

Higher-Order Process Engineering: The Technical Background

Johannes Neubauer

21.04.2014

TU Dortmund

Contents

1	Introduction	1
2	Preliminaries	3
2.1	Conventions	3
2.2	ChainReaction	3
2.3	Extreme Model-Driven Design	4
3	Higher-Order XMDD	7
3.1	Top-Down View	7
3.1.1	Execution Context & Constants	8
3.1.2	IO Activities	10
3.1.3	User Interface	13
3.1.4	Dependency Management	15
3.2	Canonical Mapping	15
3.2.1	Atomic Activities	16
3.2.2	Service Browser	20
3.2.3	Class Chooser	21
3.3	Hierarchy	22
3.3.1	Parameterization	24
3.3.2	Abstraction Activities	26
3.3.3	Stateless and Stateful Processes	27
3.3.4	Type Parameters	28
3.3.5	Constructors	30
3.3.6	Graph Inspector	31
3.3.7	Graph Browser	32
3.3.8	Graph Chooser	33
3.4	Configuration	33
3.4.1	Configuration Graphs	34
3.4.2	IO Browser & Preconfigured Activities	35
3.5	Variability	37
3.5.1	Interface Graphs	38
3.5.2	Interface Graph SIBs	40
3.5.3	Constructor Activities and Interface Graphs	42
3.5.4	Configuration Interface Graphs	42
3.6	Inversion of Control	44
3.6.1	On the Target Language Level	45
3.6.2	On the Modeling Level	45
3.7	Transition from jABC3 to jABC4	46

Contents

3.8 The Interpreter	48
4 Conclusion	51

List of Figures

2.1	A screenshot of the open source computer board game ChainReaction in retro-look.	4
3.1	Excerpt of the control- and data-flow information for an activity (i.e., “activity n”) in a <i>dKTS_C</i>	8
3.2	Screenshot of the jABC4 main window showing a process model of a simple game strategy (GS) for the computer board game ChainReaction.	13
3.3	Mapping of a method to an atomic SIB.	15
3.4	Both tabs of the settings dialog for service browsers.	20
3.5	A view of a service browser offering some services of the JRE.	20
3.6	Class chooser for setting the type of a context variable labelled “iterable”.	21
3.7	Input and output parameterization of a service graph using the example of an technical SLG for iterating through an iterable (e.g. a collection).	23
3.8	Example configuration graph for the ChainReaction Scenario.	35
3.9	Example IO browser for the configuration graph shown in Fig. 3.8.	35
3.10	Interface graph for cell evaluations in the ChainReaction scenario.	39
3.11	Service graph interacting between jABC4 and ChainReaction.	41
3.12	Starter graph for the instantiation of two GSs and execution of ChainReaction via the graph depicted in Fig. 3.11 in activity “start game”.	42
3.13	Example for a configuration interface graph combining the configuration graph shown in Fig. 3.8 and the interface graph of Fig. 3.10 for the ChainReaction scenario.	43
3.14	The example GS (cf. Fig. 3.2) in a step-by-step execution via the tracer plugin.	49

1 Introduction

The *higher-order process engineering* (HOPE) approach is a consequent evolution of the XMDD [MS04, MS09a, MS12, SMCB96] approach. The main goal of XMDD is to involve the application expert, who knows the requirements on an application, into the system design process [MS06, SM99, SN07]. This still holds for HOPE. But application experts are not necessarily well-versed in technical realization, so that technical experts implement the requirements. On the contrary, technical experts in general lack expertise in the application domain. According to concepts like extreme programming, lean design, and agile computing, this is tackled by strengthening the communication between technical experts and application experts. Unfortunately the state-of-the-art solutions to this are either

1. based on informal communication [SS06, STA05], which is error-prone and puts forward a *semantic gap* between the participants in terms of terminology and experience,
2. use a formal, executable language like BPEL+BPMN [AAA⁺07], but it burdens the modeler to juggle with technical details like web service endpoints, or
3. use the formal description language BPMN 2.0 [OMG11] which is less technical, but introduces a *semantic gap* between the description and its realizations, as the standards as well as their current realizations have no proper support for the integration of business activities in a service-oriented fashion [DS12].

In the XMDD approach and its incarnation the jABC framework *immediate user experience and feedback* and *seamless acceptance* has been realized introducing a hierarchical coordination layer, i.e. the SLGs, with its components, i.e. the SIBs, which are decoupled from the service implementations beneath. Hence the application experts design the application behavior according to the requirements in coarse-grained, easy-to-understand process models, and the technical experts implement the needed SIBs and services. Furthermore the role of *domain experts* is situated in between application experts and technical experts as they are responsible for bundling SIBs from technical experts to libraries and present them to the application experts tailored to a specific domain.

The HOPE approach extends XMDD via runtime variability capabilities, in order to meet the growing requirements on (business) process modeling in terms of supporting system evolution beyond the state-of-the-art of design-time variability like product-lining and variability modeling with the flexibility to safely add new functionality at runtime. Key to this intent is to enhance the in essence *control-oriented* XMDD paradigm with *data-orientation* and to adapt well-known paradigms from

programming languages in a *simplicity-first* fashion [SN07, MS11], ranging from features of object-orientation [KA90] to functional programming [Ses12]. The new HOPE approach is type-aware, so that type-safety can be validated at design-time. But the central achievement is the introduction of higher-order semantics by treating services and processes as *first-class citizens*. They may be moved around *just like data* and *plugged and played* into activities at runtime, thus enabling higher-order process engineering. This leads to comprehensible and concise models still manageable for application experts which are in general non-programmers. The realization has been carried out minimally invasively on top of the incarnation of XMDD the modeling environment jABC3 [SMN⁺06, SM08] facilitating its simplicity-oriented plugin framework as described in [NNL⁺13].

The HOPE approach has been validated in varying scenarios, where the dynamic exchange of processes, services, and service implementations is essential:

- dealing with the *combinatorial explosion* regarding variant rich systems as well as the flexibility needed to be able to react to requirements or environmental changes at run-time in [NS13b] and [NS13a],
- applying the approach to *active automata learning* [Ang87, SHM11, NMS13, NSB⁺12, WNS⁺13] and (risk-based) testing [FR14, GT02, RSM08] integrating process models into the active automata learning framework LearnLib [RSBM09, MSHM11] via full-code generation [Jö13, JS12] in [NS14] as well as [NWS14],
- run-time enabling *process model synthesis* in [NSM13], i.e. loading tailored, synthesized processes at run-time and plug them into running processes as appropriate, and
- several bachelor and master theses as well as research projects ranging from reverse engineering, over modeling of dynamic data bases, benchmark generation, automata learning, to modeling game strategies in [Neu14].

This document accompanies the thesis [Neu14]. Thus it describes the extensions of the formal model behind XMDD for HOPE and the realization of HOPE's incarnation the jABC4 along a running example: modeling game strategies for the computer board game ChainReaction. The structure of this document is as follows: Chap. 2 introduces the extreme model driven design (XMDD) and one thing approach (OTA). The concepts of HOPE and the prototypic implementation jABC4 on top of the jABC3 framework are delineated in Chap. 3 by means of the running example ChainReaction. Chap. 4 concludes this report.

2 Preliminaries

In the following the conventions used throughout this document, the XMDD approach, and the running example ChainReaction are described.

2.1 Conventions

The following conventions will be used throughout this document:

new notion

New notions will be written emphasized.

“label”

Labels of activities, branches, and context variables will be presented in quotation marks. The differentiated types of labels will be disambiguated in the text.

CClassName, **I**InterfaceName, and **E**EnumerationName

Simple as well as full qualified names of Java types will be prefixed with a corresponding icon, and written in a type writer font.

GServiceGraphName and **G**InterfaceGraphName

Simple as well as full qualified names of graph types will be prefixed with a corresponding icon, and written in a type writer font.

A domain X is a structure $X = (X_1, X_2, \phi, R)$ where ...

A structure will be defined as a domain (e.g. X) and is identified with an identifier set or structure (e.g. X_1). The other sets or structures (e.g. X_2) are related to the entities via a function (cf. ϕ) or relation (cf. R).

2.2 ChainReaction

In order to show the impact of HOPE a running example will be used, which bases on project weeks held at recurring events at TU Dortmunds to attract pupils on lower secondary education level for computer science. The pupils have been asked to create a game strategy for a computer opponent in the open source computer board game *ChainReaction*¹ with jABC4.

ChainReaction (cf. Fig. 2.1) is a two player game with a 6×5 board. The opponents place one token – denoted by “atom” – on a cell each turn alternately. An atom may be placed either on cells of the current player (i.e., he has at least one atom on the cell) or on empty cells. Each cell has a capacity depending on its

¹<http://cr.freewarepoint.de>

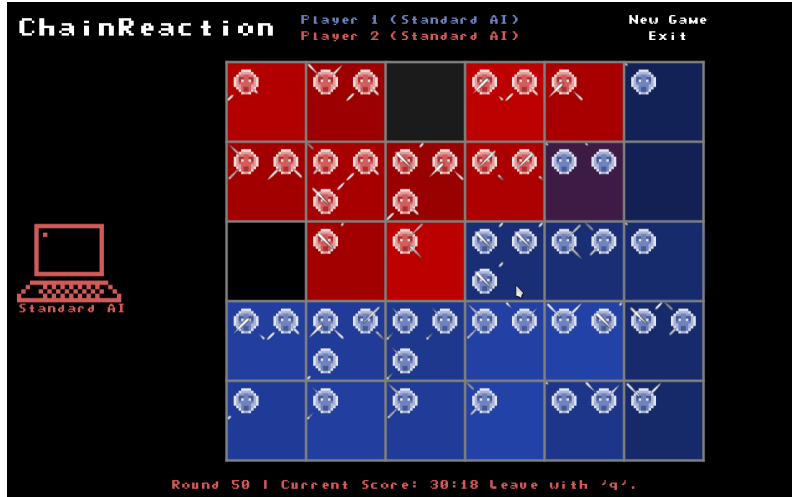


Figure 2.1: A screenshot of the open source computer board game ChainReaction in retro-look.

horizontal and vertical neighbors. Hence a corner cell has a capacity of two, an edge cell a capacity of three, and a center cell a capacity of four atoms.

If a cell reaches its capacity, it will explode and each atom spreads to one neighbor cell and assimilates it with all its atoms. The exploded cell will be empty and is no longer owned by the player. If one of the neighbor cells reaches its capacity because of the new atom, it will explode, too. This may lead to chain reactions changing the complete board and this is where the game’s name comes from. The goal of the game is to reach a board configuration where the oponent has no cells anymore.

The task for the pupils was to create a jABC4 process which evaluates a given cell in a given board configuration regarding the benefit to place an atom. Both the game strategies created by the pupils as well as ‘bridging processes’ for interacting with the API of the game will be used throughout this document.

2.3 Extreme Model-Driven Design

The basis for higher-order process engineering (HOPE) is the extreme model-driven design paradigm (XMDD) [MS04, MS09a, MS12] which embodies ideas from

1. service orientation [MSR05],
2. model-driven design [VG07], and
3. the end-user-centeredness advocated in extreme programming [BA04].

Combining these strands enables application experts to control the design and evolution of processes during their whole life-cycle according to their own level of technical competence and business responsibility. The *one thing approach* (OTA) [SN07, MS09b] provides the conceptual modeling infrastructure for XMDD that enables all the stakeholders (application experts, designers, component experts, implementers, quality assurers, ...) to closely cooperate in the design process.

In particular it enables *immediate user experience and feedback* and thereby *seamless acceptance*: all stakeholders know, refine, and modify one and the same “thing”, without duplications or need to juggle with different modeling languages or paradigms. It allows them to observe the progress of the development and the implications of decisions at their own level of expertise, which is a central trait of the one thing approach.

The language for this comprehensive model where all the information converges are executable process models called *service logic graphs* (SLGs). Operationally, the process models are similar to control flow graphs: the nodes represent activities and the branches describe how to continue the execution depending on the result of the previous activity. Following the terminology of telecommunication systems [SMC⁺96], these activities are called *service-independent building blocks* (SIBs).

SIBs may represent a single functionality (i.e., a service) or a whole subgraph (i.e., another SLG) introducing hierarchy [SMBK97], thus serving as a macro that hides more detailed process models. SIBs are parameterizable and communicate resources via shared *execution contexts*, a hierarchical concept. The application expert is equipped with a collection of SIBs, which forms the available domain of reusable, configurable processes and components shaping a kind of *domain specific language* (DSL).

SLGs are also directly formal models: they are semantically interpreted as Kripke Transition Systems (*KTS*), a generalization of both Kripke structures (*KS*) and labeled transition systems [MSS99] (*LTS*) that allows labels both on nodes and edges.

Definition 2.1. A *KTS* over a finite set of atomic propositions AP is a structure $KTS = (S, s_0, Act, R, \mathcal{I})$, where

- S is a finite set of *states*.
- $s_0 \in S$ a dedicated *start state*.
- Act is a finite set of *actions*.
- $R \subseteq S \times Act \times S$ is a total *transition relation*.
- $\mathcal{I} : S \rightarrow 2^{AP}$ is an *interpretation function*.

In the following the underlying formal model is incrementally enhanced for better presentation of the new concepts of HOPE. Therefore an alternative formal definition of the core elements in an SLG is introduced, providing a more natural (canonical) interpretation. In a *KTS* the action labels are interpreted as the active part and the states are idle. In SLGs the activities are the active part (cf. the *states* in a *KTS*), and decide which branch (cf. the *actions* in a *KTS*) is followed after execution to find the successor activity, leading to the following definition:

Definition 2.2. A *dKTS* (say *dual KTS*) over a finite set of atomic propositions AP is a structure $dKTS = (\mathcal{A}, a_0, \mathcal{B}, \delta, \mathcal{I})$, where

- \mathcal{A} is a finite set of *activities* (instantiations of SIBs).

