = \{S1, S2\}$ *and type*$(S0) = AND$*), let* $G_1 = (V_1,\ v_{0_1},\ E_1,\ inc_1,\ label_1)$ *and* $G_2 = (V_2,\ v_{0_2},\ E_2,\ inc_2,\ label_2)$ *be digraphs*[25] *representing S1 and S2, respectively. Then* $G_0 = (V_0,\ v_0,\ E_0,\ inc_0,\ label_0)$ *is the* **associated**

[25] Note that G_1 and G_2 are associated simple OR-graphs (cf. Definition 5.20).

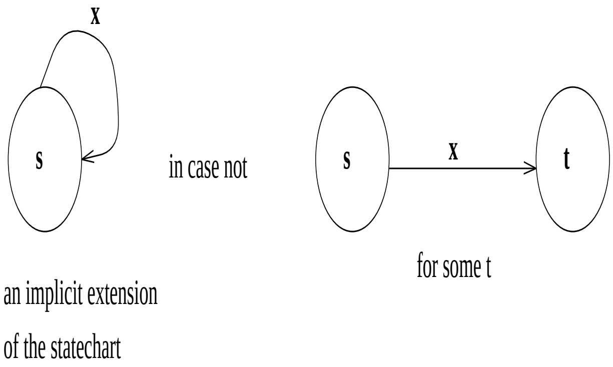


Figure 5.19: Illustrating Remark 5.10

simple OR-graph, *also called* **simple OR-graph product**, *where*

(a) $V_0 := (S_1 \times S_2)$,

(b) $v_0 := (v_{0_1} v_{0_2})$,

(c) $E_0 := (E_1 \times E_2) = \mathcal{T}_0$ *(where* $\mathcal{T}_0$ *is the set of transitions of orthogonal statechart S0),*

(d) $inc_0 : E_0 \rightarrow V_0 \times V_0$ *and* $label_0 : E_0 \rightarrow L_0$ *are defined as follows:* *for a transition* $t_0 = (x_1 x_2)\ l_0\ (y_1 y_2) \in \mathcal{T}_0 = E_0$*:* $inc(t_0) = ((x_1 x_2), (y_1 y_2))$ *and* $label(t_0) = l_0$.

Remark 5.11 *In Definition 5.35, the set of edges* $E_1 \cup E_2$ *represent* $\mathcal{T}_{orth_{arcs(s0)}}$ *the set of transition arcs of orthogonal statechart S0 (cf. Definition 5.12). The set of edges* E_0*, which represents the set of transitions of orthogonal statechart S0 will be denoted by* $\boldsymbol{\mathcal{T}_{orth}}$ *instead of* $\mathcal{T}_0$.

In the next step, we establish an algorithm which will be used to determine the simple OR-graph product described in Definition 5.35 above. This algorithm is called **statechart_product_matrix** and is defined in the following.

Definition 5.36 *For an orthogonal statechart S0 consisting of two orthogonal components S1 and S2 (i. e.,* $children(S0) = \{S1, S2\}$ *and* $type(S0) = AND$*), let* G_1 *and* G_2 *be digraphs representing S1 and S2, respectively. Then assume A and B, the corresponding* **statechart matrices** *(VSSctMs) representing* G_1 *and* G_2*, respectively, consist of basic states only.*

A mapping

$$\textbf{auto\_prod} : M(n,n;2^L) \times M(m,m;2^L) \rightarrow M(n \cdot m, n \cdot m; 2^L) \tag{5.1}$$

is defined, where $M(k,k;X)$ *denotes the set of all* $k \times k$*-matrices with entries in* X *for* $k \in I\!N$.
Given $A \in M(n,n;2^L)$ *and* $B \in M(m,m;2^L)$*, let*

$$C = (c_{ij}) = auto\_prod\,(A,B) \tag{5.2}$$

be defined as follows:

For $i,j \in \{1,\ldots,n\}$, $i',j' \in \{1,\ldots,m\}$*, element* $c_{\varphi(i,i'),\varphi(j,j')}$ *of C is defined as follows*[26]*, where* φ *is the mapping*

$$\varphi : \{1,\ldots,n\} \times \{1,\ldots,m\} \rightarrow \{1,\ldots,n \cdot m\} \text{ with } \varphi(k,l) \mapsto (k-1)\cdot m + l :$$

(1) if $a_{ij} = \emptyset$ *and* $b_{i'j'} = \emptyset$*, then* $c_{\varphi(i,i'),\varphi(j,j')} = \emptyset$

(2) if $a_{ij}\ \cap\ b_{i'j'} \neq \emptyset$*, then* $c_{\varphi(i,i'),\varphi(j,j')} = a_{ij}\ \cap\ b_{i'j'}$

[26] Note that φ is a *bijective* function.

(3) otherwise (i. e., if $a_{ij} \cap b_{i'j'} = \emptyset$ *and* $a_{ij} \cup b_{i'j'} \neq \emptyset$*)*

   (a) if $a_{ij} \neq \emptyset$ *and* $b_{i'j'} \neq \emptyset$

      i. if $(i \neq j$ *and* $i' \neq j')$ *then* $c_{\varphi(i,i'),\varphi(j,j')} = \emptyset$

      ii. if $(i = j$ *and* $i' = j')$ *and* $\exists k : a_{ik} \cap b_{i'j'} \neq \emptyset$ *then* $c_{\varphi(i,i'),\varphi(j,j')} = \emptyset$ *(i. e., NB.:* **Conflict**[27]*between transitions!!!)*
      otherwise $c_{\varphi(i,i'),\varphi(j,j')} = a_{ij} \cup b_{i'j'}$

      iii. if $(i \neq j$ *and* $i' = j')$ *then* $c_{\varphi(i,i'),\varphi(j,j')} = a_{ij}$

      iv. if $(i = j$ *and* $i' \neq j')$ *then* $c_{\varphi(i,i'),\varphi(j,j')} = b_{i'j'}$ *(this case is dual to the former one)*

   The parts ii. through iv. handle loop problems.

   (b) if $b_{i'j'} = \emptyset$ *(and* $a_{ij} \neq \emptyset$*):*

      i. if $i' \neq j'$ *then* $c_{\varphi(i,i'),\varphi(j,j')} = \emptyset$

      ii. if $i' = j'$ *and* $\forall k' : a_{ij} \cap b_{i'k'} = \emptyset$ *then* $c_{\varphi(i,i'),\varphi(j,j')} = a_{ij}$

      iii. otherwise (i. e., if $i' = j'$ *and* $\exists k' : a_{ij} \cap b_{i'k'} \neq \emptyset$*)* $c_{\varphi(i,i'),\varphi(j,j')} = \emptyset$

   (c) otherwise (i. e., if $b_{i'j'} \neq \emptyset$ *and* $a_{ij} = \emptyset$*, this case is dual to case (b))*

      i. if $i \neq j$ *then* $c_{\varphi(i,i'),\varphi(j,j')} = \emptyset$

      ii. if $i = j$ *and* $\forall k : a_{ik} \cap b_{i'j'} = \emptyset$ *then* $c_{\varphi(i,i'),\varphi(j,j')} = b_{i'j'}$

      iii. otherwise (i. e., if $i = j$ *and* $\exists k : a_{ik} \cap b_{i'j'} \neq \emptyset$*)* $c_{\varphi(i,i'),\varphi(j,j')} = \emptyset$

The adjacency-matrix $C = auto\_prod(A, B)$ *is called the* **automata product matrix** *of A and B and is abbreviated to APM.*

The automata product matrix *APM C* represents G_0, the simple OR-graph product of the orthogonal statechart *S0*.

The following two statechart matrices A and B represent the orthogonal components $S1$ and $S2$, respectively, in the orthogonal statechart $S0$ of Figure 5.17 (without default states). In the matrices singleton sets $\{x\}$ are represented as x and $\emptyset$ is denoted as 0. Rows and columns are labelled by the original state names (instead of a numbering of states i. e., vertices, as in Definition 5.33) and e. g., $a_{S3,S5}$ denotes row $S3$ and column $S5$ of the statechart matrix A, and e. g., $b_{S6,S7}$ denotes row $S6$ and column $S7$ of the statechart matrix B.

$$A: \quad \begin{array}{c} \\ S3 \\ S5 \\ S4 \end{array} \begin{array}{c} \begin{array}{ccc} S3 & S5 & S4 \end{array} \\ \begin{pmatrix} 0 & F & G \\ 0 & 0 & H \\ D & I & 0 \end{pmatrix} \end{array} \qquad B: \quad \begin{array}{c} \\ S6 \\ S7 \end{array} \begin{array}{c} \begin{array}{cc} S6 & S7 \end{array} \\ \begin{pmatrix} 0 & F \\ G & 0 \end{pmatrix} \end{array}$$

[27] Two transitions are in **conflict** if there is some common state that would be exited if any one of them were to be taken (cf. Subsection 5.4.5).

Notation 5.3 *Note that the following four expressions may be used if* $c_{\varphi(S3,S6),\varphi(S5,S7)} = \{F\}$ *and denote that if in state* $S3$ *and* $S6$ *and event* F *occurs then the next configuration will consist of* $S5$ *and* $S7$.

$$S3 \times S6 \xrightarrow{F} S5 \times S7$$

$$S3, S6 \xrightarrow{F} S5, S7$$

$$S3S6 \xrightarrow{F} S5S7$$

$$\varphi(S3, S6) \xrightarrow{F} \varphi(S5, S7)$$

(if φ *is appropriately chosen as in the footnote 26 of Definition 5.36 above).*

In the following, Definition 5.36 is illustrated.

Case (1) of Definition 5.36

$$S3S6 \xrightarrow{\emptyset} S3S6$$

$$\left.\begin{array}{r} a_{S3,S3} = \emptyset \\ b_{S6,S6} = \emptyset \end{array}\right\} \Rightarrow c_{\varphi(S3,S6),\varphi(S3,S6)} = \emptyset,$$

which means that there is no transition from $S3S6$ to $S3S6$.

Case (1) of Definition 5.36

$$S5S6 \xrightarrow{\emptyset} S3S6$$

$$\left.\begin{array}{r} a_{S5,S3} = \emptyset \\ b_{S6,S6} = \emptyset \end{array}\right\} \Rightarrow c_{\varphi(S5,S6),\varphi(S3,S6)} = \emptyset$$

which means that there is no transition from $S5S6$ to $S3S6$.

Case (2) of Definition 5.36

$$S3S6 \xrightarrow{F} S5S7$$

$$\left.\begin{array}{r} a_{S3,S5} = F \\ b_{S6,S7} = F \end{array}\right\} \Rightarrow c_{\varphi(S3,S6),\varphi(S5,S7)} = \{F\} \cap \{F\} = \{F\},$$

hence $S3S6 \xrightarrow{F} S5S7$ (cf. Notation 5.3).

Case (3)(a)i. of Definition 5.36

$$S3S6 \xrightarrow{\emptyset} S4S7$$

$$\left.\begin{array}{r} a_{S3,S4} = G \\ b_{S6,S7} = F \end{array}\right\} \Rightarrow c_{\varphi(S3,S6),\varphi(S4,S7)} = \emptyset$$

which means that there is no transition from $S3S6$ to $S4S7$.

Now consider Figure 5.20 which depicts another orthogonal statechart *S0*. We use it to illustrate case (3)(a)ii.

Case (3)(a)ii. of Definition 5.36

$$S11S21 \xrightarrow{\emptyset} S11S21$$

$$\left.\begin{array}{l} a_{S11,S11} = E \\ b_{S21,S21} = F \\ a_{S11,S12} = F \end{array}\right\} \Rightarrow (\{E\} \cup \{F\}) \cap \{F\} \neq \emptyset \ \Rightarrow c_{\varphi(S11,S21),\varphi(S11,S21)} = \emptyset,$$

which means that nondeterminism occurs when the OR-graph product is constructed. This is clearly shown by Figure 5.21.

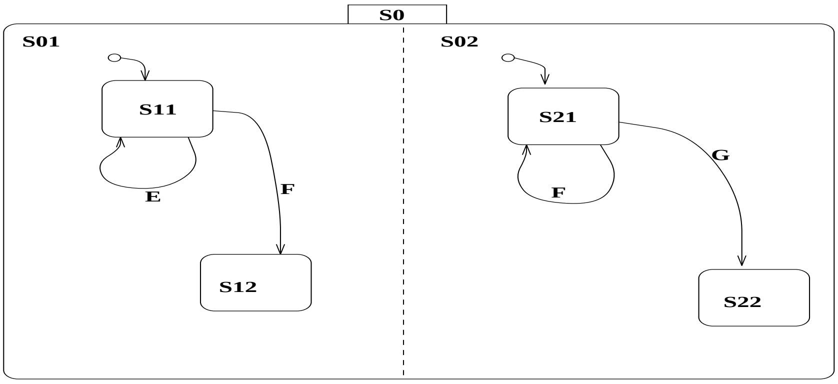


Figure 5.20: Another orthogonal statechart *S0* used to illustrate Case (3)(a)ii

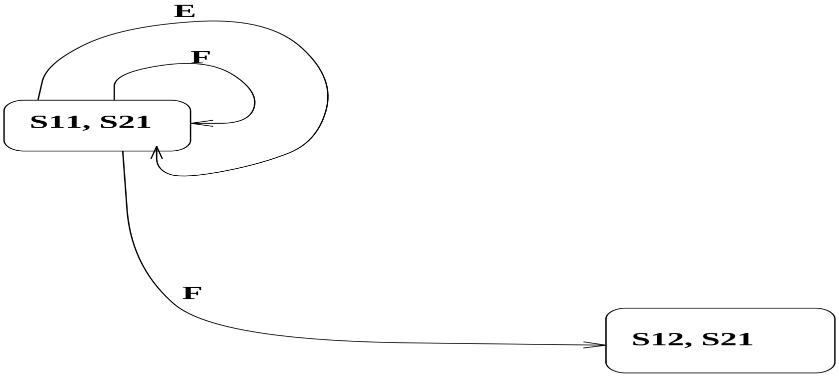


Figure 5.21: An example of nondeterminism

Now consider Figure 5.22. We use it to illustrate Case (3)(a)iii.

Case (3)(a)iii. of Definition 5.36

$$S11S21 \xrightarrow{F} S12S21$$

$$\left.\begin{array}{l} a_{S11,S12} = F \\ b_{S21,S21} = H \end{array}\right\} \Rightarrow c_{\varphi(S11,S21),\varphi(S12,S21)} = F,$$

which means that there is a transition from $S11S21$ to $S12S21$. This is shown by Figure 5.23.

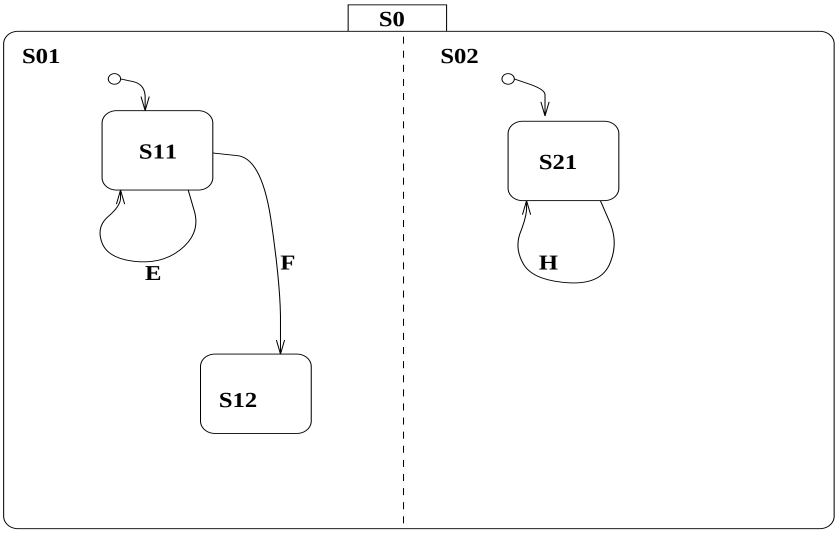


Figure 5.22: Statechart *S0* used to illustrate Case (3)(a)iii

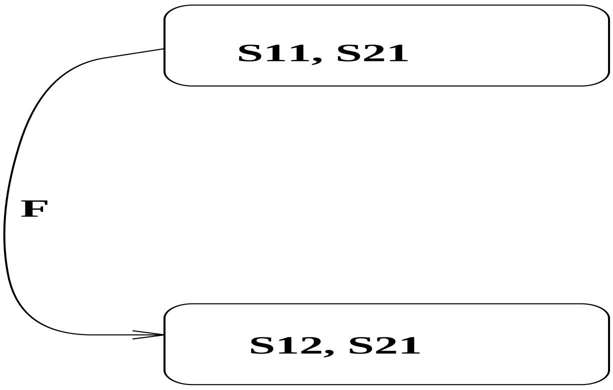


Figure 5.23: Illustrating Case (3)(a)iii more

Now consider Figure 5.17 again.
Case (3)(b)ii. of Definition 5.36

$$S3S7 \xrightarrow{\emptyset} S5S7$$

$$\begin{array}{l} a_{S3,S5} = F \\ b_{S7,S7} = \emptyset \end{array}$$

$$\left.\begin{array}{l} b_{S7,S6} = G \\ b_{S7,S7} = \emptyset \end{array}\right\} \Rightarrow \{F\} \cap (\{G\} \cup \emptyset) = \emptyset \Rightarrow c_{\varphi(S3,S7),\varphi(S5,S7)} = F,$$

which means that there is a transition from $S3S7$ to $S5S7$.

Case (3)(b)iii. of Definition 5.36

$$S3S7 \xrightarrow{\emptyset} S4S7$$

$$\begin{array}{l} a_{S3,S4} = G \\ b_{S7,S7} = \emptyset \\ b_{S7,S6} = G \Rightarrow \{G\} \cap \{G\} \neq \emptyset \Rightarrow c_{\varphi(S3,S7),\varphi(S4,S7)} = \emptyset \end{array}$$

which means that there is no transition from $S3S7$ to $S4S7$.

Here we have to check whether there is at least one transition leaving $S7$ with the label G. In this particular case, there is a transition from $S7$ to $S6$ with label G. Therefore, it is not possible to execute G only for the transition $S3$ to $S4$.

Case (3)(c)i. of Definition 5.36

$$S5S7 \xrightarrow{\emptyset} S3S6$$

$$\begin{array}{l} a_{S5,S3} = \emptyset \\ b_{S7,S6} = G \\ S5 \neq S3 \Rightarrow c_{\varphi(S5,S7),\varphi(S3,S6)} = \emptyset, \end{array}$$

which means that there is no transition from $S5S7$ to $S3S6$. If there is no transition from $S5$ to $S3$ and $S5 \neq S3$, the transition from $S5S7$ to $S3S6$ is obviously undefined. Therefore, there is no need to bother computing the value of $b_{i'j'}$.

Case (3)(c)ii. of Definition 5.36

$$S5S6 \xrightarrow{F} S5S7$$

$$\begin{array}{l} a_{S5,S5} = \emptyset \\ b_{S6,S7} = F \end{array}$$

$S5S5$ creates no problems because for the statechart product matrix, it is assumed that $(s,s) \in \tau_x$ is true if $(s,t) \notin \tau_x$ for all $t \neq s$. Hence there is a self loop.

$$\left.\begin{array}{l} a_{S5,S3} = \emptyset \\ a_{S5,S5} = \emptyset \\ a_{S5,S4} = H \end{array}\right\} \Rightarrow (\{H\} \cup \emptyset \cup \emptyset) \cap \{F\} = \emptyset \Rightarrow c_{\varphi(S5,S6),\varphi(S5,S7)} = \{F\},$$

which means that there is a transition from $S5S6$ to $S5S7$.

Here we have to check whether there is no transition leaving $S5$ with label F.

Case (3)(c)iii. of Definition 5.36

$$S3S7 \xrightarrow{\emptyset} S3S6$$

$$\begin{aligned}
&a_{S3,S3} = \emptyset\\
&b_{S7,S6} = G\\
&a_{S3,S4} = G \Rightarrow \{G\} \cap \{G\} \neq \emptyset \Rightarrow c_{\varphi(S3,S7),\varphi(S3,S6)} = \emptyset,
\end{aligned}$$

which means that there is no transition from $S3S7$ to $S3S6$.

Here we have to check whether there is at least one transition leaving $S3$ with the label G. In other words, when the external event G occurs, it must be received by both automata. Therefore, it is not possible to execute G for only one of the states.

Case (3)(c)iii. of Definition 5.36

$$S3S6 \xrightarrow{\emptyset} S3S7$$

$$\begin{aligned}
&a_{S3,S3} = \emptyset\\
&b_{S6,S7} = F\\
&a_{S3,S5} = F \Rightarrow \{F\} \cap \{F\} \neq \emptyset \Rightarrow c_{\varphi(S3,S6),\varphi(S3,S7)} = \emptyset,
\end{aligned}$$

which means that there is no transition from $S3S6$ to $S3S7$.

The application of Definition 5.36 leads to the following (6 × 6) entries:

$$\begin{aligned}
C[S3S6, S3S6] &:= 0\\
C[S3S6, S3S7] &:= 0\\
C[S3S6, S5S6] &:= 0\\
C[S3S6, S5S7] &:= F\\
C[S3S6, S4S6] &:= G\\
C[S3S6, S4S7] &:= 0
\end{aligned}$$

$$\begin{aligned}
C[S3S7, S3S6] &:= 0\\
C[S3S7, S3S7] &:= 0\\
C[S3S7, S5S6] &:= 0\\
C[S3S7, S5S7] &:= F\\
C[S3S7, S4S6] &:= G\\
C[S3S7, S4S7] &:= 0
\end{aligned}$$

$$\begin{aligned}
C[S5S6, S3S6] &:= 0\\
C[S5S6, S3S7] &:= 0\\
C[S5S6, S5S6] &:= 0\\
C[S5S6, S5S7] &:= F\\
C[S5S6, S4S6] &:= H\\
C[S5S6, S4S7] &:= 0
\end{aligned}$$

$$\begin{aligned}
C[S5S7, S3S6] &:= 0\\
C[S5S7, S3S7] &:= 0\\
C[S5S7, S5S6] &:= G\\
C[S5S7, S5S7] &:= 0\\
C[S5S7, S4S6] &:= 0\\
C[S5S7, S4S7] &:= H
\end{aligned}$$

$$\begin{aligned}
C[S4S6, S3S6] &:= D\\
C[S4S6, S3S7] &:= 0\\
C[S4S6, S5S6] &:= I\\
C[S4S6, S5S7] &:= 0\\
C[S4S6, S4S6] &:= 0\\
C[S4S6, S4S7] &:= F
\end{aligned}$$

$$\begin{aligned}
C[S4S7, S3S6] &:= 0\\
C[S4S7, S3S7] &:= D\\
C[S4S7, S5S6] &:= 0\\
C[S4S7, S5S7] &:= I\\
C[S4S7, S4S6] &:= G\\
C[S4S7, S4S7] &:= 0
\end{aligned}$$

Here is the resulting matrix:

$$M_3 = C$$

$$\begin{array}{c|cccccc|}
 & S3S6 & S3S7 & S5S6 & S5S7 & S4S6 & S4S7 \\
\hline
S3S6 & 0 & 0 & 0 & \textcircled{F} & \textcircled{G} & 0 \\
S3S7 & 0 & 0 & 0 & \textcircled{F} & \textcircled{G} & 0 \\
S5S6 & 0 & 0 & 0 & \textcircled{F} & \textcircled{H} & 0 \\
S5S7 & 0 & 0 & \textcircled{G} & 0 & 0 & \textcircled{H} \\
S4S6 & \textcircled{D} & 0 & \textcircled{I} & 0 & 0 & \textcircled{F} \\
S4S7 & 0 & \textcircled{D} & 0 & \textcircled{I} & \textcircled{G} & 0
\end{array}$$

In addition to Definition 5.36, an important control mechanism for reachable states RS is stated in the following definition.

Definition 5.37 *The* **reachable automata product matrix** *$RAPM$ of an automata product matrix APM is the* **restriction** *of the APM to reachable states. The* **set of reachable states** *RS is constructed as follows:*

Let $S_{default_1}$ and $S_{default_2}$ be the default states of $VSSctM_1$ and $VSSctM_2$, respectively. Then the sets RS_i, $i \geq 0$, are defined recursively as follows:

$RS_0 := \{(S_{default_1}, S_{default_2})\}$

$RS_{l+1} = \{(S, S') \mid \exists S''S''' \in RS_l : c_{\varphi(S,S''),\varphi(S',S''')} \neq \emptyset\} \cup RS_l$ *where* $l \in I\!N_0$.

If i is the smallest number such that $RS_{i+1} = RS_i$ then $RS := RS_i$.

Referring to the example above, Definition 5.37 gives the following result.

$RS_0 = \{S3S6\}$

$RS_1 = \{S4S6, S5S7\} \cup RS_0$

$RS_2 = \{S5S6, S4S7\} \cup RS_1$

$RS_3 = \{S3S7\} \cup RS_2$

$RS_4 = RS_3,$

hence $RS = \{S3S6, S4S6, S5S7, S5S6, S4S7, S3S7\} = \{S3, S4, S5\} \times \{S6, S7\}$.

Theorem 5.4 *Let G_0 be a simple OR-graph product of the orthogonal statechart $S0$ constructed by Definition 5.36 and Definition 5.37. Then the nodes of G_0 deliver the set of all (possible) basic configurations of the orthogonal statechart $S0$. In addition, the edges of G_0 deliver $\mathcal{T}_{orth}$, the set of legal transitions of orthogonal statechart S0.*

Proof idea The proof idea is based on the fact that G_0 is a simple OR-graph (cf. Definition 5.20).

5.4.4 Notion of scope of transitions

In Subsection 5.3.1 problems caused by *inter-level transitions* were discussed. It was noted that when an inter-level transition is dealt with, one has to accurately define the hierarchical states that are exited by taking the transition and those that are entered. Recalling the example of Figure 5.13, page 59, the question is still whether the system exits and reenters state $S1$ and which substate of $S2$ is (re-)entered when transition arc $a2$ is executed?

The main purpose of this Subsection 5.4.4, therefore, is to formalise a concept — the notion of 'scope' of transitions — that can be used to deal with interlevel transitions which in general deny a structural operational semantics for statecharts.

Harel and Naamad introduced the concept of the *scope* of transitions to overcome ambiguities as mentioned above. For a full compound transition $t \in \mathcal{T}_{legal}$, "the *scope* of a transition t is the lowest OR-state in the hierarchy of the states that is a proper common ancestor of all the sources and targets of the transition arcs appearing in t" (cf. [54]).

This informal description of the scope concept will be formalised by a function *scope* as follows:

Definition 5.38 *Let $Sct = (S, r, children, type, \mathcal{H}, hist, default, \mathcal{T})$ be a statechart. The function $scope : \mathcal{T} \rightarrow S$ which maps a transition $t = (X, l, Y) \in \mathcal{T}$ to a state, called its* **scope***, is a function satisfying the following requirements:*

(i) $\forall s \in (X \cup Y) : s \leq scope(t) \wedge type(scope(t)) = OR$, *and*

(ii) $\forall s' \in S : ((\forall s \in (X \cup Y) : s \leq s') \wedge type(s') = OR) \Rightarrow scope(t) \leq s'$.

Remark 5.12 Note that the following is true: if all considered states (in the partial order relation $\leq$) are OR-states then the *lowest common ancestor* (LCA) for the union of the source set and target set of the transition is the *scope* (see also Subsection 5.4.2, Definition 5.29 (2)).

In [54], Harel and Naamad stated, "When the transition t is taken, all proper descendants of its scope in which the system resided at the beginning of the step are exited, and all proper descendants of that scope in which the system reside as a result of executing t are entered." In other words, the scope is the lowest OR-state in which the sytem stays without exiting and reentering when taking the transition.

As an example consider Figure 5.13 on page 59. In the following, the transition in which the transition arc $a2$ is taken, will be called $t2$. Since $scope(t2) \geq S5$ and $scope(t2) \geq S7$ is demanded by Definition 5.38, the scope of $t2$ is $S0$ (and not $S1$ which is of type AND). Taking $t2$ implies exiting $S5$, $S2$, $S1$, $S3$ and either $S6$ or $S7$, depending on which one of them the system was in at the beginning of the step, and implies entering states $S1$, $S2$, $S4$[28], $S3$, $S7$. State $S0$ is not exited.

To apply Definition 5.38 to statecharts which are not of type OR, i. e., to statecharts of type AND (orthogonal statecharts, cf. Definition 5.12), it is necessary to develop an

[28] Since $S4$ is the default state of $S2$, executing $t2$ implies entering state $S4$.

algorithm which computes the source set X and target set Y for a transition corresponding to "executing a transition **arc**". This is particularly important when the sets X and Y consist of more than one state. For example, in case of a direct exit of an *AND*-state, X contains more than one state.

Let $\mathcal{T}_{orth_{arcs(s)}}$ be a set of transition arcs of an *AND*-state s (cf. Definition 5.12). Let $\mathcal{T}_{orth}$ be a set of transition arcs of an *AND*-state (cf. Remark 5.11).

For example, given the transition arc $a2 = (S5, ev2, S7) \in \mathcal{T}_{orth_{arcs(S1)}}$ of Figure 5.13 on page 59.

Result: A transition $t2 = (\{S5, S6\},\ ev2,\ \{S4, S7\}) \in \mathcal{T}_{orth}$

(When $t2$ is taken, then $a2$ is executed.)

Problem: How are the sets $X = \{S5, S6\}$ and $Y = \{S4, S7\}$ obtained?

Since it is known that the transition arc $a2 = (S5, ev2, S7) \in \mathcal{T}_{orth_{arcs(S1)}}$ exits an *AND*-state, then the first step is to determine the exited *AND*-state. That is, $scope(a2) = S0$, $children(S0) = \{S1\}$ and $type(S1) \ = \ AND$.

The next step can be summarized by the following procedure:

Procedure 1

Let S1 be an orthogonal statechart consisting of two orthogonal components, S11 and S12 (i. e., $children(S1) = \{S11, S12\}$ *and* $type(S1) = AND$*), and let* $a_x = (s_{i_1}, ev_z, s_{i_2}) \in \mathcal{T}_{orth_{arcs(S1)}}$ *be a transition arc. Furthermore, let* $t_x = (\{s_{i_1}, s_{j_1}\},\ ev_m,\ \{s_{i_2}, s_{j_2}\}) \in \mathcal{T}_{orth}$ *be a transition which when taken, executes the transition arc* a_x*, with* $\{ev_z\} \cap \{ev_m\} \neq \emptyset$*. Note that* ev_m *can have more than one element (cf. Case (3)(a)ii., the* **otherwise branch** *of Definition 5.36). Then we compute the legal configurations (i. e., the source set* $X = \{s_{i_1}, s_{j_1}\}$ *and target set* $Y = \{s_{i_2}, s_{j_2}\}$*) of* t_x *as follows:*

I. *Construct the automata product matrix of* $S1$ *according to Definition 5.36 and Definition 5.37.*

II. *Let* C *be the automata product matrix constructed in step I above. Let* $G0$ *be the simple OR-graph product corresponding to the automata product matrix* C*. Let* L_e *be an empty edge list. Then beginning with the first vertex of* $G0$ *(i. e., start node* $v_0 := (v_{0_1}, v_{0_2})$*), use a* **depth first search**[29] *(dfs) to determine the legal configurations as follows:*

check if s_{i_1} *is element of the configuration* (v_{0_1}, v_{0_2})

CASE 1*: if true, then select if possible an outgoing edge with label* ev_z*. Let the target node* v_0' *of* ev_z *be* $(v_{0_1'}, v_{0_2'})$*, check if* s_{i_2} *is element of the configuration* $(v_{0_1'}, v_{0_2'})$*; if true, then insert this edge in the list* L_e*.*

The next node to be considered is $(v_{0_1'}, v_{0_2'})$*, (i. e.,* $v_0 := (v_{0_1'}, v_{0_2'})$*. For each such vertex we repeat the procedure above (CASE* 1*) until either not true or all edges are visted.*

CASE 2*: if false, (i. e.,* s_{i_1} *is not element of the configuration* (v_{0_1}, v_{0_2}) *then traverse the edges by* **depth first search** *(dfs) until either* s_{i_1} *is element of the*

[29] cf. Subsection 7.7.2

configuration (v_{0_1}, v_{0_2}) *or all edges are visted. If* s_{i_1} *is element of the configuration* (v_{0_1}, v_{0_2}) *proceed as above* (*CASE* 1).

CASE 3: *otherwise all edges are visted, then we are done.*

III. *The list* L_e *of step* II *contains all the possible configurations of the transition* t_x.

Figure 5.24 shows the simple OR-graph product $G0$ which represents $S1$ of Figure 5.13 on page 59. In our example above, $a_x = a2 = (S5, ev2, S7)$. $t_x = t2 \in L_e = \{(\{S5, S6\},\ ev2,\ \{S4, S7\}),\ (\{S5, S7\},\ ev2,\ \{S4, S7\})\}$.

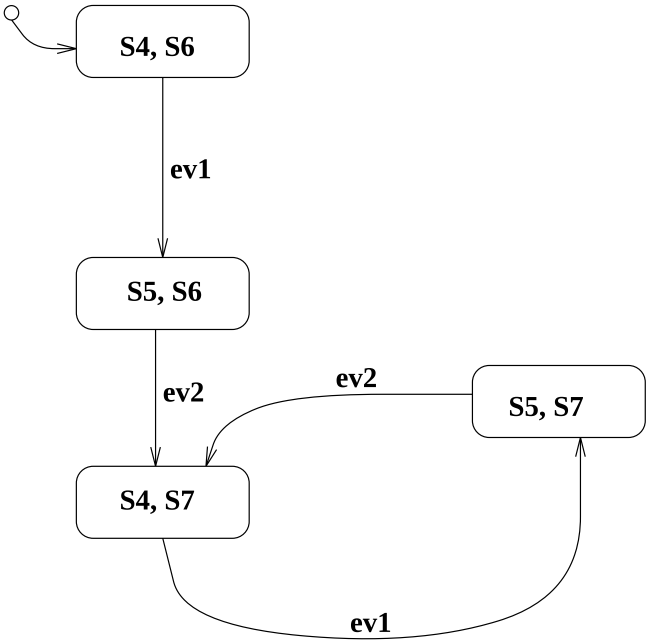


Figure 5.24: The simple OR-graph product $G0$ which represents $S1$ of Figure 5.13 on page 59.

Remark 5.13 *Note that the notion of scope does not depend on the way the arrow is drawn, but solely on its sources and targets.*

The sets of *entered* and *exited* states are determined as follows:

Definition 5.39 *The function* $exit_{top} : \mathcal{T}_{legal} \rightarrow S$ *determines for a taken transition* $t = (X, l, Y)$ *the* **highest exited state** (*on the statechart level*) $exit_{top}(t) \in S$ *where*

(i) $exit_{top}(t) \in children(scope(t))$ *and*

(ii) $LCA(X) \leq exit_{top}(t)$.

Definition 5.40 *The function* $exit : \mathcal{T}_{legal} \times \mathcal{C}_{full} \rightarrow 2^S$ *determines for a taken transition* $t = (X, l, Y)$ *and a configuration* $\mathcal{C}$ *the* **set of exited states** *by* $children^*(exit_{top}(t)) \cap \mathcal{C}$.

Definition 5.41 *The function* $enter_{top} : \mathcal{T}_{legal} \rightarrow S$ *determines for a taken transition* $t = (X, l, Y)$ *the* **highest entered state** (*on the statechart level*) $enter_{top}(t) \in S$, *where*

(i) $enter_{top}(t) \in children(scope(t))$ *and*

(ii) $LCA(Y) \leq enter_{top}(t)$.

Consider Figure 5.13 on page 59. Let $t2 = (\{S5, S6\},\ ev2,\ \{S4, S7\}) \in \mathcal{T}_{legal}$, then $scope(t2) = S0$ (cf. page 83); $exit_{top}(t2) = enter_{top}(t2) = S1$, since $LCA(X) = S1 = LCA(Y)$ and $S1 \in children(S0)$.

Now, given $\mathcal{C} = \{r, S0, S1, S2, S5, S3, S6\}$, then $exit(t2, \mathcal{C}) = \{S1, S2, S5, S3, S6\}$, since $children^*(exit_{top}(t2)) = \{S1, S2, S3, S4, S5, S6, S7\}$.

In Definition 5.23 and 5.24 on page 55 we described the *syntactical* transformations of a statechart which consists of linked transition segments into a statechart in CNF (see also Remark 5.7). Before we can give the definition of the function *enter*, it is necessary to establish the *semantical* definition of a *full compound transition* (full CT). The main idea behind the semantical definition of full compound transitions is that the (default) connectors 'vanish' from the original statechart. In other words, for every entered *OR*-state one default transition is added and the label of the full compound transition is the combination of the transition segments.

To illustrate this more precisely, consider Figure 5.25 which depicts a statechart that describes the behaviour of the telefax machine $BEAUTY$ in terms of its human interface. The combination of transition arcs from *SERVICE* to *SOURCE* with label *DISCONNECT_CABLE*, *BT_RM / dc!(MSG);fs!(POW_ON); fs!(BT_ON)* constitutes a full compound transition which in the following is denoted by $t1$. Of course, in the process of carrying out the set of transition arcs {*DISCONNECT_CABLE*, *BT_RM*}, the actions *dc!(MSG);fs!(POW_ON);fs!(BT_ON)* are carried out simultaneously. The meaning of $t1$ is that when events *DISCONNECT_CABLE* and *BT_RM* occur, then the telefax machine $BEAUTY$ leaves its *AND*-state *SERVICE* and enters the other *AND*-state *SOURCE*. Then entering *BATT_OFF* and *POWER_OFF* is determined by transition arcs with labels /fs!(BT_ON) and /fs!(POW_ON). After the execution of a full compound transition a new configuration is reached.

For the *semantical* definition, we characterize a *full CT* by a *condition* on the target set that determines a substate of an *OR*-state which will be entered when an *OR*-state is entered as follows:

Definition 5.42 *A transition* $t = (X, l, Y)$ *is a* **full CT** *iff for all substates* $s \in enter_{top}(t)$ *with* $type(s) = OR$

(a) either an entered substate s is determined by Y, *i. e.,*
$Y \cap children^*(children(s)) \neq \emptyset$,

(b) or the state s itself is not entered, i. e.,
$\exists s' \in parents^*(s), s'' \in parents^*(Y) : s' \neq s'' \wedge parents(s') = parents(s'') \wedge type(parents(s')) = OR$.

Notation: *The set of all legal full CTs will be denoted by* $\mathcal{CT}_{legal}$.

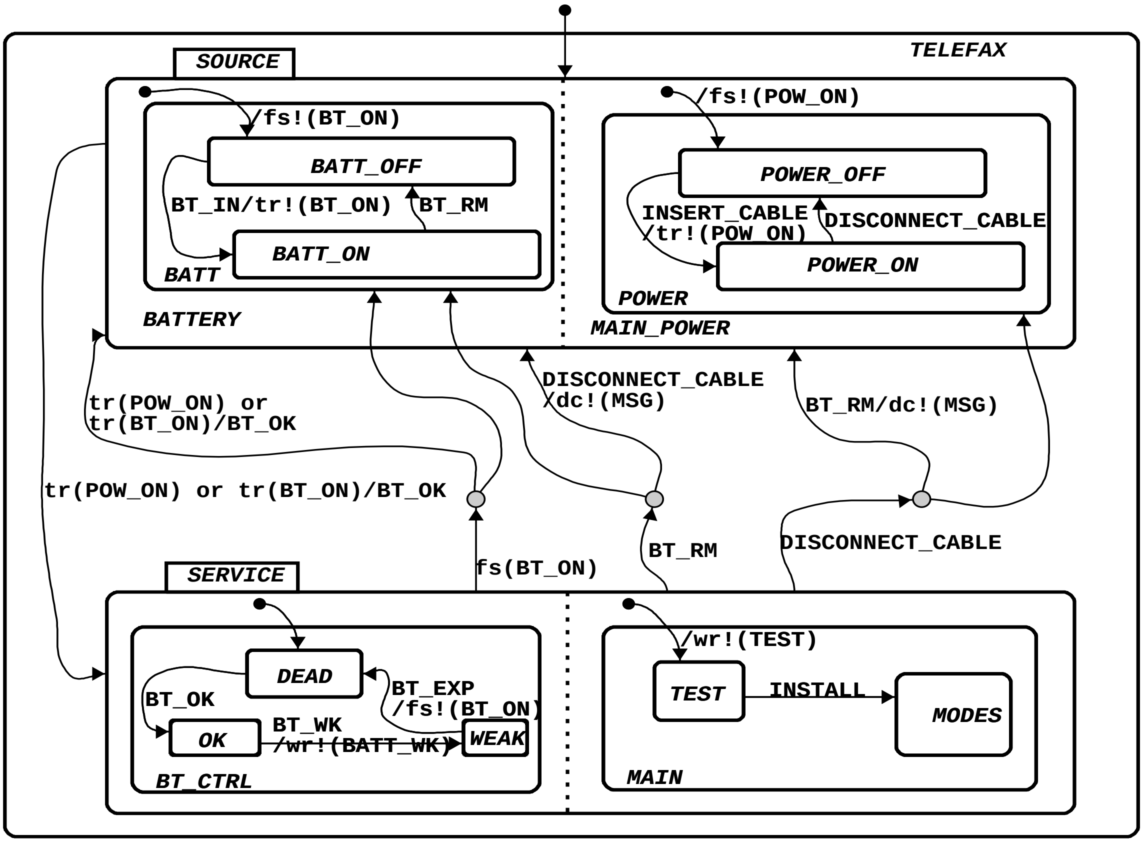


Figure 5.25: The statechart for the telefax machine *BEAUTY*.

Example for Definition 5.42

Consider again Figure 5.13 on page 59.

$t2 = (\{S5, S6\},\ ev2,\ \{S4, S7\}) \overset{?}{\in} \mathcal{CT}_{legal}$

$enter_{top}(t2) = S1$

$children(S1)$ ={S2, S3}

$type(S2) = OR = type(S3)$

According to Definition 5.42, transition $t2 \in \mathcal{CT}_{legal}$, since

for part (a): {S4, S7} $\cap$ $children(S2) = S4 \neq \emptyset$,

for part (a): {S4, S7} $\cap$ $children(S3) = S7 \neq \emptyset$.

Now the definition of the function *enter* is given as follows:

Definition 5.43 *The function enter* : $\mathcal{CT}_{legal} \to 2^S$ *maps a taken transition* $t = (X, l, Y)$ *to the set of entered states*

$enter(t) = parents^*(Y) \cap children^*(enter_{top}(t)).$

Recall Figure 5.13, where given $\mathcal{C} = \{r, S0, S1, S2, S5, S3, S6\}$, arc $a2 = (S5, ev2, S7)$ leads to the transition $t2 = (\{S5, S6\},\ ev2,\ \{S4, S7\}) \in \mathcal{C}T_{legal}$ and $enter_{top}(t2) = S1$. Therefore, $enter(t2) = parents^*(\{S4, S7\}) \cap children^*(S1) = \{S1, S2, S4, S3, S7\}$.

The above definitions can thus be used to determine the entered (target) set and exited (source) set. In addition, the problem of transitions which cross the boundaries of nested states as mentioned in Subsection 5.3.1 (page 59) can also be dealt with by applying the definitions above (see next Subsection 5.4.5).

5.4.5 Conflict between transitions (nondeterminism)

As stated in Subsection 1.2.2.1 on page 5, in the design of real life complex systems there is a variety of nontrivial situations. One of the most delicate issues are conflicts between transitions. Informally, two transitions are in **conflict** if there is some common state that would be exited if any one of them were to be taken.