- $a_0 \in \mathcal{A}$ a dedicated *start activity*.
- \mathcal{B} is a finite set of branching labels denoted by *branches*.
- $\delta : \mathcal{A} \times \mathcal{B} \rightarrow \mathcal{A}$ is a *transition function*.
- $\mathcal{I} : \mathcal{A} \rightarrow 2^{AP}$ is an *interpretation function*.

A *dKTS* is called dual *KTS*, since edge labels and nodes swap their semantics. Moreover, in order to have a clearer separation of notions the term service independent building block will be used for templates of *activities*, i.e., an activity is an instantiation of a SIB component.

3 Higher-Order XMDD

The *higher-order process engineering* (HOPE) approach is a consequent evolution of XMDD adding data-orientation to an in essence control-oriented approach as well as higher-order semantics allowing to select, modify, construct and then pass processes during process execution as if they were data. These enhancements have been accompanied by profound changes to the meta-model of SLGs and the binding of implementations to activities.

In the following Sec. 3.1 describes the revised meta-model for SLGs. Sec. 3.2 illustrates the dynamic integration of services, whilst Sec. 3.3 depicts the realization of hierarchical modeling with well-defined input/output parameterization as well as how SLGs may be bundled to libraries. Sec. 3.4 will show how new SIBs can be created by preconfiguring existing ones, without introducing technological breaks. Sec. 3.5 deals with higher-order semantics of SLGs following the HOPE approach, and Sec. 3.6 introduces using dependency injection in order to realize environmental objects in the execution context of process models without global variables or scopes. The last two sections 3.7 and 3.8 dwell into the realization of jABC4 (i.e., the incarnation of HOPE) on top of the existing jABC3 framework as well as the adaption to the integrated interpreter for rapid prototyping.

3.1 Top-Down View

An SLG in the HOPE approach has some additional information as compared to a *dKTS* for modeling the data-flow and data-type information: Every SLG has a type-aware local execution context for sharing resources between the activities. Each activity has a set of input parameters, branches, and output parameters per branch. Input parameters can either read a value from the shared resources or a constant value defined during modeling. Output parameters write to a context variable.

In Fig. 3.1 an exemplary, combined control- and data-flow view for a single activity is shown. Figures showing excerpts of an SLG will have a similar visualization like Fig. 3.1. Its semantics are described in the following paragraphs:

The context is shown as a grey box with context variables depicted in white boxes with rounded edges. The icon next to the label of the variable picture the seminal type of the variable, i.e., Java class **C**, Java interface **I**, service graph **G** (cf. Sec. 3.3), or interface graph **G** (cf. Sec 3.5). After the label of the context variable the simple name of its type follows in parantheses, e.g., the java class **C**Class1 for context variable “variable1”.

The input parameters of the activity of interest “activity n” are shown in the small parameter window “inputs”. A small icon on the left of each variable express, whether the parameter is *dynamic* **D** (i.e., read from the context) or *static* **S** (i.e.,

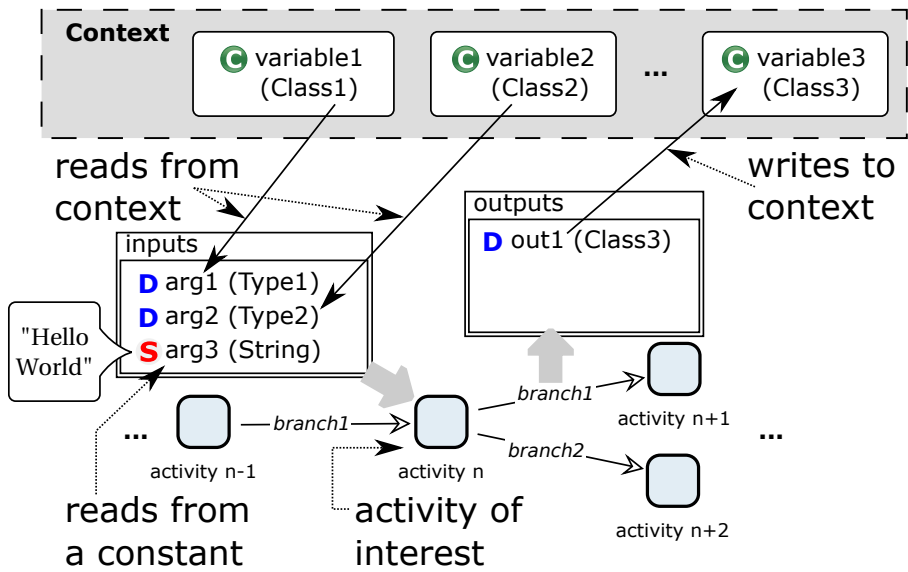


Figure 3.1: Excerpt of the control- and data-flow information for an activity (i.e., “activity n”) in a $dKTS_C$.

read from a constant value). The simple name of a type is denoted in parentheses right after the label of the parameter.

Constant values are displayed as little speech bubbles (cf. “Hello World”). A directed edge with a filled arrow head starting in the variable and pointing at a dynamic parameter, denotes from which context variable the parameter reads.

Output parameters of a branch of activity “activity n” are shown in a small output window labelled “outputs” (here pictured exemplarily for branch “branch1”). As all output branches write to a context variable they are always marked as dynamic **D** and the simple name of its type is given subsequently in parantheses. The context variable, an output parameter writes to, is noted via a directed edge (with a filled arrow head) starting from the output parameter and pointing at the variable.

These concepts of a type-aware execution context and input/output parameterization of activities are defined more formally in the following subsections.

3.1.1 Execution Context & Constants

The execution context consists of a set of typed variables and at runtime each variable has a mutable value associated to it. These variables have a auto-generated unique name which is coupled with a domain-specific, human-readable simple name. Together the unique name and the domain-specific name are denoted by *label*. Labels will be used for other components like activities, input and output parameters as well as branches, too.

Definition 3.1. A *label domain* is a structure $\mathcal{L} = (ID, \mathcal{N}, id)$ where

- ID is a set of identifiers. ID is a local set containing only the ids for the respective structure, but the identifiers itself are universally unique identifiers

(UUIDs) ¹. This is especially relevant for the implementation of the higher-order process modeling approach for differentiating elements with the same name.

- \mathcal{N} is a set of human readable, domain specific names, that are not necessarily unique.
- $id : \mathcal{ID} \rightarrow \mathcal{N}$ is a bijective function mapping identifiers to names.
- A label $l \in \mathcal{L}$ is a structure $l = (i, n)$ with $i \in \mathcal{ID}$ and $n \in \mathcal{N}$. For better presentation $l \mapsto_{\mathcal{L}} n$ will be used and the name of a label is employed as its representative.

In the following the *variables* are employed as the elements of the execution context (i.e. context variables) and formal as well as actual input/output parameters of activities. The general concept of a variable consists of a label being associated with a type. The underlying type system is simply the Java type system augmented with graph types for service- (cf. Sec. 3.3) and interface graphs (cf. Sec. 3.5.1). If a graph is generated to code, it will accordingly be represented via a Java class or Java interface.

Definition 3.2. A *variable domain* over a type system \mathcal{T} is a structure $\mathcal{V} = (\mathcal{L}_{\mathcal{V}}, \nu_{\mathcal{T}})$ where

- $\mathcal{L}_{\mathcal{V}}$ is a label domain.
- $\nu_{\mathcal{T}} : \mathcal{L}_{\mathcal{V}} \rightarrow \mathcal{T}$ is an injective function mapping a type to each variable.
- A variable $v \in \mathcal{V}$ is structured as follows: $v = (l_v, t)$ where $l_v \in \mathcal{L}_{\mathcal{V}}$, and $t =_{\text{def}} \nu_{\mathcal{T}}(l_v)$. For better presentation $v \mapsto_{\mathcal{V}} t$ will be used.

The *execution context* then relates a value to each variable, which may be written by output parameters and read by input parameters. These values are called *mutable* as the referenced value of a context variable may change (i.e. it is reassigned) due to a write operation of an output parameter.

Definition 3.3. An *execution context* (i.e. a *context variable domain*) over a type system \mathcal{T} is a structure $\mathcal{C} = (\mathcal{V}_{\mathcal{C}}, \mathcal{V}_{ar}, \nu_{\mathcal{V}_{ar}})$ where

- $\mathcal{V}_{\mathcal{C}}$ is a variable domain (see Def. 3.2). Each $v \in \mathcal{V}_{\mathcal{C}}$ represents exactly one context variable.
- \mathcal{V}_{ar} represents the set of all objects available in the current runtime environment each of any type $t_{mv} \in \mathcal{T}$ including **null** indicating the absence of a value. These values represent the system state at runtime.
- $\nu_{\mathcal{V}_{ar}} : \mathcal{V}_{\mathcal{C}} \rightarrow \mathcal{V}_{ar}$ is an injective function mapping a value to each context variable.

¹<http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html>

- A context variable $c \in \mathcal{C}$ is structured as follows: $c = (v, mv)$, where $v \in \mathcal{V}_{\mathcal{C}}$, and $mv =_{\text{def}} \nu_{\mathcal{V}ar}(v)$. For better presentation $c \mapsto_{\mathcal{C}} mv$ and $c \mapsto_{\mathcal{C}\mathcal{V}} t_{mv}$ will be used.
- The type of the mutable value mv has to be compatible to the type t_{mv} defined for the variable v as follows $v \mapsto_{\mathcal{V}} t_{mv}$.

Each SLG has an execution context \mathcal{C} which is represented by a context variables domain.

A second kind of resource are the *constants*. They correspond to immutable values, i.e., they are declared at modeling time and cannot be reassigned at runtime. In most cases it is desirable that the value itself is a constant, too (e.g., using an `int`, `String`, or `Collections.unmodifiableMap(Map<K, V>)`).

Definition 3.4. A *constant domain* over a type system \mathcal{T} is a structure $Const = (\mathcal{V}al, \nu_{Const})$ where

- $\mathcal{V}al$ is a finite set of constant values. Example:

$$\mathcal{V}al = \{1, 1.0, \text{"Hello World!"}, \text{"Hello"} \rightarrow \text{"World!"}\}$$

- $\nu_{Const} : \mathcal{V}al \rightarrow \mathcal{T}$ a function linking each value of the *set of constants* to its type. Example:

$$\nu_{Const} = \{(1, \text{int}), (1.0, \text{float}), (\text{"Hello World!"}, \text{String}), (\text{"Hello"} \rightarrow \text{"World!"}, \text{UnmodifiableMap})\}$$

- A constant $const \in Const$ is a tuple $const = (val, t)$ with $val \in \mathcal{V}al$, and $t =_{\text{def}} \nu_{Const}(val) \in \mathcal{T}$. For better presentation $const \mapsto_{Const} t$ will be used.

The current implementation of jABC4 supports primitive types, strings, file handles, and some more. But conceptually arbitrary objects may be used.

3.1.2 IO Activities

In the HOPE approach the main entities representing the services/processes to be executed are the activities. They are the instantiations of SIBs, which carry the actual parameter definitions. They may be compared to a statement in conventional programming languages. According to the advised data-flow information, they have a well-defined input/output parameterization. Their *input parameters* may either read a mutable value from a context variable at runtime or a constant value. The association to a variable may be changed at modeling time as well as switching between static and dynamic parameters, and setting a constant value as input.

Definition 3.5. An *input parameter domain* over a context \mathcal{C} and constants $Const$ is a structure $\mathcal{I}n = (\mathcal{V}_{\mathcal{I}n}, \sigma_{\mathcal{C}}, \sigma_{Const})$ where

- $\mathcal{V}_{\mathcal{I}n}$ is a (input) variable domain (see Def. 3.2). Each $v \in \mathcal{V}_{\mathcal{I}n}$ represents exactly one input parameter.

- $\sigma_{\mathcal{C}} : \mathcal{V}_{\mathcal{I}n} \rightarrow \mathcal{C}$ is a partial function describing from which variable of the context \mathcal{C} an input parameter reads the current mutable value. In the following i reads c will be used with $i \in \mathcal{V}_{\mathcal{I}n}$ and $c \in \mathcal{C}$. Parameters that read from a context variable are called *dynamic variables*.
- $\sigma_{\mathcal{C}onst} : \mathcal{V}_{\mathcal{I}n} \rightarrow \mathcal{C}onst$ is a partial function describing the input parameters being assigned with a constant value. In the following i represents $const$ will be used. Parameters representing a constant value are called *static parameters*.
- A parameter is called *undefined*, whenever its variable $v \in \mathcal{V}_{\mathcal{I}n}$ is in neither of these sets:

$$v \notin (\text{domain}(\sigma_{\mathcal{C}}) \cup \text{domain}(\sigma_{\mathcal{C}onst}))$$

- One has $\text{domain}(\sigma_{\mathcal{C}}) \cap \text{domain}(\sigma_{\mathcal{C}onst}) = \emptyset$, i.e., a parameter can either be static, dynamic, or undefined.
- If $\text{domain}(\sigma_{\mathcal{C}}) \cup \text{domain}(\sigma_{\mathcal{C}onst}) = \mathcal{V}_{\mathcal{I}n}$ the set of input parameters $\mathcal{I}n$ is called *complete*.
- A dynamic input parameter $i_d \in \mathcal{I}n$ is a tuple $i_d = (v, c)$ with $v \in \text{domain}(\sigma_{\mathcal{C}})$ and $c =_{\text{def}} \sigma_{\mathcal{C}}(v)$. For better presentation write $i_d \mapsto_{\sigma_{\mathcal{C}}} c$ will be used.
- A static input parameter $i_s \in \mathcal{I}n$ is a tuple $i_s = (v, const)$ with $v \in \text{domain}(\sigma_{\mathcal{C}onst})$ and $const =_{\text{def}} \sigma_{\mathcal{C}onst}(v)$. For better presentation $i_s \mapsto_{\sigma_{\mathcal{C}onst}} const$ will be used.

Output parameters on the contrary may only be dynamic and a modeler declares to which context variable each writes the corresponding value being returned from the execution of the underlying service or process.

Definition 3.6. A *output parameter domain* over a context \mathcal{C} is a structure $\mathcal{O}ut = (\mathcal{V}_{\mathcal{O}ut}, \omega)$ where

- $\mathcal{V}_{\mathcal{O}ut}$ is a (output) variable domain (see Def. 3.2). Each $v \in \mathcal{V}_{\mathcal{O}ut}$ represents exactly one output parameter.
- $\omega : \mathcal{V}_{\mathcal{O}ut} \rightarrow \mathcal{C}$ is a partial function describing to which variable of the context \mathcal{C} an output parameter (over-)writes the mutable value (cf. Def. 3.3). In the following o writes c will be used with $o \in \mathcal{V}_{\mathcal{O}ut}$ and $c \in \mathcal{C}$. All output parameters are *dynamic variables* as they either write to a context variable or are undefined².
- If ω is total the set of output parameters is called *complete*:

$$\omega : \mathcal{V}' \rightarrow \mathcal{C}, \text{ and } \mathcal{V}' = \mathcal{V}$$

- An output parameter $o \in \mathcal{O}ut$ is a tuple $o = (v, c)$ with $v \in \mathcal{V}_{\mathcal{O}ut}$ and $c =_{\text{def}} \omega(v)$. For better presentation $o \mapsto_{\mathcal{O}ut} c$ will be used.

²An output parameter is undefined, iff its variable $v \in \mathcal{V}_{\mathcal{O}ut}$ does not write to a context variable $v \notin \text{domain}(\omega)$.

An output parameter is always associated to a labelled *branch* of the activity. Different from conventional programming languages, there may be more than one output parameter per branch, although generally but not necessarily an output parameter has only one branch. The semantic behind relating output parameters to branches is, that depending on the outcome of the execution of the activity in terms of control-flow, it may return different values. If, e.g., a non-**void** Java method returns successfully, a value of its return type (or **null**) will be returned. But if an exception is thrown, an instance of the exception type will be the result.

Definition 3.7. A *branch domain* over a context \mathcal{C} is a structure $\mathcal{B} = (\mathcal{L}_{\mathcal{B}}, \mathcal{O}ut, \omega_{\mathcal{B}})$ where

- $\mathcal{L}_{\mathcal{B}}$ is a label domain representing the branch names.
- $\mathcal{O}ut$ is a output parameter domain (cf. Def. 3.6).
- $\omega_{\mathcal{B}} : \mathcal{L}_{\mathcal{B}} \rightarrow 2^{\mathcal{O}ut}$ is a function assigning a subset of the outputs to each branch label.
- One has $\{\omega_{\mathcal{B}}(l) \mid l \in \mathcal{L}_{\mathcal{B}}\} = \mathcal{O}ut$, i.e., every output parameter is assigned to at least one branch.
- A branch $b \in \mathcal{B}$ is a tuple $b = (l_b, \mathcal{O}ut_b)$ with $l_b \in \mathcal{L}_{\mathcal{B}}$ and $\mathcal{O}ut_b =_{\text{def}} \omega_{\mathcal{B}}(l_b)$. For better presentation $b \mapsto_{\mathcal{B}} \mathcal{O}ut_b$ will be used.

With input parameters, branches, and output parameters, all ingredients for *input/output activities* (IO activities) are available. There are different kinds of IO activities which will be introduced and discussed in Sec. 3.2, 3.3, and 3.5.

Definition 3.8. The *IO activity domain*³ over a context \mathcal{C} and constants Const is a structure $\mathcal{A} = (\mathcal{L}_{\mathcal{A}}, \mathcal{I}n, \mathcal{B}, \mathbf{i}, \mathbf{b})$ where

- $\mathcal{L}_{\mathcal{A}}$ is label domain (cf. Def. 3.1) identifying each activity as well as giving it a domain specific name.
- $\mathcal{I}n$ is a input parameter domain (cf. Def. 3.5). For the static input parameters the constants of Const are used.
- \mathcal{B} is branch domain (cf. Def. 3.7).
- $\mathbf{i} : \mathcal{L}_{\mathcal{A}} \rightarrow 2^{\mathcal{I}n}$ is a function mapping input parameters to activities.
- $\mathbf{b} : \mathcal{L}_{\mathcal{A}} \rightarrow 2^{\mathcal{B}}$ is a function mapping branches to activities.
- An *activity* $a \in \mathcal{A}$ is a tuple $a = (l_a, \mathcal{I}n_a, \mathcal{B}_a)$ where
 - $l_a \in \mathcal{L}_{\mathcal{A}}$ is the label of the activity,
 - $\mathcal{I}n_a =_{\text{def}} \mathbf{i}(l_a) \subseteq \mathcal{I}n$ is the input parameter domain of activity a , and
 - $\mathcal{B}_a =_{\text{def}} \mathbf{b}(l_a) \subseteq \mathcal{B}$ is the branch domain of activity a .

³IO activity and activity will be used synonymously in the following unless it is explicitly differentiated.

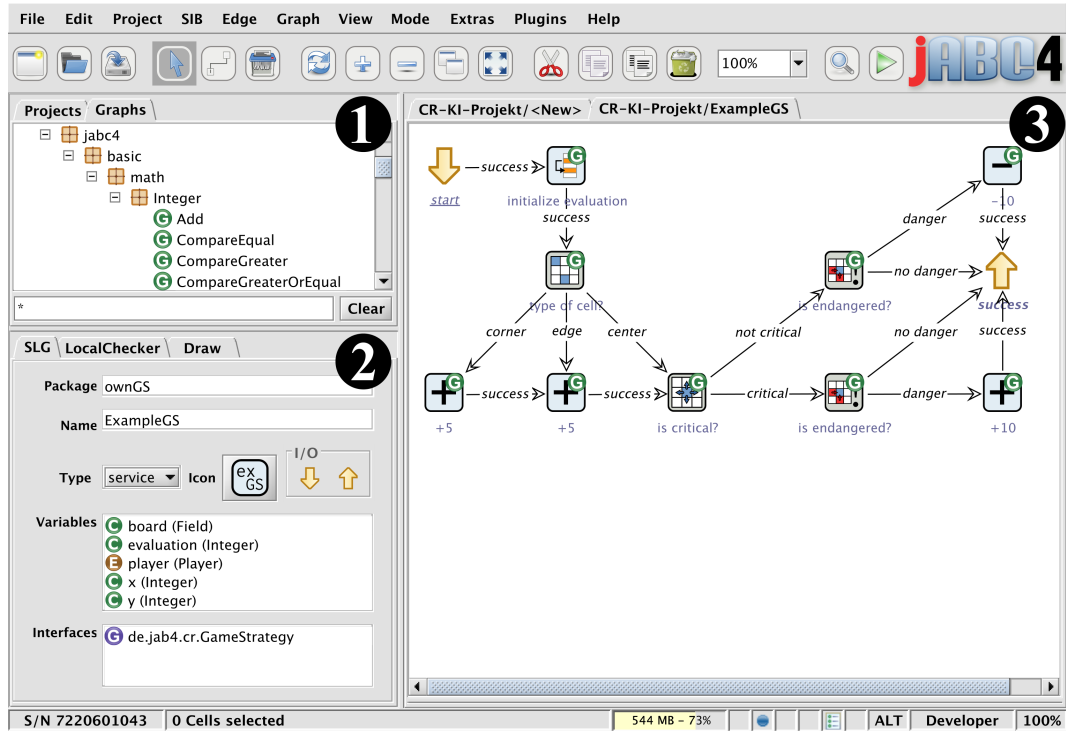


Figure 3.2: Screenshot of the jABC4 main window showing a process model of a simple game strategy (GS) for the computer board game ChainReaction.

For better presentation $a \mapsto_{\mathcal{A}} (\mathcal{I}n_a, \mathcal{B}_a)$ will be used.

With execution contexts, constants and IO activities, a much richer meta-model may be defined introducing the *dual kripke transition system with context*, which incorporates the data-flow as well as data-type information.

Definition 3.9. A *dual kripke transition system with context* over a type system \mathcal{T} and a finite set of atomic propositions AP is a structure $dKTS_{\mathcal{C}} = (\mathcal{A}, a_0, \mathcal{B}, \delta, \mathcal{I}, \mathcal{C})$, where

- \mathcal{A} is an *activity domain* (cf. Def. 3.8).
- $a_0 \in \mathcal{A}$ a dedicated *start activity*.
- \mathcal{B} is a *branch domain* (cf. Def. 3.7).
- $\delta : \mathcal{A} \times \mathcal{B} \rightarrow \mathcal{A}$ is a *transition function*.
- $\mathcal{I} : \mathcal{A} \rightarrow 2^{AP}$ is an *interpretation function*.
- \mathcal{C} is an *execution context* shaping the context of shared resources (cf. Def. 3.3).


3.1.3 User Interface

The user interface of the development environment jABC4 (based on the jABC3 framework) is shown in Fig. 3.2. It is divided in three main areas:

1. The *browser area* consists of different tabs with tree-views for resources namely the *project browser*, *graph browser* (cf. Sec. 3.3.7), *service browsers* (cf. Sec. 3.2.2), and *IO SIB browsers* (cf. Sec. 3.4.2).
2. The *graph canvas* is the main modeling area, where the SLGs are depicted as well as modified.
3. The *inspector area* contains panels showing meta-information regarding currently selected activities or the currently shown SLG in the graph canvas. For changeable attributes the inspectors provide editors.




In the project browser a modeler selects the jABC4-project to work on. The other browsers (denoted by *SIB browser*) show project-specific the available resources in terms of SIBs (i.e., kinds of activities), that may refer to a method call (atomic activities), a process invocation (abstraction activity), or a preconfigured activity, which may be either an atomic or an abstraction activity.



An arbitrary SIB may be instantiated as an activity by drag&drop from a leaf node of a SIB browser to the graph canvas. An activity is visualized on the graph canvas via an icon representing the node and a text label beneath it, showing the initial label of the activity, which may be changed later on.


To create a new context variable the *graph inspector* may be used, which is shown in the inspector area (cf. bottom-left of Fig. 3.2). A right-click in the list of “Variables” and click on the menu item “Add new...” opens a popup dialog where the modeler enters a name for the variable. Initially the type of a variable is  Object. Sec. 3.2.3 and 3.3.8 describes how the type of a context variable is declared.

When a context variable $c \in \mathcal{C}$ is dragged to and dropped on an activity a , a popup-menu opens where each entry represents an input parameter. With the selection of an input parameter $i_d \in \mathcal{In}_a$ the relation $i_d \mapsto_{\sigma_c} c$ is set.

The *input parameter window* (cf. “inputs” in Fig. 3.1) of an activity can be opened via right-clicking on the activity and selecting “Manage input parameters...” or

1. holding the left-mouse on the activity for about half-a-second,
2. a blue  and an orange arrow  appear left and right beneath the mouse cursor,
3. moving to the left (i.e., on ) , opens the input parameter window.

In the popup menu of an input parameter it is possible to switch between *static*  and *dynamic*  parameters (iff the type of the parameter is supported for constant values). For a static parameter $i_s \in \mathcal{In}_a$ a menu point “set value...” is available, which opens a corresponding editor for entering its constant value setting the relation $i_s \mapsto_{\sigma_{Const}} const$ with $const \in Const$.

Choosing to move on the aforementioned orange arrow  opens a popup menu with all branches of the activity. If a branch is selected, the corresponding *output parameter window* (cf. “outputs” in Fig. 3.1) will be opened. Analogously to the inputs, there is a menu item “Manage output parameters” – which has a sub menu

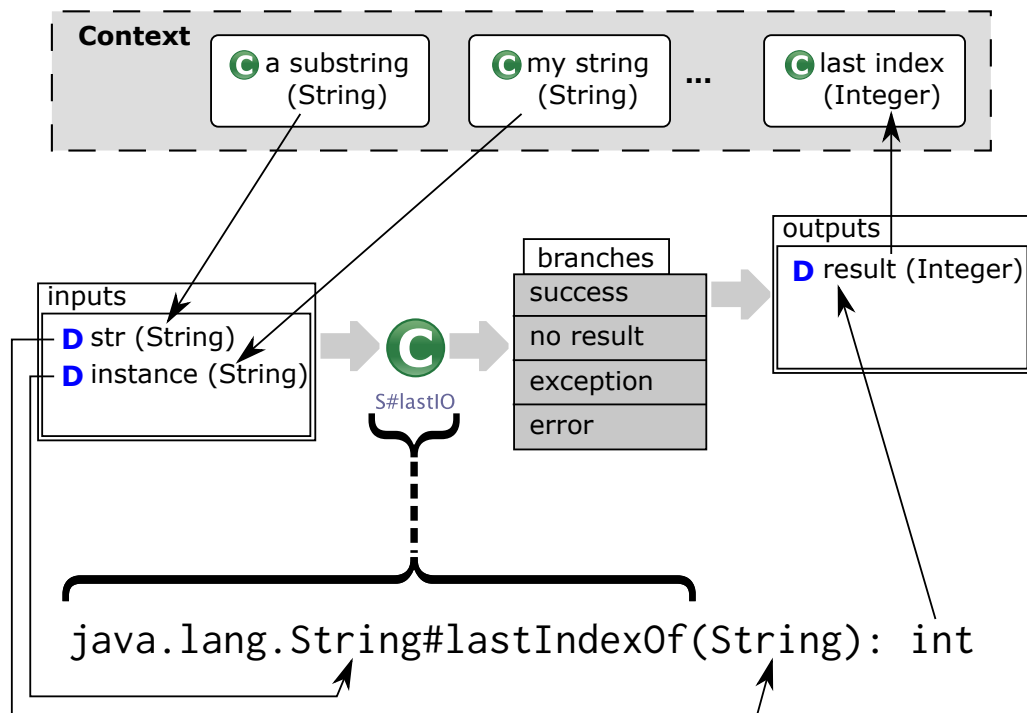


Figure 3.3: Mapping of a method to an atomic SIB.

with all branches – leading to the same output parameter window like using the orange arrow →.