In Figure 5.26, e. g., $t1$ and $t2$ are in conflict because — in case of $ev1$ occuring — they each imply exiting state $S11$. In addition, $t4$ is in conflict with all of $t1, t2$ and $t3$ because if and when $t4$ is taken, the system must have been in $S1$ and thus also in one of its substates. In this particular case, it follows e. g., that $t2$ and $t4$ cannot be taken in the same "step". This notion can be defined in two ways, *with respect to a configuration* or *independent of a configuration*, as follows:

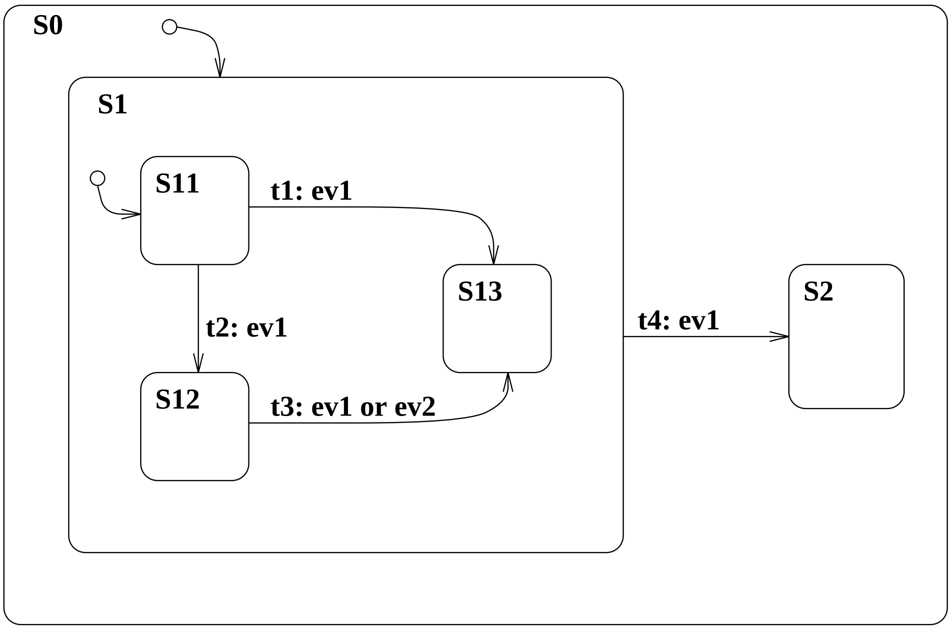


Figure 5.26: Illustrating some conflicts

Definition 5.44 *Let a statechart* $Sct = (S, r, children, type, \mathcal{H}, hist, default, \mathcal{T})$ *be given. Two transitions* $t_1 = (X_1, l_1, Y_1)$, $t_2 = (X_2, l_2, Y_2) \in \mathcal{T}$ *with* $t_1 \neq t_2$ **are in conflict**

(1) **with respect to a configuration** $\mathcal{C} \in \mathcal{C}_{full}$ *iff*

(a) $exit(t_1, \mathcal{C}) \cap X_2 \neq \emptyset \vee exit(t_2, \mathcal{C}) \cap X_1 \neq \emptyset$, *or*

(b) $exit(t_1, \mathcal{C}) \cap exit(t_2, \mathcal{C}) \neq \emptyset$ *holds.*

(2) **independent of a configuration** *iff*

(a) $exit_{top}(t_1) \leq exit_{top}(t_2) \vee exit_{top}(t_2) \leq exit_{top}(t_1)$,

(b) $type(LCA(\{scope(t_1), scope(t_2)\})) = OR$, *and*

(c) $\neg(scope(t_1) \perp scope(t_2))$.

5.4.6 Dealing with conflict between transitions (nondeterminism)

As a solution to conflicting transitions, the notion of *priority* has been proposed in [54]. Assume that t_1 and t_2 are two conflicting transitions with respect to a configuration, and that $scope_1$ and $scope_2$ are their scopes, respectively. According to Definition 5.44, case (2), when two transitions are in conflict, there must be a common state in their source states. This implies that their scopes cannot be orthogonal or exclusive. In other words, either they are equal or one of the scopes is an ancestor of the other in the state hierarchy (cf. Definition 5.44 case (2), part (a) and (b)).

In [54], priority is given to the transition whose scope is higher in the hierarchy. Of course, if $scope_1 = scope_2$, neither t_1 nor t_2 has priority over the other in which case *nondeterminism* occurs. In this particular case, one of the possible transitions is chosen nondeterministically. A formal definition of priority will not be given in this thesis. Instead an example is given. Nonetheless, in Subsubsection 6.2.1.1 we describe an algorithm $\boldsymbol{Generate\_\mathcal{T}_{orth}}$ which will be used to determine the sets of maximal nonconflicting transitions $\boldsymbol{\mathcal{T}_{orth}} \subseteq \mathcal{T}$ (cf. Remark 5.11, page 74).

Example for the notion of priority

In Figure 5.27, e. g., the scope of $t3$ and $t4$ is $S21$, the scope of $t5$ is $S2$ and the scope of $t1$ is $S1$. Thus $t3$ and $t4$ have the same priority, and $t5$ has a higher priority than $t3$ and $t4$. Let $\mathcal{C} = \{r, S00, S0, S1, S11, S2, S21, S211, S2112\}$ be a configuration in Figure 5.27. Then, $t1 = (\{S11\}, ev1, \{S12\})$ and $t3 = (\{S2112\}, ev3, \{S2122\})$ are not in conflict with respect to $\mathcal{C}$, because $exit(t1, \mathcal{C}) \cap \{S2112\} = \emptyset$ as well as $exit(t3, \mathcal{C}) \cap \{S11\} = \emptyset$.

Recall static reactions (SRs) which were briefly introduced in Subsection 4.2.1.5. An "enabled" SR defined in a state $s \in S$ is executed if the system was in s at the beginning of the step but s was not exited by any full compound transition (CT) during the step. Following the notion of scope, one might say that an SR defined in a state s is in conflict with all the CTs that exit s or one of s's ancestors. In addition, CTs have higher priority than SRs since if a state s is exited as a result of some CT, its SRs will not be executed in that step.

Throughout the rest of this thesis, unless stated otherwise, only legal full CTs ($\mathcal{C}T_{legal}$) will be considered. However, we will use $\mathcal{T}_{legal}$ instead of $\mathcal{C}T_{legal}$.

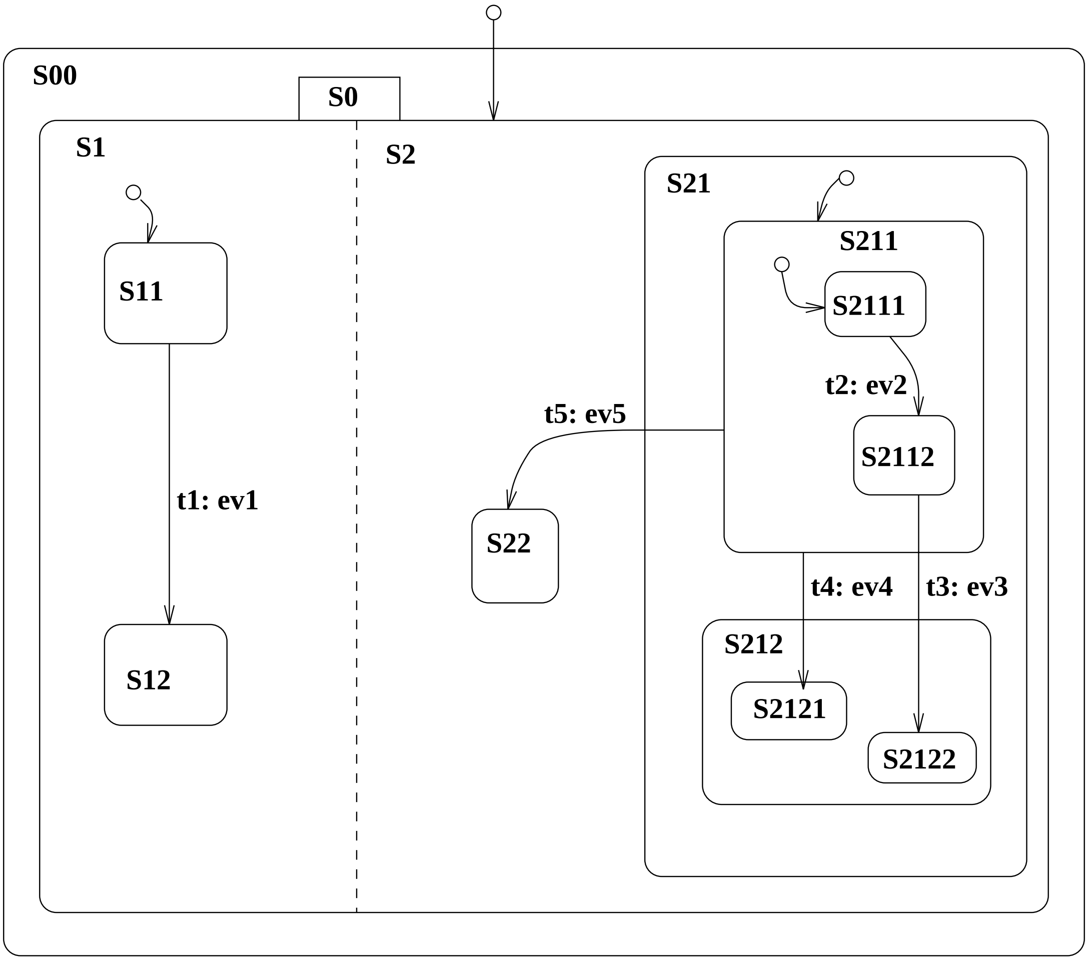


Figure 5.27: Illustrating priority

5.4.7 Summary of Section 5.4

In this subsection we present a summary of Section 5.4. The main goal of Section 5.4 is to provide concepts which can be used to overcome some of the problems, e. g., inter-level transitions discussed in Section 5.3. In addition, Section 5.4 deals with the formal aspects which will be required to describe and formalise the step semantics based on a *qualitative time model* and the one based on a *quantitative time model* in Chapter 6.

In Subsection 5.4.1 we present the concept of *legal configuration*. The formal description of a configuration is necessary for the definition of the *scope of transitions* which is presented in Subsection 5.4.4. The scope concept of transitions is used to handle nondeterminism and to associate priorities to transitions. Above all, the configuration definitions will be used to determine *legal statuses* defined in Subsection 6.2.1.

In Subsection 5.4.2 we consider the case of a direct exit of an *AND*-state and give the formal concept of a 'legal' transition. In case of a direct exit of an *AND*-state the definition of a *simple statechart* presented in Subsection 5.2.3.1 cannot be used to compute transitions in parallel states. Therefore, we present a new definition of a *complex statechart* (cf. Definition 5.28). Furthermore, we remark that the requirements stated in the definition of 'legal' transitions in [51] are necessary, but they are not sufficient to define a *legal* transition. Consequently, we develop a formal concept of *legal* transitions (cf. Definition 5.32).

In Subsection 5.4.3 we present techniques which can be used to transform AND-states into OR-states. The transformation of an AND-state into an OR-state corresponds to building the automata product of an orthogonal statechart (cf. Definition 5.12). This subsection, therefore, is devoted to the construction of this automata product. In particular, it is shown how the adjacency-matrix technique can be used for determining the corresponding automata product of an orthogonal statechart. Since the constructed automata product is different from the conventional automata product we refer to it as *modified automata product.*

The main goal is to achieve an OR-graph which represents the OR-states derived from the AND-states. The OR-graph is used in the computation of all legal configurations of the AND-state. These legal configurations are necessary for the illustration of the concept of scope of transitions (cf. Subsection 5.4.4). Theorem 5.4 summarises that "the nodes of a simple OR-graph product $G0$ of an orthogonal statechart $S0$ deliver the set of all (possible) basic configurations of the orthogonal statechart $S0$". In addition, the edges of $G0$ deliver $\mathcal{T}_{orth}$, the set of legal transitions of orthogonal statechart *S0*.

In Subsection 5.4.4 we formalise a concept — the notion of scope of transitions — that can be used to deal with interlevel transitions which in general deny a structural operational semantics for statecharts.

The definitions presented in this subsection thus can be used to determine the entered (target) set and exited (source) set. In addition, the problem of transitions which cross the boundaries of nested states as mentioned in Subsection 5.3.1 can also be dealt with by applying these definitions (cf. Subsection 5.4.5).

In Subsection 5.4.5 we formally describe one of the most delicate aspects in the design of real life complex systems, namely *conflicts between two transitions.* In this context, we describe the notion of *priority* which has been proposed in [54] as a solution to conflicting transitions.

Chapter 6

Step Semantics of Statecharts

Following the STATEMATE specification, the statechart behaviour is described in terms of *steps* (cf. Figure 1.3, page 5). The semantics of statecharts described in this chapter, therefore, deal in general with defining the behaviour precisely in terms of transitions "which execute a step", i. e., describing a step formally, with all its ramifications and side effects. In [54], some of the general principles that have been adopted in defining the semantics are stated informally as follows:

(a) Reactions to external and internal events, and changes that occur in a step, can be sensed only after completion of the step.

(b) Events "live" for the duration of one step only (the one following the step in which they occur) and are not "remembered" in subsequent steps.

(c) Calculations in one step are based on the situation at the beginning of the step.

(d) A maximal subset of 'non-conflicting' transitions and 'static reactions' is always executed.

In this thesis, we describe two basic models of execution of a step. The first one is a *qualitative time abstraction*, i. e., the environment can be seen as a *discrete* process, namely as an infinite sequence of inputs $I_1, I_2, \ldots$ occuring at successive instants of time. The system is faster than the environment, namely the reaction of the system to the inputs I_i is completed before the inputs I_{i+1} are produced. The second model is a *quantitative time abstraction* based on an internal clock.

6.1 Overview of step semantics of statecharts

Figure 6.1 presents an overview of the approach which is developed in this chapter to establish a formal step semantics of statecharts. According to Figure 6.1, on one side, STATEMATE offers a simulation tool to examine the behaviour of statechart model. On the other side, we require a formal step semantics of statecharts.

Simulation tool (STATEMATE)

STATEMATE offers a **simulation tool** to examine the behaviour of model.

'Execution of a step' $\hat{=}$ "run" the model in a step-by-step interactive fashion.

Static Tests: E. g.,

♣ *Reachability Test*
♣ *Detection of nondeterminism*
♣ *Detection of deadlock*
♣ *Usage of transitions*

Step Semantics

Initial Situation:
Use simple OR-graph product $G0$ (cf. 5.4.3)
to develop test paths (TSs)
which allow a systematic analysis of,
e. g., detecting nondeterminism

Problem:
Executability of the transitions $\mathcal{T}_{orth}$ *of* $G0$
Each transition $t \in \mathcal{T}_{orth}$ is **legal**
but not necessarily 'executable', i. e.,
the TSs of $G0$ are not suitable for the detection
of the other system flaws e. g., deadlock.

Required:
formalisation of
step *with all its ramifications and side effects*

⇓

Step semantics of statecharts
Basic step algorithms

⇓

Go-Step and *Go-Repeat* algorithms
based on a *qualitative time model*

⇓

Go-Step and *Go-Repeat* algorithms
based on *quantitative time model*

Figure 6.1: Overview of step semantics of statecharts

.

The most basic way of STATEMATE to execute or to "run" the model is in a step-by-step interactive fashion (cf. Chapter 1, pp. 4ff.). At each step, the user generates external events, changes conditions and carries out other actions (such as changing the values of variables) at will, thus emulating the environment of the system. The tool (STATEMATE) in turn responds by transforming the system into the new resulting status. Typically, there will be one or more statecharts on the screen during this process, and often also an activity-chart (cf. Subsubsection 4.1.1.2, page 25). The change in status will be reflected visually by changes in colour of the currently active states and activities. In fact, by animating charts, simulation allows one to see how and why the model transforms from one state to another at a stage of the development. Being able to simulate the model (i.e., to run through dynamic scenarios) gives the designer the ability to verify that the specifications satisfy the functional requirements of the system — *long before final implementation.* If he/she finds that the system's response is not exactly as expected, he/she goes back to the model, changes it (by modifying a statechart, for example), and runs the same scenario again.

Description of static tests: Besides, STATEMATE offers a number of static tests, such as the ones shown in Figure 6.1. In the following, we give a brief description of these static tests (see also Subsection 4.1.2).

- *Reachability of conditions* — examines a set of conditions to determine whether they may be reached by the STATEMATE specification.
- *Nondeterminism* — checks the STATEMATE specification for a scenario that leads to an ambiguous system reaction (cf. Subsection 5.4.6).
- *Detection of Deadlock* — Deadlock refers to a reachable condition that remains true forever after it has been reached (not necessarily for the first time) and there are no processes (components) which require the condition to become false in order to proceed.
- *Usage of Transitions* — looks for static reactions and transitions that can never be taken and for states that the specification never enters.
  - A transition is not taken if its source is never reached or if its trigger never occurs when the system is in its source $state(s)$.
  - Static reactions are not used if the state in which the reaction takes place is never entered or if its trigger never occurs when the system is in this state (cf. Subsection 4.2.1.5, and see Subsubsection 6.2.1.2 "Example for nonconflicting transitions", page 100).

Now let us briefly consider the formal concepts and techniques we have developed so far with respect to the above described static tests.

Initial situation: Using the simple OR-graph product of an orthogonal statechart, constructed by Definition 5.36 and Definition 5.37 we are able to develop test paths which allow the system designer/tester to make a systematic analysis of, e. g., detecting nondeterminism (cf. Illustration of Case (3)(a)ii. of Definition 5.36, pp. 77ff.).

Problem: These test paths of a simple OR-graph product, however, are not suitable for the detection of the other system flaws like deadlock because they do not consider the "enabledness" of the transitions. In other words, each transition $t \in \mathcal{T}_{orth}$ delivered by the simple OR-graph product is legal but not necessarily 'executable'.

Required: To be in a position of dealing with system flaws, such as deadlock, therefore, it is necessary to formalise *the step with all its ramifications and side effects* as stated at the beginning of this Chapter 6. Hence, in next Section 6.2 and Section 6.3 we deal with the formalisation of the *step* algorithms presented in the informal paper of [54].

In the previous Section 5.4 we dealt with the formal aspects which will be required for the description and formalisation of the step semantics based on *qualitative time model* in Section 6.2 and the step semantics based on *quantitative time model* in next Section 6.3.

Chapter 6 is organised as follows. In Section 6.2 we give a formal description of the step semantics based on a *qualitative time model abstraction.* After that, we formalise in Section 6.3 the step semantics based on a *quantitative time model abstraction.* We then give a brief view of statecharts in object-oriented designs in Section 6.4 before we discuss, in Section 6.5, the suitability of statecharts/STATEMATE for the development of large and complex reactive systems. This discussion is based on the personal experience of the author.

6.2 Step semantics of statecharts based on the model of qualitative time abstraction

The purpose of this section is to formalise the parts of the *basic step* algorithms presented in the informal paper of [54].

An *operational semantic* describes the algorithm by which the results of a *step* are calculated, given a configuration and external stimuli. The results of a step are the configuration at the end of the step, and the output, e. g., actions generated by the statechart during the step. According to Definition 5.26 on page 67, the *configuration* of a statechart is the set of basic states in which a statechart currently resides. The *system status* of a statechart at a given instance, however, consists of its configuration, and the values of external events and condition variables.

Recalling Definition 1.1 on page 1, a reactive system is one that is in continual interaction with its environment and executes at a pace determined by that environment. In Figure 6.2 we present a functional view of a reactive system and its environment. In it, the system part consists of a system processing unit *System_Step*, a system status *System_Status*, and a unit *Merge*. The environment part consists of an environment status *Environment_list*, and a unit *Operate*.

The unit *Operate* is an extension of the basic step algorithm in [54]. The interactions between the system and its environment take place over *Environment_list*, where unit *Operate* provides inputs to the system and unit *Merge* adds[1] inputs from the environment to the system status. *System_Status* contains the last system status on which the system processing unit *System_Step* depends. The result of the unit *System_Step* is a new system status.

[1] see Merge step on page 102

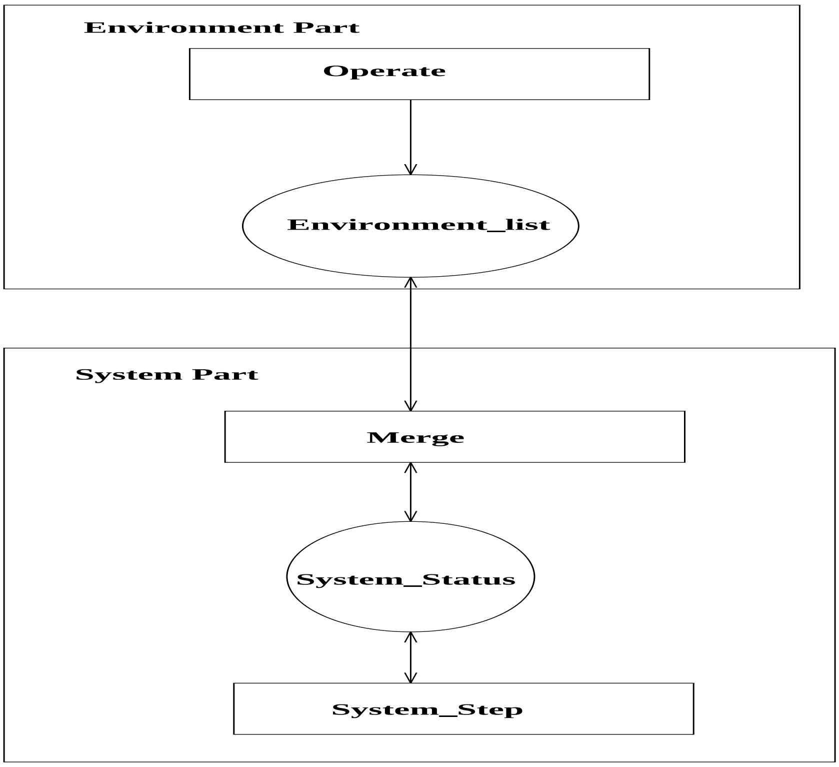


Figure 6.2: A functional view of a reactive system and its environment

In Subsection 6.2.1 we particularly establish the formal definitions of the **system part**: *System_Status* and *System_Step*. In Subsection 6.2.2 we then develop the formal definitions of the **interface part**, namely the *status of the interface list* called *Environment_list* and *Merge step* of unit *Merge* as shown in (cf. Figure 6.2). Last but not least, we give the definition of *Put* step which models how the environment provides inputs to the system. In Figure 6.2, *Put* is represented by unit *Operate*. Using these formal definitions we give in Subsection 6.2.3 a formal description of the *basic step* algorithms (untimed part), the *Go-Step* and *Go-Repeat* algorithms of [54].

6.2.1 The basic step algorithm (system part)

In Subsection 1.2.2.1, page 4, rather informal definitions of **step** and **status** were given. In the following, we give formal definitions which are used to compute the contents of the basic step part of the algorithm.

Let a statechart $Sct = (S, r, children, type, \mathcal{H}, hist, default, \mathcal{T})$ be given. In the following context, $\mathcal{T}$ is the set of full compound transitions[2] containing at least one *initial transition*[3] Remark also that static reactions (SRs)[4] may be executed.

[2] cf. Definition 5.42

[3] cf. Definition 5.22

[4] cf. Subsubsection 4.2.1.5, page 31

Definition 6.1 *A* **system status** *is a triple* $(\mathcal{C}, SE, SV) \in \mathcal{C}_{full} \times 2^{SE_p} \times 2^{SV_p}$ *consisting of a* **configuration** $\mathcal{C} \in \mathcal{C}_{full}$*, a set of ("enabled") events* $SE \in 2^{SE_p}$ *and a set of ("true") conditions* $SV \in 2^{SV_p}$*.*
Notation*:* $\boldsymbol{System\_Status_{all}}$ *denotes the set of statuses. A system status* $(\mathcal{C}, SE, SV) \in System\_Status_{all}$ *with* $\mathcal{C} = \{r, init\}, SE = \emptyset$ *and* $SV = \emptyset$ *is called* **initial status**. *The set of initial statuses is denoted by* $\boldsymbol{System\_Status_{init}}$*.*

The system processing step has the function of executing "enabled" transitions. According to [54], "*a transition is said to be* **enabled** *in a step if at the beginning of the step the system is in all its source states and its triggers,* (*e. g., events*) *are true*". For example, in Figure 6.3, assuming that at the beginning of the step the system was in state $S1$ and events $e1$ and $e2$ as well as conditions $c1$ and $c2$ were generated during the previous step, $t1$ becomes enabled but $t2$ not. This is due to the fact that although $e2$ and $c2$ were true at the beginning of the step and that the source state of $t2$ is entered during the step, the system was not in $t2$'s source at the beginning of the step.

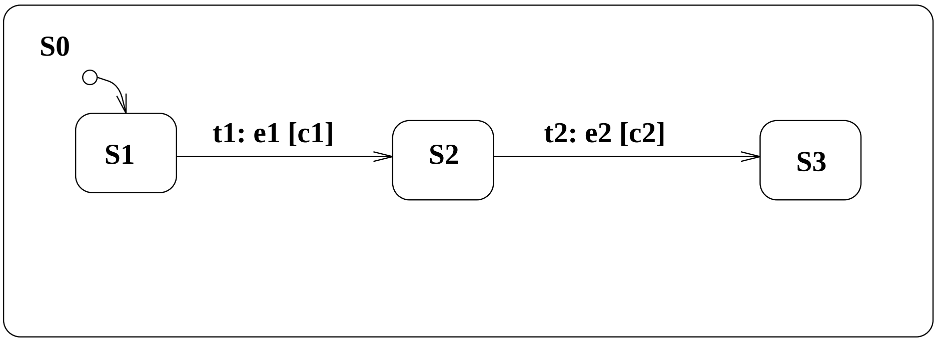


Figure 6.3: Illustrating an *enabled* transition

In other words, a transition $t = (S1, e[c], S2)$ is said to be enabled in a status $(\mathcal{C}, SE, SV)$ iff its source states $S1$ are part of the actual configuration $\mathcal{C}$ and both the event e as well as the condition expression c evaluates to true.

Before the formal definition of a transition being *enabled* is given, it is necessary to establish the *evaluation of the event and condition expressions.* Assignments to condition variables can be regarded in STATEMATE as the set of internally generated tuples (SV, val), where val is a new value for SV (cf. Definition 5.8 and Definition 5.7).

Let $Val = SV_p \times \{true, false\}$. Val is the assignment (cf. Definition 5.8 and Definition 5.7. $Val^\star$ denotes lists over Val.

Informally speaking, a transition with label $e[c]/a$ is said to be **enabled** in a system status of the statechart iff the event expression e evaluates to true for the set of availabe events in the current system status and the condition expression c evaluates to true (in the current system status) for the set of condition variables.

For the evaluation of event and condition expressions, the following relations are introduced.

Definition 6.2 *The relations* $EvalEv \subseteq \mathcal{L}_e \times 2^{SE_p}$ *and* $EvalCo \subseteq \mathcal{L}_c \times 2^{SV_p}$ *are used* *to describe the* **evaluation of event and condition expressions** *in a system status* $(\mathcal{C}, SE, SV)$*, respectively.*

For the evaluation of action expressions, the functions $EvalAc$ and $EvalVlist$ are defined as follows:

Definition 6.3 *Function* $EvalAc : \mathcal{L}_a \times 2^{SV_p} \rightarrow 2^{(2^{SE_p} \times Val^\star)}$ **evaluates for an action expression and values of condition variables** *a set, where each element consists of a set of generated events and a list of new values for condition variables.*
The evaluation of an action expression $a_1; a_2$ *(a non-deterministic execution of* a_1 *and* a_2*) is given by*

$$EvalAc(a_i; a_2) = \{(SE_1 \cup SE_2, SVlist_i * SVlist_j) \mid i, j \in \{1, 2\}, i \neq j,$$

$$\textit{where } (SE_1, SVlist_1) \in EvalAc(a_1),\ (SE_2, SVlist_2) \in EvalAc(a_2)\}.$$

Definition 6.4 *Function* $EvalVlist : Val^\star \times 2^{SV_p} \rightarrow 2^{SV_p}$ *computes for the previous set of true conditions and a list of new values for condition variables a set of true conditions.*

Now the formal definition of a transition being *enabled* is given.

Definition 6.5 $Enable \subseteq \mathcal{T} \times System\_Status_{all}$ *denotes the set of tuples* $(t, (\mathcal{C}, SE, SV))$ *such that transition* t *is* **enabled** *in the system status* $(\mathcal{C}, SE, SV)$*:*
For $t = (X, e[c]/a, Y)$, $(t, (\mathcal{C}, SE, SV)) \in Enable$ *iff* $X \subseteq \mathcal{C}$ *and both* $EvalEv(e, SE)$ *and* $EvalCo(c, SV)$ *evaluate to true (cf. Definition 6.2).*

6.2.1.1 Computation of maximal nonconflicting sets

The operational semantics of statecharts as described in STATEMATE is a maximal parallelism semantics inspired by the synchrony hypothesis [11]. Consequently, part of the basic step algorithm consists of the execution of a set of transitions that can be executed simultaneously. Obviously, such a set of transitions $\boldsymbol{\mathcal{T}_{orth}} \subseteq \mathcal{T}$ (cf. Remark 5.11, page 74) similarly depends on the actual system status $System\_Status \in System\_Status_{all}$. In the following, we describe an algorithm $\boldsymbol{Generate\_\mathcal{T}_{orth}}$ which will be used to determine the set of transitions $\mathcal{T}_{orth}$.

Let $Max \subseteq System\_Status_{all} \times 2^{\mathcal{T}}$ and $(System\_Status, \mathcal{T}_{orth}) \in Max$. Then proceed as follows

(1) /* Compute for actual state $System\_Status$ the maximal set of enabled transitions */
$Enable_{System\_Status} = \{t \in \mathcal{T} \mid Enable(t, System\_Status)\}$.

(2) /* Remove from $Enable_{System\_Status}$ those transitions which are in conflict with other enabled transitions of higher priority*/
$Enable^{-}_{System\_Status} = Enable_{System\_Status} \setminus$
$\{t_{confl} \in Enable_{System\_Status} \mid \exists t \in Enable_{System\_Status} : t_{confl} < t \wedge conflict(t_{confl}, t)\}$

(3) /* Compute T_{orth} as a maximal subset of $Enable^{-}_{System\_Status}$ containig no conflicting transitions */
$\mathcal{T}_{orth}$ must satisfy the following requirements:

(a) for all $t_1, t_2 \in \mathcal{T}_{orth} : t_1 = t_2 \vee \neg conflict(t_1, t_2)$, and

(b) for all $t_{confl} \in Enable^{-}_{System\_Status} : t_{confl} \in \mathcal{T}_{orth} \vee$ $(\exists t \in \mathcal{T}_{orth} : (t_{confl} \neq t \wedge conflict(t_{confl}, t)))$.
In other words, each enabled transition not included in the T_{orth} is in conflict with at least one transition in the T_{orth}.

Remark 6.1 In part (3) of the above computation of the set transitions $\mathcal{T}_{orth}$, being *maximal* means that each enabled transition not included in the set is in conflict with at least one transition in the T_{orth}.

6.2.1.2 Example for nonconflicting transitions

As an example, Figure 6.4 is used to demonstrate the nonconflicting sets. Note that the various *sr*'s indicated in it denote the static reactions associated with the corresponding states. Assume that the system is in the configuration depicted by the shaded basic states, and that incidentally all the transitions and static reactions are enabled.

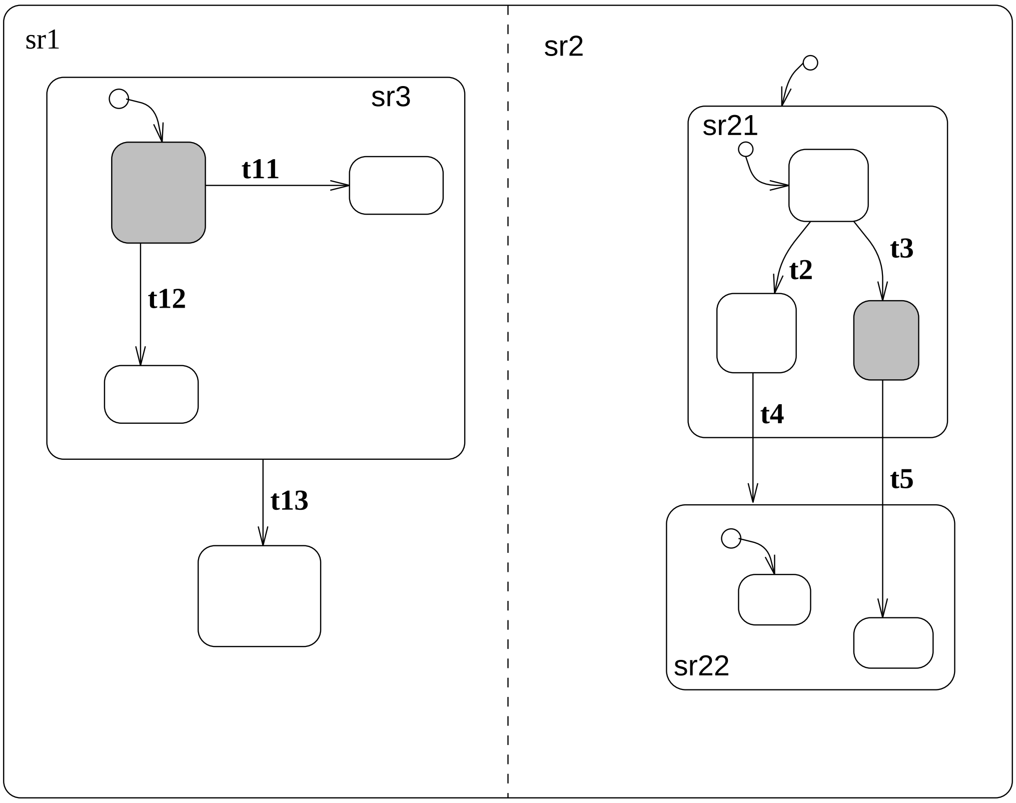


Figure 6.4: Nonconflicting transitions

Recall the notion of priority which was discussed in Subsection 5.4.6. Indeed, $t13$ has higher priority than $t11$ and $t12$ in the left-hand component of Figure 6.4. In the right-hand component, $t4$ and $t5$ have the same priority but higher than that of $t2$ and $t3$. Thus, in the left-hand component $t13$ will be taken which implies that $sr1$ will be carried out in this step but not $sr3$. In the right-hand component $t4$ or $t5$ will be taken, and $sr2$ will be carried out. As a result the maximal nonconflicting sets are as follows:

$$\{t13, t4, sr1, sr2\}$$

$$\{t13, t5, sr1, sr2\}$$

6.2.1.3 Execution of a step

At the beginning of this Chapter 6 we stated general principles adopted in defining the semantics of statecharts. In the following, we define the principle (b) which deals with the duration of events is modelled by the step *System_Step*.

Definition 6.6 *The* **system step** *System_Step is defined as the relation* $System\_Step \subseteq (System\_Status_{all} \times 2^{\mathcal{T}} \times System\_Status_{all})$, *such that* $((\mathcal{C}_0, SE_0, SV_0), \mathcal{T}_{orth}, (\mathcal{C}', SE', SV')) \in System\_Step$ *iff* $((\mathcal{C}_0, SE_0, SV_0), \mathcal{T}_{orth}) \in Max$ *and exactly one of the following conditions is satisfied:*

(1) If $|\mathcal{T}_{orth}| = 0$,
then the step is empty and $(\mathcal{C}', SE', SV') = (\mathcal{C}_0, \emptyset, SV_0)$.
/ There is no transition enabled */*

(2) Otherwise,
let $\{t_1, \ldots, t_{|\mathcal{T}_{orth}|}\} = \mathcal{T}_{orth}$, *and* $t_i = (X_i, e_i[c_i]/a_i, Y_i)$ *for* $i \in \{1, \ldots, t_{|\mathcal{T}_{orth}|}\}$. *Then, for all* $i \in \{1, \ldots, t_{|\mathcal{T}_{orth}|}\}$ *there exist tuples* $(\mathcal{C}_i, SE_i, SVlist_i) \in \mathcal{C}_{full} \times 2^{SV_p} \times SV^*_{cond}$ *(cf. Definition 5.9), such that* $(SE_i, SVlist_i) \in EvalAc(a_i, SV_0)$ *(cf. Definition 6.3)* $\wedge\, \mathcal{C}_i = (\mathcal{C}_{i-1} \backslash exit(t_i, \mathcal{C}_0)) \cup enter(t_i)$ *and* $\mathcal{C}' = (\mathcal{C}_{|\mathcal{T}_{orth}|}$, $SE' = \bigcup_i i \in \{1, \ldots, t_{|\mathcal{T}_{orth}|}\} SE_i$, $SV' = EvalVlist(SV_0, SVlist_1 * \ldots * SVlist_{|\mathcal{T}_{orth}|}))$ *(cf. Definition 6.4).*

In fact, the above definition formalises most sensitive part of the basic algorithm.

Denotation 6.1 *We write* $((\mathcal{C}_0, SE_0, SV_0), (\mathcal{C}', SE', SV'))$ *instead of* $((\mathcal{C}_0, SE_0, SV_0), \mathcal{T}_{orth}, (\mathcal{C}', SE', SV')) \in System\_Step$.

6.2.2 The basic step algorithm (interface part)

The interaction between the system and its environment is described by a list which contains the changes generated by the environment[5] since the last internal system step.

Definition 6.7 *A* **status of the interface list** *is a pair* $(SE, SV) \in 2^{SE_p} \times SV^*_{cond}$, *where SE is a set of events and SV is a list of actions over condition variables. The set containg all statuses of the interface list is denoted by* $Environment\_list_{all}$. *A status of the interface list satisfying the predicate* $\Theta_{Env}(SE_{env}, SV_{env}) \equiv SE_{env} = \emptyset \wedge SV_{env} = \epsilon$ *is the* **initial status**.

[5] the external events and internally generated events

6.2.2.1 Merge step

Indeed, the basic step algorithm adds the external events to the list of internally generated events and in addition executes all actions implied by the external changes (cf. [54] page 18.) This part is described by the *Merge* step in the following definition.

Definition 6.8
The **Merge step** *is a relation Submit* $\subseteq$ $(System\_Status_{all} \times Environment\_list_{all} \times System\_Status_{all} \times Environment\_list_{all})$, *such that* $((\mathcal{C}, SE, SV), (SE_{env}, SVlist_{env}), (\mathcal{C}', SE', SV'), (SE'_{env}, SVlist'_{env})) \in$ *Submit iff the new system status* $(\mathcal{C}', SE', SV') = (\mathcal{C}, (SE \cup SE_{env}), EvalVlist(SV, SVlist_{env})$ *and the new interface status* $(SE'_{env}, SVlist'_{env}) = (\emptyset, \epsilon)$.

In other words, the *Merge* step adds the interface status to the actual system.

Last but not least the definition of *Put* step which models how the environment provides inputs to the system is given.

Definition 6.9 *The relation Put* $\subseteq Environment\_list_{all} \times Environment\_list_{all})$ *describes the* **changes of the environment status**.

6.2.3 The Go-Step and Go-Repeat algorithms (untimed part)

Next, the communication of statecharts and environment is investigated. Based on the report of Harel and Naamad in [54], section 9, there are two communication modes:

- the **synchronous time model** in which case the communication with the environment is performed after each basic step and
- the **asynchronous time model** where the communication is achieved by allowing several basic steps to take place within a single point in time. Such a collection of steps is referred to as a **super-step**.

Note that in both models, the execution of the step describes a sequence of discrete events that cause abrupt changes (taking no time) in the state of the system, separated by intervals in which the system's state remain unchanged. This approach has proven effective for describing behaviour of programs and other digital programs. This discrete event approach benefits from the assumption that the environment, similar to the system itself, can as well be modelled as a discrete process. The advantage of this assumption is that it allows a completely symmetrical treatment of the system and its environment, and encourages modular analysis of systems. What is considered an environment in one phase of the analysis may be considered a component of the system in the next phase.

By the comparison of statecharts variants, in Subsection 5.3.2 we learnt that the main differences between *RSML* and statecharts are syntactical. Nevertheless, there exists as well a significant difference between RSML's and statecharts' dynamic semantics in the use of the terms.

The *RSML* semantic definition of a step is based on the notion of steps in [131]. *A* **step** *is completed when no more internal events are generated or there are no more transitions triggered by events that were generated, i. e., the model has stablized in a state.*

In statecharts semantics, a *step* is the basic atomic operation while a *super-step* composes of several steps. In the RMSL approach, however, a statecharts step can be regarded as a *micro-step*, whereas a statecharts super-step is their step (basic atomic operation). The main advantage of the statecharts step approach is that one can easily show that each step construction will always terminate since the set of enabled transitions $Enable \subseteq \mathcal{T}$ (cf. Definition 6.5) is finite. On the contrary, the step construction in RSML state machines can pontentially be infinite as is shown in Figure 6.5. In Figure 6.5, the events $e1$ and $e2$ will be generated forever in an alternating sequence.

Nevertheless, Leverson et al, argued that the RSML step (super-step of STATEMATE) has the advantage that it can assist the analyst to discover specification inconsistences. For example, consider Figure 6.6. The disadvantage of the "basic step" concept is that it allows transition $t3$ to be executed, even though event $e2$ is generated after $e1$. The analyst might get a wrong impression, i.e., to believe that the specification works correctly; not realizing that the specification is inconsistent with what is intended.

Conlusion: To achieve the best effect as far as specification analysis is concerned, one should use the basic step semantics as well as super-step semantics.

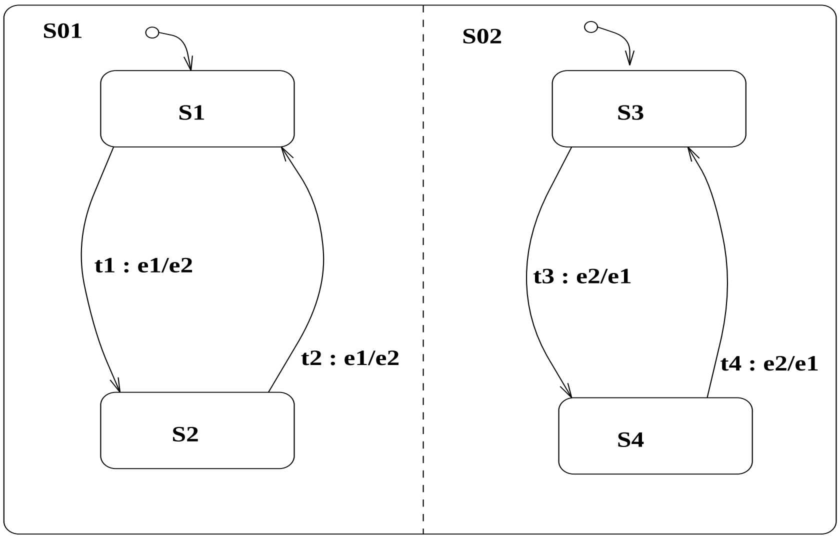


Figure 6.5: An RSML state machine that will not terminate

In the next subsection the fair transition system semantics are used to describe the untimed parts of algorithm *Go-Step* which is based on the synchronous time model. This is followed by the most important Go command, the algorithm *Go-Repeat* that is based on asynchronous time model.

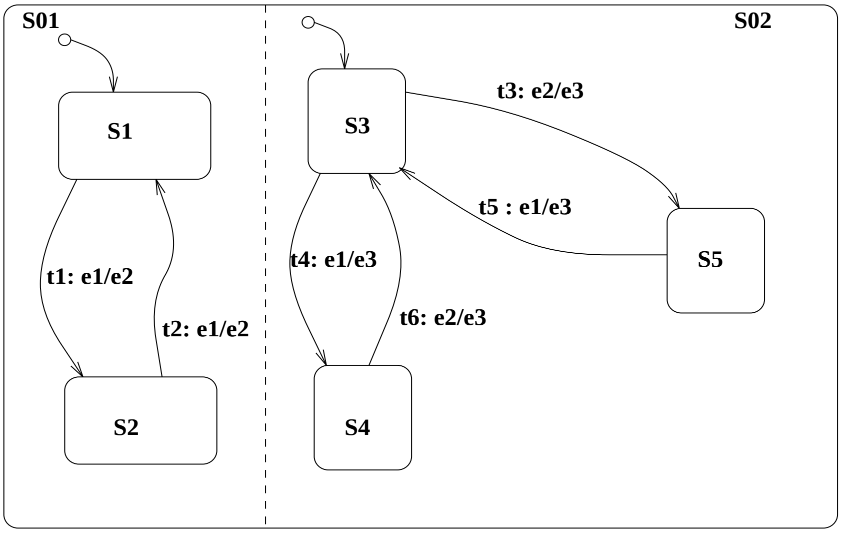


Figure 6.6: A statechart and RSML example

6.2.4 A fair transition system for Go-Step (FTS_{gostep})

In the following, the untimed parts of the algorithm Go-Step are formalised in terms of a *fair transition system*[6] denoted FTS_{gostep}. The above definitions of system status and environment status are used.

FTS_{gostep} consists of the set of variables $V_{sys} = \{\mathcal{C}, SE, SV\}$ and $V_{env} = \{SE_{env}, SV_{env}\}$. Let σ_{Sys} and σ_{Env} be mappings that give the actual system status by $(\sigma_{Sys}(\mathcal{C}), \sigma_{Sys}(SE), \sigma_{Sys}(SV))$ and the actual environment status by $\sigma_{Env}((SE_{env}), \sigma_{Env}(SV_{env}))$, respectively.

The untimed part of algorithm Go-Step in [54] is stated as follows:

(1) execute all external changes reported since the completion of the previous step; (This corresponds to the *Submit* step described by Definition 6.8).

(2) execute one step; (This corresponds to the *System_Step* described by Definition 6.6).

Furthermore, a relation *Put* is introduced to model that the environment has provided input to the system (cf. Definition 6.9).

Definition 6.10 *The* **fair transition system** FTS_{gostep} *consists of the following components:*

(1) The set of variables V*, is partitioned into* $V = V_{sys} \cup V_{env} \cup V_{pos}$*, where*

(a) $V_{pos} = \{\pi\}$ *and*

(b) $\sigma_{pos}\{\pi\} \rightarrow \{1, 2, 3\}$.

Let Σ *be a set of mappings with domain* V *consistent with* σ_{Sys}*,* σ_{Env} *and* σ_{pos}.

[6] For a summary about Fair transition systems, see [129], pages 148-150.

(2) The set of initial states $\Theta \subseteq \Sigma$ *is computed by the predicate* $\Theta(V)$, *such that* $\Theta(V) \equiv \Theta_{Sys}(V_{sys}) \wedge \Theta_{Env}(V_{env}) \wedge \pi = 1$.

(3) $\mathcal{T}$ *is the set of transitions which consists of the relations* $\tau_{12}, \tau_{23}, \tau_{31} \subseteq \Sigma \times \Sigma$. *These relations are determined by the following assertions:*

(a) $\alpha_{12}(V, V') : \pi = 1 \wedge Put(V_{env}, V'_{env}) \wedge V'_{sys} = V_{sys} \wedge \pi' = 2$

(b) $\alpha_{23}(V, V') : \pi = 2 \wedge Merge(V_{sys}, V_{env}, V'_{sys}, V'_{env}) \wedge \pi' = 3$

(c) $\alpha_{31}(V, V') : \pi = 3 \wedge System\_Step(V_{sys}, V'_{sys}) \wedge V'_{env} = V_{env} \wedge \pi' = 1$

Remark 6.2 *The set of transitions is well-defined since for every state* $s \in \Sigma$ *there exists a transition* $\tau \in \mathcal{T}$ *which is enabled.*

Figure 6.7 demonstrates the algorithm extended to the transition system FTS_{gostep} of Definition 6.10.

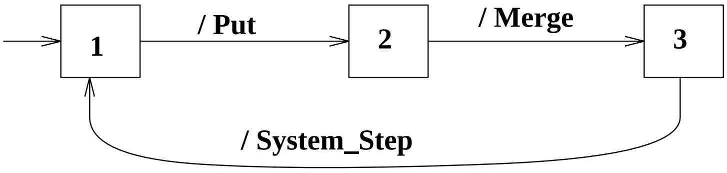


Figure 6.7: Fair transition system FTS_{gostep}

6.2.5 A fair transition system for Go-Repeat ($FTS_{gorepeat}$)

The untimed parts of the algorithm *Go-Repeat* which are formalised in this subsection are as follows:

(1) execute all external changes reported since the completion of the previous step;

(2) repeatedly execute one step until the system is in a stable state, i. e., there are no generated events and no enabled compound transitions or static reactions.

Procedure

Let *stable* $\subseteq System\_Status_{all}$ be the set consisting of all system statuses $(\mathcal{C}, SE, SV) \in System\_Status_{all}$ in which the system is in a stable state.

That is, $SE = \emptyset \;\wedge\; \neg\exists t \in \mathcal{T} : Enable(t, (\mathcal{C}, SE, SV))$ holds.

Let $c \in \mathbb{N}$ be a constant which restricts the allowed number of repetitions of $System\_Step$.

The idea behind this constant is to ensure that a system reaction as modelled by the Go-Repeat terminates. The simulation algorithm will be modelled by $c = \infty$.

Definition 6.11 *The* **fair transition system** $FTS_{gorepeat}$ *consists of the following components:*

(1) The set of variables V, is partitioned into $V = V_{sys} \cup V_{env} \cup V_{pos}$, where $V_{pos} = \{\pi, x\}$ such that

(a) $\sigma_{pos} : \{\pi\} \rightarrow \{1, 2, 3, 4\}$.

(b) $\sigma_{pos}\{x\} \rightarrow I\!N$.

Let Σ be a set of mappings with domain V consistent with σ_{Sys}, σ_{Env} and σ_{pos}.

(2) The set of initial states $\Theta \subseteq \Sigma$ is computed by the predicate $\Theta(V)$, such that

$\Theta(V) \equiv \Theta_{Sys}(V_{sys}) \wedge \Theta_{Env}(V_{env}) \wedge \pi = 1 \wedge x = 0.$

(3) $\mathcal{T}$ is the set of transitions which consists of the relations

$\tau_{12}, \tau_{23}, \tau_{33}, \tau_{31}, \tau_{34}, \tau_{44} \subseteq \Sigma \times \Sigma$. The relations τ_{12} and τ_{23} are defined as in FTS_{gostep} (cf. Definition 6.10). The rest are determined as follows:

(a) $\alpha_{33}(V, V') : \pi = 3 \wedge x \leq c \wedge \neg stable(V_{sys}) \wedge System\_Step(V_{sys}, V'_{sys}) \wedge V'_{env} = V_{env} \wedge x' = x + 1 \wedge \pi' = 3$

(b) $\alpha_{31}(V, V') : \pi = 3 \wedge x \leq c \wedge stable(V_{sys}) \wedge (V'_{sys}, V'_{env}) = (V_{sys}, V_{env}) \wedge x' = 0 \wedge \pi' = 1$

(c) $\alpha_{34}(V, V') : \pi = 3 \wedge x > c \wedge (V'_{sys}, V'_{env}) = (V_{sys}, V_{env}) \wedge x' = x \wedge \pi' = 4$

(d) $\alpha_{44}(V, V') : \pi = 4 \wedge V' = V \wedge \pi' = 4$

Remark 6.3 *The set of transitions is well-defined since for every state $s \in \Sigma$ there exists a transition $\tau \in \mathcal{T}$ which is enabled.*

Figure 6.8 demonstrates the algorithm extended to the fair transition system FTS_{gostep} of Definition 6.11.

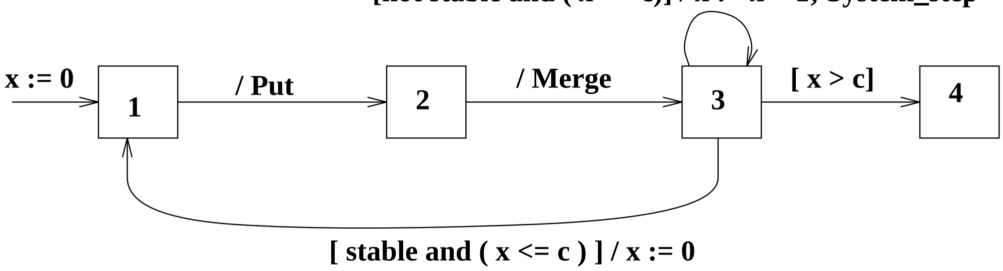

[not stable and (x <= c)] / x := x + 1; System_step
x := 0
1
/ Put
2
/ Merge
3
[x > c]
4
[stable and (x <= c)] / x := 0

Figure 6.8: Fair transition system $FTS_{gorepeat}$

6.3 The step semantics of statecharts based on the model of quantitative time abstraction

So far the algorithms defined above do not consider the aspect of time. In both models, the execution of a step may be viewed as taking zero time as far as the environment is concerned since during the execution of the step itself no external changes have any effect to the system. Such discrete event approach fits systems that are highly synchronous. However, there are certainly many types of systems in which such discrete processing greatly distorts reality, and may lead to unrealiable conclusions. For example, a control program for driving a real-time embedded system (or an industrial robot or a patient monitoring system or an aircraft) must take into account that the environment with which it interacts follows continuous rules of change. Therefore at least an extension to semantics that deal with quantitative time abstraction is required, and will be described here in this Section 6.3.

In particular, this section will be devoted to the formalisation of parts of the basic step algorithm related to timeout events and the internal clock. In addition, the timed parts of the algorithms (cf. [54]) will be formalised in terms of *clocked transition systems*. *Clocked Transition Models* (CTMs) provide an effective representation of realizable systems. Statecharts (and Petri-Nets) can be mapped into CTMs (cf. [79, 78, 99]).

6.3.1 Timeout events

Definition 6.12 *A special event* $\boldsymbol{timeout(e, d)}$ *denoted* $\boldsymbol{tm(e, d)}$ *occurs iff the last occurence of event e is* $d \in I\!N$ *units ago.*

Definition 6.13 *The set of all timeout events defined in the given statechart is denoted by* $\boldsymbol{\mathcal{T}mSet}$. *A* **timeout status** *is a set of tuples* $(tm(e, d), next) \in \mathcal{T}mSet \times I\!N^{\infty}$ *where* $tm(e, d)$ *is a timeout event and next is the time of its next occurence. The relation* $\mathcal{T}mSet \times I\!N^{\infty}$ *is denoted by* $\boldsymbol{\mathcal{T}md_{all}}$. *A special timeout status* $\boldsymbol{\Theta_{\mathcal{T}m}} \subseteq \mathcal{T}md_{all}$ *is called* **initial timeout status** *with* $\Theta_{\mathcal{T}m}(\mathcal{T}m) \equiv \forall tm(e, d) \in \mathcal{T}mSet : (tm(e, d), \infty) \in \mathcal{T}m \wedge |\mathcal{T}m| = |\mathcal{T}mSet|$.

In [54], page 18, the basic step algorithm changes the timeout status depending on the set of available events and the current time, and generates the timeout events the time of which is due:

```
For each pair $(E, next)$ in the timeout status with $E = tm(e, d)$, do the following:
    if $e$ is generated
        then set $next := current\_time + d$;
        else if $next \leq current\_time$
            then generate $E$ and set $next := \infty$.
```

Comment: Logically *equality* (i. e., =) is used for time comparison (and not $\leq$), but in practice, because of floating point inaccuraces, and particularly, because in real life steps do take time, $\leq$ is used here.

Nonetheless, in the algorithm Go-Step the condition $next \leq current\_time$ is replaced by $next \in (current\_time, current\_time + 1)$. In order to be able to capture these differences a parameterized timeout evaluation step will be defined as a function over the set of all time intervals $\mathcal{I}$.

Definition 6.14 *The function* ***TMgen***: $\mathcal{I} \rightarrow ((2^{\mathcal{T}md_{all}} \times 2^{Enames} \times I\!N) \rightarrow (2^{\mathcal{T}md_{all}} \times 2^{Enames}))$, *with* $TMgen(I)((\mathcal{T}md, E, T)) = (\mathcal{T}md', E')$ *defines the* **change of the timeout status** *and the* **generation of timeout events** *with respect to an interval* $I \in \mathcal{I}$ *iff it satisfies the following:*

(i) $\forall (tm(e,d), next) \in \mathcal{T}md$:

(a) $(e \in E \Rightarrow (tm(e,d), T + d) \in \mathcal{T}md' \wedge$

(b) $(e \notin E \wedge next \in I \Rightarrow (tm(e,d), \infty) \in \mathcal{T}md' \wedge tm(e,d) \in E' \wedge$

(c) $(e \notin E \wedge next \notin I \Rightarrow (tm(e,d), next) \in \mathcal{T}md')$,

(ii) $\forall e1 \in E' : (e1 \in E \vee (\exists (tm(e2,d), next) \in \mathcal{T}md : e2 \notin E \wedge next \in I))$.

6.3.2 The clocked transition systems of statecharts

The realization of the timed parts of algorithms Go-Step and Go-Repeat employs an internal clock T with discrete domain (cf. Definition 6.14). For this purpose clocked transition systems will be used to formalise the clocks over $I\!N$.

The notions of clocked transition systems are based on [79, 78]. A system to be formalised/analysed is associated with

(i) $V = D \cup C$: A finite set of variables, where

(a) $D = \{u_1, \ldots, u_n\}$ is the set of discrete variables and

(b) $C = \{c_1, \ldots, c_m\}$ is the set of clocks.
Note: Clocks[7] are of type $I\!N$ whereas discrete variables can be of any type. A special clock $T \in C$ represents a *master clock*.

(ii) Θ : An initial condition characterized by an assertion with $\Theta \Rightarrow T = 0$.

(iii) $\mathcal{T}$: A finite set of transitions.

(iv) θ : A *time progress condition.* $\mathcal{T}_T = \mathcal{T} \cup \{tick\}$ is then the set of transitions obtained by augmenting $\mathcal{T}$ with *tick* which is given by $\lambda_{tick} \equiv \exists \Delta : (\Delta > 0 \wedge \forall t \in [0, \Delta) : \theta(D, C + t) \wedge D' = D \wedge C' = C + \Delta)$, where

(a) $C' = C + \Delta$ abbreviates $c_1' = c_1 + \Delta, \ldots, c_m + \Delta$ and

(b) $\theta(D, C+t)$ abbreviates $\theta(u_1, \ldots, u_n, c_1 + t, \ldots, c_m + t)$. Note that this requirement ensures that each discrete variable's change is precisely timed. In other words, discrete variable and time do not change at the same time.

[7] In [79] clocks have always the type real. In this thesis, however, the clocks have the type $I\!N$ in order to simplify the development of the analysis methods. This however, has a drawback when modelling.

6.3.2.1 A clocked transition system for Go-Step (CTS_{gostep})

Now, the timed parts of the algorithm Go-Step are formalised in terms of a clocked transition system denoted CTS_{gostep}.

The algorithm Go-Step is stated as follows:

(i) execute all external changes reported since the completion of the previous step; (This is formalised as in the untimed case (cf. Definition 6.8)).

(ii) increment the clock by one time-unit;

(iii) execute all timeout events whose time falls within $(T, T+1]$; (This is formalised using the definitions in Subsection 6.3.1 above).

(iv) execute one step; (This corresponds to the $System\_Step$ of Definition 6.6).

The complete formalisation of the algorithm Go-Step is given by the clocked transition system CTS_{gostep} which is shown in Figure 6.9. Note that the clock T is able to progress simultaneously with timer t.

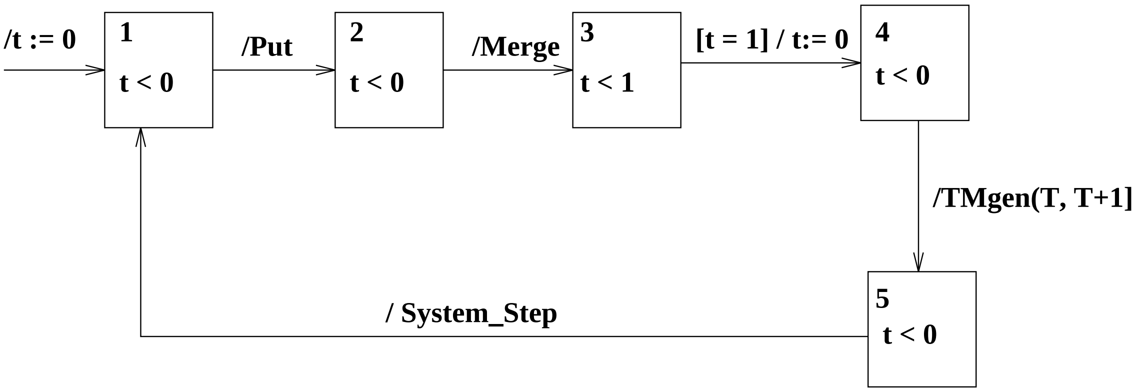


Figure 6.9: Clocked transition system CTS_{gostep}

Definition 6.15 *The* **clocked transition system** CTS_{gostep} *consists of the following components:*

(i) The set of variables V*, is partitioned into* $V = V_{sys} \cup V_{env} \cup \{\mathcal{T}m\}, \{T, t\}$*, where*

(a) V_{sys} *and* V_{env} *are defined as in* FTS_{gostep} *(cf. Definition 6.10),*

(b) $\mathcal{T}m$ *expresses a list of timeout events and their time occurrence,*

(c) T *is the current time of the internal clock, and*

(d) t *is the timer.*
The domain of T *and* t *is* $I\!N$*.*

(ii) The set of initial states is computed by the predicate $\Theta(V)$*, such that*
$\Theta(V) \equiv \Theta_{Sys}(V_{sys}) \wedge \Theta_{Env}(V_{env}) \wedge \Theta_{\mathcal{T}m}(\mathcal{T}m) \wedge T = t = 0.$

(iii) $\mathcal{T} = \tau_{12}, \tau_{23}, \tau_{34}, \tau_{45}, \tau_{51}$*: is the set of transitions, where time does not progress. The relations* τ_{12} *and* τ_{45} *e. g., are defined by the following assertions:*

(a) $\alpha_{12}(V, V') : \pi = 1 \wedge Put(V_{env}, V'_{env}) \wedge V'_{sys} = V_{sys} \wedge \pi' = 2 \wedge \mathcal{T}m = \mathcal{T}m' \wedge (T', t) = (T, t)$

(b) $\alpha_{45}(V, V') : \pi = 4 \wedge TMgen((T, T+1])((\mathcal{T}m, V_{sys}, T), (\mathcal{T}m', V_{sys})) \wedge V'_{env} = V_{env} \wedge \pi' = 5 \wedge (T', t) = (T, t)$

(c) θ *: A time progress condition which describes the global restrictions on the progress of time. Here* $\theta(t) : (\pi \in \{1, 2, 4, 5\} \rightarrow t < 0) \wedge (\pi = 3 \rightarrow t < 1)$*. In this case, progress of time is captured by a transition* τ_{tick} *given by*

$\alpha_{tick} : \exists \Delta \in I\!N_{>0} \forall d \in [0 \ldots \Delta) : \theta(t + d) \wedge (T', t') = (T + \Delta, t + \Delta) \wedge (V'_{sys}, V'_{env}, \mathcal{T}m') = (V_{sys}, V_{env}, \mathcal{T}m)$

which equivalent to

$\pi = 3 \wedge t = 0 \wedge (T', t') = (T' + 1, t' + 1) \wedge (V'_{sys}, V'_{env}, \mathcal{T}m') = (V_{sys}, V_{env}, \mathcal{T}m)$.

6.3.2.2 Clocked transition system of Go-Repeat ($CTS_{gorepeat}$)

Finaly, the timed parts of the algorithm Go-Repeat are similary formalised in terms of a clocked transition system denoted $CTS_{gorepeat}$.

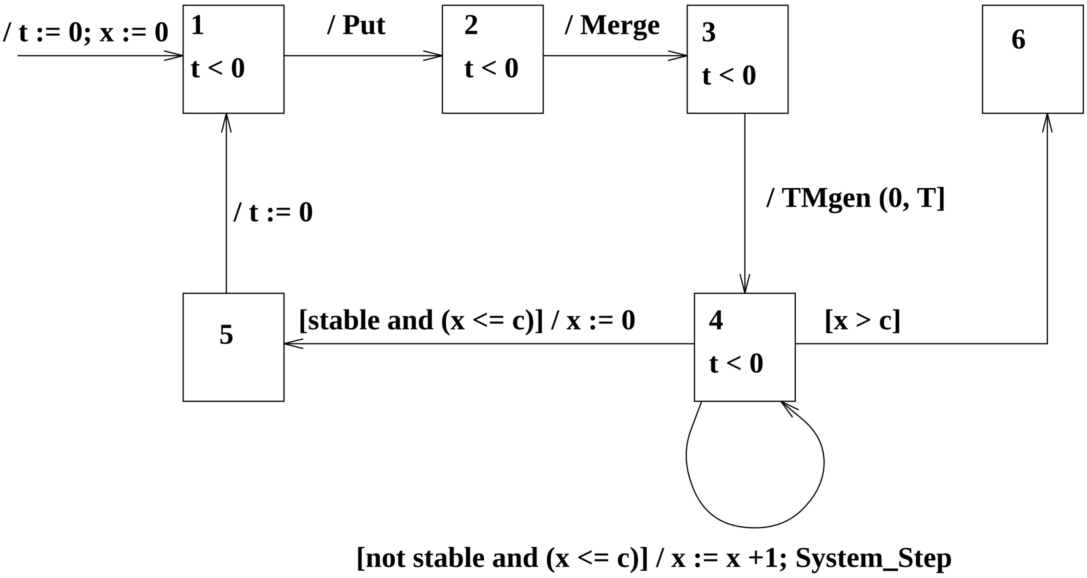


Figure 6.10: Clocked transition system $CTS_{gorepeat}$

The algorithm Go-Repeat is stated as follows:

(i) execute all external changes reported since the completion of the previous step;

(ii) execute all timeout events whose time is due (up to and including the current time);

(iii) repeatedly execute one step till the system is in a stable state.

Note that there is no increment of the internal time clock in these steps. The steps (i) and (iii) are formalised as in the untimed case (cf. Definition 6.8) or $System\_Step$ in Definition 6.6, respectively. Step (ii) is formalised using the definitions in Subsection 6.3.1.

The resulting formalisation of the algorithm Go-Step is given by the clocked transition system $CTS_{gorepeat}$ which is shown in Figure 6.10.

6.4 Statecharts in object-oriented designs

In Section 4.1, a brief introduction to the STATEMATE tool was given. The language set which was built around statecharts is based on the function-oriented structured analysis paradigm (SA) (cf. [159]). SAs, however, are widely criticised as suffering from a discontinuity problem when it comes to certain aspects of system development, e. g., transition to design and re-use, many software engineers/developers recommend the object-oriented (OO) paradigm. In fact, this convention is one of the most notable development in recent years.

Most object-oriented modelling methodologies use graphical notations for specifying the model (cf. [151, 161]). Typically, entity relationship (ER) diagrams are used in specifying classes of objects and their entity-relationships, and means for describing the interface and capabilties of the objects themselves. In addition, extended state machines are often adopted to specify class behaviour. Nevertheless, most of these methodologies do not address dynamic semantics adequately, so that the real behaviour of models over time is in many cases not well-defined.

Therefore, recently a lot of effort is being made to use statecharts in object-oriented designs to overcome such crucial shortcoming (see [27] and [53, 29]). In [27], a notation called "Objectcharts" for specifying the behaviour of object classes as state machines is presented. Objectcharts are a combination of object-oriented analysis and design techniques and statecharts to give a digrammatic specification technique for objects. Objectcharts offer a formal and declarative extension to the state-machines, and therefore are more appropriate for analysis and design.

In addition, a Computer-Aided Software Engineering (CASE) tool *RHAPSODY*[8] based on an object-oriented framework has been constructed (cf. [53]). It constitutes the subset of Unified modelling Language (UML[9]) (cf. [135]) and enables model execution and full code synthesis in object-oriented languages such as C++ (see [44, 96, 149]). The main difference between the way statecharts are used in a function-oriented framework e.g. STATEMATE and in the object-oriented framework proposed in [53] is in the role of transitions. In the latter, events are treated in a 'run-to-completion' style, i. e., in which the transitions can be compound and multiple. In other words, it is required that all parts of a transition are fully executed before the statechart becomes stable and the system can respond to new event. This leads to non-zero-time semantics approach when compared to STATEMATE's approach (cf. Section 5.4).

[8] trademark of i-logix, Inc.

[9] The UML is a third-generation state-of-art object modelling language that defines a set of notations, and more importantly, defines the semantics of those language elements (see also [29]).

6.5 Experiences with statecharts/STATEMATE

The main purpose of this section is to discuss the suitability of statecharts/STATEMATE for the development of large and complex reactive systems, based on the personal experience of the author. The first experiences with the statecharts/STATEMATE go as far back as the phase of my *diploma thesis* (cf. [84]). During the research for the *diploma thesis*, several examples were considered including the following:

- a digital watch,
- a television set,
- a video set,
- airport activities,
- traffic lights,
- different electronic systems e.g.,
  - automatic braking system of a car,
  - automatic window functions of a car

  and particularly,
- the telefax machine *BEAUTY*.

6.5.1 Positive experiences with statecharts/STATEMATE

The existence of the supporting tool STATEMATE was a vital criterion for specification and analysis of different statecharts models. In this thesis, the telefax statechart models of [84] are modified. In the following, some of the greatest experiences of the author with the statecharts are given:

(1) Statecharts allowed the designer to specify even complex models, such as dynamical behaviour of the *telefax machine BEAUTY* which consists of three devices in one cabinet:

   (a) Fax device
   (b) Telephone
   (c) Answering machine.

   Comment: The statechart model of the telefax machine *BEAUTY* has relatively large number of states and transitions (cf. [84] chapter 5). Therefore, it can be considered as a large and complex system. The author of this monograph does not doubt that statecharts as implemented in STATEMATE/RHAPSODY can be used to design behaviour of large and complex reactive systems.

(2) Dynamic execution of statecharts in a simulation environment offers the analyst with means for early validation of specification. The simulation capability of the tool allows one to test complex logical statements to ensure whether they produce the desired effect.

Comment: The procedure used to simulate complex logical statements was often of incremental type. In other words, by building them up one by one, testing them in simulation as they developed.

(3) Statecharts modelling facilitates clear communication between designers (researchers) and companies.

Comment: Discussions with various outstanding companies (in particular in the automobile industry) and software houses, including institutes for software quality and reliability, and application of software methods have emphasized the importance of clear communication. Many of them recommend the STATEMATE tool as being a requirement validation tool not only valuable in facilitating error discovery but because of its ultimate help to ensure that the system being built will meet the end users expectations (see also [113]).

6.5.2 Negative experiences with statecharts/STATEMATE

One of the main disadvantages of STATEMATE/RHAPSODY is, as in several CASE-tools, the problem of not enabling 'systematic' test methods. In this context, systematic refers to providing concepts or at least heuristics which assist the tester, e. g., by the generation of test cases.

Statecharts enable viewing the description at different of levels detail[10], and make even very large and complex specifications manageable and comprehensible. However, the *test problem/verification problem* remains difficult:

(1) because of the *interaction* between the components, and

(2) because testing a *complete* system, in many cases, leads to "space explosion" problem.

An alternative is to divide the system test into two phases:

(i) Perform detailed tests for individual components (*module test*).

(ii) Then test the proper integration of an increasing number of components, adding one by one until all components are integrated. This, nevertheless, leads to *integration test* problems.

Comment: Discussions with several outstanding companies and software houses show that there is still a great problem to be solved in finding and applying software methods to overcome/eliminate space explosion problems. In particular, there are hardly any practical methods for *integration* tests.

[10] because hierarchical statecharts allow decomposition of components/states into subcomponents/substates

The main goal of the next Chapters 7 and 8, therefore, is to develop methods and concepts which can aid the system designer/tester to detect and/or eliminate reactive system flaws. Above all, the developed methods and concepts should reduce the space explosion and integration problems outlined above.

Chapter 7

Module Test Approach

Often testing is undertaken when the whole system has been coded. The main disadvantage of such a preceeding, however, is that the most expensive errors to correct are often introduced early in the development. Therefore methods and concepts which allow the system designer/tester to (automatically) test the partially designed and implemented system are of theoretical as well as practical interest. In this chapter as well as in Chapter 8 we address these problems and establish practical methods which can aid the system designer/tester to detect and/or eliminate reactive system flaws. Specifically, in this chapter we provide a *module test approach* which can be applied to reduce the space explosion problems that has been discussed at the end of the previous chapter (cf. Section 6.5.2). This is achieved by generating test cases on different system levels of *hierarchical*, structured system models. Hence the methods developed here can be applied to large and complex models. These methods are basically established on the concepts developed in Chapters 5 and 6.

This chapter is organised as follows. In Section 7.1 we present an overview of *verification* methods and *validation* methods adopted for ensuring the correct operation of reactive systems. After that, we give in Section 7.2 the motivation as well as the aim of the *module test approach*. In Section 7.3 we give the family of test criteria required by the test strategies. In order to give an idea of the various test selection techniques we use Section 7.4 to present an overview of the components of test selection strategies. In Section 7.5 we describe the test principles which can be used in the validation of the statechart model. To place the methods and concepts which are developed in this thesis in an appropriate context, we review in Section 7.6 the related work. In particular, testing based on finite state machines (FSMs) is described and several criticisms of FSMs analysis/testing from literature are discussed. After the review of existing test selection methods, we notice that the test effort of the most well-known strategies is very large (e. g., an automata equivalence test (cf. criterion $C3$, Definition 7.4 requires at least a cubic number of tests (in the number of states or events)). Therefore, we devote Section 7.7 to the establishment of module concepts which operate on the *hierarchical* statecharts. These can be applied to reduce the space explosion problem by generating test cases on different system levels (cf. [88]). In Section 7.8 we present methods or heuristics to generate test paths that satisfy the given criterion (cf. Criterion 7.1). Finally, in Section 7.9

we formalise the algorithms which are described in Subsection 7.8.2.

7.1 Background on verification/validation of reactive models

The purpose of this section is to present an overview of the *verification* methods and *validation* methods in order to place *testing* which is the central part of this thesis in an appropriate context. The difficulties of ensuring the correct operation of concurrent and reactive systems have led to the development of several verification and validation methods. In the community of the software engineering, the terms verification and validation have been used inconsistently. Therefore, before tackling the details of these methods, an overview of the different use of the terms verification and validation is given.

How are the terms verification and validation applied in the engineering community?

The usually adopted academic convention of the term "verification" means "giving a complete proof" while "validation" stands for "giving evidence". In the practical sense, the goal of verification is to ensure that the artifact[1] satisfies properties of interest whereas the goal of validation is to ensure that the artifact satisfies the user's intention. In this sense, proving a property about a specification is a form of verification whereas simulation is a form of validation. In general, however, verification does not only apply to implementations which must be checked against the specifications in order to verify the correctness of the implementation but also to specifications themselves: the specifications e. g., must be consistent and satisfy critical application properties.

This section is divided into three subsections. In the first Subsection 7.1.1 we describe the different forms of verification. We then introduce in Subsection 7.1.2 the problem of testing and give some of the different concerns related to testing. In Section 7.1.3 we specifically describe tests performed by the *Check Model* tool of STATEMATE.

7.1.1 Verification methods

In this subsection we describe the different forms of verification. Verification can be performed with different levels of confidence and in different ways. Many of the verification methods consist of formulating and proving theorems about a formal model of a system. Once the system and its properties have been stated in a suitable mathematical notation, the idea is to check that the system satisfies the properties, and this may be proved as a mathematical theorem called *formal* verification. Through the verification process the confidence in the correctness[2] of the system is increased.

[1] that is, an implementation or a specification

[2] In the literature, the special correctness properties of reactive or concurrent systems are often classified into two broad classes: *safety properties* and *liveness properties* (cf. [129]).

Often, verification, mainly performed through testing is associated with informal design methods, and correctness proofs[3] with formal methods. In [61] Heitmeyer and Mandrioli state that the situation in practice is far more complex. They argue that during verification as well as during requirements specification and design, one may adopt several strategies for checking correctness, each with its pros and cons. Depending on many factors, including the characteristics of the application and the designers experience and taste, the following stategies have been suggested:

- Informal reasoning can be applied even to a formal specification design.
  For example, traditional *walkthroughs* and *inspections* can be applied to any informal design notion as well as to code written in some high level language.

- Formal specifications may be used to derive sample executions of the system. The tester can experiment with these system models (see also Subsection 3.2.2) to determine whether the specifications capture the intended meaning. This is called *simulation* [55] or *specification testing* [77].

- Formal methods can increase the reliability of more traditional techniques, such as testing. For example, test cases can be derived from formal specifications either automatically or semiautomatically (cf. [84]).

- The most popular formal verification technique is mathematical deduction: basic system features are stated as axioms and related properties are then proven as theorems [28].

7.1.2 Background on path testing

This subsection introduces the problem of testing and gives some of the different concerns related to testing. It focuses on the central concern of this thesis, the test selection problem. In particular, we use the validation and verification terms discussed above to place *testing* in an appropriate context and hence provide background information for the test methods and concepts developed in this thesis.

Testing is widely used in software validation and verification. This process consists of

(i) selecting a set of certain features of the specification[4], and

(ii) finding appropriate data that exercise paths in the control flow graph of the implementation which verify or falsify the validity of the features.

During the implementation phase one may choose e. g., a test set in which every program statement is executed at least once, or a broader test set that would exercise all exits from the branch statements. Now when a program is represented as a digraph, these strategies

[3] sometimes called *formal verification*

[4] Depending on the test phase, specification can be seen as an informal specification, formal specification (model) up to implementation (program).

correspond to the problems of finding sets of source to the sink paths, called ***s*-*t* paths** that cover the vertices or edges of the digraph (cf. [116]).

The aim of testing is to detect errors in the program or specification. However, to certify a program, one must test all execution sequences in the program. As mentioned earlier (cf. Section 1.2.5), even for small programs (models) the number of execution sequences can be extremely large. This makes exhaustive testing an impossible task. Therefore a compromise is to test a subset of execution sequences which is manageable but at the same time holding the ability to cover interesting or potentially troublesome interactions among the code segments.

Assuming that the program (specification) or model is represented as a digraph, the problem is to find minimum path covers for the vertices (or the edges) of the digraph. Although this minimum requirement is important, it is far from *effective*. An idea for more effective requirement can be formulated as follows:

An *effective criterion* requires paths (test paths) with a high probablity of revealing faults, i. e., when the model is simulated with test data that cause the selected test paths to be executed, there is a 'high probability' that faults, if they exist, are revealed by those test runs.

Generally, the effectiveness of such criterion depends not only on the selected paths but also on the test data for those test paths. Therefore for an *intensive test*, a more thorough test criterion is required (cf. Definition 7.4 on page 121).

Since the beginning of the early 1970's path coverage criteria have widely been used to measure the thoroughness of test cases (cf. [67, 92, 114, 138, 160]). In [24], efforts have been made to compare these criteria. One of the major criticisms of all these criteria is that they are solely based on syntactic information and do not consider semantic issues (cf. Section 5.4 and Chapter 6), e. g., infeasible paths.

Yet, another point which has to be taken into account is that while applying a path selection criterion a module (program or subprogram) is represented as directed graph (control flow graph) with two unique nodes, a start node usually termed as the *source* and final node called *sink* of the program. In the case of *reactive systems* (*models*), however, it is clear that this is not usually the case, in particular, there is no unique final node. Therefore other test methods are required to deal with such systems (see Section 7.2).

7.1.3 Tests performed using the Check Model tool

In this subsection we describe tests performed by the *Check Model* tool of STATEMATE. The Check Model tool allows the system designer/tester to examine his/her "System Under Development" (SUD) specification for consistency and completeness. Generally, the *Check Model* performs two type of tests:

(1) *Correctness* — Checks for inconsistencies in the model

(2) *Completeness* — Checks for superfluous and incompleteness in the model.

7.1.3.1 Correctness examples

In this subsubsection we give some examples of correctness checks. The Check Model tool of STATEMATE detects violations in the statechart model, e. g.,

(a) *Loops in compound transitions*:
loops in element definition, e. g., compound event $SE1$ is defined as '$SE2$ or $SE3$' $SE2$ is defined as '$SE1$ or $SE4$'.

(b) *Data-items with unknown types*:
Data-items with unknown types that are used inconsistently e. g., Data-item D is used both as a string and an integer, yet it is undefined in a form.

(c) *Elements using uninitialized context variables*:
e. g., a transition with the following label: $SE0/SA1 := \$X$; $\$X := Z$. $\$X$ is used before any value is assigned to it.

7.1.3.2 Completeness examples

In this subsubsection we give some examples of completeness checks. The completeness checks detect missing or superfluous information in the statechart model, e. g.,

(a) A transition without a trigger is incomplete.

(b) An event defined in the data dictionary, but not used anywhere in the specification, is considered superfluous.

7.1.3.3 Test problems

In Subsection 6.1 we gave a brief description of the static tests provided by the STATEMATE Test tool. Amongst them, we described *reachability test, detection of nondeterminism* and *deadlock* as well as *usage of transitions*. All these and the above tests should carry out (or check) exhaustive sets of execution sequences. However, the number of possible sequences is often not manageable. In order to get an idea about some of the largest problems when testing a statechart, one may consider the following example of a behavioural model:

- that contains 40 concurrent components, where
- each component has about 10 states.

In this case there are so many state configurations that the number of possible scenarios is not manageable. (in worst case there are 10^{40} states). This example illustrates that even for small statecharts, the number of execution sequences can be extremely large.

In the next Section 7.2, therefore, we establish a *module test approach* which will be applied to reduce the space explosion problem by generating test cases on different system levels.

7.2 Motivation and aim of the module test approach

Almost all the methods currently used for testing large and complex reactive systems are experience based rather than theoretically founded methods. In particular, very few methods allow us to make an objective statement about the problem of generating sufficient test paths, e. g., for all used transitions and/or all used states in a statechart. We recall that STATEMATE offers a number of static tests, such as reachability test, detection of nondeterminism, deadlock, and usage of transitions. All of these tests are useful for checking general properties of the system.

Nevertheless, testing is usually carried out using the common, rather inefficient test methods. The specification of test cases based on the requirements specification is carried out on a largely intuitive and unsystematic basis. In other words, the system tester/designer has no information or even hints from the tool how to make a systematic test. Tool support is available for routine activities such as test execution, monitoring and test documentation (e. g., [46, 68]).

In fact, the CASE tools are not in a position to carry out sufficient tests for system models. Tests are offered under particular scenarios. This is often not enough for detection of all system flaws. The statecharts' analysis capability (e. g., offered by STATEMATE) is confined to using reachability analysis to check a small set of properties which is quite limited.

It would be better if the analysis tool were in a position to simulate all possible combinations of events, conditions, and values of variables in order to make a global test e. g. for nondeterminism. Unfortunately, this leads even for small statecharts models to an unimaginable number of variations so that a practical application is not possible[5].

Indeed, as already observed in Subsection 6.5.2 there is still a great problem to be solved in finding and applying software methods to overcome/eliminate space explosion problems. In addition, an *exhaustive testing*, i.e., testing every execution to prove the system's correctness, often cannot be performed even for models of moderate complexity.

It is important to keep in mind that even if we limit the values of variables to finite sets, the number of scenarios that have to be tested in an exhaustive execution rapidly becomes unmanageable (see example given in Subsubsection 7.1.3.3).

The *module test approach* given in this chapter describes methods and concepts which allow the system designer/tester to (automatically) test the partially designed and implemented system (cf. Sections 7.7 and 7.8).

The **aim** of the *module test approach* is mainly to establish an acceptable small set of test paths that reaches all used states and carries out all used transitions in a component. The idea behind this is to apply as few tests as possible, but of course, providing the ability to test all *crucial* dynamic properties in the model. The term "crucial" here depends on the abstraction degree of observation and on the required test cases. This involves determining *necessary test criteria* which are defined in Section 7.3.

By applying this test method, we thus illustrate that the module test concepts can

[5] The number rises exponentially with the number of the input and output state variables.

provide a more convincing approach to the problem testing on a systematic basis.

7.3 Definitions of the test criteria

In this section we give the family of test criteria required by the test strategies.

Definition 7.1 *The* **C0 test criterion** *requires a subset of test sequences such that every state in a statechart model is executed at least once.*

Definition 7.2 *The* **C1 test criterion** *requires a subset of test sequences such that every transition in a statechart model is executed at least once.*

Test criterion $C1$ is necessary since every transition can be faulty. On the other hand, the interaction of transitions in a sequence can produce new faults. Hence the following criteria are necessary too.

Definition 7.3 *The* **C2 test criterion** *considers that all possible transition sequences of length two are executed at least once.*

As stated in Section 7.1.2 on page 118 the effectiveness of a criterion depends not only on the selected paths but also on the test data for those test paths. Therefore for an *intensive test*, a more thorough test criterion is required. This is given by the following definition.

Definition 7.4 *The* **C3 test criterion** *requires automata equivalence test (cf. [22]).*

Note however, criterion $C3$ is only possible if certain assumptions are made, e. g., when the statechart model to be tested is assumed to have a 'limited' number of states.

7.4 Components of test selection strategies

Software reliability presents a remarkable problem for the development of large computer systems. Various methods have been developed to increase the software reliability. In the area of software testing, many different test selection techniques have been proposed. Each test selection technique has its own advantages and disadvantages. Therefore, none of these techniques should be applied in isolation. Rather, an effective testing methodolgy should use a combination of the techniques. In order to give an idea of these test selection techniques, we use this Section 7.4 to present an overview of the components of test selection strategies. The test selection problem is concerned with finding techniques that are inexpensive for testing, but in any case assure a certain grade of correctness. We divide this subsection into three parts. In Subsection 7.4.1 we compare *manual test selection techniques* with *automatic test selection techniques*. Since the *module test approach* developed in this thesis is based on an *automatic test selection technique*, we describe the components of the automatic test selection used in this thesis in Subsection 7.4.2. Lastly, in Subsection 7.4.3 we give an overview of some of the most well known test selection strategies proposed in the literature.

7.4.1 Manual test selection versus automatic test selection

The usual approach in selecting tests is to define test purposes and construct tests to satisfy each test purpose. Such test purposes are derived manually, perhaps from the statements of the functionality of the sytem (e. g., from user's manuals), or data definitions in data dictionaries, etc.

One fundamental benefit of test purposes is that it allows customers as well as developers to understand what is being tested, and allows even the use of unstructured information, such as informal descriptions, perhaps unstructured knowledge, e. g., common programming mistakes.

On the contrary, automatic test selection strategies define which kind of tests are required, and in practice these are based on formal models of the program or design and need no human effort. The idea behind the automatic test selection technique is that when the model of the program/design has been created and the necessary test criteria are chosen, then the necessary tests are given automatically.

The effectiveness of automatic strategies lies in the fact that the knowledge of test purposes can be codified in the form of test criteria. Consequently, automatic strategies are easier to apply, and above all, do not depend as much on the system user. Therefore, they are easier to compare and evaluate, and thus provide a better basis for improvement.

7.4.2 Components of automatic test selection

In this subsubsection we describe the components of the automatic test selection used in this thesis. Figure 7.1 shows two main components which are necessary for an automatic test selection technique: the *Statechart Model* which comprises the knowledge about the behaviour of the reactive system, and the *Test Criteria* which comprises the knowledge about how reactive system flaws occur in relation to the statechart model. These are useful for the prediction of how the statechart will behave (on e. g., inputs not in the test set); without them, it can be very difficult to analyse the results of test cases in order to assess the correctness of the whole model.

Using the statechart model and test criteria we are able to define a *test selection* stategy. This stategy selects a test to detect each reactive system flaw that is possible under the test criteria. Apart from selecting tests, the technique should as well provide some means for *assessing the correctness* of the statechart after executing the tests.

7.4.3 Common test selection strategies

In this subsection we give an overview of some of the most well known test selection strategies proposed in the literature. They can be considered 'standard' due to the fact that they have been used for a long time, included in many software textbooks, e. g.,

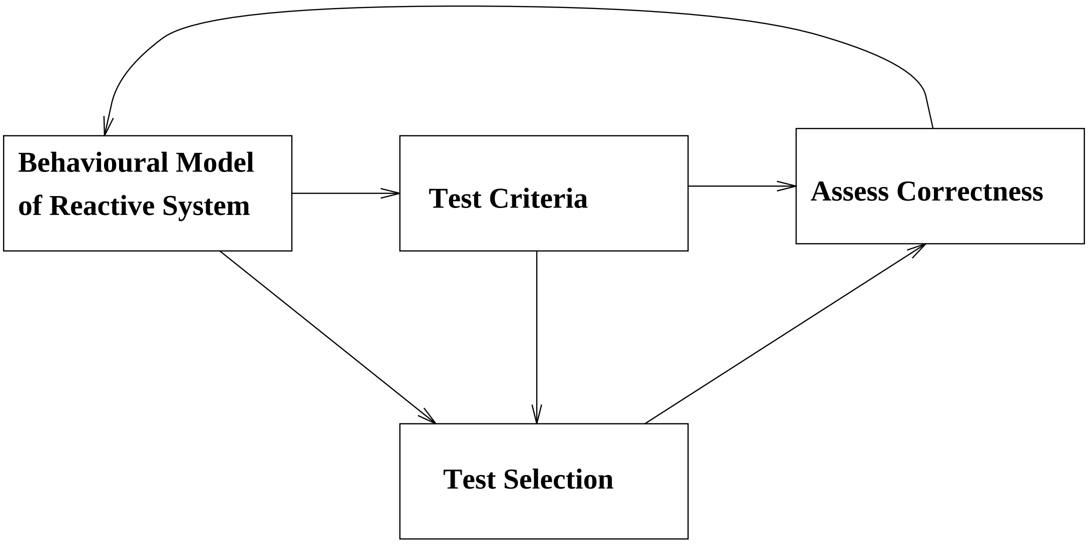


Figure 7.1: Components of test selection Legend: An arc between component A and B describes that A embodies some knowledge which is necessary for B.

[5, 132], and above all form the basis for almost all testing techniques. The three most important standard techniques are *path testing*, *state-based testing* and *logic testing*.

Path testing: The most important and widely-used techniques are based on path testing. This kind of testing uses the flow-graph model which represents a digraph that captures the flow of the processing of the program (model). We refer the reader to Subsection 7.1.2 for more details. Coverage criteria for path testing aim to cover all nodes, all edges and all paths[6] (compare criteria given by Definition 7.1, Definition 7.2 and Definition 7.3 in Section 7.3).