An output parameter $o \in \mathcal{O}ut_b$ of the selected branch $b \in \mathcal{B}_a$ writes to context variable $c \in \mathcal{C}$ denoted $o \mapsto_{\mathcal{O}ut} c$ by dragging the parameter and dropping it onto the variable in the graph inspector.

3.1.4 Dependency Management

Apache maven⁴ support has been added to jABC4-projects, so that if a `pom.xml` is in the root folder of a project, the dependencies defined there will be added to the *project class path*. The different SIB browser may then rely on this enhanced class path for finding services (i.e., Java methods) and SLGs⁵, which may be delivered as SLG libraries each library bundled in a standard Java archive via maven artifacts.

3.2 Canonical Mapping

This section will show how to dynamically bind services to activities [NS14]. In our running ChainReaction example we need a basic functionality like adding a value to an integer in order to realize a bonus to the current cell evaluation, or iterating through the cells of the game board. This is realized via a direct mapping between

⁴<http://maven.apache.org>

⁵SLGs are serialized to XML files and may then be put into a JAR (Java archive). The archive can be added to the class path and the XML files may be retrieved via standard mechanisms of the Java runtime environment (JRE) and deserialized to an SLG.









	method	constructor	static	abstract
class				–
abstract class				
interface	–	–	–	

Table 3.1: Icons for atomic activities.

constructs of the target language and the *atomic activities*. Services are represented by static, constructor, and dynamic⁶ methods. Fig. 3.3 depicts an example for an atomic activity representing the method `lastIndexOf(String)` of the class `CString`. The details of the input and output parameterization as well as the binding of methods to atomic activities is described in the following. Later on Sec. 3.3 will delineate how SLGs abstracting from technical service bindings may be bundled to SLG libraries to be reused by application experts guaranteeing executability.

A target language may be any object-oriented language supporting a more or less one-to-one mapping between the structures defined here and constructs of the language. A prototype for modeling and interpreting the process models has been designed and implemented in this thesis on top of the jABC3 platform for Java and a codegenerator following the full-code generator principle of [Jö13] for both Java and Scala⁷. The approach heavily bases on the existing type system of the respective target language. An easy-to-use subset of modern object-oriented languages is supported on an ‘as needed’ basis, i.e., it may be enhanced in future versions in order to fulfill some requirements.

3.2.1 Atomic Activities

Technically there are the following three different variations of method types:

1. the *static methods*⁸ represented by *static SIBs*,
2. *constructor methods* which are also realized as static SIBs, and
3. *dynamic methods*⁹ realized via *dynamic SIBs*.

A constructor method is static too in that it is not called on an instance but on a class and therefore it is interpreted a special form of a static method, although the syntax for calling a constructor method and a static method differs in most languages, e.g., in Java the `new` operator is used to invoke constructors. Additionally, a constructor is named alike its declaring class and always returns either a corresponding instance

⁶Dynamic methods are often denoted by *instance methods* in object-oriented programming.

⁷The Scala generator has been implemented with support of a student assistant.

⁸In object-oriented programming a static method is associated to a class and not to an object.

⁹A dynamic method is associated to an instance of a class (i.e. an object).

or throws an exception. But from the point of view of the HOPE approach they are comparable.

In the user interface I differentiate whether an atomic activity is either a normal, constructor, static, or abstract method and whether it is declared in a normal, abstract, or interface class. This is reflected in the activities' icon (cf. Table. 3.1).

Regarding the return type of a method I introduce three SIB variants:

1. the *service SIB* represents a method call with an arbitrary return type including `void` for methods which have no return type, except `boolean` and enumerations (sub classes of `Enum`),
2. the *decision SIB* handles methods with a return type `boolean` used to realize if-then control-structures, and
3. the *enumeration SIBs* deal with methods returning Java enumerations realizing a switch statement.

This is not meant as an exhaustive list of variants and it therefore may be enhanced in the future.

Definition 3.10. The atomic activities \mathcal{A}_a shape a subset of the IO activities \mathcal{A} . The atomic activities inherit the structure of IO activities (cf. Def. 3.8) and bind the name, instance parameter, input parameters and return type of the method to these structures. Additionally the method itself is bound to an atomic activity. Hence a domain of atomic activities is a structure $\mathcal{A}_a = (\mathcal{L}_{\mathcal{A}_a}, \mathcal{In}_{\mathcal{A}_a}, \mathcal{B}_{\mathcal{A}_a}, \mathbf{i}_{\mathcal{A}_a}, \mathbf{b}_{\mathcal{A}_a}, \mathcal{M}, \mathbf{m})$ where

- $\mathcal{L}_{\mathcal{A}_a} \subseteq \mathcal{L}_{\mathcal{A}}$, $\mathcal{In}_{\mathcal{A}_a} \subseteq \mathcal{In}$, $\mathcal{B}_{\mathcal{A}_a} \subseteq \mathcal{B}$, $\mathbf{i}_{\mathcal{A}_a} : \mathcal{L}_{\mathcal{A}} \rightarrow \mathcal{In}_{\mathcal{A}_a}$, and $\mathbf{b}_{\mathcal{A}_a} : \mathcal{L}_{\mathcal{A}} \rightarrow \mathcal{B}_{\mathcal{A}_a}$ are defined similar to the corresponding structures in Def. 3.8. The binding of structures of the associated method is described separately for the different types of atomic SIBs in the following.
- \mathcal{M} is a set of (public) methods in the target language.
- $\mathbf{m} : \mathcal{L}_{\mathcal{A}_a} \rightarrow \mathcal{M}$ is a bijective function mapping a method to each atomic activity.

An atomic activity $a \in \mathcal{A}_a$ represents a *method call* in a target language (i.e. Java) and is called *atomic SIB* which is the superset of static and dynamic SIBs in the different shapes service, decision, and enumeration SIBs. Each maps at least the following constructs from a *method* in the target language (Java), to a structure $a = (l_a, \mathcal{In}_a, \mathcal{B}_a, m)$ as follows

- l_a is a label that is composed of (in this order)
 1. the uppercase letters of the class name¹⁰,
 2. a hash character,
 3. all characters until the first uppercase letter of the method name¹¹, and

¹⁰With 'class name' I refer to the simple name of a Java class, i.e., for the class `java.lang.String` the simple name is `String`.

¹¹With 'method name' the simple name of a method is meant, i.e., for the method `java.lang.String.lastIndexOf(String)` the simple name is `lastIndexOf`.

4. the uppercase letters of the rest of the method name.

Hence for the method `java.lang.String.lastIndexOf(String)` the label is: “S#lastIO” (cf. the activity in Fig. 3.3). This is only a predefined label. It may be changed by the modeler later on.

- $\mathcal{I}n_a =_{\text{def}} \mathbf{i}_{\mathcal{A}_a}(a)$ is a set of input parameters (cf. the *inputs*-window in Fig. 3.3), that is directly connected to the formal input parameters of the method. Each input parameter has a position in the parameter array and a type $t \in \mathcal{T}$. The position is translated to a name label `arg_$pos` for the input parameter. If there is a name present in the target language, it is favored. For Java if available we use the Javadoc information which contains the name of each parameter, since it is not directly accessible through the Java reflection API¹².
- For dynamic SIBs, which are executed on an instance, there is an additional special input parameter *instance parameter* $i \in \mathcal{I}n_a$ (cf. the parameter “instance” in the activities’ inputs in Fig. 3.3). Hence a dynamic SIB $a_d \in \mathcal{A}_a$ is denoted as $a_d = (l_a, i, \mathcal{I}n_{a_d}, \mathcal{B}_a)$ with $\mathcal{I}n_{a_d} = (\mathcal{I}n_a \setminus i)$.
- $\mathcal{B}_a =_{\text{def}} \mathbf{b}_{\mathcal{A}_a}(a)$ is the set of branches that handle the return value of a method as well as the *exceptions* or more generally the *throwables* as this is in Java the superset of exceptions and *errors* (cf. the *branches*-popup-window and the *outputs*-window in Fig. 3.3).

The exception handling is the same for all atomic SIBs. There is one branch for each throwable in the list of *defined throwables*¹³. The label of such a branch is composed of a lower case version of the class name where the elements separated via camel case letters are detached via a space character. In addition the word “exception” – or “error” respectively – is omitted. For `NullPointerException` the resulting label is: “null pointer”.

In addition there is one branch for all subclasses of `Exception`, which handles all exceptions not defined in the method definition (e.g. the *runtime exceptions* like `NullPointerException` which can be thrown although they are not explicitly stated in the method definition). Furthermore one branch deals with errors (i.e. subclasses of `Error`) as, e.g. `OutOfMemory`, which are not necessarily defined by the method but may be thrown at any time as they describe a completely unexpected situation like running out of memory.

Each of these *exceptional branches* has exactly one output parameter: the respective throwable. It is typed with the explicitly stated class (for throwables defined by the method) as well as `Exception` and `Error` for the latter general exceptional branches.

The output branches for the return type differ for service, decision, and enumeration SIBs. Constructor methods are always represented via service SIBs. Dynamic and static methods may arise in all three forms:

¹²The Java reflection API enables to inspect the meta-model of Java and execute methods dynamically.

¹³The explicitly defined throwables of a method are listed in the `throws` clause of the method declaration.

Service SIBs have a dedicated branch labeled “success” for successful execution, which has exactly one output parameter labeled “return” typed with the return type of the method. They further on have a “no result” branch devoid of any output parameter, followed if a `null` is returned. In contrast, `void` methods have a “success” branch without any output parameter (and no “no result” branch).

Decision SIBs distinguish the branches “true” and “false” representing the respective output of the `boolean` return type. Both branches define exactly one output parameter returning the respective boolean return value of the method execution. Additionally a “no result” branch without any output parameter deals with a `null` result¹⁴.

Enumeration SIBs provide one branch per enumeration constant labeled alike the constants string representation (i.e. `constant.toString()`). Each branch has exactly one output parameter returning the respective constant. Furthermore a “no result” branch without any output parameter will be followed if a `null` has been returned.

The semantics of an atomic SIB is as follows. If the execution of an SLG reaches an atomic SIB:

1. the input parameters $\mathcal{In}_{\mathcal{A}_a}$ are evaluated (including the instance for dynamic SIBs),
2. the method is executed with the input parameters as actual parameters,
3. the return value or throwable is collected as the method has returned,
4. the corresponding output branch $b = (l_b, \mathcal{Out}_b) \in \mathcal{B}_{\mathcal{A}_a}$ is selected,
5. the output parameters \mathcal{Out}_b of the branch – which are either one $\mathcal{Out}_b = o$ or none $\mathcal{Out}_b = \emptyset$ for atomic SIBs – are evaluated and if appropriate (i.e. $\#\mathcal{Out}_b \hat{=} 1$) the throwable or return value is written to the connected context variable c for $o \mapsto_{\mathcal{Out}} c$, and
6. the successor SIB $a_{succ} = \delta(a_{curr}, b)$ is evaluated.

Hence an atomic SIB is like a statement in the target language performing a single method call without any nested calls. The evaluation of input parameters differs for the static and dynamic variants:

- for a static input parameter $i_s \mapsto_{\sigma_{const}} const$ the constant $const$ is retrieved and used as actual parameter for the method, and
- for a dynamic input parameter $i_d \mapsto_{\sigma_c} c$ the mutable value mv defined as $c \mapsto_c mv$ of the context variable c is retrieved and used as actual parameter for the method.

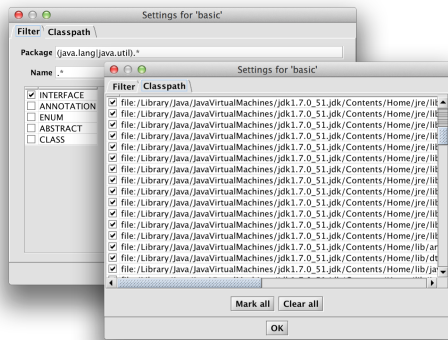


Figure 3.4: Both tabs of the settings dialog for service browsers.

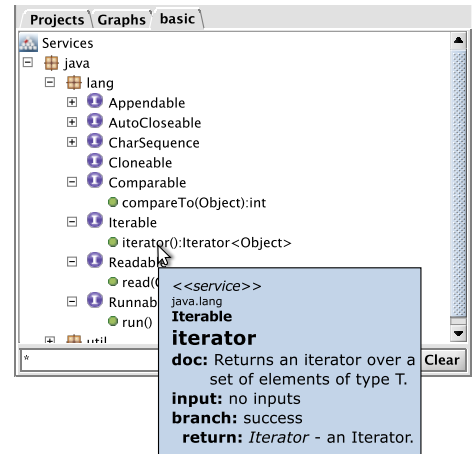


Figure 3.5: A view of a service browser offering some services of the JRE.

3.2.2 Service Browser

Atomic activities are created from *service SIBs*, i.e., static and dynamic SIBs. These each represent a method of a class, abstract class, or interface. The *service browsers* are responsible for making available these services. In the project browser, a new service browser may be created via a right-click on the project node, followed by selecting “Create service browser...” and entering a name for the new browser. Afterwards a new browser tab is added, which is stored to the project configuration, so that it is available beyond a restart of the environment.

In a service browser the offered services may be prefiltered via some configurations in a settings dialog (right-click on a node and select “Settings...”), which is depicted in Fig. 3.4. The dialog has two tabs. The first tab “Filter” offers to constrain the package and simple name of classes via regular expressions. Please note that in the spirit of *separation of concerns*, these are settings that are done by very few technical experts preparing a jABC4-project for domain experts who then will use the services to build technical SLGs as a library for application experts. Hence in the settings dialog, using regular expressions is adequate for the target audience. In addition the kind of returned classes may be specified in the table below (i.e., “interface”, “class”, ...). The second tab “Classpath” allows to choose, in which libraries the service browser should search for services. The classpath entries are built from the project classpath, i.e., it contains the transitive hull of dependencies defined in the projects maven `pom.xml` (cf. Sec. 3.1.4).

An example of a service browser offering some basic JRE services is depicted in Fig. 3.5. It consists of a tree view showing the package structure of the prefiltered Java types, and a *fast filter component*, which allows to do further filtering in a convenient way. The public methods of these classes shape the leaf nodes of the tree. It does not offer complete regular expressions as it should be used by domain

¹⁴I support this, since a method that returns the wrapper class `Boolean` of the primitive type `boolean` may return a `null` in Java.

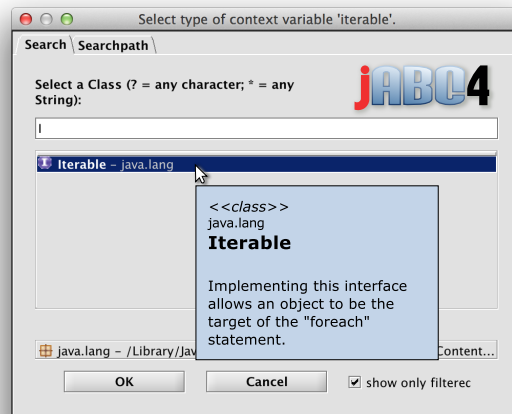


Figure 3.6: Class chooser for setting the type of a context variable labelled “iterable”.

experts and not necessarily by technical experts. Instead such a filter may contain wildcards like “?” for an arbitrary character and “*” for any number of arbitrary characters. The mouse cursor in Fig. 3.5 points at the method `iterate()` of the interface `Iterable` so that a tooltip with the documentation of the service is shown. I created a corresponding importer for the Javadoc of Java classes, which is then shown in the service browser as well as in input and output parameter windows of a respective activity.

Via drag and drop of a leaf node – representing a Java method – to the graph canvas a new atomic activity is created with an icon as illustrated in Table. 3.1. The input parameter, branches and output parameter are created automatically from the reflective information of the corresponding `Method` object as described in Sec. 3.2.1 as well as the corresponding type information.

3.2.3 Class Chooser

A context variable may be configured in the graph inspector (cf. Fig. 3.2). Along its label and mutable value (at runtime) it has a type (cf. Def. 3.3), which may be set via the *class chooser* in the case of Java types (cf. Sec. 3.3.4 for graph type parameters and Sec. 3.3.8 for graph types). This is done via a right-click on a context variable followed by “Edit type (<simple name of current type>)...”. Fig. 3.6 displays the class chooser for a context variable labelled “iterable”. The chooser consists of two tabs. The tab labelled “Searchpath” is the same as the “Classpath” tab of the settings dialog for service browsers (cf. Sec. 3.2.2). The selected class path elements are stored in the project configuration so that this can be prepared by a technical expert for a domain expert.

The currently shown tab labelled “Search” features a search panel, which has the same easy-to-use filter capabilities as the fast filter component of a service browser. The results are presented in a list below. Via tooltips, the documentation of the corresponding class can be inspected (again imported from the Javadoc of the class).

The central convenience feature of the class chooser is activated via the checkbox “show only filtered” situated in the bottom right of search panel. If it is selected, the results will contain classes that are compatible with the parameters connected to the context variable, only.

That means, if a variable is read by an input parameter of an activity with the interface **I**Iterable, then only every descendant in the inheritance tree – or *directed acyclic graph* (DAG) for interfaces – on the selected search path and **I**Iterable itself will be in the result list, i.e., every collection type that may be used in a “for each”-statement in Java. This is because the type of an actual parameter of a method has to be “below” the type of its formal parameter.

In contrast, if a variable is written by an output parameter of an activity with the interface **I**Iterable, then only **I**Iterable itself as well as every ancestor in the inheritance tree (or DAG) on the selected search path will be in the result list, i.e., **I**Iterable and **O**bject. This is because the type of an actual result of a method has to be “above” its formal result type.

If a context variables has some connected parameters, the matching class will often either be determined or there is no match at all, which is a sign for a mistake in the data-flow modeling. In both cases the user does not have to choose at all. In a future release, unambiguous types may even be assigned automatically¹⁵.

By selecting a class in the result list and hitting the “OK”-button, the class is set for the context variable. If the class has type parameters (like the **T** in **I**Iterable<**T**>) it is initially set to its upper bound (which is **O**bject for iterables). In the context menu of a context variable, a new menu item appears labelled “Type parameters. . .”.

For each type parameter it is then possible to use the class chooser to declare a concrete class for it. If the chosen type for a type parameter again has type parameters, they can be recursively set via the class chooser. Input and output parameters of activities connected to a context variable will automatically adapt the selected type arguments. If, e.g., the output parameter of an atomic activity representing the method `Iterator<T>#next():T` is connected to a String, the type Parameter **T** is automatically set to `String: Iterator<String>#next():String`.

Array types may be created, by choosing the context menu item “is array” on a context variable. A dialog asks for the dimensions of the array; the default is 1. Array types may be configured as type arguments, too.

3.3 Hierarchy

In this section I will show how I support hierarchical modeling [NSM13]. In short, the so called *graph SIBs* are templates for *abstraction activities* $\mathcal{A}_g \subseteq \mathcal{A}$ that reference a sub model. If the control flow reaches such an activity, it executes the sub model which returns the branch to impel to the next activity. Fig. 3.7 shows exemplarily two hierarchy levels, where on the n th level an abstraction activity references a technical sub model **G**Iterate<**T**> on the $n - 1$ th hierarchy level, which is part of

¹⁵Such a feature is called *type inference* in type theory [PS91].

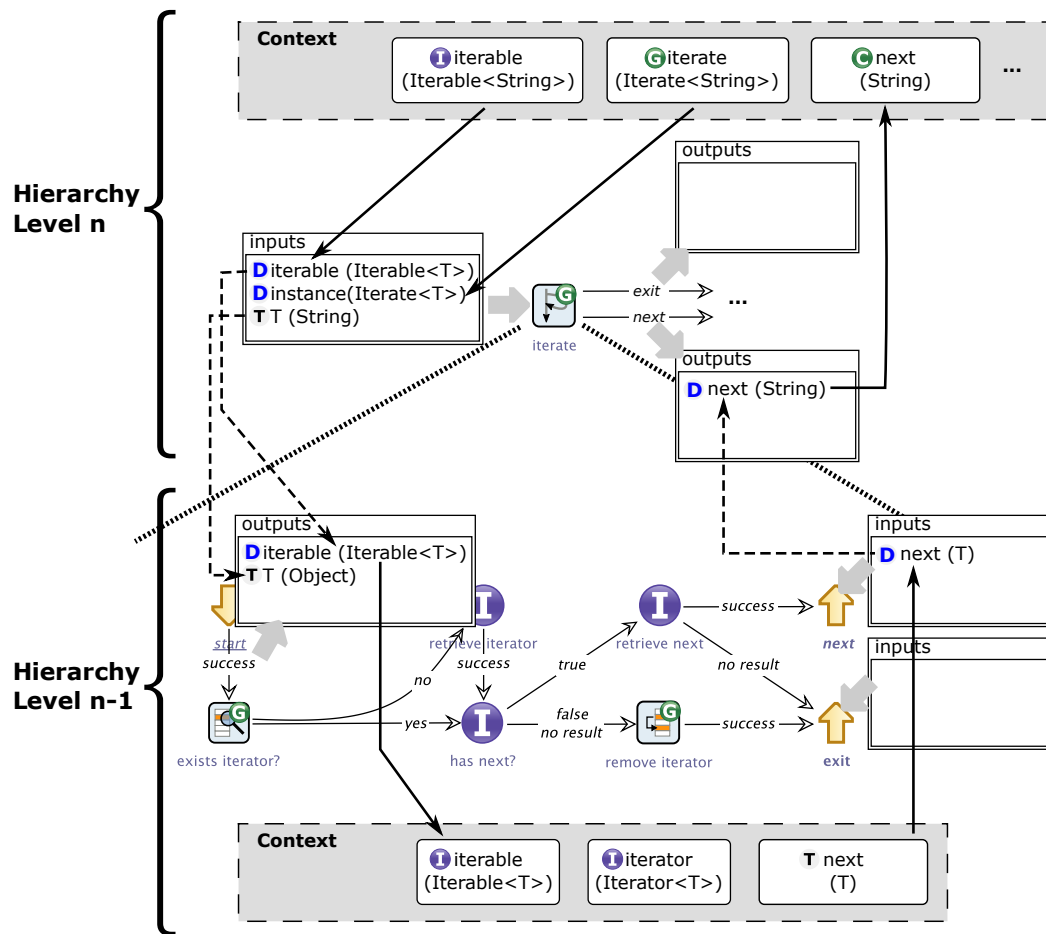


Figure 3.7: Input and output parameterization of a service graph using the example of an technical SLG for iterating through an iterable (e.g. a collection).

jABC4’s basic SLG library. The latter iterates through an arbitrary Java collection or array (i.e. anything that implements `java.lang.Iterable<T>`), one element each time it is executed. The first time `Iterate<T>` is invoked it retrieves the underlying iterator of the collection of type `Iterator<T>`. In each invocation `Iterate<T>` tries to get the next element of the iterator and returns the element via the “next” branch or chooses the “exit” branch if no more elements are in the collection. Afterwards it can be reused to iterate again through the same or another iterable.

In the example shown in Fig. 3.7 on the n th hierarchy level an abstraction activity labelled “iterate” is used to iterate through an iterable with `String` elements. For the running example `ChainReaction` the same graph may be used for iterating through the board of type `Field` implementing `Iterable<Cell>` which returns the cells from left to right and row by row from top to bottom. The corresponding concepts are described in the following.

3.3.1 Parameterization

Unlike jABC3 I explicitly do not support global variables for the communication between hierarchy levels. As stated in [WS73], global variables are harmful as they make a program less clear and hamper its static analysis. I go one step further and do not allow scopes, too. Instead the communication between hierarchy levels is completely defined via input and output parameters and each SLG has a local execution context, only. Since a consequent enforcement of this rule would lead to a vast amount of parameters not being easier to handle than global variables or scopes, I adopt the paradigm *inversion of control* (cf. Sec. 3.6), in order to inject values into local context variables from the environment.

The input and output parameterization of a graph is defined by a dedicated *input SIB* and one *output SIB* per branch.

Example 1: Input/Output parameterization of SLG $\textcircled{G}\text{ExampleGS}$

In the SLG $\textcircled{G}\text{ExampleGS}$ shown in Fig. 3.2 the input SIB has four input parameters, namely the “board” of type $\textcircled{C}\text{Field}$, “player” of the enumerated type $\textcircled{E}\text{Player}$ as well as an “x” and “y” coordinate of the current cell with type $\textcircled{C}\text{Integer}$. These parameters write to the corresponding context variables with same name and type (cf. the graph inspector in Fig. 3.2), so that they can be used throughout the cell evaluation.

Furthermore the graph has exactly one output SIB “success” with a single output parameter “evaluation” of type $\textcircled{C}\text{Integer}$. The output parameter reads from the equally labelled and typed context variable and is the score for the cell evaluated by the process with the given x and y coordinates, board configuration, and player.

In the following I will enhance the formal definition of SLGs accordingly.

Definition 3.11. A *dual IO kripke transition system with context* – denoted by *service graph* – over a type system \mathcal{T} and a finite set of atomic propositions AP is a structure $dKTS_C^{I/O} = (n, p, \mathcal{A}, a_0, \mathcal{A}_{Out}, \mathcal{B}, \delta, \mathcal{I}, \mathcal{C})$, where

- n is the simple name of the service graph,
- p is the package name of the service graph, which is a tree structure just like the Java package structure,
- together n and p build the full qualified name of the graph,
- \mathcal{A} is an activity domain (cf. Def. 3.8),
- $a_0 \in \mathcal{A}$ a dedicated start activity, i.e., an *input SIB*,
- $\mathcal{A}_{Out} \subseteq (\mathcal{A} \setminus a_0)$ a finite set of *end activities*, i.e., *output SIBs*,
- \mathcal{B} is a *branch domain* (cf. Def. 3.7),
- $\delta : \mathcal{A} \times \mathcal{B} \rightarrow \mathcal{A}$ is a *transition function*,

- $\mathcal{I} : \mathcal{A} \rightarrow 2^{AP}$ is an *interpretation function*, and
- \mathcal{C} is an execution context shaping the context of shared resources (cf. Def. 3.3).

Further on, a modeler may add meta-information to a service graph like an icon and documentation for the graph itself as well as its input and output parameterization. All this information can be added to the components directly in the jABC4 development environment.