State-based testing: State-based testing uses finite state machines (FSMs) and various extensions. Though FSMs are typically found in the specification of behaviour of systems or objects, they can also be derived from the code with difficulty. FSM models can also be tested using path coverage criteria, but there are usually more special-purpose testing techniques which are specifically developed for them. The best known state-based test selection methods are called *Transition tour* [110], *W-method* [22], *Distinguishing Sequence Method* (*DS*) [43], and *Unique-Input-Output* (*UIO*) [148].

Predicate testing: Predicate testing requires certain types of tests for the properties that are formulated as predicates in the specification or implementation of a program (statechart). For example, a decision table relates the input predicates to the outputs. The exhaustive testing criteria requires all the combinations of values for all predicates.

[6] Generally, when the graph contains loops the criterion for all-paths is not satisfiable because it demands an infinite set of paths.

7.5 The test principles for validating the statechart model

Validation of the model using tests requires *test cases.* A test case generator will be responsible for the creation of test cases from the formal specifications (statechart models). The main purpose of the generator is to produce a set of *suitable* test cases for the given model. Generally, the test cases can be developed on the following test principles:

(1) *Test principle A* — the statechart model is validated against its specification;

(2) *Test principle B* — the statechart model is verified against its implementation.

Testing a statechart model against the specification:

The specification against which the statechart model is to be tested may be stated informally in the form of prose or formally as set of axioms (of the *requirement statements* of the customer or developer (cf. Section 3.2)). In this thesis, the test cases will be considered *suitable*[7] for the *test principle A*, if for each error in the statechart model, violating, e. g., the informally specified requirements, a test case will be created which is capable of detecting this error. In this particular case, it will be assumed that the informally specified requirements are correct. This case involves the application of the test criterion $C0$, criterion $C1$ and criterion $C2$ (cf. Definition 7.1, Definition 7.2 and Definition 7.3 in Section 7.3).

Testing statechart model against the implementation:

When testing a statechart model Sct, it is assumed that the implementation behaves like some, unknown, statechart model Sct'. In this particular case, the test should attempt to determine whether Sct and Sct' are equivalent. This case is actually the *automata equivalent test* (cf. criterion $C3$, Definition 7.4)

7.6 Related work

In this section related work is reviewed to place the concepts which are developed in this thesis in an appropriate context. In particular, testing based on finite state machines (FSMs) is described and several criticisms of FSMs analysis/testing from literature are discussed.

The problem of testing software has been approached in a variety of ways. Most test selection techniques assume the existence of a specification S as basis for the development of an implementation (formal model) I. The problem is then to know whether or not I is a correct implementation of S. The procedure involves prescribing a way to generate a set of tests from the specification S which is used to test the implementation I.

[7] Note that in this sense, *suitable* describes the "ideal" case (cf. [143], chapter 2).

The **basic idea** of this proceeding is that one would like to construct a test set D such that the output produced by I on D shows that I correctly implements S.

Actually, the **aim** of all these techniques is to find faults, but unfortunately, in most cases, there is no guarantee that the system is *fault-free* after testing has been completed.

This rather justifies to a certain extent Dijkstra's old claim that "all a test can tell us is that a system has failed — it cannot tell us that a system is correct".

In order to get out of this dilemma, we initially give the following definition.

Remark 7.1 The claim "the system is **fault-free** after testing has been completed" can only be made if and only if the (finite) test set generated by the method is proven to reveal **all** the faults of the implementation.

Assumption 7.1 *Let S and I be the specification and implementation, respectively.*

(i) If S and I are (partial) functions $S,\ I:\ D \to R$, where

  (a) D is the **input domain***, and*

  (b) R the **result domain***;*

(ii) and $X \subseteq D$ is the (finite) test set,

then the **implication**

$$[\forall x\ (x \in X \implies S(x) = I(x))] \implies [\forall x\ (x \in D \implies S(x) = I(x))]$$

must be shown correct.

To approach this problem, both the specification and the implementation can be represented as some computational models or machines. A testing method would then try to ascertain if these two machines compute the same function. However, if S and I are assumed to be arbitrary Turing machines, it is well known that this problem is unsolvable, i. e., no algorithm exists that can determine whether the two arbitrary machines compute the same function.

One way to solve this problem is to consider more restrictive models, such as finite state machines (FSMs). Therefore, in Subsection 7.6.2 testing based on FSMs is described and several criticisms of FSMs analysis/testing from literature are discussed. In addition, it will be indicated how these criticisms are addressed in this thesis. But before that, in the next Subsection 7.6.1 we give some fundamental definitions used in the research area of test selection methods based on FSMs.

7.6.1 Preliminaries for methods based on FSMs

In this subsection we introduce some notations and concepts related to FSMs that are used to allow a formal comparison and discussion of various test selection methods, e. g., *Transition tour* [110], *W-method* [22], *Distinguishing Sequence Method* (*DS*) [43], and *Unique-Input-Output* (*UIO*) [148].

Definition 7.5
Given a nonempty finite set Σ, *called* **alphabet**. $x \in \Sigma$ *is called a* **symbol** (*or* **letter**).

(1) Then $\Sigma^\star$ *is defined as follows:*

$$\Sigma^\star =_{def} \{w = x_1x_2 \ldots x_n \mid x_i \in \Sigma, where\ i = 1 \ldots n \wedge n \in I\!N_0\}.$$

(a) $w \in \Sigma^\star$ *is called a* **string** (*also called* **word**).

(b) A string $w = x_1x_2 \ldots x_n$ *is called an* **empty string** *if* $n = 0$. *An empty string is denoted by* ϵ.

Let U *and* V *be sets of* **strings** *over* Σ.

(2) The **concatenation** *of* U *and* V, *denoted* $U \bullet V$, *is the set* $\{w \mid u \in U \ \wedge v \in V \ \wedge \ w = uv\}$.

(3) The **union** *of* U *and* V, *denoted* $U \cup V$ *is the set* $\{w \mid u \in U \ \vee \ v \in V\}$.

(4) The **length** *of a string is given by* $|w|$. *Let* $\Sigma^0 =_{def} \{\epsilon\}$. *For* $i \geq 1$ Σ^i *is defined as* $\{w \mid w \in \Sigma^\star \ \wedge \ |w| = i\}$.
Note: Σ *is a set of strings, each having length* 1, *the set* Σ^i *is the set of all strings over* Σ *having length* i, *and* $\Sigma^\star$ *is the set of all strings over* Σ. *Hence, the following concatenation notation:*

(a) Σ^i *is the set* $\Sigma \bullet \Sigma^{i-1}$, *and*

(b) $\Sigma^\star$ *is the set* $\bigcup_{i=0}^{\infty} \Sigma^i$.

Definition 7.6
A **finite state machine** (**FSM**) *is a 6-tuple* $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$ *where:*

(1) Q *is a nonempty finite set of so called* **states**;

(2) Σ *is an* **input alphabet** (*cf. Definition 7.5*);

(3) Γ *is a nonempty finite set of* **output alphabet**;

(4) $\delta : Q \times \Sigma \rightarrow Q$ *is a* **transition function**;

(5) $\gamma : Q \times \Sigma \rightarrow \Gamma$ *is an* **output function**, *and*

(6) $q_0 \in Q$ *is called a* **starting state** *or* **initial state**.

As an example we consider Figure 7.2 which depicts a 'minimal' finite state machine. In it $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{a, b\}$, where q_0 is the initial state.

Notation 7.1
If $p, q \in Q$, $s \in \Sigma$, $t \in \Gamma$, $\delta(p, s) = q$, *and* $\gamma(p, s) = t$, *then an arc from* p *to* q *is labelled by* s/t. *A transition of this kind is denoted by* $p \xrightarrow{s/t} q$.

In order to compute an input sequence, Definition 7.6 is extended as follows:

Definition 7.7
For a FSM $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$,

(1) the function δ can be extended to a function on strings $\delta_{ext} : Q \times \Sigma^\star \rightarrow Q$ such that if $q \in Q$, $a \in \Sigma$, $y \in \Sigma^\star$, and $w = ya$, then define $\delta_{ext}(q, \epsilon) = q$ and $\delta_{ext}(q, w) = \delta(\delta_{ext}(q, y), a)$.

(2) the function γ can be extended to a function $\gamma_{ext} : Q \times \Sigma^\star \rightarrow \Gamma^\star$ such that if $q \in Q$, $a \in \Sigma$, $y \in \Sigma^\star$, and $w = ya$, then define $\gamma_{ext}(q, \epsilon) = \epsilon$ and $\gamma_{ext}(q, w) = \gamma_{ext}(q, y)\gamma(\delta_{ext}(q, y), a)$.

Nevertheless, in the following, the notations δ and γ will be used for δ_{ext} and γ_{ext}, respectively. That is, if $q \in Q$ and $w \in \Sigma^\star$, then $\delta(q, w)$ will be the next state of M after processing the string w beginning from state q. $\gamma(q, w)$ will be the output when M processes w beginning from state q. In example of Figure 7.2, $\delta(q_0, 00110) = q_1$ and $\gamma(q_0, 00110) = abbba$

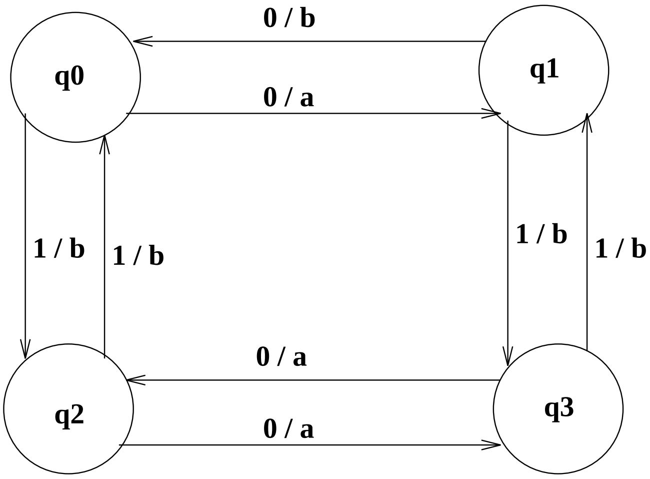


Figure 7.2: A finite state machine

Definition 7.8
For a FSM $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$, Let $w = w_0 w_1 \dots w_{n-1}$ be a string, and let $q \in Q$ be a state. Then the k **successor** *of q (with respect to w) is the state $p = \delta(q, w_0 w_1 \dots w_{k-1})$, where $k \geq 1$.*

Definition 7.9
For a FSM $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$, Let $p, q \in Q$ and $w \in \Sigma^\star$ Then w is said to **distinguish** *between p and q iff $\gamma(p, w) \neq \gamma(q, w)$.*

In other words, if two different computations of M on input w are considered, one starting in state p and another starting in state q, and if these computations produce different outputs, then w distinguishes p and q. In example of Figure 7.2, the string 110 will distinguish q_1 from q_2.

Definition 7.10
A **characterisation set** *W for FSM $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$ is a set $W \subseteq \Sigma^\star$ such that $\forall q_i, q_j \in Q$, where $i \neq j$, there exists a string $w \in W$ such that $\gamma(q_i, w) \neq \gamma(q_j, w)$, and no other $v \in \Sigma^\star$ with $|v| < |w|$ distinguishes q_i and q_j.*

For example, in Figure 7.2, $\{110, 010, 10\}$ is a characterisation set.

Definition 7.11
Let $M_1 = (Q_1, \Sigma, \Gamma, \delta_1, \gamma_1, q_{0_1})$ and $M_2 = (Q_2, \Sigma, \Gamma, \delta_2, \gamma_2, q_{0_2})$ be two FSMs; let $q_i \in Q_1$, $p_j \in Q_2$, and let $W \subseteq \Sigma^\star$. If $\gamma_1(q_i, w) = \gamma_2(p_j, w)$, for all $w \in W$, then q_i and p_j are **W-equivalent** *(denoted by $q_i \sim_W p_j$). If q_i and p_j are W-equivalent for all $W \subseteq \Sigma^\star$, then q_i and p_j are* **equivalent** *(denoted by $q_i \sim p_j$). If q_{0_1} and q_{0_2} are W-equivalent then M_1 and M_2 are* **W-equivalent**. *If q_{0_1} and q_{0_2} are equivalent then M_1 and M_2 are* **equivalent**.

Furthermore, let $f : Q_1 \to Q_2$ be a one-to-one and onto function such that

(1) $f(q_{0_1}) = q_{0_2}$ and

(2) $\forall q_i \forall p_j \forall x \forall y\ (q_i \in Q_1 \wedge p_j \in Q_2 \wedge x \in \Sigma \wedge y \in \Gamma \wedge (q_i \xrightarrow{x/y} p_j \text{ iff } f(q_i) \xrightarrow{x/y} f(p_j)))$. Then f is said to be an **isomorphism** *from M_1 to M_2. If there is an isomorphism from M_1 to M_2, then M_1 and M_2 are* **isomorphic**.

Definition 7.12
An FSM $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$ is **minimal** *if the number of states in M is less than or equal to the number of states for any machine $M' = (Q', \Sigma, \Gamma, \delta', \gamma', q_0')$ which is equivalent to M.*

Remark 7.2 If a FSM M has a characterisation set W, then M is said to be **minimal**.

7.6.2 Existing solutions based on finite state machines (FSMs)

In this subsection some test selection methods based on FSMs are described and several criticisms of FSMs analysis/testing from literature are discussed. Many test approaches have been based on finite state machines (FSMs) (cf. [8, 15, 22, 23, 37, 45, 57, 58, 59, 97, 103, 137]). This has been especially motivated by the fact that FSMs are a recognised and a very wide spread model for the description of system behaviour. In addition, many of the present CASE-tools support formal specification with FSMs. The study of testing finite state machines is to discover aspects of their behaviour and to ensure their correct functioning. In this research area, a lot of effort is being devoted to finding efficient algorithms or heuristics, which, for instance, satisfy certain test criteria (cf. [37, 45, 22]).

This Subsection 7.6.2 is organised as follows. We begin by giving some definitions of faults in a FSM model in Subsubsection 7.6.2.1. Then in Subsubsection 7.6.2.2 we review the most important test selection methods for FSMs.

7.6.2.1 Definitions of faults in a FSM model

This Subsection 7.6.2.1 is devoted to describing some of the faults which are detected by the test selection methods discussed in Subsection 7.6.2.2.

Definition 7.13
A transition is said to have an **output fault***, if for the corresponding state and received input, the implementation under test* (*IUT*) *provides an output different to the one specified by the output function γ.* (*cf. Definitions 7.6 and 7.7*)*.*

Definition 7.14
A transition is said to have a **transfer fault***, if for the corresponding state and received input, the IUT enters a different state than the one specified by the transition function δ.* (*cf. Definitions 7.6 and 7.7*)*.*

Definition 7.15
An IUT is said to have an **additional** (*respectively,* **missing**) **transition fault***, if for a pair of considered state and input, at least one more* (*respectively, one less*) *transition* (*with respect to the specification*) *is defined.*

Remark 7.3 An IUT is said to have **multiple faults** iff one of its transitions has one or several faults described above (cf. Definitions 7.13, 7.14 and 7.15).

7.6.2.2 The most best known test selection methods for FSMs

In this subsubsection we review the most important test selection methods for FSMs. The four best known state-based test selection methods are called *T-method* [110], *W-method* [22], *Distinguishing Sequence Method* (*DS*) [43], and *Unique-Input-Output* (*UIO*) [148]. All these methods are similar in such a way that they provide a technique for generating a set of one or more strings $\{p_1, p_2, \dots, p_k\}$ which is often referred to as *test suite*. The implementation under test (IUT) I is then tested by applying these test strings (test sequences) to verify that it conforms[8] to the specification S.

Assumptions about the specification S and the implementation I: Usually, all the four above mentioned test selection techniques assume the existence of a *specification* S as basis for the development of an *implementation under test* (UIT) I. In the following, let the specification S and the implementation I be two FSMs $M_1 = (Q_1, \Sigma, \Gamma, \delta_1, \gamma_1, q_{0_1})$ and $M_2 = (Q_2, \Sigma, \Gamma, \delta_2, \gamma_2, q_{0_2})$, respectively. It is assumed that M_1 is *minimal* and *strongly connected*[9], i. e., each state p of the FSM M_1 is reachable from any other state q in the machine M_1. In addition, it is assumed that $|Q_1| = n$ and $|Q_2| = m$, where $m \geq n$. These assumptions are especially considered a necessary (and sufficient) condition for the existence of a *characterisation set* W of M_1.

The four important approaches are now reviewed in the following.

[8] This term originates from the area of testing communication protocols (cf. [148, 137]), but it can as well be used in software engineering.

[9] cf. Definition 7.30

T-method: The T-method [110] generates a test suite of a single test called a “transition tour” More specifically, given the FSM M_2 (i. e., which represents the IUT I), a **transition tour** is an input sequence which takes the machine M_2 from its initial state, traverses every transition at least once, and returns to its initial state. The T-method is thus concerned with checking the transitions between the IUT I. It has the power of detecting all output faults (in the absence of transfer faults), but there is no guarantee of detecting any transfer fault.
Note that the T-method is based on the test criterion which corresponds to our Definition 7.2 in Section 7.3.

DS-method: The DS-method [43] uses a **distinguishing sequence** (DS) for state identification (cf. Definition 7.9). It uses a two-phase approach. The tests of the first phase check that each state defined by the specification M_1 also exists in the implementation M_2. The tests of the second phase check all remaining transitions defined by the specification M_1 for correct output and transfer in the implementation M_2. According to Definition 7.9, however, a distinguishing sequence w is capable of verifying state p, but not necessarily any other state.

Nevertheless, under the assumption that the number of states of the implementation M_2 is not larger than that of the specification M_1 (i. e., $m = n$), the DS-method guarantees detecting any transfer fault. In this particular case, M_2 is as well minimal. Unfortunately not every minimal FSM has a distinguishing sequence (cf. [57]).

UIO-method: The UIO-method [148] uses a set of **unique input/output** (UIO) sequences for state identification. Specifically, a UIO for a state q_i is an input sequence that when executed from q_i produces a different output than when executed from any other state q_j. Thus the UIO-method generates test sequences which check whether each transition has the correct **next-state** and the correct **output**. To check for the correctness of the next-state reached by the FSM M_2 after the execution of the transition under consideration, the corresponding UIO sequence is applied. Generally, test sequences which are generated by UIO-method are shorter than the ones produced by the DS-method.

However, a UIO for a state q_i may not be able to identify any state other than q_i. A DS is clearly a UIO for each state. Whereas not every FSM has a UIO for each state, some FSMs without a DS have a UIO for each state (cf. [57]).

W-method: The W-method [22] comprises the selection of two set of input sequences: **W-set** known as the **characterisation set** (cf. Definition 7.10) and **P-set** called the **transition cover set**.

An *informal* algorithm for constructing P is presented by Chow [22]. In the following, we give the formal description of P.

Definition 7.16
A **transition cover set** *P for FSM $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$ is a set $P \subseteq \Sigma^\star$ of input sequences constructed as follows:*

Let $x_r \in \Sigma, p_s \in Q$

$P = \{w = x_0 x_1 x_2 \dots x_{n-1} \mid$

$$
\begin{aligned}
&n \in [1, |Q|] \wedge \\
&\exists p_0 x_0 p_1 x_1 \dots x_{n-2} p_{n-1} x_{n-1} p_n \\
&(p_0 = q_0 \wedge \\
&\forall i \in [1, n](\delta(p_{i-1}, x_{i-1}) = p_i) \wedge \\
&\forall j \forall k (j, k \in [0, n-1] \ \wedge \ j \neq k \implies p_j \neq p_k) \\
&) \\
&\} \cup \{\epsilon\}.
\end{aligned}
$$

Note: We assume the existence of a reset with no output $(r/-)$, leading to the initial q_0 for every state q in M_1, i. e., $q \xrightarrow{r/-} q_0$.

It is easy to see that the resulting set P will generate $n(n-1)/2$ strings since a string will consist of at most $n-1$ symbols.

Let W be a characterisation set (cf. Definition 7.10). The test sequences generated by the W-method are formed by the concatenation of the transition cover set P and the **distinguishing set** Z (i. e., $P \bullet Z$), where Z is defined as follows:

$$Z =_{def} \ W \cup \Sigma^1 \bullet W \cup \Sigma^2 \bullet W \cup \ldots \cup \Sigma^{m-n} \bullet W = \bigcup_{i=0}^{m-n} \Sigma^i \bullet W. \tag{7.1}$$

Remark 7.4 The application of the distinguishing set Z instead of the characterisation set W is due to the bound number of states m in the implementation M_2 which may larger than number of states n in the specification M_1 (i. e., $m > n$). In case $m = n$, one obtains $Z = W$, and as a result the set of test sequences is the set P and W (i. e., $P \bullet W$). In fact, each test sequence starts with the initial state after the application of the 'reset operation' (See the set of partial paths in the testing tree described by Definition 7.16 above. In this case, to identify a reached state (*next-state*) after a transition, all the sequences in the set W are applied separately. The length of the test suite resulting from concatenation of these test sequences is "propotional" to $|W|$. According to Chow [22], the set of test sequences $P \bullet W$ detects any output or transfer error in the FSM M_2 provided number of states m in M_2 is bounded to $m = n$.

The following theorem summarizes the fault detection power of the W-method due to [22].

Theorem 7.1 *Let M_1 be a minimal FSM with n states, and let M_2 be a (minimal) FSM with m states ($m \geq n$). Asssume that both machines have the same input alphabet Σ and output Γ. Let P be a transition cover set for M_1. Let W be a characterisation set for M_1. Then M_1 and M_2 are equivalent iff M_1 and M_2 are $P \bullet W$-equivalent.*

Remark 7.5 In [22], M_2 is assumed to be minimal. However, in Theorem 7.1 we do not necessarily have to assume the requirement of minimality for M_2. In case M_2 is not minimal, we can reduce M_2 to M_2'. Then a similar proof can be done for M_2' which is equivalent to M_2.

The validity of Theorem 7.1 is justified by proving the four lemmas which are given in the following.

Proof

According to Definition 7.16 a transition cover set P must satisfy the following requirements:

(1) $\epsilon \in P$ and

(2) $\forall p \forall q \forall x \exists w \; (p \in Q \;\wedge\; q \in Q \;\wedge\; x \in \Sigma \;\wedge$
$(\delta(p,x) = q \;\wedge\; w \in P \;\wedge\; wx \in P \;\wedge\; (\delta(q_0, w) = p)))$.

In other words, the set P is any set of strings such that for any transition from a state p to another state q on input symbol x, there exist input strings w and wx in P such that w causes M_1 to enter state p from the initial state q_0. The string wx results thence in a computation that initiates in q_0, proceeds to p, and then traverses the transition to q.

We use these observations to give an initial basis which will be necessary in the process of proving Lemma 7.1.

Initial Basis 7.1
Let $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$ be a FSM. Let $p \in Q \;\wedge\; q \in Q \;\wedge\; w \in \Sigma^\star \;\wedge\; z \in \Sigma^\star$, where w is a prefix of z, and suppose that w distinguishes p and q. Then z will distinguish p and q.

Lemma 7.1
Let $M_1 = (Q_1, \Sigma, \Gamma, \delta_1, \gamma_1, q_{0_1})$ and $M_2 = (Q_2, \Sigma, \Gamma, \delta_2, \gamma_2, q_{0_2})$ be FSMs, where $n = |Q_1|$, $m = |Q_2|$, and $m \geq n$. Let W be a characterisation set for M_1. Let $p, q \in Q_2$, then $p \sim_W q \Leftrightarrow \forall w \in W : \gamma_2(p, w) = \gamma_2(q, w)$. The characterisation set W partitions Q_2 into at least σ equivalence classes, where $1 \leq \sigma \leq m = |Q_1|$. Then for all $0 \leq i \leq m - \sigma$ $W \cup \Sigma^i$ partitions Q_2 into at least $\sigma + j$ equivalence classes, where $0 \leq j \leq m - \sigma$.

Proof of Lemma 7.1

(1) *Induction Basis:*
$i = 0$. Since $W \subset \Sigma^\star$ partitions the states of M_2 into at least σ equivalence classes, it follows directly that $W \cup \Sigma^i = W \cup \{\epsilon\}$ partitions the states of M_2 into at least $\sigma + i = \sigma$ equivalence classes.

(2) *Induction Hypothesis:*
Assume that there exists a k, where $0 \leq k \leq m - \sigma - 1$, such that $W \cup \Sigma^k$ partitions Q into at least $\sigma + k$ equivalence classes.

(3) *Induction Step:*
To show that $W \cup \Sigma^{k+1}$ partitions Q into at least $\sigma + k + 1$.

By induction hypothesis $W \cup \Sigma^k$ partitions Q into at least $\sigma + k$ equivalence classes. It may be the case that $W \cup \Sigma^k$ partitions Q into at least $\sigma + k + 1$ equivalence classes, so that in this particular case, by Initial Basis 7.1, it follows that $W \cup \Sigma^{k+1}$ also partitions Q into at least $\sigma + k + 1$ equivalence classes. Now suppose that $W \cup \Sigma^k$ does not partition Q into at least $\sigma + k + 1$ equivalence classes. This implies by the induction hypothesis that $W \cup \Sigma^k$ partitions Q into exactly $\sigma + k$ equivalence classes. Moreover, since $0 \leq k \leq m - \sigma - 1$, it follows that the sum of the number of equivalence classes is $\sigma + k \leq m - 1$ which is less than the number of states in Q. This implies that there must be a pair of states $q_i \in Q$ and $q_j \in Q$ that is not distinguished by any string in $W \cup \Sigma^k$. But, since M_2 is minimal, it follows that q_i and q_j are distinguished by some string. Suppose that the shortest string is of length $r > k$ (cf. Definition 7.16). That is, q_i and q_j are distinguished by some string of length r but not any string of length $r - 1$ or less. This means that the pair q_i and q_j is distinguished by some string in Σ^r but by no string in Σ^{r-1}. Now let $w \in \Sigma^r$ be a string that distinguishes q_i and q_j, let $q'_i \in Q$ and $q'_j \in Q$ be the $(r - k - 1)$ successors of q_i and q_j, respectively, when executing w. Then q'_i and q'_j must be distinguished by a string of length $k + 1$ but by no string length of k. This implies that q'_i and q'_j are distinguished by Σ^{k+1} but not by Σ^k. Therefore, $W \cup \Sigma^{k+1}$ partitions the states in Q into at least $\sigma + k + 1$ equivalence classes. □

Lemma 7.2

Let $M_1 = (Q_1, \Sigma, \Gamma, \delta_1, \gamma_1, q_{0_1})$ and $M_2 = (Q_2, \Sigma, \Gamma, \delta_2, \gamma_2, q_{0_2})$ be FSMs, where $n = |Q_1|$, $m = |Q_2|$, and $m \geq n$. Let P be a transition cover set for M_1, and let W be a characterisation set for M_1. If M_1 and M_2 are $P \bullet W$-equivalent, then for each $q_i \in Q_1$ there exists at least one $p_j \in Q_2$ such that q_i and p_j are W-equivalent. It is as well true that for no other $q_r \in Q_1$, where $r \neq i$, is it possible that q_r and p_j are also W-equivalent.

Proof of Lemma 7.2

Suppose that M_1 and M_2 are $P \bullet W$-equivalent. By definition of P holds $\forall q_i (\exists w \in P \wedge (\delta_1(q_{0_1}, w) = q_i))$ (see requirement (2) at the beginning of proof of Theorem 7.1). Let $\delta_2(q_{0_2}, w) = p_j$. Since M_1 and M_2 cannot be distinguished by any string in $P \bullet W$, it follows that q_i and p_j cannot be distinguished by a string in W. Thus q_i and p_j are W-equivalent. On the other hand, suppose there exists another q_r such that $r \neq i$ and that q_r and p_j are also W-equivalent. This would imply that q_i and q_r would be W-equivalent which is a contradiction to the fact that W is a characterisation set for M_1. □

Lemma 7.3

Let $M_1 = (Q_1, \Sigma, \Gamma, \delta_1, \gamma_1, q_{0_1})$ and $M_2 = (Q_2, \Sigma, \Gamma, \delta_2, \gamma_2, q_{0_2})$ be FSMs, where $n = |Q_1|$, $m = |Q_2|$, and $m \geq n$. Let P be a transition cover set for M_1. Let W be a characterisation set for M_1. In addition, let $Z = W \cup \Sigma^{m-n}$. If M_1 and M_2 are $P \bullet W$-equivalent, then Z will partition the states Q_2 into $m = |Q_2|$ equivalent classes.

Proof of Lemma 7.3

It follows from Lemma 7.2 that every state in Q_1 is W-equivalent to a unique state in Q_2. Hence, W-equivalence partitions the states in Q_2 into at least $n = |Q_1|$ classes. The derived number of classes follows from Lemma 7.1. □

Lemma 7.4

Let $M_1 = (Q_1, \Sigma, \Gamma, \delta_1, \gamma_1, q_{0_1})$ and $M_2 = (Q_2, \Sigma, \Gamma, \delta_2, \gamma_2, q_{0_2})$ be FSMs, where $n = |Q_1|$,

$m = |Q_2|$, and $m \geq n$. Let W be a characterisation set for M_1. Let $q_i, q_j \in Q_1$, $x \in \Sigma$ and $y \in \Gamma$, where $q_i \xrightarrow{x/y} q_j$. In addition, let $Z = W \cup \Sigma^{m-n}$. If M_1 and M_2 are $P \bullet Z$-equivalent, then there exist states $p_r, p_s \in Q_2$ such that p_r and p_s are Z-equivalent to q_i and q_j, respectively, and $q_r \xrightarrow{x/y} q_s$.

Proof of Lemma 7.4

Suppose that $q_i, q_j \in Q_1$, $x \in \Sigma$ and $y \in \Gamma$, where $q_i \xrightarrow{x/y} q_j$. Following definition of P, there exists $w \in \Sigma^\star$ and $z \in \Gamma^\star$ such that $w \in P$ and $wx \in P$ and $q_{0_1} \xrightarrow{w/z} q_i \xrightarrow{x/y} q_j$. So let $p_r, p_s \in Q_2$, $z' \in \Gamma^\star$, and $y' \in \Gamma$ be such that $q_{0_2} \xrightarrow{w/z'} p_r \xrightarrow{x/y'} p_s$. Now, since M_1 and M_2 are $P \bullet Z$-equivalent, and in this case $\{w\} \bullet Z$-equivalent and $\{wx\} \bullet Z$-equivalent, it follows that q_i and q_j are Z-equivalent to p_r and p_s, respectively, and $y = y'$. □

The proof to justify that M_1 and M_2 are equivalent is trivial, and therefore, is omitted here.

7.6.2.3 Complexity of W-method

Following Remark 7.2 and Theorem 7.1, every minimal FSM has a characterisation set. Unfortunately, this identification method requires a number of inputs to be executed from each state, and hence requires more effort. Specifically, let M_1 and M_2 have n states and m states, respectively, where $m \geq n$. Asssume that both machines have the same input alphabet Σ where $|\Sigma| = k$. Then the W-method will generate a test set consisting of

$$|P||W|(k^{m-n+1} - 1)/(k - 1) \tag{7.2}$$

strings.

Nonetheless, though the W-method [22] has the largest generated test set, it is generally accepted that the W-method has the best fault detection capability. Therefore, this approach has received the widest attention in the research area of software testing. Indeed, there have been several approaches to develop more efficient algorithms which generate a smaller number of tests than the W-method (cf. [37, 8, 45]). The main idea of all these approaches is to provide versions of the W-method that have the same fault detection capability as the W-method but at the same time generate a smaller number of tests than the W-method. In this thesis, due to space limitations, we cannot consider the details of these approaches.

In Table 7.1 we summarize the results of the four best known state-based test selection methods.

7.6.3 Criticisms of the proposed state-based techniques

In this subsection we discuss the problems/weaknesses of the proposed state-based techniques and indicate how these criticisms are addressed in this thesis. To allow a theoretical comparison of the FSM-based techniques, e. g., developed by Chow [22] and Fujiwara et. al. [37] with our test concepts (cf. Subsection 7.7.3), we found it necessary to provide

Fault detection				size of test set	reset capability	formal proof	require distinguishing entity
Method	output	transfer	additional states				
W-method	+	$+^\star$	+	largest	+	+	characterrisation set (polynomial size)
DS-method	+	$+^\star$	−	medium	+	−	distinguishing string (PSPACE-complete)
UIO-method	+	−	−	medium	+	−	unique input-output string (PSPACE-complete)
T-method	+	−	−	smallest	+	−	none

Table 7.1: Comparing the four methods

$+^\star$: This requires that the number of states of the implementation must remain within a certain bound.

the full theoretical background information of reactive models, especially to classify statecharts in terms of *nondeterminism*, *pure parallelism* and *bounded cooperative concurrency*, and above all, to remain as close as possible to classical finite automata (cf. Chapter B).

Recall the Assumption 7.1 at the beginning of this Section 7.6. We suggested that one way of solving the problem formulated in Assumption 7.1 is to consider more restrictive models, such as FSMs. The main problem of such a proceeding, however, is that FSMs are too restrictive for many applications, e. g., real world concurrent computing.

A **solution** suggested by Chow is to separate the control structure of a program from the data structure and to represent the former as a FSM. Then the state-based techniques, e. g., developed by Chow [22] and Fujiwara et. al. [37] could be used to test the control structure.

Nonetheless, the assumption that the control structure of the system can be modelled separately from the data variables is not realistic in many cases. It would imply that the next state depends solely on the current state and the input which is not usually the case. The idea that the variables that affect the program control could be replaced by a number of additional states is also impractical since in many cases this number can be very large.

In Appendix Subsection B.1.2 we discuss the classical automata with respect to their suitability for modelling real world concurrent computing. The application of *existential* and *universal features*[10] which are perhaps the most well-known notions for modelling concurrency in complexity theory have two drawbacks:

(1) No communication exists in the spawned processes, except when time comes to decide whether the input should be accepted.

[10] These features are described in Subsection B.1.1 on page 194

(2) Existential and universal branching are unbounded; new processes can be spawned without limit as the computation proceeds (i. e., as the length of the input word grows).

As a solution to overcome the first drawback, in particular to capture real world concurrency, we argue that one requires cooperative concurrency (cf. [32, 31]). The idea of *cooperative concurrency* is that during a single computation, the mechanism can be in more than one state (component) and that these are able to cooperate (communicate) while achieving a common goal.

In thinking about the second drawback we have to remember that in the real world the number of processes participating in concurrent computations is bounded, and therefore cannot be assumed to grow as the size of input grows. Hence, the main question is to determine how *bounded cooperative concurrency* progresses with respect to the two classical kinds of branching in the realm of finite automata and their extensions. The idea of bounded cooperative concurrency is formulated in Appendix Section B.2, and the defined classes correspond to different kinds of statecharts.

In other words, statecharts were proposed to overcome the limitations of FSMs, such as the ones seen above. Therefore, to come back to the problem formulated in the Assumption 7.1 on page 125 it is necessary to find tests which are based on the model of statecharts.

We are not trying to say that the test selection methods based on FSMs, e. g., developed by Chow [22] and Fujiwara et. al. [37] are not useful and important in software testing, but they are particularly suitable for testing models with a limited number of states.

In [15] a rather *informal* testing method for statecharts is described. This method is based on a method of testing X-machines [97]. According to Singh [15] the method should test that the implementation of a statechart design produces the same output as the design when exposed to the same inputs. The basic idea behind Singh's method is to apply inputs derived from the design to an implementation under test and making sure that the output observed is the same as that required by the design. Before we can outline Singh's testing method we need the following definition of a **state cover set**.

Definition 7.17
A **state cover set** *C for FSM $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$ is a set $C \subseteq \Sigma^\star$ such that $\forall q_i \in Q$, where $i \in [1, \ |Q|]$, there exists a string $w \in C$ such that $q_0 \xrightarrow{w} q_i$. For the initial state q_0 we have $q_0 \xrightarrow{\epsilon} q_0$. The empty sequence ($\epsilon$) belongs to C.*

Specifically, to construct a set of test cases, Singh [15] requires three auxiliary sets:

(1) Σ (cf. Definition 7.5 on page 126)

(2) State cover set (denoted by C) (cf. Definition 7.17 above)

(3) Characterisation set (denoted by W) (cf. Definition 7.10 on page 128).

With these sets, a set of test sequences is constructed as usual:

$$T = C \bullet (\{\epsilon\} \cup \Sigma \cup \Sigma^2 \cup \ldots \cup \Sigma^{m-n+1}) \bullet W \tag{7.3}$$

where n is number of states in a design; m is the maximum expected number of states in the implementation under test.

Without doubt, this method can be fruitful for testing very 'simple' statecharts. In this context, *simple* is used to refer to a statechart with a very small number of states and transitions. For complex statecharts, Singh [15] proposes an approach of *incremental test case development*. This should be achieved as follows in two phases:

(1) *Phase* 1*:*

   (a) begin with a generation of a triple $(\Sigma_{MAIN}, C_{MAIN}, W_{MAIN})$ called the *test case basis* for the main statechart

   (b) then continue with a generation of a triple $(\Sigma_{OR-state_{id}}, C_{OR-state_{id}}, W_{OR-state_{id}})$ for each non-basic $OR - state_{id}$[11].

   considering all their substates as basic ones.

(2) *Phase* 2*:*

   (a) combine the triples from Phase 1 (i. e., (1)(a) and (1)(b)) to get a resulting triple (Σ, C, W).

   (b) Finally from the resulting triple (Σ, C, W) a set of test cases can be constructed following the Equation 7.3 above.

Note that since the Equation 7.3 above corresponds to the W-method given by Equation 7.1 on page 131, whose test effort is known to be very large (cf. Equation 7.2 on page 134), it is fair to say that the problem of testing large and complex concurrent cannot be not quite solved by the above approach. In other words, such tests should be applied to special parts of the system model (e. g., 'critical' parts) to allow a practical use.

To make a step further, we established new methods based on path testing.

Our *module test approach* is a *graph-theoretic* based technique. Using this approach, we will show that an acceptable small set of test paths that detects all used states and executes all used transitions in a component can be established efficiently. The problem of determining the test paths of large and complex reactive system models[12] is solved by deriving tests from the *level* of the statecharts. In this way, our module test approach can be used to reduce the space explosion problem of large and complex concurrent system models. By applying this test method, we thus illustrate that the module test concepts can provide a more convincing approach to the problem testing on a systematic basis.

[11] where the index *id* stands for state identifier

[12] hierarchical, structured statecharts

7.7 Ways of approaching the module test concepts

The purpose of this section is to establish the module concepts which operate on the *hierarchical* statecharts. In other words, the components of a statechart to be tested are derived from the considered *level* of the statechart. This section is organised as follows: In Subsection 7.7.1 we consider the problems of testing hierarchical, structured system models. In Subsection 7.7.2 we introduce some formal notations and concepts related to digraphs which are necessary for a formal development of graph-theoretic based methods. This aim of Subsection 7.7.3 is to find a solution to overcome the problems, e. g., the space explosion problem, discussed in Subsection 7.7.1.

7.7.1 Problems of testing hierarchical system models

In this subsection we consider the problems of testing hierarchical, structured system models. When testing large models, the individual modelled components (modules) are first tested in isolation (cf. [88]). In particular, when testing *hierarchical*, structured system models, it is useful and vital to analyse components (*modules*) in isolation. The *module test* (also known as *unit test*), hence allows the system designer or tester to develop a subset of execution sequences whose size is relatively manageable as a test set.

In Subsubsection 7.1.3.3, an example has been given to illustrate the space explosion problem in the field of testing (concurrent) reactive systems. Here, *hierarchical*, structured statecharts surely present an additional difficult task to be dealt with. Let us consider an *OR*-statechart (cf. Definition 5.13) with many *levels* which will be tested using the criterion stated by Definition 7.2, i. e., finding a test suite which executes every transition in the *OR*-statechart.

- The number of execution sequences which are possible “inside” (the descendants) of such a clustered state can be unmanageable.
- The execution duration of single tests might be too long because the length of the generated sequences might be very long.

Therefore, the concepts which will be developed in Subsection 7.7.3 should mainly attempt to provide a solution which overcomes the space explosion problem, and above all, should deal with the generation of test paths (test sequences) for a *hierarchical* (structured) statecharts model.

7.7.2 Preliminaries for graph-theoretic based methods

In Subsection 7.1.2 we presented an informal background on path testing. In this subsection we introduce some formal notations and concepts related to digraphs which are necessary for a formal development of graph-theoretic based methods.

Let $G = (V, v_0, E, inc, label)$ be a labelled digraph[13]. In the following, we write $G = (V, E, inc, label)$ instead of $G = (V, v_0, E, inc, label)$.

[13] cf. Definition 5.17, page 49

7.7.2.1 Definition of subgraph

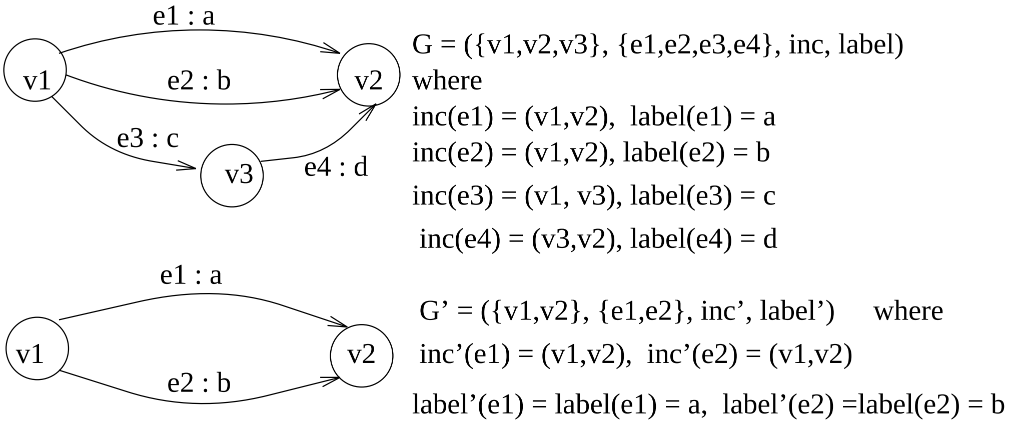
Definition 7.18 *Let $G = (V, E, inc, label)$ be a labelled digraph.*
A digraph $G' = (V', E,' inc', label')$ is said to be a **subgraph** *of G iff*

(1) $V' \subseteq V$ and

(2) $E' \subseteq E$ is a subset of E such that
$inc' =_{def} {}^{inc}|_{E'}$ maps only into $V' \times V'$, i.e.,
$inc' : E' \to V' \times V'$.

Example of subgraph G' of digraph G

As an example, figure 7.3 shows a subgraph G' of digraph G.



G = ({v1,v2,v3}, {e1,e2,e3,e4}, inc, label)
where
inc(e1) = (v1,v2), label(e1) = a
inc(e2) = (v1,v2), label(e2) = b
inc(e3) = (v1, v3), label(e3) = c
inc(e4) = (v3,v2), label(e4) = d

G’ = ({v1,v2}, {e1,e2}, inc’, label’) where
inc’(e1) = (v1,v2), inc’(e2) = (v1,v2)
label’(e1) = label(e1) = a, label’(e2) =label(e2) = b

Figure 7.3: G' is a subgraph of graph G.

7.7.2.2 Path Definitions

Definition 7.19 *Let $G = (V, E, inc, label)$ be an arbitrary digraph. An edge $(v, w) \in E$ is said to* **leave** *vertex v and* **enter** *vertex w and that vertex w is* **adjacent** *to v. It is said that v is a* **predecessor** *of w, and w a* **successor** *of v.*

Definition 7.20
A **successor function** *$Adj^+ : V \to \wp(V)$ is defined such that $Adj^+(v_i) =_{def} \{v_j | (v_i, v_j) \in E\}$. This set is called the set of* **immediate successors** *of a node. It may be empty. The inverse of the successor function is a* **predecessor function** *$Adj^- : V \to \wp(V)$, which gives the* **immediate predecessors** *of a node: $Adj^-(v_j) =_{def} \{v_i | (v_i, v_j) \in E\}$. It too may be empty.*

Definition 7.21 *Let $G = (V, E, inc, label)$ be an arbitrary digraph. The* **out-degree** *of a vertex v, i.e., the number of edges leaving v, denoted by $d^+(v)$ is defined as $d^+(v) =_{def} |Adj^+(v)|$.*

The **in-degree** *of a vertex v, i.e., the number of edges entering v, denoted by $d^-(v)$ is similarly defined as $d^-(v) =_{def} |Adj^-(v)|$.*
A vertex whose out-degree (in-degree) equals zero is called a **sink** (**source**). *If both $d^+(v) = 0$ and $d^-(v) = 0$, then v is an* **isolated vertex**

Definition 7.22 *Let $G = (V, E, inc, label)$ be an arbitrary digraph.*
A sequence of vertices $(v_0, v_1, v_2, \dots, v_k)$, $k \geq 0$, is a **path** *(of length k) from vertex v_0 to vertex v_k iff there is an edge which leaves v_{i-1} and enters v_i for all $i : 1 \leq i \leq k$.*

Definition 7.23 *A path is* **simple** *if all vertices on the path, except possibly the first and last, are distinct.*

Definition 7.24 *A* **cycle** *is a simple path $(v_0, v_1, v_2, \dots, v_k)$ with $k \geq 1$ in which $v_0 = v_k$. If (v, v) is an edge, we say node v has a* **loop**.