An *input SIB* is the start activity a_0 of a $dKTS_C^{I/O}$ visualized via a yellow arrow pointing downwards (cf. activity “start” on *hierarchy level* $n - 1$ in Fig. 3.7). The intuition is, that it defines the formal parameters of the process model, which are passed as actual parameters from an activity one hierarchy level above. Therefore a_0 defines the input parameters of the SLG as output parameters of its sole branch “success”. At runtime it is executed first in the process, writes the actual input parameters to context variables, and follows the “success” branch to the successor activity. An input SIB has the following properties:

- There is exactly one input SIB for each SLG.
- An input SIB has no ingoing transitions: $\nexists a \in \mathcal{A}, b \in \mathcal{B}. \delta(a, b) = a_0$.
- An input SIB is structured as follows $a_0 = (l_{a_0}, \mathcal{B}_{a_0})$ where
 - $l_{a_0} \in \mathcal{L}_{\mathcal{A}}$ is the label of the activity which is always “start”,
 - there are no input parameter for an input SIB,
 - $\mathcal{B}_{a_0} =_{\text{def}} \mathbf{b}(l_{a_0}) \subseteq \mathcal{B}$ consists of one branch b labelled “success”, and
 - $Out_{success}$ is the finite set of output parameters (i.e. $b \mapsto_{\mathcal{B}} Out_{success}$) that defines the formal input parameters of the process model.

For better presentation $dKTS_C^{I/O} \mapsto_{a_0} Out_{success}$ will be used.

The *output SIBs* are the end activities $a_{e1} \cup \dots \cup a_{en} = \mathcal{A}_{Out}$ (cf. activities “next” and “exit” on *hierarchy level* $n - 1$ in Fig. 3.7). Intuitively, each output SIB defines a branch of the calling graph SIB. The input parameters of the output SIB declare the formal output parameters of the branch, so that each branch may have its own outputs. As the control-flow reaches an end activity it evaluates its input parameters (either from the context or constants) and provides them together with the selected branch as actual output parameters to the calling activity. An output SIB has the following properties:

- There is exactly one output SIB per branch, arbitrary many for an SLG.
- An output SIB is structured as follows $a_{ex} = (l_{a_{ex}}, \mathcal{I}n_{a_{ex}})$
 - $l_{a_{ex}}$ is the label of the activity which corresponds to the name of the branch it defines,
 - $\mathcal{I}n_{a_{ex}}$ is the input parameter domain, defining the formal output parameters of the branch this output SIB represents, and
 - an output SIB has no branches $\mathbf{b}(l_{a_{ex}}) = \emptyset$ and therefore no successors on the current hierarchy level.

New SLGs are equipped with a start activity as well as end activities via dragging from the “I/O” area in the graph inspector (cf. Fig. 3.2) the corresponding icon (cf. Table 3.2) and dropping them to the graph canvas. For end activities a dialog pops up for entering the label of the activity defining the name of the branch it represents.

The input and output parameter for atomic activities (cf. Sec. 3.2.1) as well as abstraction activities (cf. Sec. 3.3.2) are automatically created from the respective underlying structure, i.e. a Java method for atomic activities and the input/output parameterization of an SLG, as described in the current section, for an abstraction activity. This is different for start and end activities of an $dKTS_c^{I/O}$. The output parameters $Out_{success}$ of a start activity as well as the input parameters In_{aex} are empty at the beginning. In the corresponding output (input) parameter window, a new parameter is created via the context menu and the menu item “Add input...” for a start activity as well as “Add output...” for an end activity. As for context variables a name has to be entered for a new parameter. Initially a parameter is of type **Object**. This may be changed via the context menu analogous to context variables (cf. Sec. 3.2.3 and 3.3.8).

3.3.2 Abstraction Activities

The input and output SIBs of the sub process define the formal parameters of an SLG. An invocation of a sub process is realized via an activity (i.e. abstraction activities) that references the sub process and defines the actual parameters similar to the atomic activities.

Definition 3.12. The *abstraction activities* (i.e. instantiation graph SIBs) \mathcal{A}_g shape a subset of the activities \mathcal{A} . Each represents a sub model. If an abstraction activity references a service graph its’ template is denoted by *service graph SIB* (cf. Sec 3.5 for more variants of abstraction activities). They inherit the structure of IO activities (cf. Def. 3.8) and adopt the simple name of the sub model as label as well as bind the activities’ input parameters to the output parameters $Out_{success}$ of the input SIB, and the branches to the output SIBs of the sub model.





Additional an instance parameter is bound to the activity which references a process instance of the respective service graph to be executed. Thus a domain of abstraction activities is a structure $\mathcal{A}_g = (\mathcal{L}_{\mathcal{A}_g}, \mathcal{In}_{\mathcal{A}_g}, \mathcal{B}_{\mathcal{A}_g}, \mathbf{i}_{\mathcal{A}_g}, \mathbf{b}_{\mathcal{A}_g})$ where

- $\mathcal{L}_{\mathcal{A}_g} \subseteq \mathcal{L}_{\mathcal{A}}$, $\mathcal{In}_{\mathcal{A}_g} \subseteq \mathcal{In}$, $\mathcal{B}_{\mathcal{A}_g} \subseteq \mathcal{B}$, $\mathbf{i}_{\mathcal{A}_g} : \mathcal{L}_{\mathcal{A}_g} \rightarrow \mathcal{In}_{\mathcal{A}_g}$, and $\mathbf{b}_{\mathcal{A}_g} : \mathcal{L}_{\mathcal{A}_g} \rightarrow \mathcal{B}_{\mathcal{A}_g}$ are defined similar to the corresponding structures in Def. 3.8.
- An abstraction activity (cf. activity “Iterate” on *hierarchy level n* in Fig. 3.7) maps the formal input and output parameters of a sub model to actual parameters of the activity, and represents the invocation of a sub process. It is a structure $a = (l_a, i, \mathcal{In}_a, \mathcal{B}_a)$ where
 - l_a is initially set to the simple name n of the sub model, but may be changed to an arbitrary domain-specific label lateron,
 - $i \in \mathbf{i}_{\mathcal{A}_g}(a)$ is an *instance parameter* referencing a process instance in the context,

- $\mathcal{I}n_a =_{\text{def}} (i_{\mathcal{A}_g}(a) \setminus i)$ is the input parameter domain. Each input parameter is either read from the context or a constant value and stored to the context of the sub process via its start activity, and
- $\mathcal{B}_a =_{\text{def}} b_{\mathcal{A}_g}(a)$ is the branch domain, one branch for each end activity of the sub process. At runtime it takes the outputs provided by the chosen output SIB (either read from the context or a constant value) and stores them to the context as defined in the abstraction activity.

The aforementioned icon configured for a graph is used as node visualization on the corresponding abstraction activities (e.g. activity “Iterate” on *hierarchy level n* in Fig. 3.7). The documentation is available context sensitively at several points in the jABC4.

Example 2: Description of SLG ExampleGS

Fig. 3.2 shows in the graph canvas a graph labelled ExampleGS, which uses abstraction activities from different SLG libraries (illustrated via the little green ‘G’ in the top right of the corresponding node icons) in order to evaluate the worthiness of a cell for the ChainReaction scenario. At first it uses the basic SLG PutToContext<Integer> in order to store the initial value 0 to the context variable “evaluation” in activity “initialize evaluation”. Then the activity “type of cell?” is used to decide for a bonus of +10 for corner cells and +5 for edge cells. The bonus is added via the graph Add from the basic SLG library. In short, ExampleGS then checks whether

- the cell is critical (i.e., has $\#capacity - 1$ many atoms) and endangered (i.e., at least one neighbor cell of the opponent is critical) which leads to a malus of -10, or
- the cell is not critical, but endangered which leads to a bonus of +10.

3.3.3 Stateless and Stateful Processes

In the HOPE approach the values in the execution context are first class citizens, i.e., they do not only hold data values, but may be executed. This is true for Java objects as well as process instances. Via the instance parameter of an abstraction activity the modeler can steer the behavior for sub process invocations.

If the instance parameter does not read a context variable, a new process instance will be created on every occasion the control-flow reaches the activity. This is the default behavior as it had been in the jABC3 framework. I call the resulting process instances *stateless processes* since they (and therefore their execution context) last for a single execution, only.

The process instance for an instance parameter that reads from a context variable will be created and stored to the respective variable if it is undefined. If there is already a process instance in the context, it will be reused. E.g., on *hierarchy level n* in Fig. 3.7 the first time activity “iterate” is executed a new process instance

of (graph) type $\mathbb{G}\text{Iterate}\langle\text{String}\rangle$ is created and stored to the context variable “iterate”. If the control-flow reaches the abstraction activity “iterate” again, the existing process instance with its current state (execution context) is executed again.

A process instance might be available in the context, because the instance has been

- injected from the environment (cf. Sec. 3.6),
- created for the execution of an abstraction activity,
- instantiated via a constructor activity (cf. Sec. 3.3.5), or
- passed in as an input parameter to the current process

As such a process instance and its execution context may be reused for repeated execution I denote them as *stateful processes*.

3.3.4 Type Parameters

Modern object-oriented programming languages like Java, C#, and Scala have a feature known as *parametric polymorphism* in type theory¹⁶. In all three languages mentioned before this feature is called *generics*. Subtype polymorphism allows to have virtual methods whose definition can be overridden in sub classes so that at runtime it is decided which implementation is executed due to the instance on which it is executed. In contrast, parametric polymorphism adds type parameters to class (and method) declarations, which may be set or constrained during inheritance or instantiation via type arguments.

The HOPE approach interprets processes as a special form of functions reminiscent to functional programming and allows type parameters in their declaration. This enables generic service graphs like the $\mathbb{G}\text{Iterate}\langle T \rangle$ -graph (cf. Fig. 3.7), which has a type parameter for the elements of the collections it may iterate. In an abstraction activity, the type parameters can be instantiated, without touching the underlying process model. This already gives a great amount of modeling-time variability, which is brought to another level by runtime variability in Sec. 3.5. The additional complexity in the useability for non-programmers – whom I adress with HOPE – is coped in Sec. 3.4 via preconfiguration of activities, hiding complex information (and their configuration) from users with less technical expertise, and hence supporting *separation of concerns*. This comprises presetting type arguments.

I provide a very simplified version of “generics” for graphs, which enables to define a *graph type parameter* and its upper bound (which is $\mathbb{G}\text{Object}$ by default), which can then be used in the execution context of the graph and its activities as if it is of the type set as upper bound. Hence a type parameter declaration enhances the type system \mathcal{T} via the type parameters $\mathfrak{P}\text{aram}$ to an “augmented” type system $\mathcal{T}_{\mathfrak{P}\text{aram}}$ ¹⁷, which is valid in the scope of the current graph. The graph type parameters are

¹⁶

¹⁷I use the term “augment” here since I refer to adding new types to a type system, not changing its capabilities.

declared for a complete graph and therefore I enhance the definition of service graphs accordingly.

Definition 3.13. A *Generic Dual IO Kripke Transition System with Context* – denoted by *generic service graph* – over an enhanced type system $\mathcal{T}_{\mathfrak{P}aram}$ and a finite set of atomic propositions AP is a structure $gKTS_C^{I/O} = (n, p, \mathfrak{P}aram, \mathcal{A}, a_0, \mathcal{A}_{Out}, \mathcal{B}, \delta, \mathcal{I}, \mathcal{C})$, where

- $n, p, \mathcal{A}, a_0, \mathcal{A}_{Out}, \mathcal{B}, \delta, \mathcal{I}, \mathcal{C}$ are defined analogously to $dKTS_C^{I/O}$ in Def. 3.11,
- $\mathfrak{P}aram$ is a set of *graph type parameters* (the formal parameters for parametric polymorphism), which are used to shape the “augmented” type system for this graph: $\mathcal{T}_{\mathfrak{P}aram} = \mathcal{T} \cup \mathfrak{P}aram$. The corresponding semantics is similar to the *generics* in Java [Blo08].
- A graph type parameter $p \in \mathfrak{P}aram$ is a structure $p = (l_p, u)$ where
 - l_p is a label for the graph type parameter like `T` or `Element`, and
 - $u \in \mathcal{T}_{\mathfrak{P}aram}$ is the upper bound of the type parameter.

In an abstraction activity referencing a generic sub model – denoted by *parameterized abstraction activities* – (and therefore invoking a parameterized sub process at runtime) the type parameter have to be set to a type of the current type system, i.e., either a type parameter of the current graph or any other type in the type system. These are the actual parameters $\mathfrak{A}rg$ for the parametric polymorphism and denoted by *graph type arguments*. On *hierarchy level n* in Fig. 3.7 activity “iterate”, e.g., parameterizes the type parameter `T` to `String`.

Definition 3.14. The *parameterized abstraction activities* \mathcal{A}_{pg} are a subset of the abstraction activities \mathcal{A}_g . Each represents a generic sub model (i.e. a $gKTS_C^{I/O}$). They inherit the structure of abstraction activities (cf. Def. 3.12) and bind type arguments for the type parameters of the underlying process model. Thus domain of parameterized abstraction activities is a structure $\mathcal{A}_{pg} = (\mathcal{L}_{\mathcal{A}_{pg}}, \mathfrak{A}rg, \mathcal{I}n_{\mathcal{A}_{pg}}, \mathcal{B}_{\mathcal{A}_{pg}}, \mathfrak{a}, \mathfrak{i}_{\mathcal{A}_{pg}}, \mathfrak{b}_{\mathcal{A}_{pg}})$ where

- $\mathcal{L}_{\mathcal{A}_{pg}} \subseteq \mathcal{L}_{\mathcal{A}_g}$, $\mathcal{I}n_{\mathcal{A}_{pg}} \subseteq \mathcal{I}n_{\mathcal{A}_g}$, $\mathcal{B}_{\mathcal{A}_{pg}} \subseteq \mathcal{B}_{\mathcal{A}_g}$, $\mathfrak{i}_{\mathcal{A}_{pg}} : \mathcal{L}_{\mathcal{A}_{pg}} \rightarrow \mathcal{I}n_{\mathcal{A}_{pg}}$, and $\mathfrak{b}_{\mathcal{A}_{pg}} : \mathcal{L}_{\mathcal{A}_{pg}} \rightarrow \mathcal{B}_{\mathcal{A}_{pg}}$ are defined similar to the corresponding structures in Def. 3.12.
- $\mathfrak{A}rg$ is a finite set of type arguments (the actual parameters for parametric polymorphism),
- $\mathfrak{a} : \mathcal{L}_{\mathcal{A}_{pg}} \rightarrow \mathfrak{A}rg$ maps type arguments to parameterized abstraction activities,
- a type argument $r \in \mathfrak{A}rg$ is a structure $r = (l_r, p, t)$ where
 - l_r is a label set to the name of the referenced type param (i.e. it has its own UUID but adopts the name of the label),
 - $p \in \mathfrak{P}aram_{sub}$ is the referenced type parameter of the set of type parameters $\mathfrak{P}aram_{sub}$ declared in the referenced service graph `sub`,

- $t \in \mathcal{T}_{\text{param}_{\text{curr}}}$ is the argument type for p , which is a type in the “augmented” type system of the current service graph **curr** (on the hierarchy level of the activity). The type t has to be a subtype of the upper bound of the type parameter p .
- A parameterized abstraction activity maps the formal input and output parameters as well as the type parameter to actual parameters and type arguments of the activity, and represents the call of a generic sub process. It is a structure $a = (l_a, \mathcal{A}rg_a, i, \mathcal{I}n_a, \mathcal{B}_a)$ where
 - $l_a, i, \mathcal{I}n_a, \mathcal{B}_a$ is defined analogously to abstraction activities (cf. Def. 3.12), and
 - $\mathcal{A}rg_a =_{\text{def}} \mathbf{a}(l_a)$ is a set of type arguments, which are used to parameterize the type parameter of the underlying process model.