Notation 7.2 *A simple path from v_0 to v_t, where v_0 is a source node and v_t is sink node (terminal) is referred to as an* **s-t path**.

Notation 7.3 *The terms* **circuit** *or* **closed path** *are frequently used in different literatures for a* **cycle**.

Definition 7.25 *A digraph is* **rooted** *if there exists at least one vertex r such that for each vertex v there is a path p from r to v. The vertex r is called* **root** *of the graph.*

Definition 7.26 *A digraph that contains no cycle is called* **acyclic** *or* **dag** (**directed acyclic graph**).

Note *Throughout this work, unless stated otherwise it will be assumed that the labelled digraph $G = (V, E, inc, label)$ is finite. The term graph shall be used interchangeably with digraph.*

7.7.2.3 Traversals of digraphs

Suppose G is a digraph we wish to explore. We begin at some vertex of G, the *start vertex*. The start vertex is marked *visited*. All other vertices of the graph are marked *unvisited*. All edges of the graph are marked *unexplored*. The goal is to visit all vertices of the graph, and in doing so, to explore all edges of the graph. *Exploring* an edge means walking along it.

Search Step:

(1) Select some unexplored edge (v, w) such that v has been visited.

(2) Mark the edge *explored*, and mark w *visited*.

Note that, if (v, w) is explored, then both v and w will be visited, but the reverse does not have to hold, if there are other paths from v to w or from w to v.

Notation: The repeated execution of search steps until all edges are explored is called a *search* of G.

7.7.2.4 Representation

Adjacency structure: Several data structures may be used to represent a digraph G. We assume that the graph G is given by an *adjacency structure* A, which is of the type $array[vertex]$ **of list of vertex**.
For each vertex v, $A[v]$ is a list of all direct successors of v in some order. If a vertex is a direct successor of v via several "multiple" edges, then it occurs in $A[v]$ with this multiplicity.

Note that a single graph may have many adjacency structures; i.e. each ordering of the edges around the vertices of G gives a unique adjacency structure[14], and each adjacency structure corresponds to a unique ordering of edges at each vertex. We shall use the adjacency structure for a graph to perform **depth-first search**. At this juncture, we are not yet interested in the representation of the labels.

7.7.2.5 Depth-first search

There are many ways of searching a graph, depending upon the way in which edges to search are selected. One of the most popular strategies is the depth-first search. It can be used as a foundation in the construction of a large number of efficient, mostly-linear time graph algorithms (see [Tar72]). The data structure D for the depth-first search is a *stack*. Thus, the depth-first search procedure is a recursive procedure.

Suppose an adjacency structure $A[v]$ of a digraph to be searched is given. In the following, we consider a simple digraph (cf. definition 2.2 on page 14). All vertices of G are initially marked *unvisited*. Depth-first search selects one vertex v of G as a *start* vertex; v is marked *visited*. Then each unvisited vertex adjacent to v is searched in turn, using depth-first search recursively. Once all vertices that can be reached from v have been visited, an unvisited vertex is selected as a new start vertex. This process is repeated until all vertices have been visited. *Algorithm 1* is a template for depth-first search. The rule used by depth-first search may be summarized as follows:

> **When selecting an edge to traverse, always choose an edge leading from the vertex most recently visited which still has unexplored edges.**

The main program consists of two FOR loops.

(∗ The first FOR loop is used for the initialization ∗).

(∗ The second FOR loop is only necessary for graphs which do not have a single root. Otherwise it suffices to invoke dfs(r), for the root r ∗).

```
FOR $v := 1$ TO $n$ DO
        $mark[v] := unvisited$;
FOR $v := 1$ TO $n$ DO
        IF $mark[v] = unvisited$ THEN
            dfs($v$)
```

[14] since only the set $Adj^{+}(v)$ is fixed for a vertex (cf. Definition 7.20)

The set of visited vertices with possibly unexplored edges may be stored on a stack. This however, is not explicit in the template because this is implied by the recursive structure of the procedure *dfs*.

```
PROCEDURE dfs($v$ : $vertex$);
    VAR
        $w$ : $vertex$;
    BEGIN
        $mark[v] := visited$;
        FOR $w \in A[v]$ DO
            IF $mark[w] = unvisited$ THEN
                dfs($v$)
    END
```

Algorithm 1 **Depth-first search.**

Let a digraph G be given. Now we consider what happens when a *dfs* is performed on the digraph G. The structure which results from the *dfs* of a digraph has four types of edges. These edges may be classified as follows:

Definition 7.27

(1) The set of edges which lead to a non-visited vertex when traversed during the search form a **tree**. *These edges are called* **tree edges**.

(2) The set of edges which run from **descendants** *to* **ancestors** *in the tree. These edges are called* **back edges**.

(3) The set of edges which run from **ancestors** *to* **descendants** *in the tree. These edges are called* **forward edges**.

(4) The set of edges which run from a **subtree** *to another in the tree. These edges are called* **cross edges**.

In definition 7.27, the reader is assumed to be familiar with the terms *tree*, *descendants*, *ancestors* as well as *subtree* [cf. Len9o]. Before a detailed depth-first search *DFS* is given, we define some *procedures* used by this *DFS*.

```
PROCEDURE initialize($v$ : vertex);
BEGIN
FOR $v := 1$ TO $n$ DO
    $\alpha[v] := 0$;
    (* Initializes all vertices to ‘free’ state ($\alpha(v) = 0$). *)
    (* I.e. each vertex is marked unvisited. *)
END initialize;
```

```
PROCEDURE previsit($v$ : vertex);
STATIC VAR
        $S$ : stack;
BEGIN
        $S \Leftarrow v$;
        (* Put $v$ on stack of vertices (i.e. $Push(v, S)$. *)
        $\alpha(v) := 1$;
        (* Change $v$'s state to 'waiting' state ($\alpha(v) = 1$). *)
END previsit;


PROCEDURE postvisit();
STATIC VAR
VAR
        $S$ : stack;
        $v$ : vertex
BEGIN
        $v \Leftarrow S$;
        (* Delete $v$ from stack (i.e. $Pop(S)$. *)
        $\alpha(v) := 2$;
        (* Change $v$'s state to 'finished' state ($\alpha(v) = 2$). *)
END postvisit;


PROCEDURE newvisit($v, w$ : vertex);
STATIC VAR
        $T$ : stack;
BEGIN
        $T \Leftarrow (v, w)$;
        (* Add $(v, w)$ to stack of tree edges ($Push((v, w), T)$). *)
END newvisit;


PROCEDURE revisit($v, w$ : vertex);
STATIC VAR
        $F, B, C$ : stack;
BEGIN
        IF $num[w] \neq 0$ and $num[w] > num[v]$ THEN
            $F \Leftarrow (v, w)$;
            (* Add $(v, w)$ to stack of forward edges ($Push((v, w), F)$). *)
        ELSE IF $num[w] \neq 0$ and $num[w] < num[v]$ and $\alpha(w) = 1$ THEN
            $B \Leftarrow (v, w)$;
            (* Add $(v, w)$ to stack of back edges ($Push((v, w), B)$). *)
        ELSE IF $num[w] \neq 0$ and $num[w] < num[v]$ and $\alpha(w) = 2$ THEN
            $C \Leftarrow (v, w)$;
            (* Add $(v, w)$ to stack of cross edges ($Push((v, w), C)$). *)
END revisit;
```

Now we are in a position of completing the depth-first search DFS.

```
$i$ : $integer$;
$num$ : $array$[vertex];
$S, T, F, B, C$ : $stack$;
$v, w$ : vertex;
BEGIN
    FOR $v := 1$ TO $n$ DO
    $num[v] := 0$;
$i := 1$; $T := \emptyset$; $F := \emptyset$; $B := \emptyset$; $C := \emptyset$;
initialize($v$);
FOR $v := 1$ TO $n$ DO
    IF $num[v] = 0$ THEN DFS($v$);
RECURSIVE PROCEDURE DFS($v$ : vertex);
BEGIN
    $i := i + 1$; $num[v] := i$;
    previsit($v$);
    FOR $w \in Adj[v]$ DO
    BEGIN
        IF $num[w] = 0$ THEN
        (* w is not yet numbered. *)
          DFS($w$);
          newvisit($v, w$);
        ELSE
          revisit($v, w$);
    END FOR;
    postvisit($v$);
END $DFS$;
END;
```

Algorithm 2 **A detailed depth-first search.**

7.7.2.6 Depth-first search on digraphs

Lemma 7.5 *A digraph $G = (V, E, inc, label)$ is* **acyclic** *if and only if a depth-first search of G yields no back edges.*

Proof (cf. [*CoLeRi*90] *pp. 486-487).*

Lemma 7.6 *Every directed acyclic graph (a* **dag** *for short) $G = (V, E, inc, label)$ consists of at least one vertex v, where $d^-(v) = 0$.*

Proof (cf. [*Bra*94] *pp. 79).*

Definition 7.28 *A* **permutation ord***, $ord : V \to V$, of the nodes of a digraph $G = (V, E, inc, label)$, with $V = \{1, 2, \dots, n\}$, is called a* **topological sorting** *iff $\forall v, w \in V : (v, w) \in E \Rightarrow ord(v) < ord(w)$.*

Lemma 7.6 implies the following theorem.

Theorem 7.2 *A digragh $G = (V, E, inc, label)$ has a topological sorting, if and only if it is acyclic. Proof (cf. [$Sim92$] pp. 53).*

Definition 7.29 *Let $G = (V, E, inc, label)$ be a digraph. Suppose that for each pair of vertices v_0, v_k, there exists paths $p1 = (v_0, \dots, v_k)$ and $p2 = (v_k, \dots, v_0)$. Then G is said to be* **strongly connected**.

Definition 7.30 *Let $G = (V, E, inc, label)$ be a digraph. An* **equivalence relation** $\sim$ *on the set of vertices is defined as follows:*

Two vertices v and w are **equivalent** *iff $v = w$ or there is $v_0 \in V$ and a cycle $p = (v_0, \dots, v_0)$ which contains v and w. Let the distinct equivalence classes under this relation be V_i, $1 \leq i \leq n$. Let $G_i = (V_i, E_i, inc_i, label_i)$, where*

(1) $E_i = \{(v, w) \in E | v, w \in V_i\}$.

(2) $inc_i =^{inc} |_{E_i}$

The subgraphs G_i are called the **strongly connected components** *of G.*

Remark 7.6

(1) Each G_i is strongly connected.

(2) No G_i is a proper subgraph of G_j where $j \neq i$.

7.7.3 Developing concepts to operate on levels of the hierarchical statecharts

This aim of this subsection is to find a solution to overcome the problems of testing hierarchical, structured system models, e. g., the space explosion problem, discussed in Subsection 7.7.1 above. As a possible solution to such problems, we develop **module concepts** which operate on the *hierarchical* statecharts, i. e., the components of a statechart to be tested are derived from the considered *level* of the statechart. Consequently, the graph generated from the statechart specification will be built by applying a top-down abstraction to a collection of states on many levels by construction of "reduced graphs".

Before the concepts "Building reduced graphs" are given, we consider again the problem of interlevel transitions which was discussed in Subsection 5.3.1 on page 59 and in Subsection 5.4.4, page 83. We recall that interlevel transitions deny to respect hierarchy states (cf. Definition 5.14 on page 45), i. e., they cross the borderlines of hierarchy states. In particular, when an inter-level transition is dealt with, one has to accurately define the states that have been exited and those that are entered by taking the transition. Mainly, one has to know which *non-basic* states have been exited and entered in the process of taking the transition.

7.7.3.1 Restriction with respect to hierarchy states

As a possible solution to respect hierarchy states, a restriction has been proposed in [84], chapter 6. The idea behind this restriction is to support "good modelling" (see also [88]). In other words, to achieve a construct which satisfies the following restriction.

Restriction 7.1 *Let a statechart $Sct = (S, r, children, type, \mathcal{H}, hist, default, \mathcal{T})$ be given. For an arbitrary state $s_i \in S$ and a super-state*[15] *$s_h \in S$, if s_i and s_h are OR-states, then the following is always true for the super-state s_h:*

(i) *For all transitions from s_i to the super-state s_h:*

- *there is no transition to the substates of the super-state s_h:*

(ii) *For all transitions from a super-state s_h to s_i:*

- *there is no transition leaving a substate of the super-state s_h to the arbitrary state s_i.*

In other words, only transitions leaving/entering the contour of a super-state are allowed.

For further details, see [84], chapter 6 and [88].

Recall Definition 5.15 in Subsection 5.2.4, which reflects the representation of the basic states and super-states on the corresponding abstraction level of any model of a component (state) of a statechart. However, this definition is very general and does not consider the Restriction 7.1 and, therefore, another definition will be required for the concepts of generation of test paths (cf. Definition 7.34).

7.7.3.2 Basic definitions of a path cover

In the following, we assume that a statechart is represented as a digraph (cf. Definition 5.18 on page 50).

Definition 7.31 *Let $G = (V, v_0, E, inc, label)$ be an arbitrary digraph. A* **path cover for the vertices** *of G is the set of paths, $P = \{p_1, p_2, \ldots, p_k\}$, such that for each $v_i \in V$, there exists a path $p_j \in P$ with $v_i \in p_j$.*

Definition 7.32 *Let $G = (V, v_0, E, inc, label)$ be an arbitrary digraph.*
A **path cover for the edges** *of G is the set of paths, $P = \{p_1, p_2, \ldots, p_k\}$, such that for each $e_i = (v_{i_s}, v_{i_t}) \in E$, there exists a path $p_j = (v_{j,0}, v_{j,1}, \ldots v_{j,l}(j)) \in P$ with $v_{j,0} = v_0$ and $m \in \mathbb{N}_0$ with $v_{i_s} = v_{j,m}$ and $v_{i_t} = v_{j,m+1}$, i. e., the edge e_i is contained in path p_j.*

Considering the high costs of program testing [13], the path test selection strategies are naturally interested in finding path covers with the minimum number of paths. Therefore, the following definition:

Definition 7.33 *The set P is a* **minimum path cover** *if there exists no path cover set P' such that $|P'| < |P|$.*

[15] cf. Definition 5.14, page 45

7.7.3.3 Building of reduced graphs

The main purpose of this subsection is to establish a part of the "framework" which will be used for the generation of test paths (path cover) which operate on different levels. In order to simplify the generation of test paths, we consider *strongly connected components*[16] (*SCCs*) as units in the first step of the generation of test paths. *SCCs* are subgraphs of G, where each node of the subgraph is reachable from any other node in the subgraph. Hence, a corresponding 'reduced graph' of G will be analysed.

Definition 7.34
Let s be a component (state) of an OR-statechart Sct. Let $G_1, G_2, \ldots, G_k$ be the strongly connected components (SCCs) of the associated OR-graph $G = (V, v_0, E, inc, label)$ of Sct with respect to s. Then $G_r^s = (V_r^s, v_{0_r}^s, E_r^s, inc_r^s, label_r^s)$ is called the **reduced graph** *of G with respect to s, where*

(1) $V_r^s = \{G_1, G_2, \ldots, G_k\}$, the set of the SCCs of the associated OR-graph G,

(2) E_r^s, inc_r^s and $label_r^s$ are defined as follows:

for all $G_i \neq G_j \in V_r^s$ and all nodes $v \in G_i, w \in G_j$, such that $(v, w) \in E$ this edge (v, w) is element of E_r^s and $inc_r^s((v, w)) = (G_i, G_j)$ and $label_r^s((v, w)) = label((v, w))$.

An SCC G_i, that consists of at least two nodes (states) will be said to be a **reduced** *node in G_r^s, otherwise it is a* **simple** *node.*

As an example, consider Figure 5.8 on page 49, and 5.9 on page 51. Figure 7.4 describes the reduced graph G_r^s of digraph G of OR-statechart *S00* (cf. Figure 5.9 and 5.8). *SCC0* is the strongly connected component of the states $S2, S3$ and $S4$ of G, *SCC1* is the strongly connected component of the states $S8$ and $S9$.

Remark 7.7 *The reduced graph G_r^s is acyclic and has a start node $v_{0_r}^s$ which corresponds to the start node v_0 in G.*

Next, we give a criterion which will allow the test paths to be performed on every abstraction level of a given OR-statechart.

Criterion 7.1 *For every component (state) of an OR-statechart Sct and the corresponding abstraction level (of states $S' = children(s)$), choose a set of test paths, such that all transitions (edges) of the corresponding reduced-graph with respect to S' are executed at least once.*

Remark 7.8 Criterion 7.1 corresponds to Definition 7.32 of a path cover for the edges.

[16] cf. Definition 7.30

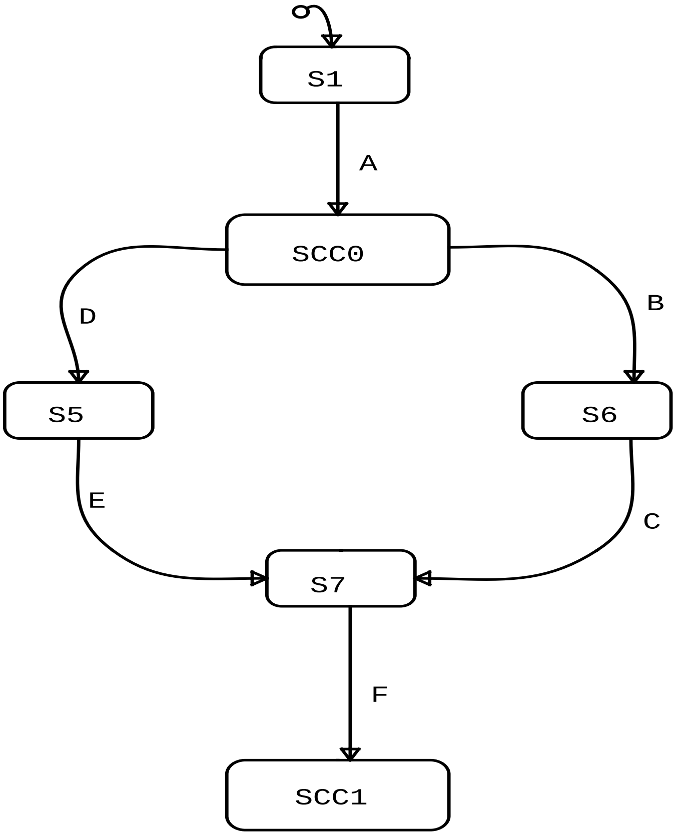


Figure 7.4: The reduced graph G_r^s of digraph G of OR-statechart *S00*

Note:
Note that this criterion is test criterion $C1$ for the super-graph[17] with respect to s (cf. Definition 7.2) The criteria $C2$ and $C3$ of Definition 7.3 and Definition 7.4, respectively, could be applied to the reduced-graphs as well, but will not be considered in the following since the test paths are *not linear* in the number of edges in the worst case.

Proof See Ntafos and Hakimi [116].

7.7.4 Levels of an OR-statechart

In the previous section we described the abstraction level of an OR-statechart with respect to its corresponding reduced graph. In the following, we introduce the general use of the term **level** which will be referred to during the test design Condition 8.1. As an illustration, recall example of Figure 5.4 on page 46. The following levels i are computed for *S0* (by the **hierarchy function** *children*^{i}, cf. Definition 5.3 in Subsection 5.2.1, $i = 0, 1, 2$) :

level 0:
The set of states on level 0 is $children^0(S0)$ = {S0}

[17] In the following, super-graph is used as the general term for reduced graph.

level 1:
The set of states on level 1 is $children^1(S0)$ = {S1, @S2, @S3, S4, S5, @S6}

level 2:
The set of states on level 2 is $children^2(S0)$ = {S21, S22, S23, S31, S32,S33, S34, S61, S62}

Note: We say level $i+1$ is *lower* than i.

7.8 Methods for module concepts

7.8.1 Test design requirement

In Chapter 5 a formal account of syntax and semantics of statecharts was given. Recall that the formal descriptions established in that chapter can be adopted in solving various problems, e. g., caused by transitions in parallel states (nodes), linked transition segments (i. e., labelled arrows that connect states and connectors), inter-level transitions and conflictness between transitions (nondeterminism).

(i) *Transitions in parallel states: AND*-states are transformed into *OR*-states (cf. Subsection 5.4.3).

(ii) *Linked transition segments:* A statechart which consists of linked transition segments is transformed into a special statechart form called CNF which is based on the notion of full compound transitions (cf. Subsection 5.2.5).

(iii) *Inter-level transitions:* Problems caused by inter-level transitions are solved by the formalisation of the notion of scope of transitions (cf. Subsection 5.4.4).

(iv) *Conflictness between transitions (nondeterminism):* The issue of conflicts between transitions is dealt with by applying the the notion of priority (cf. Subsection 5.4.6). In addition, algorithm $\boldsymbol{Generate\_\mathcal{T}_{orth}}$ is used to determine the sets of maximal nonconflicting transitions (cf. Subsection 6.2.1.1).

The establishment of such formal statechart specification guideline contributes to the simplifications of the tests to be designed for the statechart model. For example, the tester can assume that the transitions in the statechart model are 'legal'. Consequently, the number of tests performed by the system designer/tester to examine his/her "System Under Development" (SUD) specification are reduced. Moreover, the tests may not be as complicated as would be the case if we had to consider 'illegal' transitions as well.

In addition, being able to simulate the statechart model which is based on the formal step semantics (cf. Chapter 6) gives the system tester the ability to make certain assumptions which in the following will be referred to as the *test condition/requirement.*

Test condition 7.1
The **test condition** *will assume that*

(1) all transitions are modelled/implemented in conformance with the statechart specification guideline, i. e., triggers and actions are 'legal'.

(2) there is a slow environment and no livelocks (cf. Chapter 1 pp. 1ff. and Chapter 6).

(3) every transition is triggered by an input which does not depend on any external events and conditions (cf. Section 8.2 Definition 8.1).

(4) every transition is triggered by an input which does not depend on a lower level (cf. Subsection 7.7.4).

The idea behind the test condition is to simplify and thus also improve the effectiveness of the testing process. Generally, the static tests and sophisticated methods offered by STATEMATE for validating the model's behaviour against a specific scenario by simulation require an extremely large number of tests. In our case, the number of tests will become manageable due to the fact that only legal transitions will be considered. More specifically, the concepts and methods proposed in this thesis assume the existence of the algorithms developed in Chapter 6, e. g., *Generate_$\mathcal{T}_{orth}$*[18] and *System_Step*[19], and therefore ignore such aspects like *nondeterminism, detection of deadlock, reachability of conditions* and *usage of transitions.*

7.8.2 Algorithms for the generation of test paths

In the following, we present methods or heuristics to generate test paths that satisfy the given Criterion 7.1 on page 147. We solve the path generation problem by the algorithm *Generate_Paths_inG* which has the following interface.

Algorithm 7.1 Generate_Paths_inG
Input*: digraph* $G = (V, v_0, E, inc, label)$ *which corresponds to the statechart* $Sct = (S, r, children, type, \mathcal{H}, hist, default, \mathcal{T})$.
Output*: A set of test paths* $P = \{p_1, p_2, \ldots, p_n\}$*, that satisfy the given Criterion 7.1, where each test path* p_i, $0 < i \leq n$*, is a sequence of edges.*

The path generation problem can hereby be divided into three major parts, namely

I. determining test paths in the reduced graph G_r^s (cf. Definition 7.34 page 147) which consist of the *strongly connected components SCCs* (cf. Definition 7.30 page 145) of the original graph G,

II. determining test paths in each strongly connected component *SCC* and finally

III. use the test paths of part I and II to compose (we say 'merge') final test paths.

[18] cf. Subsubsection 6.2.1.1, page 99

[19] cf. Definition 6.6, page 101

7.8.2.1 I. Determining test paths in the reduced graph G_r^s

The fact that G_r^s is acyclic reduces the problem of determining paths in G to determining paths in a simpler structure. Therefore, one can heuristically determine the paths in G_r^s by using the following steps.

(1.) Find the topological sorting of an acyclic digraph G_r^s.

(2.) Find the paths of the topologically sorted graph $G_{r_{TOP}}^s$ which satisfy the given Criteria 7.1.

Idea of step (1.):
The idea behind step (1.) is to seek a linear ordering of the vertices $[v_1, v_2, \dots, v_n]$ which is consistent with the edges of G_r^s; i.e.,

$$(v_i, v_j) \in E_r^s \Rightarrow i < j \ (\forall i, j). \tag{7.4}$$

Idea of step (2.):
Let a linear ordering of the vertices $v_l = [v_1, v_2, \dots, v_n]$ as in step (1.) above be given. Let $G_{r_{TOP}}^s$ be a topologically sorted graph. Then the test paths of $G_{r_{TOP}}^s$ are determined by using the following heuristic:

Beginning with the last vertex of the topologically sorted graph $G_{r_{TOP}}^s$, build for each incoming edge a path. Each of these paths will be inserted in the empty list L_p. The next node to be considered is the one which lies before the lastly used node, with respect to the topological sorting. For each such vertex with the exception of the start node s_r, the number of elements in the list L_p will be compared with the indegree of the vertex whereby there are three cases:

1) *If the indegree equals the number of elements then each incoming edge will be appended to a test path in the list whereby each list element is used only once.*

2) *If the indegree is less than the number of elements then first it will be proceeded as in case* 1) *above. For the rest of list elements, the DIJKSTRA algorithm will be used to compute the shortest path to the start-node s_r and this path will be appended to the not yet used list element of this step.*

3) *If the indegree is greater than the number of elements then for each list element proceed as in case* 1) *above. The remaining incoming edges will be inserted as new test paths in the list.*

The detailed procedure is described in **Generate_Paths_inReducedGraph**, Algorithm 7.2 (cf. Section 7.9, Subsection 7.9.1).

To illustrate part I above, the example given by Figure 7.5 is chosen. The graph G of Figure 7.5 is acyclic. That is, the reduced graph G_r^s is the same as its original graph G. After the topological sorting of G_r^s, the Algorithm 7.2 examines all nodes of $G_{r_{TOP}}^s$ and builds the following paths:

$$\begin{aligned}\text{path1} &= D\ H\ J\\ \text{path2} &= A\ G\ I\ L\\ \text{path3} &= B\ E\ F\ M\ K\\ \text{path4} &= C\ G\ N\end{aligned}$$

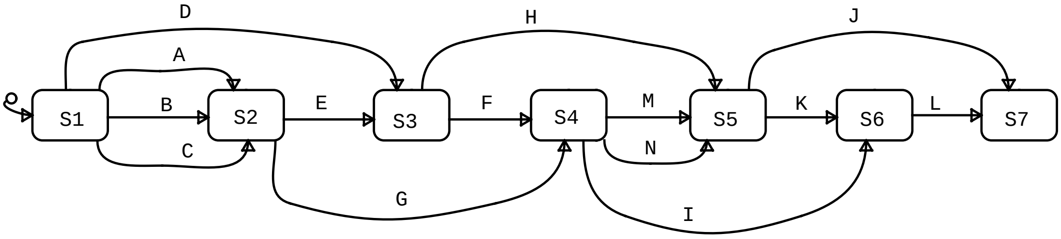


Figure 7.5: An example graph G to illustrate the method described in Algorithm 7.2

Remark: The paths above are final. Since the reduced graph G^s_r is the same as the original graph G, it follows from Definition 7.34 that there are no reduced nodes in $G^s_{r_{TOP}}$. Therefore, no further steps are required.

7.8.2.2 II. Determining test paths in the *SCCs*

For every test path derived from part I, the following will be considered:

a) *For each non-basic component SCC_j in the given path p, do find the shortest path p_s in SCC_j (in G !!) to its immediate successor in the test path p. At the beginning, all edges of SCC_j are marked unvisited. If the edge (v, v) exists for the current node v, insert (v, v) in p_s. Mark all edges in p_s which are in SCC_j as visited. Assign to the test path in the SCC_j $pathscc := p_s$.*

b) *Beginning with the last edge of the test path p_s, a backtracking procedure will be carried out. For each non-visited edge of the SCC_j, modified* **depth first search** *(dfs) will be used. The edges which are traversed by the dfs will be marked visited. As soon as a node which is has no more unvisited edges is found by the dfs, all edges marked by dfs build a path subpath. subpath will always be appended to the end of pathscc. If the last node (of last edge) in pathscc is not equal to the first node in subpath then the shortest path p_m between these two nodes is to be built and p_m is to be appended to pathscc and then subpath. The backtracking using dfs will be repeated until all remaining edges of the SCC_j are covered.*

For a detailed description see algorithm **Generate_Path_inSCC** (Section 7.9, Subsection 7.9.2).

Remark 7.9 *The advantage of using the shortest path p_m when subpath is appended to pathscc is that the result of the final pathscc is always the shortest possible test path in SCC_j (in G !!). Another optimization of the procedure is that when the computation of the shortest path p_m is undertaken, unvisited edges will always if available be selected first.*

Consider graph *G1* in Figure 7.6 and its reduced graph $G1^s_r$ in Figure 7.7. Both figures will be used to demonstrate algorithm **Generate_Path_inSCC**. After the topological sorting of $G1^s_r$, the Algorithm 7.2 examines all nodes of $G1^s_{r_{TOP}}$[20] and builds the following paths:

$$\begin{aligned} \text{path1} &= \quad A\ C\ H \\ \text{path2} &= \quad A\ D \end{aligned}$$

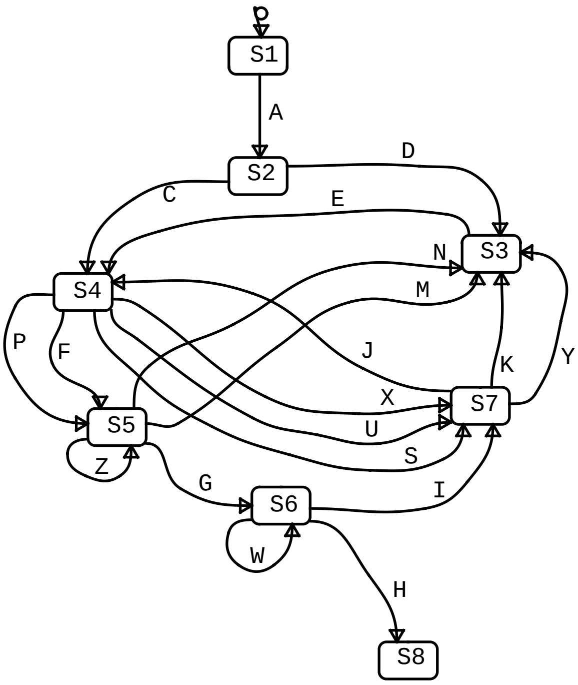


Figure 7.6: Graph *G1*.

Now since path1 consists of a non-basic component *SCC*_0, the shortest path through *SCC*_0 (in $G1$!!) to its immediate successor in the path (i. e., in $G1^s_r$) must be determined. The shortest path p_s with respect to $G1$ (see Figure 7.6) is: $F\ Z\ G$. Note that the loop Z is inserted in shortest path p_s. The insertion of the loops at this point contributes to the efficiency of the algorithm. Obviously, traversing such loops later would involve unnecessary paths to find them.

The final path (in *SCC*_0) of part *II* with respect to $G1$ (see Figure 7.6) is:

$$\underbrace{F\ Z\ G}_{\mathbf{p_s}}\ \ \underbrace{W\ I\ J\ P\ M\ E\ X\ Y}_{\mathbf{1.subpath}}\ \ \underset{\mathbf{2.subpath}}{\overbrace{\underbrace{E\ S}_{\mathbf{p_m}}\ \ \underbrace{K}}^{\mathbf{further-path}}}\ \ \underset{\mathbf{3.subpath}}{\overbrace{\underbrace{E}_{\mathbf{p_m}}\ \ \underbrace{U}}^{\mathbf{further-path}}}\ \ \underset{\mathbf{4.subpath}}{\overbrace{\underbrace{J\ F}_{\mathbf{p_m}}\ \ \underbrace{N}}^{\mathbf{further-path}}}\ \ E\ F\ G$$

[20] Note that the topologically sorted digraph $G^s_{r_{TOP}}$ is exactly the same as one in Figure 7.7

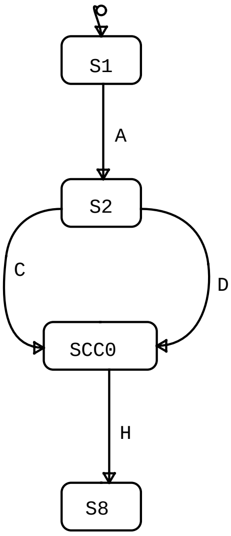


Figure 7.7: The reduced graph $G1^s_r$ of digraph $G1$.

7.8.2.3 III. Using the test paths of part I and II to merge final test paths.

If the topologically sorted graph $G^s_{r_{TOP}}$ consists of only simple nodes (basic nodes) (cf. Defition 7.34), then the paths derived from $G_{r_{TOP}}$ by **Generate_Paths_inReducedGraph**, Algorithm 7.2 are final, i.e., output paths. Otherwise the paths derived from part I are to be extended by ones generated by the heuristic of part II.

Coming back to the examples of Figure 7.6 and figure 7.7 above, the final (output) test paths after the insertion of the path in SCC_0 in path1 derived from part I are:

$$\text{path1} = \ \mathbf{A}\ \mathbf{C}\ F\ Z\ G\ W\ I\ J\ P\ M\ E\ X\ Y\ E\ S\ K\ E\ U\ J\ F\ N\ E\ F\ G\ \mathbf{H}$$

$$\text{path2} = \ \mathbf{A}\ \mathbf{D}$$

7.9 Formal Algorithms

In this Section 7.9 two formal algorithms that are informally desribed in Subsection 7.8.2 are presented.

7.9.1 Generation of test paths in a topologically sorted graph $G_{r_{TOP}}$

Algorithm 7.2 Generate_Paths_inReducedGraph

Input: Topologically sorted graph $G_{r_{TOP}}$

Output: A set of paths $P = \{p_1, p_2, \ldots, p_n\}$*, where* p_i*,* $0 < i \leq n$*, is a sequence of edges that satisfy the given criteria 7.1 [cf. section 7.7.3].*

```
$m := 0$; /* # of the current paths */
$last(\{e_1, e_2, \ldots, e_n\}) := e_n$ /* last element of list */
$DIJKSTRA(G_{r_{TOP}}, cost, dist, pred)$
$dist : V \rightarrow I\!N$
$|V|$ Elements $\rightarrow dist(0), \ldots, dist(|V|)$


FOR $i := n$ DOWNTO 2 DO
        $v := ord(i)$;
        $Q := \emptyset$; $q := 0$;
        /* $out_k(v) \hat{=} kth$ edge which leaves vertex $v$, $1 \leq k \leq outdeg(v)$ */
        /* $in_k(v) \hat{=} kth$ edge which enters vertex $v$, $1 \leq k \leq indeg(v)$ */
        /* Determination of the paths whose last element intersects with one of the out-
            degree edges of vertex $v$. */
            FOR $j := 1$ TO $m$ DO
              FOR $k := 1$ TO $outdeg(v)$ DO
                IF $last(p_j) = out_k(v)$ THEN
                  $q := q + 1$; $Q_q := p_j$; $k := outdeg(v)$;
                FI
              OD
            OD /* $Q = \{Q_1, \ldots, Q_q\}$ */
        I.  IF $|Q| = indeg(v)$ THEN
              FOR $j := 1$ TO $|Q|$ DO
                $Q_j \leftarrow Q_j \circ in_j(v)$;
                $p_l \leftarrow Q_j$
                /* $\exists l \in \{1, \ldots, m\}$ */
              OD
            FI
```

```
II.  IF $|Q| > indeg(v)$ THEN
       FOR $j := 1$ TO $indeg(v)$ DO
         $Q_j \leftarrow Q_j \circ in_j(v)$;
         $p_l \leftarrow Q_j$; /* $\exists l \in \{1, \ldots, m\}$ */
       OD
       FOR $j = indeg(v) + 1$ TO $|Q|$ DO
         /* For the rest of paths from Q, indegree edges of vertex v should be used
         more than once. Determine the shortest path to vertex v from the start-
         node. DIJKSTRA algorithm computes the predecessor node of v. The
         index of pred(v) will then be used in the calculation of $p_l$. */
         $z \in \{w | w = pred(v) \wedge \forall w' \in pred(v) : dist(w) \leq dist(w')\}$;
         $y := index(z)$;
         $Q_j \leftarrow Q_j \circ in_y(v)$;
         $p_l \leftarrow Q_j$; /* $\exists l \in \{1, \ldots, m\}$ */
       OD
     FI
III. IF $|Q| < indeg(v)$ THEN
       FOR $j := 1$ TO $|Q|$ DO
         $Q_j \leftarrow Q_j \circ in_j(v)$;
         /* $\exists l \in \{1, \ldots, m\}$ */
         $p_l \leftarrow Q_j$; /* build pairs of the first edges */
       OD
       FOR $j = 1$ TO $indeg(v) - |Q|$ DO
         $p_{m+j} \leftarrow in_j(v)$;
       OD
       $m := m + indeg(v) - |Q|$;
     FI
   OD
```

7.9.2 Generation of test paths in a strongly connected component

Algorithm 7.3 Generate_Path_inSCC

Input: G_j, the graph representing SCC_j

Output: A path p', which traverses all edges of SCC_j and satisfies the given criteria 7.1 [cf. section 7.7.3]. p' is a sequence of edges.

```
$subpath, pathscc\ :\ LIST\_OF\_EDGES$;
$start\_node : VERTEX$;
$subpath\ :=\ \emptyset$;
$pathscc\ :=\ p_s$;
(* $p_s$ is the shortest path, *)
(* which has been computed by **Step** II(a) above. *)
$start\_node\ :=\ v_{end}$;
$MOD\_DFS(start\_node, subpath, pathscc)$;

PROCEDURE $MOD\_DFS(v : VERTEX,$
        $subpath\ :\ LIST\_OF\_EDGES,$
        $pathscc\ :\ LIST\_OF\_EDGES)$;
VAR
$w, v_1, v_2 : VERTEX$;
$e, found\_edge : EDGE$;
BEGIN
   FORALL $w \in succ(v)$
   IF $mark[v, w] = unvisited$ THEN
   BEGIN
            $mark[v, w] := visited$;
            $subpath := subpath.append(v, w)$;
            $MOD\_DFS(w, subpath, pathscc)$;
   END IF;
   ELSE
   (* end of subpath *)
   (* insert subpath in pathscc *)
   BEGIN
   IF $(subpath \neq \emptyset)$ THEN
   BEGIN
      (* last node in pathscc *)
      $v_1\ :=\ target(pathscc.last())$;
      (* first node in the subpath *)
      $v_2\ :=\ source(subpath.first())$;
      IF $(v_1 \neq v_2)$ THEN
      BEGIN
         (* join $v_1$ with $v_2$ using DIJKSTRA *)
         (* append the found path to pathscc *)
         (* mark all appended edges as visited *)
```

```
            $BUILD\_SHORTEST\_PATH(subpath, v_1, v_2)$;
         END IF;
         FOR $found\_edge \in subpath$ DO
         BEGIN
               $pathscc.append(found\_edge)$;
         END FOR;
      END IF;
      END ELSE;
      $subpath.clear()$;
      END FOR;
   END $MOD\_DFS$;


PROCEDURE $BUILD\_SHORTEST\_PATH$
($p_s$ : $LIST\_OF\_EDGES, start, endnode$ : $VERTEX$);
VAR
   $edge\_new, e : EDGE$;
   (* variables for $DIJKSTRA$ *)
   $cost[G_j, 1] : int$;
   $dist[v, w] : int$;
   $pred[v] : EDGE$;
BEGIN
   (* computation of shortest distance *)
   $DIJKSTRA(G_j, start, cost, dist, pred)$;
   WHILE($endnode \neq start$) DO
   BEGIN
      $edge\_new := pred[endnode]$;
      $endnode := source(pred[endnode])$;
      (* check for unvisited edges first *)
      (* if no unvisited edge exists *)
      (* then take first edge in the list *)
      (* in case an unvisited edge is explored *)
      (* then mark this edge as visited *)
      FOR $e \in adjacent\_edges[endnode]$ DO
      BEGIN
            IF $mark[e] = unvisited$ THEN
            BEGIN
               $edge\_new := label(e)$;
               $mark[edge\_new] := visited$;
            END IF;
            $p_s := p_s.push(edge\_new)$;
      END FOR;
   END WHILE;
END $BUILD\_SHORTEST\_PATH$;
```

Chapter 8

Integration Test Approach

In the previous chapter we developed concepts and methods which can be used to generate test paths for modules of hierarchical statechart models. The remaining task of this thesis is to establish concepts and methods which can aid the testing of the interacting processes (modules) of the whole system. Therefore, this Chapter 8, is devoted to development of such methods which will also be called the *integration test approach*. The concepts for *integration test* should mainly exploit the module tests, i.e., to determine which of the established module test cases may be involved in the tests required for the particular module interfaces.

When testing large and complex system models, the individual components (modules) are usually tested in isolation during module testing. Then the interfaces of the modules are tested during one or more *integration steps* (cf. [60]). Each integration step requires that actual modules replace stubs one at a time, and that a 'basis' set of test paths are exercised through each module as it is added to the system. This test strategy is called *incremental integration test* and is described in Subsection 8.1.

8.1 Incremental integration test

In the following, it is assumed that every component has been tested in isolation and proven to be a correct refinement of its associated specification. The objective of integration testing is to organise the overall testing effort by explicitly seperating the testing of structure within a component (module) from the testing of the module interactions. To achieve this seperation, the integration of components is divided into several phases, an initial unit testing phase and several integration steps. During each step, one component is selected for integration according to an integration strategy, such as bottom-up or top-down integration (cf. [108]).

Integration testing is performed to specifically test the modules interface. The idea of the *integration test strategy* will be based on one of the well known heuristic approaches from practical testing:

I. Perform detailed tests for individual components (processes). This test corresponds to the *module test*[1] which has to be extended by test cases which do not obey the condition (3) of Test condition 7.1 on page 149.

II. Then test the proper integration of an increasing number of components, adding them one by one until all components are integrated. This test is called the *incremental integration test.*

8.1.1 Example of interacting components

Consider Figure 8.1 which depicts a statechart that describes the behaviour of the telefax machine *BEAUTY* in terms of its human interface. Figure 8.1 above shows a typical example of interacting components. The components (states) *SOURCE* and *SERVICE* are said to interact with each other. Following step *I* of the above described integration test strategy, the components *SOURCE* and *SERVICE* will first be tested in isolation, respectively. In other words, the transitions which violate the test conditions (3) and (4) are considered. The extended module test (step *I*), on the other hand, requires that the interface of the second module must be simulated using a stub. For the module test of *SOURCE* e. g., a stub for *SERVICE* should be implemented in order to simulate the behaviour of the interface. The aim of this phase is mainly to show whether the interface of modules is correct.

8.1.2 Drawbacks of the incremental integration test approach

An *incremental integration test* should 'drive' each module through a full 'basis' set of test paths in an integration context. Unfortunately, this is often practically impossible due to intermodule control coupling, i. e., incremental execution at each integration step may be costly since the number of scenarios might be very high.

Depending on the complexity of the statechart components, certain scenarios which were not represented by the stubs (because it would have been very complicated, to implement such scenarios in the stubs), must be tested elsewhere. These tests can be very complicated to build. In particular, for time dependent variables, it is necessary to know which transition ($event[condition]/action$) is taken and when it is taken. In other words, the incremental integration test described above is neither in a position to handle time dependent variables efficiently, nor to support tests which involve such variables in a reasonable way. On the other hand, exhaustively re-executing all 'basis' sets of test paths involve portions that may have no relevance for the current integration step because they may be already tested in the module test.

To address the drawbacks above, we will modify the *incremental integration* technique, in such a way that only transitions affecting the module calling relationships need to be considered for integration testing. This approach will be called the *goal oriented integration analysis*, and will be described in the Subsection 8.2.

[1] cf. Chapter 7

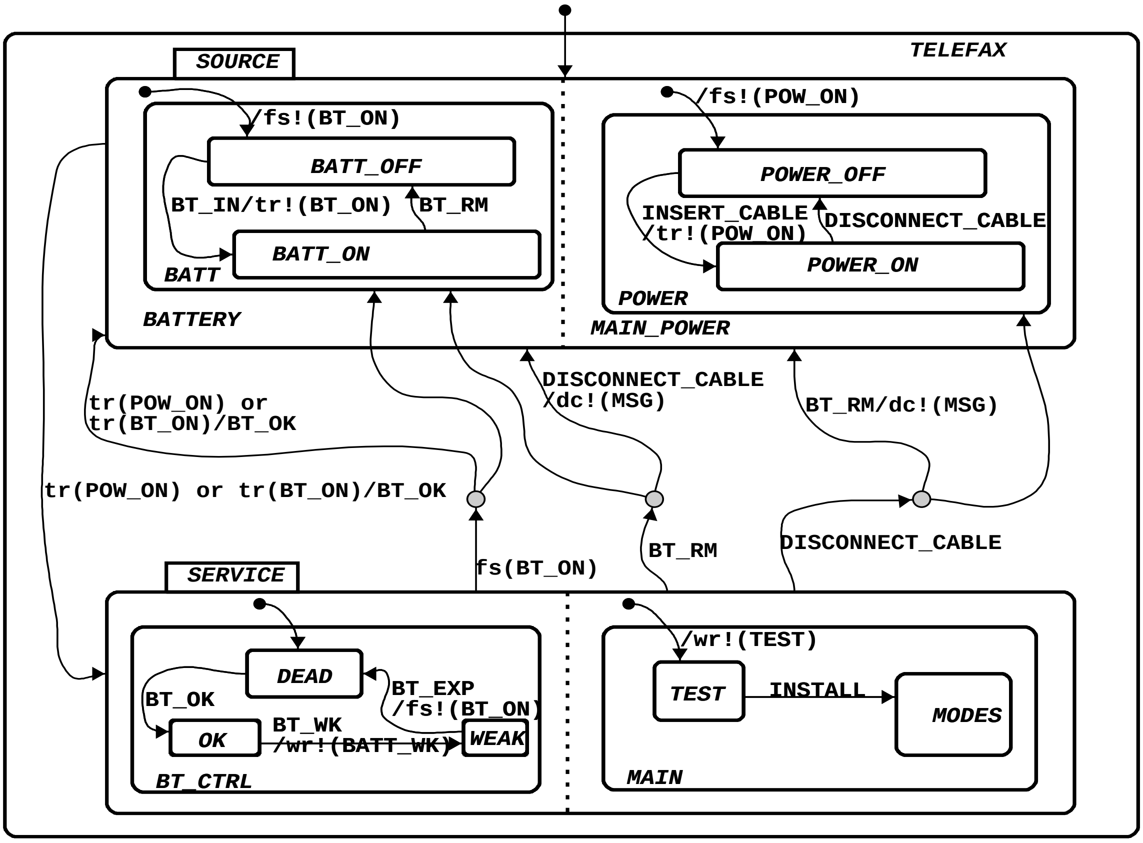


Figure 8.1: The statechart for the telefax machine *BEAUTY*

8.1.3 Integration test concepts

In this subsection the development of *integration test concepts* for hierarchical statecharts models is undertaken. The module test Criterion 7.1 suggested for performing test paths on every level of a given statechart will be extended by another Criterion 8.1 called 'integration test criterion', to exercise test paths between modules which rely on internal events and conditions which were not considered so far due to test conditions (4) on page 149.

Criterion 8.1 *For the submission of test paths between modules, it must be observed that for execution of the external transitions of the modules, eventually internal transitions must have to take place* (*see also [85, 86, 87]*).

The integration test Criterion 8.1 differs from the module test Criterion 7.1, in that, the generation of test paths based on Criterion 8.1 must consider a subtle issue which requires that execution of the external transitions between the modules take in account internal transitions (variables) whenever there exists a dependency. Indeed, this issue appears to be the most challenging problem in the development of integration test concepts.

Now, based on the incremental integration stategy, we establish the *integration test methods* on the following concepts:
(1) *Each orthogonal state* (*module*) *will be tested in isolation* by application of module

test methods developed in Section 7.8 and Section 5.4.3 (Transforming *AND*-states into *OR*-states).

Example 1
Consider Figure 8.2 which depicts an abstract form[2] of the statechart *BEAUTY* (Figure 8.1). The orthogonal states *SA* and *SB* in Figure 8.2 which correspond to the orthogonal states *SOURCE* and *SERVICE* in Figure 8.1 are to be tested in isolation, respectively. The orthogonal components *@SA1* (= *S*1) and *@SA2* (= *S*2) in Figure 8.2 will be represented by the automata product $S1 \times S2$ (in Figure 8.3). Similarly, components *@SB1* (= *S*3) and *SB21* (= *S*4) and *@SB22* (= *S*5) are represented in Figure 8.2 or Figure 8.3, respectively.

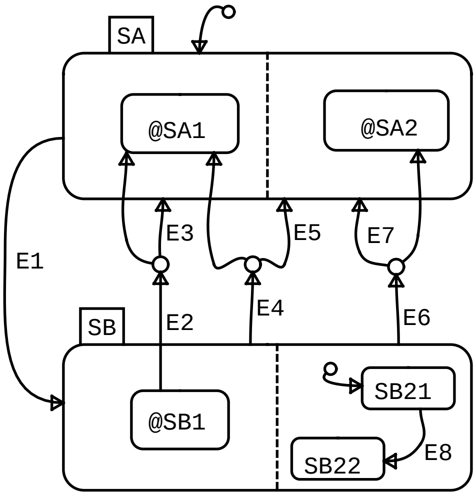


Figure 8.2: Representing the statechart *BEAUTY* (Figure 8.1) in a rather more abstract form

(2) *The interacting modules (components) will then be considered reduced (reduced graph) [cf. Definition 7.34]. As reducing methods, methods similar to the module concepts can be used. Nevertheless, it must be observed that, for AND-states which interact and build a strongly connected component, no clustering should be undertaken. This restriction is necessary, otherwise the concept of integration test cannot be applied.*

Example 2
Figure 8.3 depicts the reduced OR-graph of Figure 8.2. The modules (components) *AP*1 and *AP*2 build a strongly connected component, but are nevertheless to be handled as seperate states.

(3) *Submit all edges (test paths) between modules (reduced graph) by application of algorithm* **Generate_Paths_inReducedGraph** (cf. Section 7.9, Subsection 7.9.1) **and algorithm Generate_Path_inSCC** (cf. Section 7.9, Subsection 7.9.2). Remark: The interacting modules are assumed to be tested, and hence are now treated as black boxes.

[2] Figure 8.1 is simplified by Figure 8.2 for demonstration purposes.

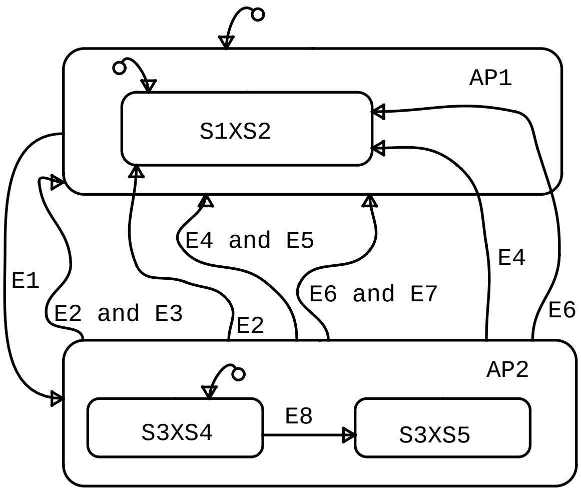


Figure 8.3: Reduced OR-graph of Figure 8.2

Example 3
Test paths[3] between orthogonal states $AP1$ and $AP2$ (of Figure 8.3) which correspond to SA and SB (cf. Figure 8.2):

$$\begin{array}{lclcl}
TS1 & : & \text{AP1} & \xrightarrow{E1} & \text{AP2} \\
TS2 & : & \text{AP2} & \xrightarrow{E2\ and\ E3} & \text{AP1} \\
TS3 & : & \text{AP2} & \xrightarrow{E2} & \text{AP1} \\
TS4 & : & \text{AP2} & \xrightarrow{E4\ and\ E5} & \text{AP1} \\
TS5 & : & \text{AP2} & \xrightarrow{E6\ and\ E7} & \text{AP1} \\
TS6 & : & \text{AP2} & \xrightarrow{E4} & \text{AP1} \\
TS7 & : & \text{AP2} & \xrightarrow{E6} & \text{AP1}
\end{array}$$

Note the difference between the representation of transition $E2$ in Figure 8.2 and that of Figure 8.3. The latter is a further simplification of the reduced node $AP2$ (cf. Figure 8.3) representing the orthogonal state SB (cf. Figure 8.2). The simplification is achieved by letting the outcoming transition $E2$ from the hierarchy state $@SB1$ to begin from the contour of SB. In this particular case, transition $E2$ can be rewritten as $E2[in(SB1)]$. But how can the states to be entered by the transition $E2$ or $E4$ or $E6$ be traced in Figure 8.3? To solve this problem, the orthogonal components in an orthogonal state can be implemented as a list of automata product. Figure 8.4 shows the implementation of the automata products as lists. List L_1 represents the automata product $\mathit{S1} \times \mathit{S2}$ in $AP1$ (cf. Figure 8.3). The list elements $A_{1,1}$, $A_{1,2}$ correspond to automata *@SA1* and *@SA2*, respectively. Now since the statechart semantics and STATEMATE specification allow each orthogonal component to consist of a default state or an init mechanism, the main idea behind holding lists is that when the transition, such as $E2$ is taken then the default state of list element $A_{1,1}$ of L_1 is entered. Taking the transition ($E2$ and $E3$) expresses the entering of all default states of L_1.

[3] transitions

Observe that, for the integration test for Figure 8.3 we consider only the transitions $E1$ to $E7$ between orthogonal states $AP1$ and $AP2$. However, the methods applied so far only satisfy Criterion 7.1, and do not consider Criterion 8.1. Therefore, we extend the concepts stated in part (3) as follows:

(a) *For all edges (transitions, the so called* critical transitions *[cf. Definition 8.1]) which require*[4] *additional* internal *transitions which violate condition* (4) *of Test condition 7.1 on page 149, methods for* goal oriented integration analysis *have to be applied.* (see Section 8.2).

(b) *For time dependent variables, methods for* analysis of time dependent behaviour *have to be applied* (see Section 8.2).

8.2 Goal oriented analysis

In particular, when testing for conditional variables (signals) in participating modules or time dependent behaviours of processes one requires more "economical and intelligent" tests. In this subsection methods for analysing additional *internal* transitions will be developed. These methods will be called *goal oriented integration analysis.* The idea of the goal oriented integration analysis is that only regions of interest undergo a special (critical) test.

The following terms are fundamental for the concept of goal oriented integration analysis. Now since the integration test is principally performed to test the modules interface, in the next definition the *critical transitions* will be restricted to transitions between two interacting components.

[4] the values of which depend on some *internal* variables

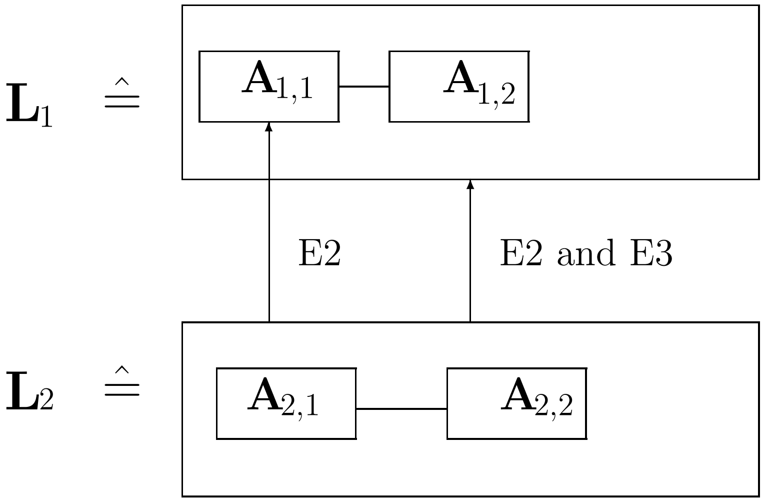


Figure 8.4: Implementing automata products as lists

Definition 8.1 *For two arbitrary components (states) $S_i, S_j \in S : \ type(S_i) = AND \wedge type(S_j) \in \{OR, AND\}$, if $t \in \mathcal{T}$ where $t = \ (X, l, Y)$ is a transition between S_i and S_j, i. e. $X \subseteq S_i$ and $Y \subseteq S_j$, then t is called a* **critical transition** *(or* **critical point***) if its label l consists of a condition expression SV_{expr} having at least one internal*[5] *condition variable (signal) $SV \in SV_{expr}$ that must be set in component S_i or S_j before the transition t is enabled (cf. Definition 6.5). Such a condition variable is called a* **critical variable** *and is denoted by SV_{cr_pt}.*

Notation*: A critical transition will be denoted by t_{cr_pt}. The set of all critical transitions will denoted by $\mathcal{T}_{cr\_pt}$. Finally, the set of all critical variables will denoted by $\mathcal{SV}_{cr\_pt}$.*

In Figure 8.1 the transition from *SOURCE* to *SERVICE* with the label $tr((BT\_ON)$ or $tr((POW\_ON))$, e. g., is a critical transition because it requires that the variables *BT_ON* or *POW_ON* are true, whose values, however, are assigned in component *SOURCE*. Consequently, the variables *BT_ON* and *POW_ON* are critical variables. By application of symbolic evaluation in the substates *BATT* and *POWER*, it can be computed that — starting with the initial states — the transitions $BATT\_OFF \overset{BT\_IN/tr!(BT\_ON)}{\longrightarrow} BATT\_ON$ and $POWER\_OFF \overset{INSERT\_CABLE/tr!(POW\_ON)}{\longrightarrow} POWER\_ON$ must take place before the (external) critical transition with the label $(tr(BT\_ON)$ or $tr(POW\_ON))$ is enabled.

On the other hand, the transitions from *SOURCE* to *SERVICE* with the label *DISCON_CBL* or with the label *BT_RM*, respectively, e. g., are not critical transitions. Obviously, these are always enabled as soon as the events *DISCON_CBL*, *BT_RM*, respectively, occur. In other words, they require no internal variables (signals) from *SOURCE*.

[5] Techniques from symbolic evaluation methods (cf. [143]) can be used in this case to compute the required internal signals.

Chapter 9

Summary of Contributions, Concluding Remarks and Open Problems

9.1 Summary of contributions

The motivation of this thesis was that though many formal methods have been proposed to address the problem of specifying and modelling reactive systems, the *test problem/verification problem* remains difficult: (1) the interaction between the components, and (2) testing a *complete* system, in many cases, leads to a "space explosion" problem. The aim of this thesis has been, therefore,

(i) to develop a *module test* which allows the system designer/tester to (automatically) test the partially designed and implemented system.

(ii) to develop concepts and methods which aid the testing of the *interacting* processes (modules) of the whole system.

In this section we give the particular research contributions to this thesis. All achievements were discussed in the preceeding chapters, but it is helpful to give a short listing of the most important ones. We used the language of *statecharts* as a framework for the development of the required methods and concepts.

9.1.1 Contributions to a statechart standard specification guideline

One of the criticisms of statechart specifications, however, is that there are many variants of syntax or semantics of statecharts proposed in the literature (cf. Section 5.3). In other words, there is no kind of standard specification guideline on which the test concepts and test methods, developed in this dissertation could be based. Another criticism is that although the original paper [51] of Harel, Pnueli, Schmidt and Sherman presents a formal syntax and semantics of statecharts, a number of issues, e. g., transitions in parallel states, hierarchy in states, etc. are not accurately defined. Other features, such as *inter-level transitions* and conflictness between transitions are not defined formally. Later, many of these problems are discussed by Harel and Naamad in their report [54] of the recent STATEMATE semantics of statecharts. Nevertheless, this report is *informal*, and thus does not provide precise definitions. Therefore, in Chapter 5 we established some kind of *statecharts standard specification guideline* which formally defines and restricts syntax and semantics of variants of statecharts. The basic idea behind such specifications is that we would like to construct a statechart in a way that it consists of only *OR*-states, i. e., *AND*-states are not allowed. This is called an *OR-statechart* and has the advantage that it can easily be represented as an OR-graph (cf. Subsection 5.2.4). The derived *OR*-graph will be required for the development of simple and efficient algorithms which are used to generate test paths in a statechart model (cf. Chapter 7). Moreover, the restriction above imposed to the statechart language is not a restriction of the modelling power of statecharts because every *AND*-state can be represented as an *OR*-state (as demonstrated in Section 5.4.3). Last but not least the proposed test of OR-statechart is more thorough than would be corresponding isolated tests of the components of the *AND*-state (i. e., *orthogonal statechart*).

The concepts which we used in the development of the statecharts standard specification guideline are summarised as follows:

(1) We defined subclasses of statecharts to simplify the test selection problem. In particular, we defined a *simple statechart* (cf. Subsection 5.2.3.1) over which all further statechart classes are defined. These definitions are used for the exploration of the relationship between a formal statechart and a rooted labelled directed graph (cf. Definition 5.18). We require the graphical definitions for the development of methods and concepts to overcome the test problems. A summary of the results of this exploration is depicted by Figure 5.12 (cf. page 57).

(2) A statechart which consists of linked transition segments is transformed[1] into a special statechart form called *Canonical Normal Form CNF* which is based on the notion of full compound transitions (cf. Subsection 5.2.5). This transformation is necessary to simplify the test selection problem and to preclude 'incomplete' transitions[2] (see the case of simulating initial compound transitions described on page 53).

(3) Next we defined a *complex statechart* in order to compute transitions in parallel states (*AND*-states). A complex situation arises when there is a direct exit of an *AND*-state because the source set contains more than one state. Thus to define a legal transition, we need to define the 'maximality' of the exited (source) set. We formalised the concept of *legal configurations* which was only informally defined in [54] to describe precisely the exited set and entered (target) set of a transition.

[1] without changing the semantics of the statechart

[2] The term incomplete is used in a context in which the execution of a transition is taken into account.

(4) We established techniques which can be used to transform *AND*-states into *OR*-states. This kind of transformation corresponds to constructing the automata product of an orthogonal statechart. This automata product is referred to as the *modified automata product* because it is different from the conventional automata product (cf. Definitions 5.36 and 5.37). Our main goal is to achieve an OR-graph which represents the *OR*-states derived from the *AND*-states. The results due to this transformation are summarised by Theorem 5.4.

(5) Statecharts allow *inter-level* transitions. The term *inter-level transitions* is used in the literature to refer to transitions that cross the borderlines of hierarchy states. As supported by STATEMATE, this mechanism can be seen as a 'goto' method which allows an arbitrary movement of control across the state hierarchy. The main question was how to accurately define the states that have been exited and those that are entered by taking the transition. Essentially, one has to know which *non-basic* states are exited and entered in the process of taking the transition. In this thesis we have formalised the notion of scope of transitions (which was only informally defined in [54]) to deal with interlevel transitions. In addition, we have suggested a restriction (cf. Restriction 7.1) as a possible solution to respect hierarchy states.

(6) One of the most delicate issues are conflicts between transitions. Informally, two transitions are in *conflict* if there is some common state that would be exited if any one of them were to be taken. In particular, conflicting transitions cannot be taken in the same "step". We defined this notion in two ways, *with respect to a configuration* or *independent of a configuration* (cf. Definition 5.44). We formalised the notion of priority (which was only informally defined in [54]) to deal with conflicts between transitions. In addition, we developed algorithm $Generate\_\mathcal{T}_{orth}$ to determine the sets of maximal nonconflicting transitions (cf. Subsection 6.2.1.1).

The establishment of such formal statechart specification guideline has the following advantages:

(i) Through the syntax restrictions and specification transformations which transform complex statecharts into simpler form, we are able to derive a suitable form for a profound testing ("design for testability").

(ii) As a consequence of (*i*), the tests to be designed for the statechart model become simple since we can make assumptions like: *all transitions are modelled/implemented in conformance with the statechart specification guideline, i. e., triggers and actions are 'legal'.*

The next major issue was to clarify the process of test execution and simulation of statecharts, the so called *step semantics* of statecharts. So far, we have developed the concept of the legal transitions. We have described the simple OR-graph product (constructed by Definition 5.36 and Definition 5.37) which can be used in the generation of test paths, which allow the system designer/tester to make a systematic analysis of, e. g., detecting nondeterminism (cf. Illustration of Case (3)(a)ii. of Definition 5.36, pp. 77ff.). These test paths of a simple OR-graph product, however, are not enough for the detection of the other system flaws like deadlocks because they do not consider the "enabledness" of the transitions. In other words, each transition delivered by the simple OR-graph product is legal (cf. Theorem 5.4) but not necessarily 'executable'. For this case, we had to formalise the statechart behaviour in terms of transitions "which execute a step", i. e., describing a step formally, with all its ramifications and side effects. In particular, we devoted Chapter 6 to describing two basic models of execution of a step. The first one is a *qualitative time abstraction*, i. e., the environment can be seen as a *discrete* process, namely as an infinite sequence of inputs $I_1, I_2, \ldots$ occuring at successive instants of time. The system is faster than the environment, namely the reaction of the system to the inputs I_i is completed before the inputs I_{i+1} are produced. The second model is a *quantitative time abstraction* based on an internal clock. Our formalisation work was based on the informal report of [54].

The last major issue was to develop test concepts and test methods for hierarchical, structured statecharts. Ways of approaching these methods and concepts can be divided into three main parts (cf. Subsections 9.1.2, 9.1.3 and 9.1.4).

9.1.2 Contributions with respect to related methods

To place the methods and concepts which are developed in this thesis in an appropriate context, we reviewed the related work in Section 7.6. In particular, testing based on finite state machines (FSMs) is described and several criticisms of FSMs analysis/testing from literature are discussed. We introduced some notations and concepts related to FSMs that can be used to allow a formal comparison and discussion of the four best well-known state-based test selection methods: *T-method* [110], *W-method* [22], *Distinguishing Sequence Method* (*DS*) [43], and *Unique-Input-Output* (*UIO*) [148].

In particular, we formally described the informally defined *W-method* [22] (usually known as the *automata equivalence test*). We gave the formal description of a *P-set* called the *transition cover set* (cf. pp. 130ff.). In addition, we summarized the fault detection power of the W-method due to [22] by Theorem 7.1. The formal proof is given on pp. 132ff.

We summarised the results of the four test selection methods above in Table 7.1. Though the W-method [22] has the largest generated test set, it is generally accepted that the W-method has the best fault detection capability (cf. Table 7.1). Therefore, this approach has received the widest attention in the research area of software testing.

Often, FSMs are criticised for being too restrictive for many applications, e. g., real world concurrent computing. A *solution* suggested by Chow is to separate the control structure of a program from the data structure and to represent the former as a FSM. Then the state-based techniques, e. g., developed by Chow [22] and Fujiwara et. al. [37] could be used to test the control structure.

Nonetheless, the assumption that the control structure of the system can be modelled separately from the data variables is not realistic in many cases. It would imply that the next state depends solely on the current state and the input which is not usually the case. The idea that the variables that affect the program control could be replaced by a number of additional states is also impractical since in many cases this number can be very large.

In Subsection 7.6.3 we discussed the problems/weaknesses of the proposed state-based techniques, and indicated how these criticisms are addressed in this thesis. To allow a theoretical comparison of the FSM-based techniques (e. g., developed by Chow [22] and Fujiwara et. al. [37]) with our test concepts (cf. Subsection 7.7.3), we found it necessary to provide the full theoretical background information of reactive models, especially to classify statecharts in terms of *nondeterminism*, *pure parallelism* and *bounded cooperative concurrency*, and above all, to remain as close as possible to classical finite automata (cf. Appendix B).

In Appendix Subsection B.1.2 we discuss the classical automata with respect to their suitability for modelling real world concurrent computing. The application of *existential* and *universal features*[3], which are perhaps the most well-known notions for modelling concurrency in complexity theory, have two drawbacks:

(i) No communication exists in the spawned processes, except when time comes to decide whether the input should be accepted.

(ii) Existential and universal branching are unbounded; new processes can be spawned without limit as the computation proceeds (i. e., as the length of the input word grows).

As a solution to overcome the first drawback, in particular to capture real world concurrency, we argue that one requires cooperative concurrency (cf. [32, 31]). The idea of *cooperative concurrency* is that during a single computation, the mechanism can be in more than one state (component) and that these are able to cooperate (communicate) while achieving a common goal.

In thinking about the second drawback we have to remember that in the real world the number of processes participating in concurrent computations is bounded, and therefore cannot be assumed to grow as the size of input grows. Hence, the main question is to determine how *bounded cooperative concurrency* progresses with respect to the two classical kinds of branching in the realm of finite automata and their extensions. The idea of bounded cooperative concurrency is formulated in Appendix Section B.2, and the defined classes correspond to different kinds of statecharts (cf. Figure B.2 on page 203). These classes were first investigated in [31, 32].

In other words, statecharts were proposed to overcome the limitations of FSMs, such as the ones seen above.

Nonetheless, there are hardly any testing method for complex statecharts. A rather *informal* testing method for statecharts is described in [15]. This is the only method known to the author of this monograph. According to Singh [15], the method should test whether the implementation of a statechart design produces the same output as the design when exposed to the same inputs. The basic idea behind Singh's method is to

[3] These features are described in Subsection B.1.1 on page 194

apply inputs derived from the design to an implementation under test and to make sure that the output observed is the same as that required by the design.

We outlined the testing method of Singh [15] on pp. 136ff. The main problem of Singh's [15] method is that the Equation 7.3 used to construct the set of test sequences corresponds to the W-method given by Equation 7.1 on page 131, whose test effort is known to be very large. In other words, such tests should be applied to special parts of the system model (e. g., 'critical' parts) to allow a practical use.

To make a step further, we established new methods based on path testing. The next two subsections present the central part of this dissertation. They include our research contributions to the testing methods and concepts in this thesis (cf. [85, 86, 87, 88]). We divided the development of concepts into *module concepts* and *integration concepts*.

9.1.3 Contributions to the module test approach

We developed *module concepts* which operate on *hierarchical* statecharts, i. e., the components of a statechart to be tested are derived from the considered *level* of the statechart. More specifically, we considered the problems of testing hierarchical, structured system models in Subsection 7.7.1. We then introduced some formal notations and concepts related to digraphs which are necessary for a formal development of graph-theoretic based methods (cf. Subsection 7.7.2). After that we established a part of the "framework" which is used for the generation of test paths "path cover for edges" which operate on different levels (cf. Definition 7.34 for construction of *reduced graph*). As a result of the establishment of the formal statechart specification guideline and the formal step semantics (cf. Subsection 9.1.1 above), we could make certain assumptions (cf. Test condition 7.1 (1) and (2) on page 149). The idea behind the additional test conditions (3) and (4) of Test condition 7.1 (cf. page 149) is to simplify and thus also improve the effectiveness of the module testing process.

In Subsection 7.8.2 we presented methods for the generation of test paths that satisfy the given Criterion 7.1 (cf. page 147). We solved the path generation problem by the algorithm *Generate_Paths_inG* (cf. Algorithm 7.1 page 150).

The path generation problem was divided into three major parts, namely

I. determining test paths in the reduced graph G_r^s (cf. Definition 7.34 page 147) which consist of the *strongly connected components* $SCCs$ (cf. Definition 7.30 page 145) of the original graph G,

II. determining test paths in each strongly connected component SCC and finally

III. use the test paths of part I and II to compose (we say 'merge') final test paths.

We gave a detailed description with examples for all the three parts. We presented their formal algorithms in Section 7.9.

9.1.4 Contributions to the integration test approach

The final objective of this thesis was to establish concepts and methods which can aid the testing of the interacting processes (modules) of the whole system. The concepts for an *integration test* should mainly exploit the module tests, i.e., determine which of the established module test cases may be involved in the tests required for the particular module interfaces.

When testing large and complex system models, the individual components (modules) are usually tested in isolation during module testing. Then the interfaces of the modules are tested during one or more *integration steps* (cf. [60]). Each integration step requires that actual modules replace stubs one at a time, and that a 'basis' set of test paths are exercised through each module as it is added to the system.

The idea of our *integration test strategy* was based on one of the well known heuristic approaches from practical testing:

(I.) Perform detailed tests for individual components (processes). This test corresponds to the *module test*[4] which has to be extended by test cases which do not obey the test condition restriction (3) on page 149.

(II.) Then test the proper integration of an increasing number of components, adding them one by one until all components are integrated. We call this test strategy the *incremental integration test.*

The main drawbacks of the incremental integration test approach were discussed in detail on page 160. The basic problem of an *incremental integration test* is that it should 'drive' each module through a full 'basis' set of test paths in an integration context. Unfortunately, this is often practically impossible due to intermodule control coupling, i. e., incremental execution at each integration step may be costly since the number of scenarios might be very high.

To address these drawbacks, we modified the incremental integration technique, in such a way that only transitions affecting the module calling relationships need to be considered for integration testing. We call this test strategy the *goal oriented integration analysis.*

For the development of *integration test concepts* for hierarchical statecharts models, we extended the module test Criterion 7.1 (cf. page 147) which was suggested for performing test paths on every level of a given statechart by another Criterion 8.1 (cf. page 161) called *integration test criterion*, to exercise test paths between modules which rely on internal events and conditions which were not considered so far due to test condition (4) on page 149.

The integration test Criterion 8.1 mainly differs from the module test Criterion 7.1, in that the generation of test paths based on Criterion 8.1 must consider a subtle issue: the execution of an external transition between the modules demands the execution of internal transitions (or the setting of variables) whenever there exists a dependency. We

[4] cf. Subsection 9.1.3

gave a detailed description of the integration test concepts including examples for all the steps (cf. Subsection 8.1.3). We described the goal oriented integration analysis in Subsection 8.2.

9.2 Concluding remarks and open problems

Concluding remarks

The test concepts and test methods which have been developed in this thesis can be applied to large and complex models. Though these methods have been developed on models of statecharts, they should however, with a little modification be applicable to other graph-based specifications. In other words, the aim here was to establish methods which are not restricted to statecharts. In fact, their application should be easily extendable to different specification languages. Our *module test approach* is a *graph-theoretic* based technique, and should therefore, be easily applied to other graph-based formal specification languages (with a little modification).

It should also be noted that the methods and concepts developed for the models are not limited to the specification phase, but can be extended to later stages of the system development.

Open problems

There is an opportunity for future work in integration testing. The main area of analysis are critical transitions (or critical points) (cf. Definition 8.1 page 164) of a system model. In particular, for time dependent variables, methods for analytical *time dependent behaviour* have to be applied.

In Section 6.3 we formalised parts of the basic step algorithm related to timeout events (which were only informally defined in [54]) and the internal clock. Another alternative would be to express the properties of time dependent variables by means of temporal logic (cf. Appendix A). In Appendix A we presented an overview of the subject of temporal logics. It should be used particularly to enable the reader to become acquainted with the basic ideas of this formal language and evaluate the advantages of applying temporal logics to program reasoning, especially to concurrent programs.

Temporal logics are nonclassical (multi-modal) logics that enable us to formulate and verify propositions about situations dynamically changing in time. In particular, temporal logic has been used and proposed as a formal language to specify and verify reactive programs (cf. [129, 130]). Nowadays, Temporal Logic is an active area of research interest.

In Appendix A Section A.6 we considered the decidability problem of the propositional linear temporal logic (PLTL). Appendix A Section A.7 was devoted to linear-time logic model checking problem (LMPCP). Such models can be extended with explicit time clocks and thus be applied to *proof-theoretic reasoning* or *model-theoretic reasoning* for time dependent variables. There are still other types of logics like temporal logic of actions (cf. [90]) which can also be extended to achieve a better technique for formal reasoning about concurrent systems.

Appendix A

Temporal logic

In Subsubsection 1.2.2.2 it was noted that most logics applied to reasoning about reactive systems, especially about concurrent programs, are either *first-order predicate logics* or *temporal logics*. The difference between the models of first-order logics and temporal logics is that temporal logics use special symbols to provide a simple and natural, but precise, way of describing the order in which interactions occur, without the adaption of absolute time measures. This Appendix presents an overview of the subject of temporal logics. It is used particularly to enable the reader to become acquainted with the basic ideas of this formal language and evaluate the advantages of adopting temporal logics to program reasoning, especially to concurrent programs. We shall study propositional linear temporal logic (PLTL) and first-order linear temporal logic (FOLTL). The PLTL dealt with in this Appendix is decidable. The FOLTL is undecidable. PLTL thus plays a big role in automated theorem proving. In recent years, a variety of algorithms have been developed to verify properties of systems modeled as state machines. In Appendix Section A.7 we shall describe the linear-time logic model checking problem (LMPCP) for PLTL. Finally, we shall see two broad classes of properties of concurrent programs, the safety and liveness properties.

A.1 Introduction to temporal logic

Temporal logic (TL) [126] is a formal specification language proposed by Pnueli for the description and analysis of time-dependent and behavioural aspects of reactive systems. Historically, TL is a branch of modal logic. It is a logic of propositions whose truth and falsity may depend on time. In classical mathematics propositions do not depend on time. Hence TL is not of much interest there. The mathematical treatment of programs, however, contains a significant dynamic aspect. A model of the execution of a program may be viewed as a sequence of states. In different states, program entities, e.g., variables, may have different values. Consequently, propositions about these values may have different truth values. The language of temporal logic provides various "modalities" to enable us to describe and reason about how truth value assertions change over time.

Any execution of a program consists of a sequence of situations, or states, determined by the program's statements. Thus any natural notion of time adequate for program specification and verification has to reflect the nature of execution sequences of programs.

Although the choice of the time structure is not quite obvious, we shall assume that time corresponds to execution sequences of programs. Due to the fact that these sequences are determined step by step by program instructions, time is assumed to be discrete, with points corresponding to natural numbers. Now the main idea is that different points may yield different (truth) values of propositions (program entities, e.g., variables).

Consider a proposition P_i. The *problem* is to describe the variety of the truth values of P_i at different times t. One way would be to introduce an explicit time parameter t in the proposition and denote it by $A(t)$. Unfortunately, this method is limited to transformational programs, whose semantics can be viewed as given by a *transformation*[1] from an initial state to a final state. Gabbay [38] illustrates drawbacks of adding an extra time variable to the language of predicate logic (cf. [38] pp. 20ff.). The constructed proposition of predicate logic is not readable, although its original form is crystal clear and understandable without any mental effort. The main reason is that the described 'sentence' contains an infinite number of time structures. For the case of nonterminating or continuously operating concurrent programs[2] such as operating systems and network protocols, we need *temporal logical* operators which enable us to formulate new propositions about the truth values of P_i at time points which are related to some 'reference point' in particular ways.

Temporal logics are nonclassical (multi-modal) logics that enable us to formulate and verify propositions about situations dynamically changing in time. In particular, temporal logic (TL) has been used and proposed as a formal language to specify and verify reactive programs (cf. [129, 130]). Nowadays, Temporal Logic is an active area of research interest.

A.2 Classification of temporal logic

This section is to give the reader an idea of the wide range of possibilities in formulating a system of Temporal Logic. Nevertheless, only those are mentioned which are the most intensively investigated types of Temporal Logics.

A.2.1 Branching versus linear time

There are two distinct views in the definition of a system of a temporal logic:

- *linear time*: at each moment there is only one possible future moment;
- *branching time*: time has a tree-like nature in which, at each instant, time may split into alternative courses representing different possible futures.

[1] This is a traditional approach which considers the relational model of programs, i.e. characterises the behaviour of processes by relations between data and results.

[2] In contrast to transformational programs, these are referred to as reactive programs.

Figure A.1 depicts the two possible views. The main difference lies in the semantics of the time structure. Whereas in a logic of linear time, temporal modalities are provided for describing events along a single time line, in a logic of branching time, the modalities reflect the branching nature of time by allowing *quantification* over possible futures. There is much discussion about whether linear or branching time is to be preferred (cf. [34, 35, 91, 127]). As a matter of fact both approaches have been applied to program specification and reasoning.

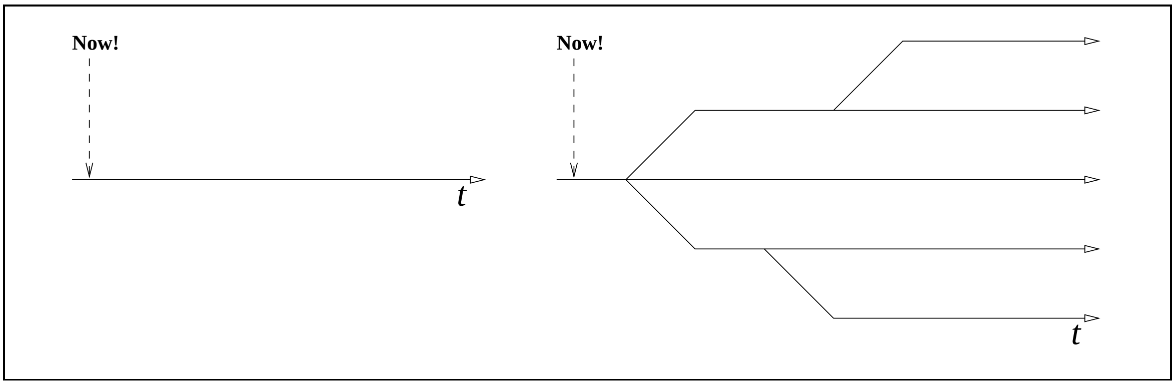


Figure A.1: Linear time versus branching time.

A.2.2 Points versus intervals

Most temporal formalisms are based on points semantics.

- *Points formalisms*: Temporal formulas are evaluated ('interpreted') as true or false with respect to a certain reference point in time.
- *Intervals formalisms*: Temporal formulas are evaluated over intervals of time.

Intervals are claimed to simplify the formulation of correctness properties (cf. [150]).

A.2.3 Discrete versus continuous

In most temporal logics used for program reasoning, time is *discrete*.

- *Discrete*: The present moment corresponds to the program's current state and the next state moment corresponds to the program's immediate successor state. The underlying time structure is isormophic to natural numbers with their ordering ($I\!N_0$, $<$).
- *Continuous*: The time structure is isormophic to reals (or rationals).

Mainly temporal (or time) logics are interpreted over a continuous time structure. Such applications have been proposed for real-time programs where strict, quantitative properties are of great interest (cf. [9, 82, 133]).

A.2.4 Past versus future

The temporal operators are partitioned into two groups:

- *Future operators*: $Futr(P, t)$ means P holds at a time t in the future with respect to the current time. In most temporal logics for reasoning about concurrency, only future operators are used, since generally program executions have a definite starting time.
- *Past operators*: $Past(P, t)$ means P holds at a time t in the past with respect to the current time. In [98], it is claimed that the past tense operators make the formulation of specifications more natural and convenient.

A.2.5 Global versus composition

- *Global reasoning*: Predicate symbols and functions are globally interpreted with regard to the application of concurrent programs.
- *Composition reasoning*: The syntax of temporal operators allows expression of correctness properties concerning different programs in the same formula. A complete program can be verified by specifying and verifying its constituent subprograms and then combining them into the complete program. The proof of correctness of the complete program thus is achieved by using the proofs of the subprograms as lemmas (cf. [4, 128]).

A.2.6 Qualitative versus quantitative

- *Qualitative*:
- *Quantitative*:

So far, we have given the various classifications to enable the reader to get a feeling about the wide range of possibilities in describing a system of Temporal Logic. Now we summarise this section by listing the classes which we do not follow up in details. These include:

- branching time temporal logic,
- intervals formalisms,
- continuous time logics,
- past operators and
- compositionality of temporal logic.

A.3 Basic aspects of linear temporal logic

In linear temporal logic (LTL), the underlying structure is a totally ordered set $(S, <)$. We assume that time

(I.) is discrete,

(II.) has an *initial* moment with no predecessors,

(III.) is *infinite* into the *future*.

We then use temporal operators to reason about truth values of propositions at time points which are related to some 'reference point' in particular ways. Some of the desirable operators with their intended meaning are shown in Figure A.2.

In modal logic, the underlying structure is described in terms of *Kripke frames* and *Kripke structures*. A Kripke frame is a pair $(S,\ R)$ where S is a set (of possible worlds) and R is binary relation on S (i.e. $R \ \subseteq \ S \times S$). In temporal logic, R is *transitive*, *irreflexive* and in particular in LTL R is *linear* (or total). A Kripke structure $\mathcal{M}$ is a triple $\mathcal{M} = (S,\ R,\ V)$ where $(S,\ R)$ is a Kripke frame and V a mapping which assigns each primitive (atomic) proposition the set of states at which it is true.

Definition A.1 *Given the set of atomic proposition $\mathcal{P}$, denoted by $P,\ Q,\ P_1,\ P_2,\ \dots$, we define a* **temporal structure** *$\mathcal{M} = (S,\ R,\ V)$, where*

S is a non-empty set of **states***,*

R is a total **binary relation** *$\subseteq\ S \times S$ i.e. $\forall s \in S\ \exists t \in S :\ (s,t) \in R$, and*

$V\ :\ \mathcal{P} \rightarrow 2^S$ assigns each atomic proposition the set of states at which it is true.

Thus $V(P)$ is the set of states at which P is "true" under the valuation of V.

Definition A.2 *Given a temporal structure $\mathcal{M}\ =\ (S,\ R,\ V)$ we define* **full path** *as an infinite sequence $s_0, s_1, s_2, \dots$ of states such that $\forall i\ :\ (s_i, s_{i+1}) \in R$. We write $\pi = \langle s_0, s_1, s_2, \dots \rangle$ to denote a full path.*

A.4 Propositional linear temporal logic

A.4.1 Syntax and notational convetions

The *formulas* of temporal logics are built from the denumerable non-empty set $\mathcal{P}$ and the symbols $\neg$ ("not"), $\implies$ ("implies"), $\bigcirc$, $\Box$, **atnext**, (,).

Definition A.3 *The set of formulas $\mathcal{F}_p$ of propositional linear temporal logic (PLTL) is obtained by the following rules:*

Temporal operators:

$\bigcirc\, p$	*next time p* p holds at the next moment
$\Box\, p$	*always p* p holds at all future time points
$\Diamond\, p$	*sometimes p* p holds at some future time point
p **until** q	*p until q* p holds at all following moments at least up to a point at which q holds
p **unless** q	*p unless q* If there is a following time point at which q holds then p holds up to that point or else p holds permanently
p **before** q	*p before q* If q holds at sometime in the future then p holds before that
p **atnext** q	*p atnext q* p will hold at next time point that q holds
p **while** q	*p while q* p holds as long as q holds

Figure A.2: Examples of temporal operators and intended meaning.

(1) each atomic proposition $P \in \mathcal{P}$ is a formula;

(2) if p and q are formulas then $\neg p$, $\Box\, p$ and $\bigcirc\, p$ are formulas;

*(3) if p and q are formulas then $(p \implies q)$ and $(p$ **atnext** $q)$ are formulas;*

$\iff$ and the other formulas of Figure A.2 can be introduced as abbreviations. For example, $\Diamond\, p$ abbreviates $\neg\, \Box\, \neg p$.

Note 1 *In order to simplify the notation we establish a priority order of the operators: The temporal operators $\bigcirc$, $\Box$, $\neg$ have the highest priority; followed by* **atnext***, followed by $\wedge$, followed by $\vee$, and finally by $\implies$.*

Example

We write

$\bigcirc\, p_1 \vee q_1 \implies \neg p_2 \wedge \Box\, q_2$ **atnext** p_3

instead of

$((\bigcirc\, p_1 \vee q_1) \implies (\neg p_2 \wedge (\Box\, q_2$ **atnext** $p_3)))$.

A.4.2 Semantics

We define the semantics of a formula $p \in \mathcal{F}_p$ of PLTL with respect to a linear-time structure $\mathcal{M} = (S,\ R,\ V)$. Assume a full path π is given in $\mathcal{M}$

Definition A.4 *$\mathcal{M}, \pi \models p$ means that "in structure $\mathcal{M}$ formula p is true in timeline π". As a short hand for $\mathcal{M}, \pi \models p$ we will write $\pi \models p$.*
For an infinite sequence $\pi = \langle s_0, s_1, s_2, \ldots\rangle$, $\pi^n = \langle s_n, s_{n+1}, \ldots\rangle$ is the n^{th} suffix of π.