A new graph type parameter is defined in the start activity of an SLG (i.e., an input SIB). As they are formal parameters of a $gKTS_c^{I/O}$, they are declared along with the input parameter of the graph as output parameters Out_{success} of the start activity’s branch “success” (cf. Sec. 3.3.1). In Fig. 3.7 on hierarchy level $n - 1$ the output parameter window of the start activity declares the graph type parameter **T** highlighted via the icon **T**.

A graph type parameter may be used as a type for context variables, input parameters, or output parameters of the respective graph and as graph type arguments. Hence besides using the class chooser for selecting a Java class, the context menu for editing types has an item denoted “Choose type parameter...”. It offers a dialog with a combo box for selecting one of the graph type parameters declared for the current SLG. In the graph **GIterate**<**T**> (cf. hierarchy level $n - 1$ in Fig. 3.7) the context variables “iterable”, “iterator”, and “next” as well as output parameter “next” of end activity “next” reference the graph type parameter **T** either directly or as a type argument (e.g., **T** in **IIterator**<**T**> or **GIterable**<**T**>).

On hierarchy level n the generic abstraction activity “iterate” (cf. Fig. 3.7) shows the graph type parameter as actual parameter (i.e., a type argument) in its input parameter window. Type arguments are analogously highlighted via the icon **T**. In the example a modeler has set the type argument to **GString**, so that it may iterate through **IIterable**<**String**> and returns an output parameter “next” of type **GString** for the branch “next”. A type arguments type is set or updated to a Java type, graph type, or type parameter of the SLG on hierarchy level n just like the type of a context variable as described in Sec. 3.1.3, 3.2.3, and 3.3.8.

3.3.5 Constructors

A new process instance is created and stored if the context variable connected to the instance parameter of an abstraction activity is undefined. The instance may be reused for execution later on (cf. Sec. 3.3.3). Often it is required to create a process instance, but delay its first-time execution. A use case is, that a process should be created and passed to a sub process, where it is executed via an abstraction activity

at some point This is discussed more detailed in Sec. 3.5 in the context of runtime variability.

For the purpose of creating new process instances I designed the *constructor activities*. At the time of writing a constructor activity has no input parameter, a type argument for each type parameter of the referenced service graph (if it is a generic service graph), and exactly one branch labelled “success” with a sole output parameter labelled “instance” referencing the respective service graph. If the execution reaches such an activity, a process instance of the corresponding (generic) service graph is created and stored to the bound context variable of the output parameter “instance” of the branch “success”. Hence a constructor activity can directly be compared to the `new`-statement in Java.

Definition 3.15. The *constructor activities* \mathcal{A}_c shape a subset of the abstraction activities \mathcal{A}_g . Each represents a (generic) sub model (i.e. either a $dKTS_C^{I/O}$ or a $gKTS_C^{I/O}$). They bind type arguments for the type parameters of the underlying process model, but do not handle any input or output parameterization of the sub model, as it constructs instances of it, but does not execute them. Thus a domain of constructor activities is a structure $\mathcal{A}_c = (\mathcal{L}_{\mathcal{A}_c}, \mathfrak{Arg}, \mathcal{B}_{\mathcal{A}_c}, \mathfrak{a}, \mathfrak{b}_{\mathcal{A}_c})$ where

- $\mathcal{L}_{\mathcal{A}_c} \subseteq \mathcal{L}_{\mathcal{A}_g}$, $\mathcal{B}_{\mathcal{A}_c} \subseteq \mathcal{B}_{\mathcal{A}_g}$, and $\mathfrak{b}_{\mathcal{A}_c} : \mathcal{L}_{\mathcal{A}_c} \rightarrow \mathcal{B}_{\mathcal{A}_c}$ are defined similar to the corresponding structures in abstract activities (cf. Def. 3.12). The binding of structures of the associated sub model is described separately in the following.
- \mathfrak{Arg} , and $\mathfrak{a} : \mathcal{L}_{\mathcal{A}_c} \rightarrow \mathfrak{Arg}$ are defined analogously to parameterized abstraction activities (cf. Def. 3.14).
- A constructor activity maps the type parameter of the referenced sub model to type arguments of the activity, creates an instance of the sub model, and stores it to the context variable bound to the output parameter “instance”. It is a structure $a = (l_a, \mathfrak{Arg}_a, \mathcal{B}_a)$ where
 - \mathfrak{Arg}_a is defined similar to a parameterized abstract activity (cf. Def. 3.14), and
 - \mathcal{B}_a consists of a single branch labelled “success” with a single output parameter parameter labelled “instance”, which is used to write the newly created process instance to a context variable.

A constructor activity is not to be confused with an atomic activity executing a Java constructor. The latter is defined as a static SIB (cf. Sec. 3.2.1).

3.3.6 Graph Inspector

The graph inspector is used to configure SLGs. It is depicted in the bottom left of Fig. 3.2:



- The “Package” and “Name” field define the full qualified name of an SLG.
- The combo box “Type” differentiates between service graphs (cf. Sec. 3.3.1) and interface graphs (cf. Sec. 3.5.1).

start	end	service	interface	constructor
				

Table 3.2: Node visualization of activities for modeling hierarchical SLGs.

- “Icon” allows to set a domain-specific icon for a graph, which is then used as node visualization for referencing abstraction activities. A domain-specific icon overrides the default visualization depicted in Table 3.2, but is overlaid with the icon for service, interface, or constructor graph SIBs, respectively (cf. the small overlay icons in the top right of the icons in columns “service”, “interface”, and “constructor”).
- Start and end activities are created from the “I/O” box as described in Sec. 3.3.1.
- The “Variables” list contains the context variables of the current SLG (cf. Sec. 3.1.3).
- For service graphs the implemented interface graph is visualized in the “interfaces” list (cf. Sec. 3.5.1). There may be only one interface per service graph at the time of writing.

3.3.7 Graph Browser

In jABC3 the SLGs are shown in the project browser as they are saved in the project folder as XML files. This is still the case, but since SLGs of jABC4 have a full-qualified name and are available via SLG libraries beyond project boundaries, the *graph browser* tab has been added. The browser is situated in the top-left of the jABC4 main window (cf. Fig. 3.2). It shows all graphs on the project class path as well as the SLGs defined in the project itself in a tree view according to their package structure. The symbol left of the simple name of an SLG depicts whether it is a service graph  or interface graph  (cf. Sec. 3.5.1). In Fig. 3.2 some basic integer math operations of the basic SLG library are shown.

As for the service browser a fast filter box assists the modeler in finding a graph. The domain expert may prepare the shown SLGs by configuring the dependencies via maven, so that the SLG libraries available are tailored to the needs for the project. The graph browser may even be hidden completely. This may be reasonable for projects that have prepared service browsers or IO browsers (cf. Sec. 3.4.2), so that a modeler does not need or should not see the graph browser.

An abstraction activity referencing an SLG is created by dragging a leaf node of the graph browser to the graph canvas. The new activity automatically has the input parameters, branches, and output parameters declared in the sub model as well as the domain-specific icon set in the graph inspector or the default icon as shown in Table 3.2 if no domain-specific icon has been configured.

New constructor activities are instantiated by holding the “CTRL”-key down while dragging and dropping a service graph onto the graph canvas. No input parameters, branches, or output parameters of the sub model are created, but the type arguments as well as a “success”-branch returning a process instance of the given type.

3.3.8 Graph Chooser

Context variables as well as input and output parameters of an SLG (via the start and end activities) are typed. As described in Sec. 3.2.3 a class chooser may be used to select a Java type, like a class, interface, or enumeration. Analogously a graph type may be configured. This is done by a right-click on a context variable or input (output) parameter of an SLG and selecting the menu item “Choose graph. . .”. This opens the graph chooser, which is a dialog similar to the graph browser component. It shows the SLGs on the project class path, offers likewise a fast filter box, and has an “OK” and “Cancel”-button. The latter does not change the type of the respective component. If the “OK”-button is chosen, the currently selected leaf node representing an SLG is selected as graph type. For context variables the graph type is inferred automatically via the connected input/output parameters of activities in the most cases, so that it is not necessary to do it manually.

If the selected SLG is a generic graph, the context menu of the respective component (i.e. context variable or parameter) will offer to configure the type of these type arguments, too (cf. the type `GIterate<String>` of context variable “iterate” on hierarchy level n in Fig. 3.7). Initially each is set to the upper bound of the type parameter. At the time of writing it is not possible to use graph types in type arguments, neither for graph types nor for Java types. This is not a technical restriction, but again a decision which may be reconsidered in the future on an ‘as needed’ basis.

3.4 Configuration

Atomic activities have already been introduced in Sec. 3.2.1, in order to bind service (i.e. method) executions to activities in technical processes as well as abstraction activities in Sec. 3.3.2 for creating SIB libraries representing service graphs in a hierarchical fashion. The former give a foundation in the target language ensuring executability. The latter abstract from technical details via hierarchical modeling and assembly of SIB libraries. Together this allows to create executable domain-specific processes staying in the same modeling language, i.e. without the need to program any SIB classes or to use any other form of description language as in former jABC versions or its predecessor ABC. Even the decision between dynamic and static input parameters can be made on the modeling level.

But still due to the lack of a SIB class or some similar construct – as it has been available in jABC3 and its predecessors – it is not possible to hide access to the execution context. Every communication with context variables has to be modeled explicitly. This limitation is tackled by a configuration layer. Furthermore it enables to abstract from additional details without introducing any technological breaks.

3.4.1 Configuration Graphs

The configuration layer is realized via *configuration graphs*, SLGs without any control-flow information, i.e. no (or empty) transition function $\delta : \mathcal{A} \times \mathcal{B} \rightarrow \mathcal{A}$ (cf. Def. 3.11), declaring activities with (partially) preconfigured data-flow and further meta-information denoted by *preconfigured activity*. A modeler may

- simply drag-and-drop activities – atomic as well as abstraction activities – to the canvas of a configuration graph,
- create context variables,
- connect context variables to input and output parameters of the activities,
- set type arguments of a generic abstraction activity,
- change the icon of an activity,
- set the label of activities, and
- add further meta-data as, e.g., validations or temporal formulae for verification.

Example 3: Produce a “Bonus”-SIB from an “Add”-SIB

In the ChainReaction scenario, e.g., it is a common task to add a bonus or subtract a malus from the context variable holding the cell evaluation “evaluation” (cf. the graph canvas in Fig. 3.2). The configuration feature described in this section enables to create a new domain-specific SIB named `de.jabc4.strategies.cr.Bonus` from an existing basic SLG `Add`. The latter adds Integer “a” to “b” and stores the result “result” to a third integer. During the preconfiguration (cf. Fig. 3.8) an abstraction activity of type `Add` is created and labeled “de.jabc4.strategies.cr.Bonus” in the graph canvas of a configuration graph named `CR` (short for ChainReaction). Then the first parameter is switched to a static parameter with initial value 1, and the second is configured to read from the context variable “evaluation”. The output parameter “result” is set to write to the variable “evaluation”, too. Finally, the icon of the activity is changed, so that it shows `+n` instead of a `+` as the SLG `Add` does (cf. activities with label “+5” or “+10” in Fig. 3.2). The new activity can be used without any configuration. If a modeler uses the activity “Bonus”, but wants to change the amount of bonus, he or she may change the value of the static parameter “a” as for any other activity. In contrast, the communication with the context must not be configured manually anymore, but is still available (and adaptable) on the modeling level, e.g., for validations.

Then a new *io browser* can be created from a configuration graph in the spirit of service- (cf. Sec. 3.2.2) and graph browsers (cf. Sec. 3.3.7), just by right-clicking on a SIB in the graph browser and selecting “create new io browser”.

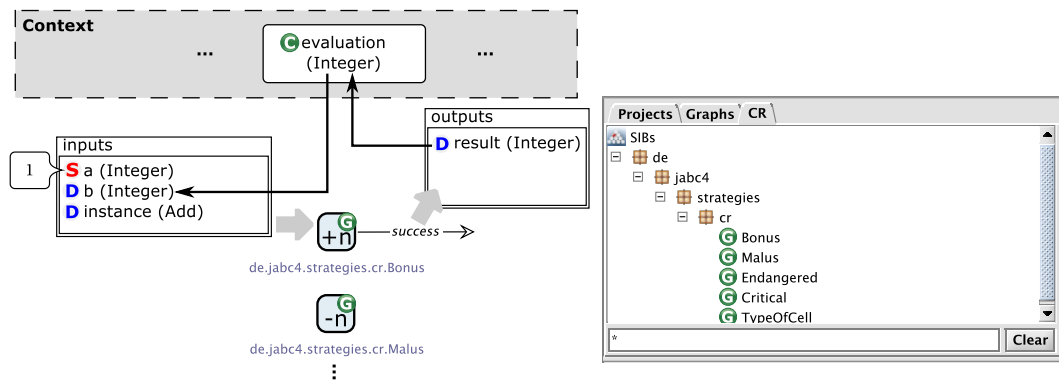


Figure 3.8: Example configuration graph for the ChainReaction Scenario.