Now we define when a formula p is valid in the full path π of the temporal structure $\mathcal{M}$ (in notation $\mathcal{M}, \pi \models p$) by induction on the structure of the formula p:

$$\begin{array}{lcll}
\mathcal{M}, \pi \models & P & \textit{iff} & P \in V(s_0) \\
\mathcal{M}, \pi \models & p \implies q & \textit{iff} & \mathcal{M}, \pi \not\models p \textit{ or} \\
 & & & \mathcal{M}, \pi \models q \\
\mathcal{M}, \pi \models & \neg p & \textit{iff} & \mathcal{M}, \pi \not\models p \\
\mathcal{M}, \pi \models & \bigcirc p & \textit{iff} & \mathcal{M}, \pi^1 \models p \\
\mathcal{M}, \pi \models & \Box p & \textit{iff} & \forall n \geq 0 : \mathcal{M}, \pi^n \models p \\
\mathcal{M}, \pi \models & p \textbf{ atnext } q & \textit{iff} & \forall n > 0 : \mathcal{M}, \pi^n \not\models q \textit{ or } \exists k > 0 : \mathcal{M}, \pi^k \models q \ \wedge \ p \textit{ and} \\
 & & & \forall i < k : \mathcal{M}, \pi^i \not\models q
\end{array}$$

The temporal operators $\Diamond$, **until**, **unless**, **before**, and **while** are also of great importance and therefore their semantics will be given explicitly:

$$\begin{array}{lcll}
\mathcal{M}, \pi \models & \Diamond p & \text{iff} & \exists n \geq 0 : \mathcal{M}, \pi^n \models p \\
\mathcal{M}, \pi \models & p \textbf{ until } q & \text{iff} & \exists n > 0 : \mathcal{M}, \pi^n \models q \text{ and} \\
 & & & \forall k, 0 < k < n : \mathcal{M}, \pi^k \models p \wedge \neg q \\
\mathcal{M}, \pi \models & p \textbf{ unless } q & \text{iff} & [\exists n > 0 : \mathcal{M}, \pi^n \models q \text{ and} \\
 & & & \forall k, 0 < k < n : \mathcal{M}, \pi^k \models p] \text{ or} \\
 & & & \forall k > 0 : \mathcal{M}, \pi^k \models p \\
\mathcal{M}, \pi \models & p \textbf{ while } q & \text{iff} & [\exists n > 0 : \mathcal{M}, \pi^n \not\models q \text{ and} \\
 & & & \forall k, 0 < k < n : \mathcal{M}, \pi^k \models p \wedge \mathcal{M}, \pi^k \models q] \text{ or} \\
 & & & \forall k > 0 : \mathcal{M}, \pi^k \models p \\
\mathcal{M}, \pi \models & p \textbf{ before } q & \text{iff} & \forall n > 0 : (\mathcal{M}, \pi^n \models q \implies \exists k, 0 < k < n : \mathcal{M}, \pi^k \models p)
\end{array}$$

Definition A.5 *We now define :*

(a) *A PLTL formula $p \in \mathcal{F}_p$ is* ***satisfiable*** *iff there exists a temporal structure $\mathcal{M} = (S,\ R,\ V)$ and a full path π in $\mathcal{M}$ such that $\mathcal{M}, \pi \models p$.*
In this case $\mathcal{M}$ is called a ***model*** *of p.*

(b) *p is called* ***valid****, written $\models p$ iff for all temporal structures $\mathcal{M} = (S,\ R,\ V)$ and all full paths π of $\mathcal{M}$ we have $\mathcal{M}, \pi \models p$.*

Note *p is valid iff $\neg p$ is not satisfiable.*

Examples

(1) $p \implies \Diamond q$
"If p is true now, then at some future moment q will be true."

$\leadsto$ This formula is satisfiable, but not valid.

(2) $\Box\,(p \implies \Diamond\, q)$
"Whenever p is true, q will be true at some subsequent time point."
$\leadsto$ This formula is satisfiable, but not valid.

(3) $p \wedge \Box\,(p \implies \bigcirc\, p) \implies \Box\, p$
"If p is true now, and whenever p is true, it will be true at the next moment, then p is always true."
$\leadsto$ This formula is valid, and is a temporal formulation of mathematical induction.

A.4.3 Notions of "Validity"

Similar to many other logics, there are a number of significant validities. In this section we present some significant validities in PLTL. In the following, we write
$p \equiv q$
instead of
$\models p \iff q$.

Duality laws

(P1) $\neg \bigcirc p \equiv \bigcirc \neg p$ (self dual)
(P2) $\neg \Box\, p \equiv \Diamond\, \neg p$
(P3) $\neg \Diamond\, p \equiv \Box\, \neg p$

We extend the list of frequently used temporal formulas by

$\Diamond\Box\, p$: Sometime p will hold permanently.
$\Box\Diamond\, p$: For every following state there is a later state where p holds, i.e. "p holds infinitely often from now on".

The semantics of the formulas above is given as follows:

$$\mathcal{M}, \pi \models \Diamond\Box\, p \quad \text{iff} \quad \exists k\ \forall n, n \geq k \geq 0 : \mathcal{M}, \pi^n \models p$$
$$\mathcal{M}, \pi \models \Box\Diamond\, p \quad \text{iff} \quad \forall k\ \exists n, n \geq k \geq 0 : \mathcal{M}, \pi^n \models p$$

Further validities:

(P4) $\Diamond\Box\, \neg p \equiv \neg \Box\Diamond\, p$
(P5) $\Box\Diamond\, \neg p \equiv \neg \Diamond\Box\, p$

A.4.4 Formal system of PLTL

A *deductive* (proof) system for temporal logic consists of sets of *axiom schemes* and *inference rules*.

Let us consider the following axioms and rules of inference:

Schemes of axioms

(Ax1) All validities of propositional logic;
(Ax2) $\neg \bigcirc p \iff \bigcirc \neg p$;
(Ax3) $\bigcirc (p \implies q) \implies (\bigcirc p \implies \bigcirc q)$;
(Ax4) $\Box\, p \implies p \wedge \bigcirc\Box\, p$;
(Ax5) $\bigcirc\Box\, \neg q \implies p$ **atnext** q;
(Ax6) p **atnext** $q \iff \bigcirc (q \implies p) \wedge \bigcirc (\neg q \implies p$ **atnext** $q)$.

Inference (derivation) rules

(R1) $\vdash p$ and $\vdash p \implies q$ then $\vdash q$;
(R2) $p \vdash \bigcirc p$;
(R3) $\vdash p \implies q, \vdash\bigcirc p$ then $\vdash p \implies \Box\, q$

The above axioms and rules of inference describe a possible deductive system of PLTL. For reference, this system is called S_{PLTL}.

Definition A.6 *A PLTL formula $p \in \mathcal{F}_p$ is* ***provable****, written $\vdash p$, iff there exists a sequence of formulas from $\mathcal{F}_p$ ending with p such that each formula is an instance of an axiom or follows from previous formulas by application of an inference rule.*

Definition A.7 *A proof system is said to be* ***sound*** *iff every provable formula $p \in \mathcal{F}_p$ is valid. It is said to be* ***complete*** *iff every valid formula is provable.*

The proof system S_{PLTL} is sound.

Theorem A.1 (*Soundness Theorem for S_{PLTL}*)
Let p be a formula. If $\vdash p$ then $\models p$.

The proof system S_{PLTL} is complete.

Theorem A.2 *(Completeness Theorem for S_{PLTL})*
For every formula p, if $\models p$ then $\vdash p$.

The proofs of A.1 and A.2 can be found in [83].

An informal discussion of soundness and completeness

Soundness is the most fundamental property of any reasonable proof system. Soundness means that all proved conclusions are semantically true. In terms of procedures, soundness can be defined as correctness of the procedure implementing the proof system. That is, all results of the procedure must be correct. On the other hand, completeness means that all semantically true conclusions can be obtained as a result of the procedure. As soundness is always required, a system is acceptable whenever it is sound and is as close to completeness as possible.

A.5 First-order linear temporal logic

First-order linear temporal logic (FOLTL) is developed from PLTL plus a first-order language $\mathcal{L}_{FO}$.

Symbols of $\mathcal{L}_{FO}$

We distinguish between:

- constants (0-ary function symbols),
- propositions (0-ary predicate symbols),
- a set of individual variables,
- n-ary function symbols ($n \geq 1$),
- n-ary predicate symbols ($n \geq 1$).

Notation:

$\varphi,\ \psi,\ \ldots$	for n-ary, $n \geq 1$ predicate symbols,
$P,\ Q,\ \ldots$	for proposition symbols,
$f,\ g,\ \ldots$	for n-ary, $n \geq 1$ function symbols,
$c,\ d,\ \ldots$	for constants symbols, and
$y,\ z,\ \ldots$	for variable symbols,
$\approx$	binary predicate symbol (equality symbol),
$\forall$ and $\exists$,	denotes universal and existential quantification, respectively.

A.5.1 Syntax of $\mathcal{L}_{FO}$

Inductive definition of *terms*

(1) Each constant c is a term.

(2) Each variable y is a term.

(3) If f is an n-ary function symbol and $t_1,\ \ldots,\ t_n$ are terms then $f(t_1,\ \ldots,\ t_n)$ is a term.

Inductive definition of *atomic formulas*

(1) Every 0-ary predicate symbol is an atomic formula.

(2) If ψ is an n-ary predicate symbol and $t_1,\ \ldots,\ t_n$ are terms then $\psi(t_1,\ \ldots,\ t_n)$ is an atomic formula.

(3) If t_1 and t_2 are terms then $t_1 \ \approx\ t_2$ is also an atomic formula.

Inductive definition of *(compound) formulas*

Informally, a variable x is called *free* in a formula p if there is no subexpression of the form $\exists x\psi$ or $\forall x\psi$ for some formula ψ in $\mathcal{F}_p$.

(1) Every atomic formula is a formula.

(2) if p and q are formulas then $\neg p$ and $(p \implies q)$ are formulas;

(3) if p is a formula and x is a free variable in p then $\exists x\ p$ is a formula and $\forall x\ p$ is a formula.

A.5.2 Semantics of $\mathcal{L}_{FO}$

For the definition of the semantics of the $\mathcal{L}_{FO}$, we use the concept of an interpretation I over some domain D. $\mathbb{B} := \{0, 1\}$ where 0 stands for $false$ and 1 stands for $true$. An interpretation I over a domain D is a mapping with the following properties:

- for an n-ary predicate symbol ψ, $n \geq 1$: $I(\psi)$ is a function $D^n \ \rightarrow \ \mathbb{B}$
- for a proposition symbol P: $I(P) \ \in \ \mathbb{B}$
- for an n-ary function symbol f, $n \geq 1$: $I(f)$ is a function $D^n \ \rightarrow \ D$
- for a constant symbol c: $I(c) \ \in \ D$
- for a variable y: $I(y) \ \in \ D$

Using interpretation I we assign by inductictive definition a value $I^*(t)$ to terms and a truth value $I^*(p)$ to formulas p. We denote that $I^*(p) = true$ by $\pi \models p$.

(1) $I^*(c) := I(c)$

(2) $I^*(x) := I(x)$

(3) $I^*(f(t_1, \ \ldots\ , \ t_n)) \ = \ I(f)(I^*(t_1), \ \ldots\ , \ I^*(t_n))$

(4) $I \models P$ $(P \in \mathcal{P})$ iff $I(P) \ = \ true$

(5) $I \models \psi(t_1, \ \ldots\ , \ t_n)$ iff $I(\psi)(I^*(t_1), \ \ldots\ , \ I^*(t_n)) \ = \ true$

(6) $I \models t_1 \ \approx \ t_2$ iff $I^*(t_1) \ = \ I^*(t_2)$

(7) $I \models p \implies q$ iff not $I \models p$ or $I \models q$

(8) $I \models \neg p$ iff not $I \models p$

(9) $I \models \exists x\ p \Leftrightarrow$ there is some $d \in D$ with $I[x \leftarrow d] \models p$ where $I[x \leftarrow d]$
$I[x \leftarrow d](x) := d$ and in all other cases equals to I.

(10) $I \models \forall x\ p \Leftrightarrow$ for each $d \in D$ with $I[x \leftarrow d] \models p$ where $I[x \leftarrow d]$
$I[x \leftarrow d](x) := d$

A.5.3 Syntax of FOLTL

We partition the set of variables into two infinite subsets:

(a) *Global (rigid)* variables (respectively propositions) which are independent of the flow of time, i.e., must have the same value in all states of computation.

(b) *Flexible (local)* variables (respectively propositions) which are time dependent symbols, i.e., may take different values in different states of the computation, e.g., control variables.

Notation A.1 *The set of all local individual variables is denoted by V_{loc} and the set of all local propositions is denoted by $\mathcal{P}_{loc}$ and*

Definition A.8 *The set of formulas $\mathcal{F}_{FO}$ of first-order linear temporal logic (FOLTL) is obtained by the following rules:*

(1) Every atomic formula is a formula;

(2) if p and q are formulas then $\neg p$, $\Box\, p$ and $\bigcirc\, p$ are formulas;

*(3) if p and q are formulas then $(p \implies q)$ and $(p$ **atnext** $q)$ are formulas;*

(4) if p is a formula and x is a free global variable in p then $\exists x\ p$ is a formula.

A.5.4 Semantics of FOLTL

The semantics of FOLTL is given by a first-order linear-time structure $\mathcal{M} = (S, R, V_T, V_F, I)$ over a domain D and a full path π. Here $V_T : S \rightarrow D^{V_{loc}}$ and $V_F : S \rightarrow 2^{\mathcal{P}_{loc}}$ assign each state s the valuation function for the local individual variables and the local propositions, respectively. I defines the interpretation of the global variables as in Subsection A.5.2 above. The interpretation $(\mathcal{M},\ \pi)$ is defined as follows. The value of a term t, respectively of a formula p, in a full path π of $\mathcal{M}$, noted by $(\mathcal{M},\ \pi)(t)$, respectively $(\mathcal{M},\ \pi)(p)$, is defined inductively in the following way.

For terms:

(1) $(\mathcal{M}, \pi)(c) \ = \ I(c)$ where c is a constant;
Note: All constants are global.

(2) $(\mathcal{M}, \pi)(y) \ = \ I(y)$ where y is a global variable;

(3) $(\mathcal{M}, \pi)(y) \ = \ V_T(s_0)(y)$ where y is a local variable, and $\pi = \langle s_0, s_1, s_2, \ldots \rangle$;

(4) $(\mathcal{M}, \pi)(f(t_1,\ \ldots\ ,\ t_n)) \ = I(f)((\mathcal{M}, \pi)(t_1),\ \ldots\ ,\ (\mathcal{M}, \pi)(t_n))$;

(We use $\mathcal{M}, \pi \models p$ as a notation for $(\mathcal{M},\ \pi)(p) = \mathit{true}$)

For atomic formulas:

(1) $\mathcal{M}, \pi \models P$ iff $I(P) = 1$, where P is a global proposition;

(2) $\mathcal{M}, \pi \models P$ iff $P \in V_F(s_0)$, where P is a local proposition and $\pi = \langle s_0, s_1, s_2, \ldots \rangle$;

(3) $\mathcal{M}, \pi \models \varphi(t_1, \ \ldots\ , t_n)$ iff $I(\varphi)((\mathcal{M}, \pi)(t_1), \ \ldots\ , (\mathcal{M}, \pi)(t_n)) = \mathit{true}$;

(4) $\mathcal{M}, \pi \models t_1 \approx t_2$ iff $(\mathcal{M}, \pi)(t_1) = (\mathcal{M}, \pi)(t_2)$.

For compound formulas:

Let $\mathcal{M} = (S, R, V_T, V_F, I)$.

$$\begin{array}{rcll}
\mathcal{M}, \pi \models & p \implies q & \text{iff} & \text{not } \mathcal{M}, \pi \models p \text{ or} \\
 & & & \mathcal{M}, \pi \models q \\
\mathcal{M}, \pi \models & \neg p & \text{iff} & \text{not } \mathcal{M}, \pi \models p \\
\mathcal{M}, \pi \models & \bigcirc\, p & \text{iff} & \mathcal{M}, \pi^1 \models p \\
\mathcal{M}, \pi \models & \Box\, p & \text{iff} & \forall n \geq 0 : \mathcal{M}, \pi^n \models p \\
\mathcal{M}, \pi \models & p \,\mathbf{atnext}\, q & \text{iff} & \forall n > 0 : \mathcal{M}, \pi^n \not\models q \text{ or } \exists k > 0 : \mathcal{M}, \pi^k \models q \wedge p \text{ and} \\
 & & & \forall i < k : \mathcal{M}, \pi^i \not\models q
\end{array}$$

$\mathcal{M}, \pi \models \exists x\, p$ iff there exists some $d \in D$ such that $\mathcal{M}[x \leftarrow d], \pi \models p$, where $\mathcal{M}[x \leftarrow d] = (S, R, V_T, V_F, I[x \leftarrow d])$.

$\mathcal{M}, \pi \models \forall x\, p$ iff for each $d \in D$ such that $\mathcal{M}[x \leftarrow d], \pi \models p$, where $\mathcal{M}[x \leftarrow d] = (S, R, V_T, V_F, I[x \leftarrow d])$.

Definition A.9

(a) A FOLTL formula $p \in \mathcal{F}_{FO}$ is ***satisfiable*** *iff there exists a first-order linear-time structure $\mathcal{M} = (S, R, V_T, V_F, I)$ and a full path π such that $\mathcal{M}, \pi \models p$.*

(b) p is called ***valid*** *iff for all first-order linear-time structures $\mathcal{M} = (S, R, V_T, V_F, I)$ and all full paths π we have $\mathcal{M}, \pi \models p$.*

A.6 Decidability of linear temporal logic

This section deals with the question of the decidability of PLTL. The propositional linear temporal logic considered here is decidable. In other words, there is an algorithm to decide whether a given formula is satisfiable. A common method for developing such algorithms is to establish the *small model property* for the logic.

Decidability problem of PLTL A.1 *Given a formula $p \in \mathcal{F}_p$ of PLTL, check whether there is a linear-time structure of PLTL $\mathcal{M} = (S,\ R,\ V)$ and a full path π satisfying p (i.e. such that $\mathcal{M}, \pi \models p$).*

Theorem A.3 *If a formula is satisfiable, then it is satisfiable by a finitely representable model. Moreover, the size of the model can be calculated from the size of the formula.*

Here is a naive procedure which — because of Theorem A.3 — can be used to check whether a given formula is satisfiable.

Procedure 2 Checking-satisfiablity

given a formula $p \in \mathcal{F}_p$,

calculate the size k *of the model* $\mathcal{M}$ *and*
generate all possible models of size k,

test whether they do satisify p.

Note *This check can be done by exhaustive search since* $\mathcal{M}$ *is finite. Moreover, such tests can be programmed easily.*

The following results on the complexity of deciding linear time are due to Sistla and Clarke (cf. [152]).

Theorem A.4 *The problem of testing satisfiability for PLTL is PSPACE-complete.*

In contrast to the propositional linear temporal logic, the first-order temporal logic is undecidable.

A.7 Model checking of linear temporal logic

Given a linear-time structure of PLTL $\mathcal{M} = (S, R, V)$, a full path π and a formula $p \in \mathcal{F}_p$ of PLTL, does $\mathcal{M}$ define a model of p? This problem has important applications to mechanical verification of finite-state concurrent systems. In recent years, a variety of algorithms has been developed to verify properties of systems modelled as state machines (cf. [3, 23, 76]).

To be in a position to describe the linear-time logic model checking problem (LTLMCP) for PLTL, we use a model. In particular, what does it mean for a structure $\mathcal{M}$ to be a model of a formula p?

Here is the definition of a model.

Definition A.10 *Given a formula* $p_0 \in \mathcal{F}_p$ *of linear-time logic, we say that a temporal structure* $\mathcal{M}$ *is a* **model** *iff it contains a* fullpath π *such that* $\mathcal{M}, \pi \models p_0$.

The problem is, given a temporal structure $\mathcal{M}$ and a formula p, can we determine a space bound so that we can decide $\mathcal{M}, \pi \models p_0$ within this bound? The linear-time logic model checking problem (LTLMCP) for PLTL is therefore as follows.

LTLMCP A.1 *Given a temporal structure* $\mathcal{M}$ *and a formula* p *of PLTL, decide if* $\mathcal{M}$ *is a model* p.

The LTLMCP for PLTL is solved as follows:

LTLMCP-Procedure A.1 *Given a finite temporal structure $\mathcal{M} = (S, R, V)$ and a formula $p \in \mathcal{F}_p$ of PLTL, determine for each state $s \in S$ whether there is a fullpath satisfying p at s and if so, label s with E_p.*

In other words, **proceed** as follows:

solve LTLMCP as in LTLMCP A.1 above

scan the states to see if one is labelled with E_p.

Lemma A.1 *The model checking problem for PLTL is polynomial-time reducable to the satisfiability problem for PLTL.*

As a consequence of Theorem A.4 we get the model checking problem for PLTL is in PSPACE. In fact we have

Theorem A.5 *The model checking problem for PLTL is PSPACE-complete.*

The proof can be found in Sistla and Clarke (cf. [152]).

A.8 The applications of temporal logic to program reasoning

In Section A.1, we saw that temporal logic was proposed as a formal language to specify and *verify* reactive programs. Reactive systems thus subsume many programs labelled as concurrent, parallel, or distributed, as well as control programs. Nonetheless, in the sequel, we will refer to a concurrent program, to mean a reactive, concurrent system.

The operators of temporal logic such as *sometimes* and *always* are quite appropriate for describing the time-varying behaviour of such programs.

A.8.1 Properties of concurrent programs

There are two broad classes of properties of concurrent programs:

(1) *Invariance (safety) properties*: Intuitively, a safety property asserts that "nothing bad happens". More precisely, a safety formula has the form $\Box\, p$. This property states that each finite prefix of a (possibly infinite) computation meets some requirements.

(2) *Eventuality (liveness) properties*: Informally, a liveness property asserts that "something good will happen". More precisely, a liveness formula has the form $\Diamond\, p$. Such a formula expresses that at least one position in the computation satisfies p.

Examples

Next we consider a few examples of program properties expressible by means of PLTL.

- **invariance (safety) properties**

  (1) $p \implies \Box\, q$
  "All states reached by a program after a state satisifying p will satisfy q."

  (2) $\Box\, (\neg p \,\vee\, \neg q)$
  "The program cannot enter critical regions p and q simultaneously (mutual exclusion)."

- **Eventuality properties**

  (1) $p \implies \Diamond\, q$
  "There is a program state satisifying q reached by a program after a state satisfying p."

  (2) $\Box\Diamond\, p \implies \Diamond\, q$
  "Repeating a request p will force a response q."

  (3) $\Box\, p \implies \Diamond\, q$
  "Permanent holding of a request p will force a response q."

  (4) $\Box\, (p \implies \Diamond\, q)$
  "If initially p then eventually q."

A.8.2 Analysis techniques

There are two major classes of techniques developed for formal reasoning about concurrent systems:

(1) *Proof-theoretic reasoning*: The basic idea is to manually compose a program and a proof of its correctness using a formal deductive system (cf. A.4.4).

(2) *Model-theoretic reasoning*: The idea is to use decision procedures that manipulate the underlying temporal models corresponding to programs and specifications in order to automate the process of specifying the tasks of programs and their verification (cf. A.7).

Appendix B

Theoretical Background of Concurrent (Reactive) Models

As we have seen in Chapter 7 which builds one of the main parts of this thesis, most test approaches have been based on finite state machines (FSMs). Therefore, to allow a theoretical comparison of the FSM-based techniques (e. g., developed by Chow [22] and Fujiwara et. al. [37]) with our test concepts (cf. Subsection 7.7.3), we find it necessary to provide the full theoretical background information of reactive models, especially to classify statecharts in terms of *nondeterminism*, *pure parallelism*, *bounded cooperative concurrency*, and above all, to remain as close as possible to classical finite automata.

In other words, this Appendix throws some light on the three outstanding features suggested for modelling concurrency, namely: *nondeterminism* and *pure parallelism* (the two facets of *alternation*) and *bounded cooperative concurrency*, where a system configuration consists of a bounded number of cooperating (or broadcasting) states of (*orthogonal*) *components* (see Section B.2, Definitions B.1 and B.2). In addition, the *descriptive succinctness* of these features, which contributes a lot to the analysis of concurrent (reactive) models, is described. In Chapter 1, a number of proposed models of computation such as statecharts, Petri nets, communicating sequential processing (CSP), calculus of communicating systems (CCS), temporal logic (TL), etc., were briefly described.

In this Appendix, however, the main idea is to remain as close as possible to classical finite automata. This decision has the advantage that a lot of research has been carried out on the general framework of finite automata. Therefore it seems fair and reasonable first to consider these features within this framework and thereafter compare their power in respect to other models. In particular, it is interesting to glance at the results concerning *upper* and *lower bounds* on the *relative succinctness*[1] of the fundamental features over $\Sigma^\star$ and Σ^ω.

[1] That is, the inherent *size* of an automaton required to accept a given language.

B.1 The basic features for modelling concurrency

B.1.1 Nondeterminism, Parallelism and Alternation

Usually, a nondeterministic Turing machine (TM) can be considered as a machine with a single process which must make choices during a computation and accepts the input provided some sequence of choices leads to an accepting state. In [80] a model of parallel computation based on a generalisation of nondeterminism in TMs was introduced. In particular, a nondeterministic TM is considered as a machine with an unlimited number of processes and a Boolean value B_c associated with each configuration c. The machine starts with a single process in the starting configuration c_0; whenever a nondeterministic choice must be made, the process spawns several independent parallel processes, each of which follows one of the possible choices. When a process enters an accept configuration c_A, the value of B_{c_A} becomes 1; when it enters a reject configuration, 0. In a bottom up manner, these values are passed back up the computation tree[2]. In the process a Boolean *OR* is computed at each choice configuration. The machine accepts only when the value of root node $B_{c_0} = 1$.

The concept of nondeterminism has played an important role in the theory of computation. Among others, it has been generalized in [20, 21] to define yet another important feature called *alternation*. Essentially, the proceeding above can be generalized by allowing Boolean *AND*'s and *OR*'s to be computed at choice configurations. The idea is that each state q of the finite control is associated with a Boolean connective $g_q \in \{\wedge, \vee\}$. If configurations $c_1, \ldots, c_n$ follow from configuration c in one step, and q is the state of the finite control associated with configuration c, then it follows that

$$B_c = \begin{cases} B_{c_1} \wedge \ldots \wedge B_{c_n} & \text{if } g_q = \wedge \ (\wedge\text{-branch}) \\ B_{c_1} \vee \ldots \vee B_{c_n} & \text{if } g_q = \vee \ (\vee\text{-branch}) \end{cases}$$

when the computation is finished and values of $B_{c_1}, \ldots, B_{c_n}$ have been determined. Similarly, the machine is said to accept the input provided $B_{c_0} = 1$.

Denotation: The $\wedge$-branch will be referred to as *universal* quantification, and the $\vee$-branch as *existential* quantification. Furthermore, the state q of the finite control associated with a Boolean connective $g_q \in \{\wedge, \vee\}$ is called a *universal* state if $g_q = \wedge$ (respectively *existential* state, if $g_q = \vee$).

Whereas in a nondeterministic computation there are only existential quantifiers, the concept of *alternation* allows *existential* and *universal* quantification to alternate. By definition if q is the state associated with configuration c, then c is said to be a *universal* (respectively *existential*) *configuration*, if q is a universal (respectively existential) state. The following idea is used to describe the acceptance of such machines:

Assume that a single process is started in the initial configuration c_0. Subsequently, if a process is in an existential configuration c and $c_1, \ldots, c_m$ are all the configurations following from c in one step, then the process spawns m distinct offspring processes which concurrently try to determine whether any of the c_i lead to an acceptance. Of course, some of the offsping computations may be infinite, but *at least if one* is accepting, this is

[2] Observe that the computation tree and B_c are not explicitly represented.

reported back to the parent process waiting at c, and in turn reports to its parent process that c leads to acceptance. On the other hand, if a process is in a universal configuration, it must determine whether *all* its offsprings lead to acceptance.

It should be noted that alternating machines without universal states exactly correspond to nondeterministic machines. Similarly, an alternating machine solely consisting of universal states is referred to as a parallel (universal) machine.

B.1.2 Application of the features to classes of automata

Alternation, nondeterminism and parallelism can be applied to classes of automata other than Turing machines. In [20, 21] *alternating finite-state automata* (AFAs) are defined and their complexity classes are characterised. Likewise, *parallel finite-state automata* are defined and their power is characterised in [80].

In the research community complexity classes have been characterized for classical automata (cf. [64, 65, 66]). Note that in the standard framework of complexity theory, the evaluation considers mainly the following aspects: time, space and perhaps number of processes required to solve various problems. In the following, however, the principal question is whether such a proceeding is suitable for real world concurrent computing. Clearly, existential and universal branching which are perhaps the most well-known notions for modelling concurrency in complexity theory have two drawbacks:

(1) There exists no communication in the spawned processes, except when time comes to decide whether the input should be accepted.

(2) Existential and universal branching are unbounded; new processes can be spawned without limit as the computation proceeds (i. e., as the length of the input word grows).

In order to overcome the first drawback, in particular to capture real world concurrency, one requires cooperative concurrency (cf. [31, 32]). The idea of cooperative concurrency is that during a single computation, the mechanism can be in more than one state (component) and that these states are able to cooperate (communicate) while achieving a common goal.

Now, consider the second drawback. In the real world the number of processes participating in concurrent computations is bounded, and therefore cannot be assumed to grow as the size of input grows. Therefore, the next step is to determine how *bounded cooperative concurrency* progresses with respect to the two classical kinds of branching, in the realm of finite automata and their extensions. The idea of bounded cooperative concurrency will be formulated in Section B.2.

B.1.3 Acceptance criteria

The next major question concerns the criteria for comparing the features above. In the first instance, it should be stated that all variants of finite automata with the above features, in general, accept the regular sets over $\Sigma^\star$ and[3] the ω-regular sets over Σ^ω. Hence, pure expressive power is not relevant.

Another issue is that time and space, in the classical complexity-theoretic manner, are also irrelevant. The point is that emphasis in the discussion below is mainly put on synchronised automata.

From these observations therefore follows that the correct criterion seems to be *succinctness*, i. e., the inherent *size* of an automaton required to accept a given language. As we saw in Section 7.2, Definition 7.4, this measure is crucial when *automata equivalence* (see also [22]) is considered.

B.1.4 Bounds for classical automata

In this subsection, known bounds for classical automata regarding succinctness are given. Nondeterministic finite automata (NFAs) are exponentially more succinct than deterministic finite automata (DFAs) in the following upper and lower bound relations based on [64, 104, 136]:

- Any NFA can be simulated by a DFA with at most an exponential growth in size.
- There is a family of regular sets L_n, for $n > 0$, such that L_n is accepted by an NFA of size $O(n)$ but the smallest DFA accepting it is at least of size $O(2^n)$.

Similarly, it is true that:

- 2^{2^n} states are necessary, in general, to simulate an n-state parallel finite automaton (universal-automaton) deterministically (cf. [80]).
- 2^{2^n} states are sufficient, in general, to simulate an n-state alternating finite automaton (AFA) deterministically (cf. [20, 21]).

Note that these results also hold in both the lower and upper bound relations described, so that if nondeterminism is denoted by E (a short form for *exists*)[4], parallelism by A (for *all*)[5] and (E, A) (for alternation)[6], these known results can be summarized as shown in Figure B.1. Thereby, solid lines are assumed to represent one-exponential upper and lower bounds, and transitivity is assumed as well, so that the line 'two-exponential'[7] would lead from (E, A) to $\emptyset$.

Remark B.1 *In the framework of finite automata, E and A are exponentially more powerful features with respect to DFA, independently of each other, and, moreover, their power is additive, i. e., the combined are double-exponentially more succinct than none.*

[3] for the acceptance criteria which are used below

[4] Let configurations $c_1, \dots, c_n$ follow from configuration c in one step. In a nondeterministic machine, the configuration c leads to acceptance if and only if there *exists* a successor c_i which leads to acceptance.

[5] In a universal machine, a configuration c can again reach several configurations $c_1, \dots, c_n$ in one step, but now c leads to acceptance if and only if *all* successors $c_1, \dots, c_n$ lead to acceptance.

[6] Remember that the concept of alternation allows *existential* and *universal* quantification to alternate.

[7] omitted for clarity

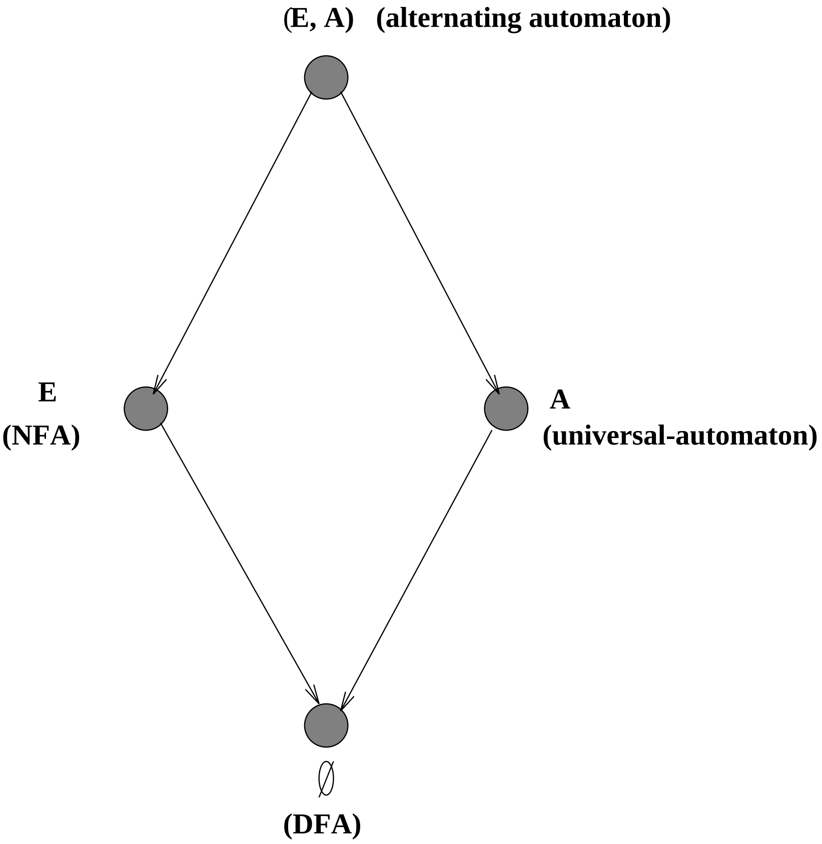


Figure B.1: Bounds for classical automata

B.2 (E, A, C)-automata

In this section, automata augmented with existential (E), universal (A) and bounded concurrency[8] (C) features are defined. The resulting automata are called (E, A, C)-automata, or (E, A, C)-machines. Note that if A and C features are absent, the E-automata are simply NFAs.

Remark B.2 *The finite cooperating automata which are constructed here correspond to statecharts (cf. Chapter 4 and 5) consisting of a single collection of orthogonal components, each of which is merely a finite automaton.*

B.2.1 Definitions

Definition B.1 (E, A, C)-automaton
Let Σ be a finite alphabet. An **(E, A, C)-automaton** *is a tuple $M = (M_1, M_2, \dots M_\nu, \Phi, \Psi)$ for some $\nu \geq 1$, where for all i, $1 \leq i \leq \nu$:*

M_i is a triple (Q_i, q_i^0, δ_i) such that

[8] where a system configuration consists of a bounded number of cooperating (or broadcasting) states of *(orthogonal) components*

Q_i is a finite set of states [9]*,*

$q_i^0 \in Q_i$ is the initial state, and

δ_i is the transition table, a finite subset of $Q_i \times \Sigma \times \Gamma \times Q_i$. Here,

Γ denotes the collection of propositional formulas over a set of propositional variables P where P contains for each state $q \in Q_j$, $1 \leq j \leq \nu$, a propositional variable v_q[10]*.*

Φ is the E-condition, and $\Phi \in \Gamma$

Ψ is the termination condition, and $\Psi \in \Gamma$.

B.2.2 Idea of cooperative automata

M consists of ν automata, the so called (*orthogonal*) *components*. Each component has its own set of states, initial state and transition table. These automata work together in a synchronous way, where transitions taken depend on the (common) input symbol being read, their internal states, and the *condition formular* from Γ. The conditions are interpreted to take on truth values according to the states of (possibly all) the ν components. Φ distinguishes between existential and universal configurations (i. e., between E and A states). Finally Ψ indicates halting configurations.

Definition B.2 Configuration of *M*
A **configuration of *M*** *is an element of $Q_1 \times Q_2 \times \ldots \times Q_\nu \times \Sigma^\star \times I\!N$, where*

$Q_1 \times Q_2 \times \ldots \times Q_\nu$ describes the current state as a ν-tuple of the states of each of the M_i,

$\Sigma^\star$ is the input word and

$I\!N$ indicates the actual position of M in processing the input word.

Remark: For any configuration $(q_1, q_2, \ldots q_\nu, x, m)$, $m \leq |x|$ must hold.

Notation B.1 *A configuration c is said to* **satisfy** *a condition $\gamma \in \Gamma$, if γ evaluates to true when each symbol therein is assigned true iff it appears in c.*

Definition B.3 Behaviour of *M*
Let $x_1 x_2 \ldots x_k$ be a finite word over Σ, let $c = (q_1, q_2, \ldots q_\nu, x, j)$ be a configuration and let $t = (q, a, \gamma, p)$ be a transition in M_i's transition table δ_i for some i, $1 \leq i \leq \nu$.
(1) t is said to be applicable to c, if $x_j = a$, $q_i = q$, and c satisfies γ.
(2) A configuration $c' = (p_1, p_2, \ldots p_\nu, x, m)$ is said to be a successor of c, if for each i there is a transition $t_i = (q_i, x_j, \gamma_i, p_i) \in \delta_i$ that is applicable to c, and $m = j + 1$.

[9] The sets Q_i for $1 \leq i \leq \nu$ are required to be pairwise disjoint.

[10] Usually, q is written instead of v_q. From the context, it will be clear if a propositional variable or a state is meant.

Definition B.4 Existential configuration
A configuration is **existential** *if it satisfies the E-condition Φ, otherwise it is* **universal**. *It is* **accepting**, *iff it satisfies the termination condition Ψ.*

Definition B.5 Computation of $\boldsymbol{M}$
A **computation of $\boldsymbol{M}$** *on an input $x \in \Sigma^\star$ is a tree where each node is labelled with a configuration. The* **root** *is labelled with the* **initial** *configuration $(q_1^0, q_2^0, \ldots, q_\nu^0, x, 1)$ and a node has one successor node for each successor configuration where the nodes are labelled by the corresponding configurations. Nodes are assigned* 1/0 (*accept/reject*) *marks, in a bottom up manner, in a way similar to that defined for alternating finite automata* (*AFAs*) (*see Section B.1.1*):

- *the marks of the successors of an existential node are ORed*[11],
- *the marks of the successors of a universal node are ANDed*[12].

Finally, the input word x is **accepted** *iff the root is marked* 1.

Definition B.6 Size of $\boldsymbol{M}$
The **size of** $M = \{M_1, M_2, \ldots M_\nu, \Phi, \Psi\}$ *is*

$$|M| = |\Phi| + |\Psi| + \sum_{i=1}^{\nu} |M_i|,$$

where the size of a formula in Γ is simply its length in symbols, and the size of each component automaton is defined by

$$|M_i| = |Q_i| + \sum_{(q,a,\gamma,p)\in\delta_i} (3 + length(\gamma)).$$

Remark: *The constant '3' is motivated by the fact that there are three components other than γ, namely q, a and p in the transition table δ_i. Note that it is also important in case $length(\gamma) = 0$ $(true/false)$.*

B.2.2.1 Special cases

Note that if $\nu = 1$, the machine M is simply an alternating finite automaton (AFA), that is, an (E, A)-automaton; if Φ is *true*, then all states are existential, so that M is an NFA (an E-automaton); if Φ is *false*, then all states are universal, so that M is an $\forall$-automaton (an A-automaton); if each configuration has at most one successor, then M is deterministic, i. e., it is defined as a C-automaton.

[11] disjuncted

[12] conjuncted

B.3 (E, A, C)-ω-automata

The (E, A, C)-automata described in the last section generate *terminating* computations which are modelled over a finite alphabet Σ. As opposed to this, (E, A, C)-ω-automata generate *ongoing* computations which are modelled over infinite words of Σ. An ω-automaton is essentially the same as a nondeterministic finite state automaton, but with the acceptance condition modified suitably so as to handle infinite input words. In particular, the acceptance terminology of ω-words over Σ for the (E, A, C)-automata must be defined. The resulting automata are called (E, A, C)-ω-automata, or (E, A, C)-ω-machines.

Generally, different types of ω-automata can be defined by adding an acceptance criterion to the definition of transition tables (cf. Definition B.1). In the following, two acceptance criteria from Rabin [26, 134] and Streett [155] are applied to obtain (E, A, C)-ω-automata. Other variants of acceptance criteria are e. g. *Büchi acceptance* and *Muller acceptance* conditions (see [156]).

Remark B.3 *Note that the* **termination condition** *is now an "accepting pair" (cf. [156]) instead of a single formula $\Psi \in \Gamma$, i. e., it is extended to*

$$\Omega \subseteq (\Gamma \times \Gamma).$$

The modification of the condition is essential so as to handle infinite input words.

Definition B.7 *A* **run** *over a word $x \in \Sigma^\omega$ is* **an infinite sequence $\boldsymbol{r}$** *of successive configurations in the sense of Definition B.3* (2).

Notation B.2 ***inf*$(\boldsymbol{r})$** *denotes the set of configurations appearing in r infinitely often.*

B.3.1 Definition of acceptance criteria

For simplicity, it will be assumed at the beginning, that the machine M is deterministic (total), so that there exists exactly one run per input word x; it will be called $\boldsymbol{r_x}$.

Definition B.8 Rabin acceptance
The machine M ***R*-accepts** *a word $w \in \Sigma^\omega$ with respect to $\Omega \subseteq (\Gamma \times \Gamma)$, if there is a pair $(\Psi_1, \Psi_2) \in \Omega$, such that there is a configuration in inf(r_x) that satisfies Ψ_1, but no configuration in inf(r_x) satisfies Ψ_2.*

Definition B.9 Streett acceptance
The machine M ***S*-accepts** *a word $w \in \Sigma^\omega$ with respect to $\Omega \subseteq (\Gamma \times \Gamma)$, if for each pair $(\Psi_1, \Psi_2) \in \Omega$, if there is a configuration in inf(r_x) that satisfies Ψ_1, then there is also a configuration in inf(r_x) that satisfies Ψ_2.*

In the next subsection, it will be assumed that the machines work on inputs from Σ^ω.

Definition B.10 *A run of a machine M over a word $w \in \Sigma^\omega$ is an* **accepting run** *iff M R-accepts or S-accepts w.*

B.3.2 Extending the definition of the acceptance criteria

Intuitively, the classical Rabin and Streett criteria for DFAs [26, 134, 155] are a special case of the Definitions B.8 and B.9. Take $\nu = 1$ and choose the pairs of conditions (Ψ_1, Ψ_2) as conjunctions that specify the pairs of sets of states required for the classical criteria.

Extending the definition of the acceptance criteria is easy:

- **For** (**E, C**)-ω-**automaton:** M accepts if there is at least one run on x which accepts.
- For (**A, C**)-ω-**automaton:** all runs on x must accept.
- For (**E, A, C**)-ω-**automaton**, the definition of a *trace* is first needed.

  Definition B.11 *A* **trace** *of an (E, A, C)-ω-automaton on a word Σ^ω is a subtree of M's computation tree on x that includes all offsprings of each universal node and one offspring of each existential node.*

  Definition B.12 *M* **accepts** *if there is a trace of M on x for which every path satisfies its corresponding acceptance criterion.*

Notation B.3 *For an (E, A, C)-ω-automaton A, let $\boldsymbol{L^R_\omega(A)} \subseteq \boldsymbol{\Sigma^\omega}$ be the set of words accepted by A on R-acceptance and let $\boldsymbol{L^S_\omega(A)} \subseteq \boldsymbol{\Sigma^\omega}$ be the set of words accepted by A on S-acceptance. Then two (E, A, C)-ω-automata A, B are said to be* **R-equivalent** *iff $L^R_\omega(A) = L^R_\omega(B)$, respectively,* **S-equivalent** *iff $L^S_\omega(A) = L^S_\omega(B)$.*

B.4 Exponential and multi-exponential relations

Next, definitions required in establishing *exponential* and *multi-exponential* relations (gaps) between the various machines are given.

Notation
Let $\boldsymbol{\xi}$ be any subset of $\{E, A, C\}$. Then $\boldsymbol{\xi}$**-automata** and $\boldsymbol{\xi}$-$\boldsymbol{\omega}$**-automata** denote the classes of machines employing the features in ξ.

Definition B.13 *Let ξ_1 and ξ_2 be any subsets of $\{E, A, C\}$. $\boldsymbol{\xi_1} \xrightarrow{\boldsymbol{P}} \boldsymbol{\xi_2}$ (respectively, $\boldsymbol{\xi_1} \xrightarrow{\heartsuit} \boldsymbol{\xi_2}$, $\boldsymbol{\xi_1} \xrightarrow{\heartsuit\heartsuit} \boldsymbol{\xi_2}$, or $\boldsymbol{\xi_1} \xrightarrow{\heartsuit\heartsuit\heartsuit} \boldsymbol{\xi_2}$) is used, if there is a polynomial p (and a constant $k > 1$, respectively) such that for any ξ_1-automaton M_1 of size n there is an equivalent ξ_2-automaton M_2 of size no more than $p(n)$ (respectively, $k^{p(n)}, k^{k^{p(n)}}$, or $k^{k^{k^{p(n)}}}$).*

Definition B.14 *Let ξ_1 and ξ_2 be any subsets of $\{E, A, C\}$. $\boldsymbol{\xi_1} \xrightarrow[\heartsuit]{} \boldsymbol{\xi_2}$ (respectively, $\boldsymbol{\xi_1} \xrightarrow[\heartsuit\heartsuit]{} \boldsymbol{\xi_2}$, or $\boldsymbol{\xi_1} \xrightarrow[\heartsuit\heartsuit\heartsuit]{} \boldsymbol{\xi_2}$) is used, if there is a family of regular languages L_n for $n > 0$, a polynomial p and a constant $k > 1$, such that L_n is accepted by a ξ_1-automaton M_1 of size $p(f(n))$ for some monotonically-increasing function f, but the smallest ξ_2-automaton M_2 accepting it is at least of size $k^{f(n)}$ (respectively, $k^{k^{f(n)}}$ or $k^{k^{k^{f(n)}}}$).*

Remark B.4 *When a small R or S is added to the arrows in the Definitions B.13 and B.14, then the corresponding relations are meant to be applied to ξ-ω-automata rather than ξ-automata, and to consider R-equivalence or S-equivalence, respectively, instead of normal equivalence.*

B.5 Results for the $\Sigma^\star$-case

In Section B.1.4, known bounds for classical automata regarding succinctness were given. In this section, the results for the $\Sigma^\star$-case are presented. Figure B.2 depicts the results for the $\Sigma^\star$-case. These were first investigated in [31, 32].

The first set of results establishes the solid lines of Figure B.2 and all transitivity consequences thereof. For instance, these include exponential upper and lower bounds for simulating nondeterministic concurrent machines (e. g. nondeterministic statecharts) on NFAs; double exponential bounds for simulating them on DFAs, and a *triple*-exponential bound for simulating *alternating* concurrent machines on DFAs. Above all, the solid lines of Figure B.2 show that bounded concurrency (C) represents a third, separate, exponentially powerful feature. The feature C is independent of conventional nondeterminism (E) and parallelism (A), since the savings remain intact in the terms of any combination of A and E. Furthermore, C is *additive* with respect to the two, by virtue of the double- and triple-exponential bounds along the appropriate lines in the figure. Actually, this result is of much interest, as it shows that among other things the unbounded nature of pure AND of alternation prevents it from being subsumed by the bounded AND of the C feature, and the cooperative nature of AND in the C feature[13] prevents it from being subsumed by noncooperative AND of alternation.

The next set of results considers a rather more delicate problem, namely, the comparison of C with A and E using the same measures, i. e., exponential discrepancies in succinctness. For instance, the results above do not reveal anything about the possibility of an exponential gap between E and C. The four thick dashed lines and the thin doubly dashed line in Figure B.2 demonstrate these relations. Each of the thick singly dashed lines denotes exponential upper and lower bounds for the simulation in the downward direction and polynomial bounds for the upper direction. In particular, the nondeterministic statecharts (cf. [50]) are shown to be exponentially more succinct than AFAs, and the same holds when nondeterminism is absent from both. The thin doubly dashed line represents upper and lower bounds in both directions, meaning that alternation and bounded concurrency can be simulated by each other with at most an exponential growth in size, and that, in general, neither exponential gap can be eliminated. Indeed, these results appear to be the most interesting, as they show that bounded concurrency is actually *more* powerful than each of pure parallelism or nondeterminism taken alone, and is comparable in power to the combination.

Step by step the results presented in Figure B.2 will be established in Subsection B.5.1 and B.5.2.

[13] as embodied by the joint transitions in Petri nets or statecharts, for example

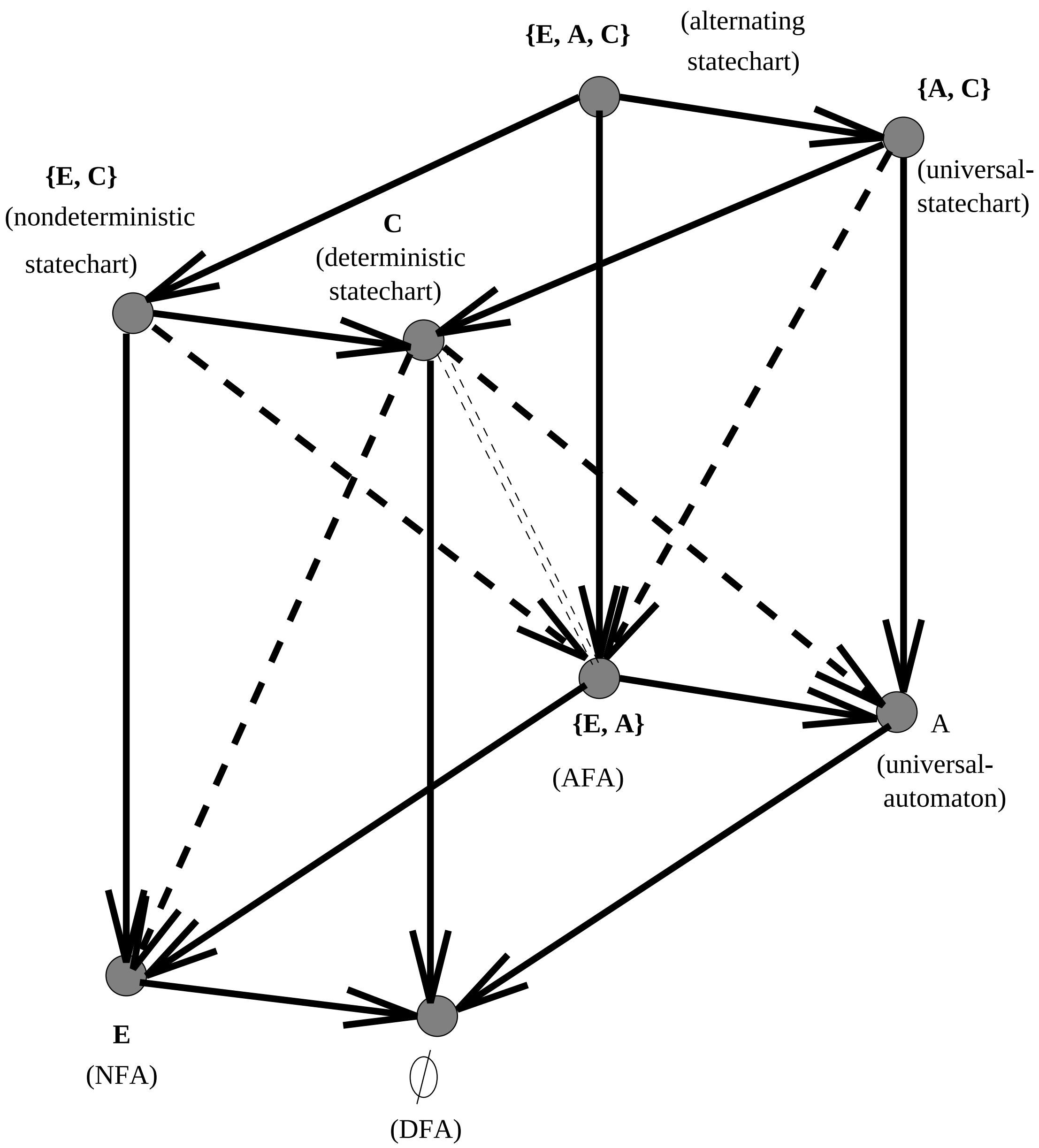


Figure B.2: Results for the $\Sigma^\star$-case

B.5.1 Upper bounds for the $\Sigma^\star$-case

The first set of results establishes the upper bounds for the vertical solid lines of Figure B.2.

Remark B.5 *In the following a* **mapping** [] *is used, which assigns* $(\xi, C)\longrightarrow\xi := \xi\cup\{C\}$, *so that for instance* $[\{E, A\}, C] = \{E, A, C\}$.

Note *In the rest of this Appendix,* **Proof** *will be used to imply the idea of the proof but not an exact proof.*

Proposition B.1 *Let* ξ *be any subset of a* $\{E, A\}$. *Then* $[\xi, C] \overset{\heartsuit}{\longrightarrow} \xi$.

Proof
The idea lies in simulating the behaviour of (ξ, C)-automaton by a ξ-automaton whose set of states is the Cartesian product of the states in component machines $M_1, M_2, \ldots, M_\nu$. By definition, if M is deterministic, nondeterministic, or alternating, the resulting machine will as well be of the corresponding type. □

Next, polynomial upper bounds for the upward direction of the four thick dashed diagonal lines of Figure B.2 are established. For example replacing nondeterminism by bounded concurrency will only polynomially increase the size.

Proposition B.2 *Let ξ be $\emptyset$ or $\{A\}$. Then $[\xi, E] \xrightarrow{P} [\xi, C]$. The same holds with A and E exchanged.*

Proof Simulating E by C involves mimicking the classical subset construction for eliminating nondeterminism using a collection of orthogonal components, one for each of the n states in the original NFA. For any state p therein, the corresponding component has two states, indicating, whether the NFA is in p or not in p, respectively. Subsets of the original state are represented by yes / no combinations. When a is received as an input, and p's component is in its no-state, it moves into its yes-state if any of its a-predecessors in the original NFA is in its yes-state right now. Dually, if p is in its yes-state, it moves to its no-state when a arrives, if all of its a-predecessors in the original NFA are in their no-state. Apparently, the machine accepts if and only if at least one of the components that represent a final state of the NFA is in its yes-state.

The simulation of an A-machine by a C-machine is identical, except for the acceptance condition, which is satisfied only if *all* the components representing the final states are in their yes-states.

Simulating an (E, A)-machine by an (E, C)-machine is a little more subtle. The basic idea is as above, i. e. to mimic the subset construction using components with yes/no-state. However, one has AND-branchings in the original machine that have to be maintained in the simulation. In this case, one has to incorporate all such branchings one step ahead of time, at the moment a universal state is entered. For a more rigorous description see [32].

The simulation of an (E, A)-machine by an (A, C)-machine is dual. □

Tracing one vertical solid line followed by an upward thick dashed diagonal leads to exponential upper bounds for the four horizontal solid lines in the upper portion of Figure B.2, in anology with what was known[14] for the lower portion of it:

Collary B.1 *Let ξ be $\emptyset$ or $\{E\}$. Then $[\xi, A, C] \xrightarrow{\heartsuit} [\xi, C]$. The same holds with A and E exchanged.*

The upper exponential bounds going downward along the four thick dashed diagonals and along the thin doubly dashed diagonal are as well obtained by moving down a vertical solid line and noticing that the resulting machines are special cases of the target ones:

[14] cf. Subsection B.1.4

Collary B.2
1) *Let* ξ *be* $\emptyset$ *or* $\{A\}$. *Then* $[\xi, C] \xrightarrow{\heartsuit} [\xi, E]$. *The same holds with* A *and* E *exchanged.*
2) $\{C\} \xrightarrow{\heartsuit} \{E, A\}$.

The exponential upper bound along the upward direction of the thin doubly dashed diagonal follows by tracing one solid and one thick dashed line in the figure:

Collary B.3 $\{E, A\} \xrightarrow{\heartsuit} \{C\}$.

B.5.2 Lower bounds for the $\Sigma^\star$-case

Now to establish the exponential lower bounds represented in all the solid lines of Figure B.2, as well as the double-exponential ones[15], it suffices to establish the *triple*-exponential lower bound for the simulation of (E, A, C)-machines by deterministic finite automata (i. e., $\emptyset$-automata). Then all the former bounds[16] follow immediately, since any violation would contradict either the *triple*-exponential lower bound or the previously established upper bounds.

Proposition B.3 $\{E, A, C\} \xrightarrow[\heartsuit\heartsuit\heartsuit]{} \emptyset$.

Proof The idea is to exhibit a family of regular sets K_n for $n > 0$ such that each K_n is accepted by an (E, A, C)-automaton of size $O(log^2\ n)$ but the smallest DFA accepting it is at least of size 2^{2^n}. For a detailed proof see [20, 32, 104]. □

The next results deal with the exponential lower bounds on both directions of the thin doubly dashed diagonal. The bound for the upward direction follows directly from the double-exponential lower bound $\{E, A\} \xrightarrow[\heartsuit\heartsuit]{} \emptyset$ and the one-exponential upper bound $\{C\} \xrightarrow{\heartsuit} \emptyset$:

Collary B.4 $\{E, A\} \xrightarrow[\heartsuit]{} \{C\}$.

To deal with the *downward* direction of the thin doubly dashed diagonal, which is more subtle, one has to exhibit a polynomial-size C-automaton. This, however, requires exponentially many states as in an alternating finite automaton. For this part, a technical lemma relating AFAs to DFAs is required (see also [21]):

Lemma B.1 *Let* M *be an AFA of size* n*, accepting the language* L. *There is a DFA with no more than* 2^n *states that accepts the language* L^r*, that is, the set consisting of the words of* L *in reverse.*

Proof see [21]. □

Proposition B.4 $\{C\} \xrightarrow[\heartsuit]{} \{E, A\}$.

[15] implicit in the appropriate compound transitive paths

[16] i. e., the exponential lower bounds as well as the double-exponential ones

Proof Construct the reversed version of the sets used in the proof of Proposition B.3. By Lemma B.1, the smallest AFA accepting the nonreversed version must have at least 2^n states. To complete the proof, one describes the operation of the C-automaton (e. g., a deterministic statechart) that accepts the nonreversed version. It "stores" the first word it reads, w, in binary form[17], and then checks each subsequent word, symbol by symbol, for equality with w. It is easy to construct this machine to be of linear size. □

A close study of the four thick dashed diagonals shows that the exponential lower bounds in the downward direction follow immediately from Proposition B.4, since E-machines and A-machines are special cases of (E, A)-machines, C-machines are special cases of (E, C)-machines and (A, C)-machines. In particular, while nondeterminism can be replaced by bounded concurrency without essential blowup in size (cf. Proposition B.2), the converse is not true; the C feature is strictly stronger than E or A.

Equally important is the study of "sideway" diagonals that involve moving from $\{A, C\}$ to $\{E\}$ and from $\{E, C\}$ to $\{A\}$.

Proposition B.5 $\{A, C\} \underset{\heartsuit\heartsuit}{\longrightarrow} \{E\}$ *and* $\{E, C\} \underset{\heartsuit\heartsuit}{\longrightarrow} \{A\}$.

Proof Exhibit a family of regular sets S_n for $n > 0$ such that each S_n is accepted by an (A, C)-automaton of size $O(log\ n)$, but the smallest E-automaton (NFA) accepting it has at least 2^n states. Define

$$S_n = \{w\$w | w \in \{0, 1\}^n\}.$$

It easy to show that any NFA that accepts S_n must have at least 2^n states: Given such an NFA, associate with each $w \in \{0, 1\}^n$ a state, by choosing some accepting computation of the word $w\$w$ and singling out the state reached after reading the first w. Call it $q(w)$. Knowing that there are 2^n different w's, if the automaton had less than 2^n states there would be $w \neq u$ with $q(w) = q(u)$. Thus the word $w\$w$ would be acceptable. On the other hand, an (A, C)-automaton of size $O(log\ n)$ can be constructed to accept S_n, similar to the (E, A, C)-automaton in the proof of Proposition B.3.

The case with E and A computed simultaneously is proved similarly, using the complement of S_n. □

The next proposition considers lower bounds in the *reverse* directions of all the solid and thick dashed lines of Figure B.2. These are the arrows with polynomial (mostly linear) upper bounds.

Proposition B.6 *There is a family of regular sets F_n for $n > 0$ such that each F_n is accepted by a DFA of size $O(n)$, but the smallest (E, A, C)-automaton accepting it is at least of size n.*

Proof Use simple one-word languages. Let

$$F_n = \{a_1 a_2 \ldots a_n\}.$$

[17] by n orthogonal yes / no components

Even the most powerful (E, A, C)-automaton requires each of the a_i to appear on at least one edge, otherwise it can easily be shown to misbehave. A trivial DFA with n states accepts F_n. $\square$

Finally, due to time factor, the results for the Σ^ω-case cannot be presented here. Moreover, these results are not quite necessary for the comparisons of our concepts developed in Subsection 7.7.3 with other related techniques discussed in Subsection 7.6.

Bibliography

[1] Alur, R.; Dill, D. L.: A theory of timed automata. In: *Theoretical Computer Science*, 126: 1994, 126–235.

[2] Berry, G.; Cosserat, I.: The Synchronous Programming Language ESTEREL and its Mathematical Semantics. In: *Proc. of the CMU Seminar on Concurrency*, Lecture Notes in Computer Science, Vol. 197, Springer-Verlag, New York, 1987, 389–449.

[3] Burch, J. R.; Clarke, E. M.; McMillan, K. L.; Dill, D. L.: Sequential Circuit Verification using Symbolic Model Checking. In: *27th ACM/IEEE Design Automation Conference*, 1990, 41–51.

[4] Barringer, H.; Kuiper, R.; Pnueli, A.: Now you may compose Temporal Logic specifications. In: *Proc. 16th Ann. Symp. on Theory of Computing*, 1984, 51–63.

[5] Beizer, B.: Software Testing Techniques. (2^{nd} *Edition*), Van Nonstrand Reinhold, 1990.

[6] Bergè, J.-M.; Levia, O.; Rouillard, J.: High-level System Modelling. Kluwer Academic Publishers, 1995.

[7] Bernhardt, M.: Ein Ansatz zur Synchronisationsanalyse nebenläufiger Programmsysteme. Diplomarbeit, Informatik LS1, Universität Dortmund, 1995.

[8] Bernhardt, P.: A reduced test suite for protocol conformance testing. In: *ACM Trans. Soft. Eng. Method.*, Vol. 3, N0: 3, 1994, 201-220.

[9] Bernstein, A.; Harter, P. K.: Proving real-time properties of programs with temporal logic. In: *Proceedings of ACM SIGOPS 8th Annual ACM Symposium on Operating Systems Principles*, December 1981, 1–11.

[10] Berthomieu, B.; Diaz, M.: Modelling and verification of time dependent systems using time Petri nets. In: *IEEE Trans. on Software Eng.*, **17**(3), March 1991, 259–273.

[11] Berry, G.; Gonthier, I.: The Esterel Synchronous Programming Language: Design, Semantics, Implementation. In: *Science of Computer Programming*, 19(2): 1992, 87–152.

[12] Bremond-Gregoire, P.; Choi, J. Y.; Lee, I.: The soundness and completeness of ACSR. Automated Test Set Generation for Statecharts. In: *Technical Report MS-CIS-93-59, University of Pennsylvannia*, June 1993.

[13] Boehm, B. W.: The high cost of software. In: *Practical Strategies for Developing Large Software Systems*, E. Horwitz, Ed. Reading, MA: Adison-Wesley, 1975.

[14] Boehm, B. W.: Software Engineering Economics. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[15] Bogdanov, K.; Holcombe, M.; Singh H.: Automated Test Set Generation for Statecharts. In: *International Workshop on Current Trends in Applied Formal Methods, Bundesamt für Sicherheit in der Informationstechnik (BSI) Boppard*, 7. - 9. October 1998.

[16] Booch, G.: Object-Oriented Analysis and Design, with Applications. Benjamin/Cummings 1991.

[17] Böhling, K. H.; Schütt, D.: Endliche Automaten II. Skripten zur Informatik, BI, Mannheim, 1969.

[18] Bruno, G.: Model-Based Software Engineering. Chapman-Hall, London, 1995.

[19] Caspi, P.; Piland, D.; Halbwachs, N.; Plaice, J.: Lustre: A declarative language for programming synchronous systems. In: *Proc. 14th ACM Symp. on Principles of Programming Languages*, 1987, 178–188.

[20] Chandra, A. K.; Stockmeyer, L. J: Alternation. In: *Proceedings of the 17th IEEE Symposium on Foundations of Computer Science*, IEEE Press, New York, 1976, 98–108.

[21] Chandra, A. K.; Kozen, D.; Stockmeyer, L. J: Alternation. *Journal of the ACM*, Vol. 28, No: 1, January 1981, 114–133.

[22] Chow, T. S.: Testing Software Design Modelled by Finite-State Machines. IEEE Transactions on Software Engineering Vol. SE-4, No. 3, May, 1978, 178–187.

[23] Clarke, E. M.; Emerson, E. A.; Sistla, A. P.: Automatic verification of finite state concurrent systems using temporal logic specifications. *Journal of the ACM Transactions of Programming Languages and systems* Vol. 8, No. 2, April 1986, 244–263.

[24] Clarke, L. A.; Podgurski, A.; et al : A Formal Evaluation of Data Flow Path Selection Criteria. *IEEE Transactions on Software Engineering* 11, Vol. 15, Nov. 1989, 1318–1332.

[25] Clements, P.; Heitmeyer, C.; Labaw, B.; Rose, A.: MT: A toolset for specifying and analysing real-time systems. In: *Proceedings, Real-Time Systems Symp.*, Raleigh, NC, 1993.

[26] Choueka, Y: Theories of automata on ω-tapes: A simplified approach. In: *Journal of Computer Syst. Sci.* 8, 1974, 117–141.

[27] Coleman, D.; Hayes, F.; Bear, S.: Introducing Objectcharts, or How to Use Statecharts in Object Oriented Design. *IEEE Transactions on Software Engineering* 18, 1992, 9–18.

[28] Craigen, D.; Gerhart, S.; Ralston, T.: Formal Methods Reality Check: Industrial Usage. In: *Proc. of Formal Methods Europe '93*, Lecture Notes in Computer Science, Vol. 670, Springer-Verlag, 1993, 250–267.

[29] Douglass, BP.: UML statecharts. *Embedded Systems Programming*, San Francisco, Calif. Vol. 12, No: 1, Addition-Wesley-Longman, Reading, MA, USA Jan. 1999, 22–42.

[30] Dittrich, G.: Spezialvorlesung "Strukturierte Petrinetze" SS 1998. *http://lrb.cs.uni-dortmund.de/Lehre/Petri2_SS98/.*

[31] Drusinsky, D.; Harel, D.: On the Power of Cooperative Concurrency. In: *Proceedings of Concurrency*, Lecture Notes in Computer Science, Vol. 335, Springer-Verlag, New York, 1988, 74–103.

[32] Drusinsky, D.; Harel, D.: On the Power of Bounded Concurrency I: Finite Automata. *Journal of the ACM*, Vol. 41, No: 3, May 1994, 517–539.

[33] Emerson, E. A.: Temporal and Modal Logic. in: *Handbook of Computer Science — Formal Models and Semantics*, Elsevier Science Publishers, Amsterdam, New York, Oxford, Tokyo, Volume B, 1990, chapter 10.

[34] Emerson, E. A.; Halpern, Joseph Y.: "Sometimes" and "Not Never" revisited: On Branching versus Linear Time Temporal Logic. *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Lanuages (POPL)*, A. Demers, ACM Press, Austin, TX US, January 1983, 127–140.

[35] Emerson, E. A.; Halpern, Joseph Y.: "Sometimes" and "Not Never" revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, Vol. 33, No: 1, january 1986, 151–178.

[36] Feldman, Y. A.; Schneider, H.: Simulating Reactive Systems by Deduction. *ACM Trans. Soft. Eng. Method.*, Vol. 2, April 1993, 128–175.

[37] Fujiwara, S.; Bochmann, G.; Khendek, F.; Amalou, M.; Ghedamsi, A.: Test Selection Based on Finite State Models. *IEEE Transactions on Software Engineering* Vol. 17, No. 6, June, 1991, 591–603.

[38] Gabbay, Dov M.; Hodkinson, I.; Reynolds, M.: Temporal Logic: Mathematical Foundations and computational Aspects. Clarendon Press, Oxford, Vol. 1, 1994.

[39] Gabow, H. N.; Maheshwari, S. N.; Osterweil, L. J.: On Two Problems in the Generation of Program Test Paths. In: *IEEE Transactions On Software Engineering*, Vol. SE-2, No. 3, September 1976, 227–231.

[40] Genrich, H.: Predicate/transition nets. In: *W. Reising and G. Rozenberg, editors, Advances in Petri Nets*, Springer-Verlag, Berlin, 1987, 254–255.

[41] Ghezzi, C.; Jazayeri, M.; Mandrioli, D.: Fundamentals of Software Engineering. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[42] Gibbs, W. W.: Software's Chronic Crisis. *Scientific American*, September 1994, 72–81.

[43] Gonenc, G.: A method for the design of fault-detection experiments. In: *IEEE Transactions Comput.*, Vol. C-19, June 1970, 551–558.

[44] Gorlen, K. E.; Orlow, S. M.; Plexico, P. S.: Data Abstraction and Object-Oriented Programming in C++. John Wiley & Sons, 1990.

[45] Gregor, G. L.; Bochmann, G.; Petrenko, A.: Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalised Wp-Method. *IEEE Transactions on Software Engineering* Vol. 20, No. 2, February, 1994, 149–162.

[46] Grimm, K.: Methoden und Verfahren zum systematischen Testen von Software. *Automatisierungstechnische Praxis*, 30(6), 1990, 271–280.

[47] Guernic, P. le; Benveniste, A.: Real-time, Synchronous, Data-Flow Programming: The Language Signal and its Mathematical Semantics. Technical Report 620, INRIA, Rennes, 1986.

[48] Halmos, P. R.: Naive Set Theory. New York, Springer-Verlag, 1960.

[49] Harary, F.: Graph Theory. Addison-Wesley, Reading, Mass., 1969.

[50] Harel, D.: Statecharts: A visual formalism for complex *systems. Sci. Comput. Program* 8, 1987, 231–274.

[51] Harel, D.; Pnueli, A.; Schmidt, J. P.; Sherman, R.: On the formal semantics of statecharts. in: *Proc. 2nd IEEE Press.* 1987, 54–64.

[52] Harel, D.; Gery, E.: Executable Object Modelling with Statecharts. *Proc. 18th Int. Conf. Softw. Eng.*, Berlin, März 1996, 246–256.

[53] Harel, D.; Gery, E.: Executable Object Modelling with Statecharts. In: *IEEE Computer*, July 1997, 31–42.

[54] Harel, D.; Naamad, A.: The STATEMATE Semantics of Statecharts. *ACM Trans. Soft. Eng. Method.* **5**:4, Okt. 1996.

[55] Harel, D.; Lachover, H.; Pnueli, A. et al.: STATEMATE: A Working Environment for the Development of Complex Reactive *Systems. IEEE Transactions on Software Engineering* 16, 1990, 403–414.

[56] Harel, D.; Pnueli, A.: On the Development of Reactive Systems. In: Logics and Models of Concurrent Systems, K. R. Apt (Ed.), Springer-Verlag, 1985, 477–498.

[57] Hierons, R: Extending test sequence overlap by invertibility to test sequences. In: *COMPJ: The Computer Journal* 39(4), 1996, 325–330.

[58] Hierons, R: Testing from a finite state machine: extending invertibility to sequences. In: *COMPJ: The Computer Journal* 40(4), 1997.

[59] Hierons, R: Testing from semi-independent communicating finite state machine with slow environment. In: *IEEE Proceedings on Software Engineering* 144(5-6), 1997.

[60] Harrold, M.J.; Soffa, M. L.: Interprocedural data flow testing. In 3^{rd} *Testing, Analysis and Verification Symp.*, December 1989, 158–167.

[61] Heitmeyer, C.; Mandrioli, D.: Formal Methods for Real-Time Computing, Volume 5 of Trends in Software. Wiley, 1996.

[62] Hoare, C.A.R.: An axiomatic basis for computer programming. *Communication of the ACM* 12(10), October 1969, 576–580.

[63] Hoare, C.A.R.: Communicating Sequential Processes. *Communication of the ACM* 21, 1978, 666–677.

[64] Hopcroft, J. E.; Ullman, J. D.: Formal Languages and their Relation to Automata. Addison-Wesley, Mass., 1969.

[65] Hopcroft, J. E.; Ullman, J. D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Mass., 1974.

[66] Hopcroft, J. E.; Ullman, J. D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, 1979.

[67] Howden, W.E.: Methodology for the the generation of program test data. *IEEE Transactions on Computers* 1, Vol C-24, No. 5. May 1975, 554–559.

[68] Howden, W.E.: Functional Program Testing and Analysis. McGraw-Hill, NY, 1987.

[69] Huizing, C.; de Roever, W. P.: Introduction to design choices in the semantics of statecharts. In: *Inf. Proc. Lett. 37*, 1991, 205–213.

[70] Huizing, C.; Gerth, R.; de Roever, W. P.: Modelling statecharts in a fully abstract way. In: *Proc. 13th CAAP*, Lecture Notes in Computer Science, Vol. 229, Springer-Verlag, 1988, 271–294.

[71] Huizing, C.; Gerth, R.: On the Semantics of Reactive Systems. In: *Proceedings of REX workshop 'Real-Time Theory in Practice'*, Lecture Notes in Computer Science, Vol. 600, Springer-Verlag, Berlin, 1992, 291–314.

[72] The languages of STATEMATE, i-Logix Inc., Burlington, MA, Tech. Rep., 1987.

[73] The semantics of statecharts, i-Logix Inc., Burlington, MA, Tech. Rep., 1989.

[74] The STATEMATE approach to complex systems, i-Logix Inc., Burlington, MA, Tech. Rep., 1989.

[75] Jahanian, F.; Mok, A.: Modechart: a specification language for real-time systems. In: *IEEE Transactions on Software Engineering*, 20(12), December 1994, 933–947.

[76] Jard, C.; Jeron T.: On-line Model Checking for Finite Linear Temporal Logic Specifications. *Automatic Verification Methods for Finite State Systems: Proceedings of the International Workshop*, Lecture Notes in Computer Science 407, Sifakis Joseph, Springer-Verlag, New York, Grenoble, France, 1990.

[77] Kemmerer, R. A.: Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering* SE-11, No. 1, January, 1985, 32–43.

[78] Kesten, Y.; Pnueli, A.: Timed and Hybrid Statecharts and their Textual Representation. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, Vol. 571, Springer-Verlag, 1992, 591–619.

[79] Kesten, Y.; Pnueli, A.: Verifying Clocked Transition Systems. In: *Hybrid Systems III*, Lecture Notes in Computer Science, Vol. 1066, Springer-Verlag, 1996, 13–40.

[80] Kozen, D.: On Parallelism in Turing Machines. In: *Proceedings of the 17th IEEE Symposium on Foundations of Computer Science*, IEEE Press, New York, 1976, 89–97.

[81] Koymans, R.: Specifying real-time properties with metric temporal logic. In: *Real-time Systems*, 2(4): 1990, 255–299.

[82] Koymans, R.; Vytopil, J.; de Roever, W. P.: Real-Time programming and asynchronous message passing. *Proceedings of the Second Annual Symposium on Principles of Distributed Computing (An extended version appeared in Information and Computation, Volume 79, Number 3, December 1988)*, Montreal, August 1983, 187–197.

[83] Kröger, F.: Temporal Logic of Programs. Springer-Verlag, New York, Berlin, Heidelberg, London, Paris, Tokyo.

[84] Kyeyune, Y.: Generating Test Sequences for Statecharts in STATEMATE. Diplomarbeit, Informatik LS1, Universität Dortmund, 1995.

[85] Kyeyune, Y.: Analysis of Statechart Models. 16. Workshop *"Interdisziplinäre Methoden in der Informatik" 16. – 19. 9. 1996, Haus Nordhelle, Meinerzhagen-Valbert.* Forschungsbericht Nr. 639, Fachbereich Informatik, Universität Dortmund, Januar 1997.

[86] Kyeyune, Y.: Konzepte für Interagierende Module. 17. Workshop *"Interdisziplinäre Methoden in der Informatik" 8. – 11. 9. 1997, Jugendburg Gemen, Borken.* Forschungsbericht Nr. 682, Fachbereich Informatik, Universität Dortmund, Dezember 1998.

[87] Kyeyune, Y.: Test Methods for Complex Systems. 18. Workshop *"Interdisziplinäre Methoden in der Informatik" 21. – 24. 9. 1998, Haus Nordhelle, Meinerzhagen-Valbert.* Forschungsbericht Nr. 718, Fachbereich Informatik, Universität Dortmund, März 2000.

[88] Kyeyune, Y.; Riedemann, E: Analyse und Test von Modellen reaktiver Systeme. 7. Kolloquium Software-Entwicklung - Methoden, Werkzeuge und Erfahrungen – 23. – 25. September, Ostfildern. Technische Akademie Esslingen, 1997.

[89] Levi, S. T.; Agrawala, A.: Real Time System Design. McGraw Hall International Editions, 1990.

[90] Lamport, L.: The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, Vol 16., No. 3, Mai 1994, 872–923.

[91] Lamport, L.: "Sometimes" is sometimes "not never": A Tutorial on the temporal logic of programs. *In: Proceedings of the Seventh Annual Symposium on Principles of Programming Languages (POPL)*, ACM SIGACT–SIGPLAN, ACM, January 1980, 174–185.

[92] Laski, J. W.; Korel, B.: A data oriented program testing strategy. *IEEE Transactions on Software Engineering* Vol. SE-9, No. 3, May, 1983, 347–354.

[93] Lengauer, T.: Combinatorial Algorithms for Integrated Circuit Layout. John Wiley & Sons, Chichester, 1990.

[94] Leverson, N. G; Turner, C. S.: An investigation of the Therac-25 accidents. *Computer*, 25(7), 1993 18–41.

[95] Leverson, N. G.; Heimdahl, M. P. E.; Hildreth, H.; Reese, J. D.: Requirements Specification for Process-Control Systems. In: *IEEE Transactions on Software Engineering*, 20(9), September 1994, 684–707.

[96] Lippman, S. B.: C++ Primer. Second Edition, Addison-Wesley, 1991.

[97] Lpate, F.; Holcombe, M.: An integration testing method that is proved to find all faults. Automated Test Set Generation for Statecharts. In: *International Journal on Computer Mathematics*. 63, 1997, 159–178.

[98] Lichtenstein, O.; Pnueli, A.; Zuck, L: The glory in the past. In: *Conference on Logics of Programs*, Lecture Notes in Computer Science (LNCS) Vol. 193, Springer-Verlag, Berlin, 1985, 196–218.

[99] Maggiolo-Schettini, A.; Peron, A.: Retiming Techniques for Statecharts. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, Vol. 571, Springer-Verlag, 1996, 55–71.

[100] Maraninchi, F.: Argonaute: Graphical Description, Semantics and Verification of Reactive Systems by Using a Process Algebra. In: *Proc. Automatic Verification Methods for Finite State Sytems*, J. Sifakis (Ed.), Comput. Sci., Vol. 407, Springer-Verlag, Berlin, 1989, 38–53.

[101] Maraninchi, F.: The Argos Language: Graphical Representation of Automata and Description of Reactive Systems. In: *IEEE Workshop on Visual Languages*, October 1991.

[102] Maraninchi, F.: Operational and Compositional Semantics of Synchronous Automaton Compositions. In: *CONCUR'92*, Lecture Notes in Computer Science, Vol. 630, Springer-Verlag, 1992, 550–564.

[103] Merlin, T. S.: Testing Software Design Modelled by Finite-State Machines. IEEE Transactions on Software Engineering Vol. SE-4, No. 3, May, 1978, 178–187.

[104] Meyer, A. R.; Fischer M. J.: Economy of description by automata, grammars, and formal systems. In: *Proceedings of the 12th IEEE Symposium on Switching and Automata Theory*, 1971, 188–191.

[105] Milner, R.: A Calculus of Communicating Systems. Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, Berlin, 1980.

[106] Milner, R.: Communication and Concurrency. Prentice-Hall, New York, 1989.

[107] Mikk, E.; Lakhnech, Y.; Siegel, M.: Hierarchical Automata as Model for Statecharts (Extended Abstract). In: *Advances in Computing Science — ASIAN'97, Third Asian Computing Science Conference*, Proceedings Springer-Verlag, Berlin, Germany, December 9–11 1997, 181–196.

[108] Myers, G. J.: Software reliabilty: principles and practices. Wiley-Interscience, New York, 1976.

[109] Myers, G. J.: Methodisches Testen von Programmen. R. Oldenbourg Verlag, München, 1989.

[110] Naito, S.; Tsunoyama, M.: Fault detection for sequential machines by transition-tours. In: *Proceedings, FTCS* (*Fault Tolerant Comput. Syst.*), 1981, 238–243.

[111] Narayan, S.; Vahid, F.; Gajski, D. D.: System Specification and Synthesis with the SpecCharts Language. In: *Proceedings, IEEE International Conference on Computer-Aided Design (ICCAD) '91*, November, 11–14 1991, 266–269.

[112] Neumann, P. G.: Computer Related Risks. Addison-Wesley, 1995.

[113] Nobe, CR.; Warner, WE.: Lessons Learned from a Trial Application of Requirements Modelling Using Statecharts. In: *International Conference on Requirements Engineering*, 2, Colorado Springs, Colo.: Proceedings Los Alamitos, Calif., 1996.

[114] Ntafos, S. C.: On required element testing. *IEEE Transactions on Software Engineering* 6, Vol. SE-10, November, 1984, 795–803.

[115] Ntafos, S. C.: A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering* 6, Vol. SE-14, June, 1988, 868–874.

[116] Ntafos, S. C.; Hakimi, S. L.: On Path Cover Problems in Digraph and Applications to Program Testing. *IEEE Transactions on Software Engineering* 5, Vol. SE-5, September 1979, 520–529.

[117] Ntafos, S. C.; Hakimi, S. L.: On Structured Digraphs and Program Testing. *IEEE Transactions on Computers* 1, Vol C-30, January 1981, 67–77.

[118] Ntafos, S. C.; Gonzalez, T.: On the Computational Complexity of Path Cover Problems. *Journal of Computer and Science System* 29, 1984, 225-242. January, 1981, 67-77.

[119] Peterson, J. L: Modelling of parallel systems, Digital System Lab., Stanford Univ., Technical Report No. SU-SEL-74-006, 1973, 241pp.

[120] Ostroff, J. S.: Temporal Logic For Real-Time Systems. Research Studies Press Ltd., Tauton Somerset, England, 1989.

[121] Ostroff, J. S.: Formal methods for the specification and design of real-time safety-critical systems. In: *Journal of Systems and Software*, 33(66), April 1992, 890–904.

[122] Ostroff, J. S.: Visual tools for verifying real-time systems. In: *T. Rus and C. Rattray, editors, Theories and Experiences for Real-Time System Development, AMAST Series in Computing*, Vol. 2, Singapore, World Scientific Publishing Co., 1994.

[123] Peterson, J. L.: Petri Net Theory and Modelling of Systems. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[124] Petersohn, C.; Urbina, L.: A Timed Semantics for the STATEMATE Implementation of Statecharts. In: *FME '97: Industrial Applications and Strengthened Foundations of Formal methods*, Lecture Notes in Computer Science, Vol. 1313, Springer-Verlag, 1997, 553–572.

[125] Pnueli, A.: The temporal logic of programs. In: *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, Springer-Verlag, Berlin, 1977, 45–57.

[126] Pnueli, A.: Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In: *Current Trends in Concurrency*, de Bakker et al. (Eds.), Lect. Notes in Comput. Sci., Vol. 224, Springer-Verlag, Berlin, 1986, 510–584.

[127] Pnueli, A.: Linear and branching structures in the semantics and logics of reactive systems. In: *12th Internat. Coll. on Automata Languages and Programming*, A. Demers, Springer-Verlag, Berlin, 1985, 15–32.

[128] Pnueli, A.: In a transition from global to modular reasoning about concurrent programs. In: *Logics and Models of concurrent Systems*, Krzysztof R. Apt, Springer-Verlag, Berlin, 1986.

[129] Pnueli, A.; Manna, Z.: The Temporal logic of Reactive and Concurrent Systems. Specification. — ISBN 0–387-97664–7, Springer-Verlag, New York, 1992.

[130] Pnueli, A.; Manna, Z.: Temporal Verification of Reactive Systems — Safety. ISBN 0–387–94459–1, Springer-Verlag, New York, Berlin, Heidelberg, London, 1995.

[131] Pnueli, A.; Shalev, M.: What is in a Step: On the Semantics of Statecharts. In: *Proceedings of the Symposium Computer Software*, Lect. Notes in Comput. Sci., Vol. 526, Springer-Verlag, Berlin, 1991, 244–264.

[132] Poston, R. M.: Automating Specification-Based Software Testing. *IEEE Computer Society Press*, Los Alamitos, CA, 1996.

[133] Ostroff, J.S.; Wonham, W.M.: A temporal logic approach to real-time control. In: *Proceedings of the 24th IEEE Conference on Decision and Control*, Florida, 1985, 656–657.

[134] Rabin, M. O.: Decidability of second-order theories and automata on infinite trees. *Trans. AMS* 141, 1969, 1–35.

[135] Rational Corp.,: Documents on UML (the Unified Modelling Language), Version 1.0 *(http://www.rational.com/ot/uml/1.0/)*, 1996.

[136] Rabin, M. O.; Scott D.: Finite automata and their decision problems. *IBM Journal Res. 3*, 1959, 115–125.

[137] Ramalingam, T.; Das, A.; Thulasiraman, K.: On testing and diagnosis of communication protocols based on the finite state machine model. *Computer Communications*. 18(5) 1995, 329–337.

[138] Rapps, S.; Weyuker, E. J.: Selecting software test data using data flow information. *IEEE Transactions on Software Engineering* Vol. SE-11, No. 4, April 1985, 367–375.

[139] Reisig, W.: Petri Nets: An Introduction. Springer-Verlag, Berlin, 1985.

[140] Reisig, W.: A Primer in Petri Net Design. Springer-Verlag, Berlin, 1992.

[141] Reusch, R.: A Note on State Reduction and Homomorphism for Incompletly Specified Sequential Machines. Forschungsbericht, Nr. 120, Fachbereich Informatik, Universität Dortmund, 1981.

[142] Reusch, R.: A Useful Lemma For Checking Experiments And A Method. Forschungsbericht, Nr. 136, Fachbereich Informatik, Universität Dortmund, 1982.

[143] Riedemann, E.: Testmethoden für sequentielle und nebenläufige Software-Systeme, Teubner, Stuttgart, 1997.

[144] Richier, J. L.; Rodriguez, C.; Sifakis, J.; Voiron, J.: XESAR: A Tool for Protocol Validation. User's Guide, LGI-Imag, 1987.

[145] Reed, G.; Roscoe, A.: Metric spaces as models for real-time concurrency. *Proceedings Mathematical Foundations and Computer Science*, Lecture Notes in Computer Science, Vol. 298, Springer-Verlag, New York, 1987.

[146] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: Object Oriented Modelling and Design. Prentice Hall, 1991.

[147] Ryant, I.: The Correctly Analysed System Which Behaves Incorrectly. ACM SIGSOFT, Software Engineering Notes, Vol. 20, No. 2, April 1995, 58–61.

[148] Sabnani, K. K.; Dahbura, A. T.: A protocol testing procedure. In: *Comput. Networks and ISDN Syst.*, volume 15, No. 4, 1988, 285–297.

[149] Sengupta, S.; Korobkin, C. P.: C++, Object-Oriented Data Structures. Springer-Verlag, New York, 1994.

[150] Schwartz, R.; Melliar-Smith, P.; Vogt, F.: An interval logic for higher-level temporal reasoning. In: *Proceedings of the 2nd Annual Symposium on Principles of Distributed Computing*, 1983, 173–186.

[151] Shlaer, S.; Mellor, S. J.: Object-Oriented Systems Analysis: Modelling the World in Data. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[152] Sistla, A. P.; Clarke, E. M.: The complexity of propositional temporal logic. In: *Journal of the ACM*, volume 32, No. 3, January 1986, 733–749.

[153] Smolarczyk, R.: Generierung von Testfällen für VHDL-Programme mittels symbolischer Methoden. Diplomarbeit, Informatik 1, Universität Dortmund, Fachbereich Informatik, 1992.

[154] Spillner, A.: Dynamischer Integrationstest modularer Softwaresysteme. Dissertation, Universität Bremen, Fachbereich Mathematik und Informatik, Dezember 1990.

[155] Streett, R. S.: Propositional dynamic logic with converse. *Inf. Cont.* 1982, 121–141.

[156] Thomas, W: Automata on infinite objects. In: Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science* volume B, Elsevier, Amsterdam Publishers, 1990, 133–191.

[157] Uselton, A.; Smolka, S.: A Compositional Semantics for Statecharts Using Labelled Transition Systems. State University of New York at Stony Brook, 1994.

[158] von der Beek, M.: A Comparison of Statecharts Variants. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, Vol. 863, Springer-Verlag, 1994, 128–148.

[159] Yourdon, E.: Modern Structured Analysis. Prentice Hall International, 1989.

[160] Woodward, M. R.; Heddley, D.; Hennel, M. A.: Experience with path analysis and testing of programs. IEEE Transactions on Software Engineering Vol. SE-6, No. 3, May 1980, 278–286.

[161] Wrifs-Brock, R.; Wilkerson, B.; Wiener, L.: Deisigning Object-Oriented Software. Prentice-Hall, Englewood Cliffs, NJ, 1990.