Figure 3.9: Example IO browser for the configuration graph shown in Fig. 3.8.

3.4.2 IO Browser & Preconfigured Activities

An io browser (cf. Fig. 3.9) presents all the activities in the configuration graph (the *templates* for preconfigured activities) as *io SIBs*, to be dragged-and-dropped onto the canvas of another graph. A new activity (the *instantiation* of a preconfigured activity) is then created in the target graph gaining the properties of the preconfigured activity.

If an input or output parameter of the preconfigured activity is connected to a context variable, a new context variable is created which references the original variable (enhancing the definition of context variables with a reference to a *source variable*), so that if the same or another preconfigured activity referencing the very context variable is instantiated in the current process model, the formerly auto-created variable will be reused. Technically this is achieved by falling back to the UUID of the label (cf. Def. 3.1) of a context variable: Both the source- and the auto-created variable have the same name, but different UUIDs, though the auto-created variable references the UUID of the source variable, in order to have distinguishable context variables that still are associated.

Auto-created context variables are matched transitively, i.e., such a variable always stores the “root” context variable as source variable. Hence if a preconfigured activity of configuration graph A (communicating with the execution context) is instantiated in configuration graph B and C, they are again preconfigured activities, which may both be instantiated in the concrete service graph D. These instantiated preconfigured activities use the same context variables defined in A. This enables to combine SLG libraries from different domains easily. This has construction-defined limitations:

- If a template for a preconfigured activity is changed in a configuration graph, its instantiations are not updated accordingly as they are copies, that are autonomous (and can be changed independently). This could be changed in future releases such that an instantiation of a preconfigured activity *references* its template and learn changes to the template. All changes to the instantiation

would have to be stored in an additional layer, so that they are not overwritten by changes to the template.

- At the time of writing every context variable has exactly one “root” source variable, so that auto-created context variables of different configuration graphs (assuming the type matches) cannot be merged to one context variable. This would be helpful in order to merge preconfigured activities (that have no common ancestor configuration graph) of different domains to communicate with each other.

Both limitations may be overcome in future releases of the jABC4.

The labels of preconfigured activities are used as the identifier of the SIBs in the IO browser. For atomic activities the package and simple name of a class, together with the method name structure the service browser as a tree, whilst for abstraction activities the package p and name n shape the tree structure in the graph browser. Analogously, the label of a preconfigured activity is evaluated as a full qualified name with a ‘.’ as the separator. The package structure is created from the elements before the last ‘.’ and the label of the IO SIB itself is the part after the last ‘.’¹⁸ (denoted as *simple name*). If there is no ‘.’ in the label, it will be put into the “default” package. When an IO SIB is dropped onto the graph canvas, the simple name will be used as initial label. All properties inherited from a preconfigured activity are may be modified/changed on the newly instantiated activity.

With configuration graphs, a modeler is now able to preconfigure activities on every hierarchy level and therefore create new activities from existing ones just by

- reorganizing them in taxonomies via a full-qualified name in the label,
- predefining the input and output parameterization, and
- setting input parameters to initial constant values.

This enables to permeate further the principle of *separation of concerns*:




1. A programmer may prepare a library as a plain old Java API.
2. A technically versed modeler creates generic technical SLGs adapting to the underlying API via service browsers (cf. Sec. 3.2).
3. A domain expert prepares service graphs (for hierarchical abstraction) as well as configuration graphs via the graph browser (cf. Sec. 3.3 and this section) tailoring the technical SLGs as well as service graphs from one hierarchy level above for a given domain.
4. These service and configuration graphs – on every hierarchy level – may be published as SLG libraries.
5. An application expert uses selected SLG libraries on a tailored abstraction level to describe his requirements to an application.

¹⁸In Java packages use the same notation.

Finally, steps (3) and (4) may be applied repeatedly resulting in a “pile” of hierarchy levels, so that the *separation of concerns* may be pervaded in a *divide-and-conquer* fashion, thereof implementation tasks become manageable for all participants. This is in particular supported by the closed-fashioned of the execution context, communicating via well-formed input/output parameterizations.

In an application-specific jABC4-project, the domain expert may then choose from the configuration graphs, which should be used for the io browsers and hide the graph as well as any service browser, so that an application expert is able to concentrate on his domain via tailored activities.


Example 4: IO browser for the ChainReaction scenario

In Fig 3.8 there is already a hint, that the configuration graph defines a complete library of activities for the ChainReaction domain via the vertical dots beneath the activity “de.jabc4.strategies.cr.Malus”. Example 3 describes how an activity for adding a bonus to the cell evaluation may be preconfigured, from a basic SLG adding two integers. The malus is realized analogously via a preconfigured negative value for the malus to be added as well as a different icon showing a -n. The corresponding IO browser is shown in Fig. 3.9 with some more preconfigured activities. They are structured regarding the package definition in the labels of the preconfigured activities. A  Bonus-SIB may be dragged and dropped to the graph canvas of an arbitrary SLG and the context variable “evaluation” is created and wired to the new activity. If a second  Bonus- or e.g. a  Malus-SIB is added to the SLG as a new activity, they will read from and write to the same context variable “evaluation”.

3.5 Variability

By allowing to bind *virtual methods*¹⁹ to dynamic SIBs and connecting the instance parameter $i =_{\text{def}} (\mathcal{I}n_a \setminus \mathcal{I}n_{ad})$ to context variables, at runtime the value of the context variable may be an instance of an arbitrary sub class of the class declared in the context variable. Thus polymorphic service calls are directly supported, since the behavior of a technical SLG may change without touching the process, just by parameterizing it with instances of different Java types at runtime [NS14]. Hence besides data objects Java objects are allowed in the context as first-class citizens reminiscent to functional programming.

Example 5: Java objects as first-class citizens in the execution context

The abstraction activity on hierarchy level n in Fig. 3.7 has the input parameter “iterable” of type  Iterable<T>. At runtime an instance of any class implementing the interface, like ArrayList, LinkedList, or HashSet may be provided as actual

¹⁹Virtual methods are dynamically bound as they override a method of a base class. Calling such a method on a variable defined with the base class executes the respective implementation of the instances declaring class. This feature is called *subtype polymorphism* in object-oriented programming.

parameter. Then on hierarchy level $n - 1$ the instance will be used to retrieve an iterator from the iterable (cf. activity “retrieve iterator”) and stores the result to the context variable “iterator”. The iterator serves activity “retrieve next” with the retrieval of the next item in the collection, no matter which implementation of $\textcircled{1}\text{Iterator}\langle T \rangle$ is in the context. This allows to change the runtime behavior of a process model in a typesafe manner without the need to adapt it.

Furthermore, it is possible to

- store process instances in the execution context,
- use them as input and output parameters of input as well as output SIBs,
- reuse them for multiple executions as described in Sec. 3.3.3, and
- to create new instances via constructor activities as presented in Sec. 3.3.5.

This enables to pass services and process instances between graphs just like data. Thus the paradigm to allow executable and polymorph “entities” in the context as well as input and output parameters is consequently permeated to the modeling level supporting true higher-order process engineering (HOPE) as described in more detail in the following [NS13b, NS13a, NSM13].

3.5.1 Interface Graphs

In order to support runtime variability on the modeling level *interface graphs* are introduced declaring the input and output parameterization of a service graph but no control-flow just like an abstract method (e.g. in Java) declares the method signatures: each implementing class has to define the method. The same holds for implementing service graphs.

Definition 3.16. A (*generic*) dual IO kripke transition system $gKTS^{I/O}$ (or a $dKTS^{I/O}$ if it does not define any type parameter) – denoted as (*generic*) *interface Graph* – over an enhanced type system $\mathcal{T}_{\mathfrak{Param}}$ and a finite set of atomic propositions AP is a structure $gKTS^{I/O} = (n, p, \mathfrak{Param}, \mathcal{A}, a_0, \mathcal{A}_{Out}, \mathcal{I})$, where

- n is the simple name of the interface graph,
- p is the package name of the interface graph,
- together n and p build the full qualified name of the graph,
- \mathfrak{Param} is a set of graph type parameters (the formal parameters for parametric polymorphism as described in Sec. 3.3.4),
- \mathcal{A} is activity domain (cf. Def. 3.8),
- $a_0 \in \mathcal{A}$ a dedicated start activity, i.e., an *input SIB*,
- $\mathcal{A}_{Out} \subseteq (\mathcal{A} \setminus a_0)$ a finite set of *end activities*, i.e., *output SIBs*,

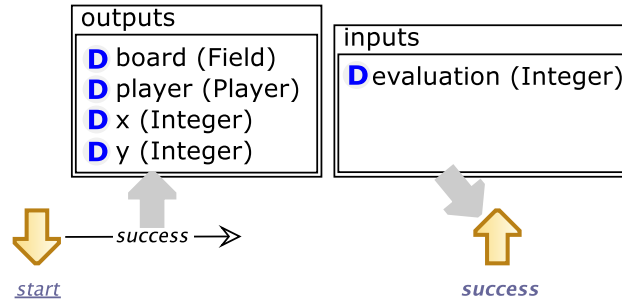


Figure 3.10: Interface graph for cell evaluations in the ChainReaction scenario.

- $\mathcal{A} = a_0 \cup \mathcal{A}_{Out}$, since an interface graph declares a process signature and therefore has a start activity and output activities only, and
- $\mathcal{I} : \mathcal{A} \rightarrow 2^{AP}$ is an *interpretation function*.

In the graph browser interface graphs are shown along with the service graphs (sorted into the tree structure by package), just showing a purple ‘G’-icon left of its label instead of a green one as for service graphs (cf. Sec. 3.3.7). An interface graph can be implemented by any service graph, just by dragging the interface graph from the graph browser to the list “interfaces” in the graph inspector shown on the bottom left of Fig. 3.2. At the time of writing exactly one interface is allowed for a service graph (although it is visualized as a list by now). In the spirit of functional programming an interface graph can be seen as a function signature and it therefore has no further identity, so that it does not make sense to let an service graph implement multiple interfaces that would be substitutable or more precisely be considered the same. Thus, it is possible to add constraints to an interface graph so that it is nevertheless reasonable to distinguish interface graphs. Still this is not supported yet and this again is a decision made on an ‘as needed’ basis, which might change in the future.

Analogously by now it is not envisaged to constrain interface graphs or service graphs, i.e., we have a flat and easy-to-understand inheritance hierarchy. An implementing service graph has the exact same input/output parameterization as its interface graph in order to be considered valid.

A context variable may represent an interface graph together with Java classes and service graphs all being a part of the type system \mathcal{T} in jABC4. The type is automatically set – just as for service graph SIBs – as soon as the context variable is connected to a corresponding activity (cf. Sec. 3.5.2), or may be set manually via the graph chooser (cf. Sec. 3.3.8). Input and output parameters of a service or interface graph (i.e., the parameters of input and output SIBs) may be configured to refer to an interface graph type via the graph chooser, too.

Example 6: An interface graph for the ChainReaction scenario

As described in Sec. 2.2, the task for the pupils in their project week was to create a process model, that evaluates the worthiness of a cell in a given board configuration. Therefore each process model should have a similar input and output param-

eterization as presented in Example 1. The corresponding interface graph named **GameStrategy** is depicted in Fig 3.10. The example GS shown in Fig. 3.2 already implements the interface (cf. the list “Interfaces” in the graph inspector). The icon in the graph inspector is set to:



3.5.2 Interface Graph SIBs

Up to here an interface graph may be used as a type for context variables and input/output parameters of abstraction activities. A process instance of any service graph implementing this very interface graph may be the mutable value of such a context variable and passed through the respective input/output parameters at runtime. Thus in order to execute the process instance polymorphous *interface graph SIBs* are introduced. They are structured analogously to service graph SIBs and are created via drag-and-drop of a leaf node in the graph browser representing the interface graph to the canvas (cf. Sec. 3.3.2 & 3.3.7).

There are some construction-conditioned limitations:

- The instance parameter “instance” has the type of the interface graph and may be connected to context variables referencing this very type, only.
- The instance parameter of an interface graph SIB must be set to a context variable, since an interface graph declares the input/output parameterization only but is not executable itself. Therefore an instance of a concrete implementation is definitively necessary and cannot be created on demand.
- It is not allowed, that no instance is available at runtime, since the interpreter (or the generated code respectively) does not know which implementation of the interface graph it should instantiate (cf. Sec. 3.6).
- Finally, no constructor SIB is available for interface graphs for the same reason.

At runtime, if the control-flow reaches an interface graph SIB, it will retrieve the respective instance from the connected context variable and execute it just as described in Sec. 3.3. This is always possible since the input/output parameterization of the interface graph and all its implementing service graphs is exactly the same and therefore surely substitutable. This way, it is possible to create and pass around process instances just like data in a higher-order fashion and execute them polymorphous without touching the executing process model, just by parameterizing it with a respective implementation of the interface graphs.

Example 7: The bridge between jABC4 and ChainReaction

A bachelor student of mine prepared an SLG embedding the pupils’ implementations of **CellEvaluation** into ChainReaction as an GS, namely **StartCR**. It is illustrated in Fig. 3.11. The start activity declares two input parameters for the

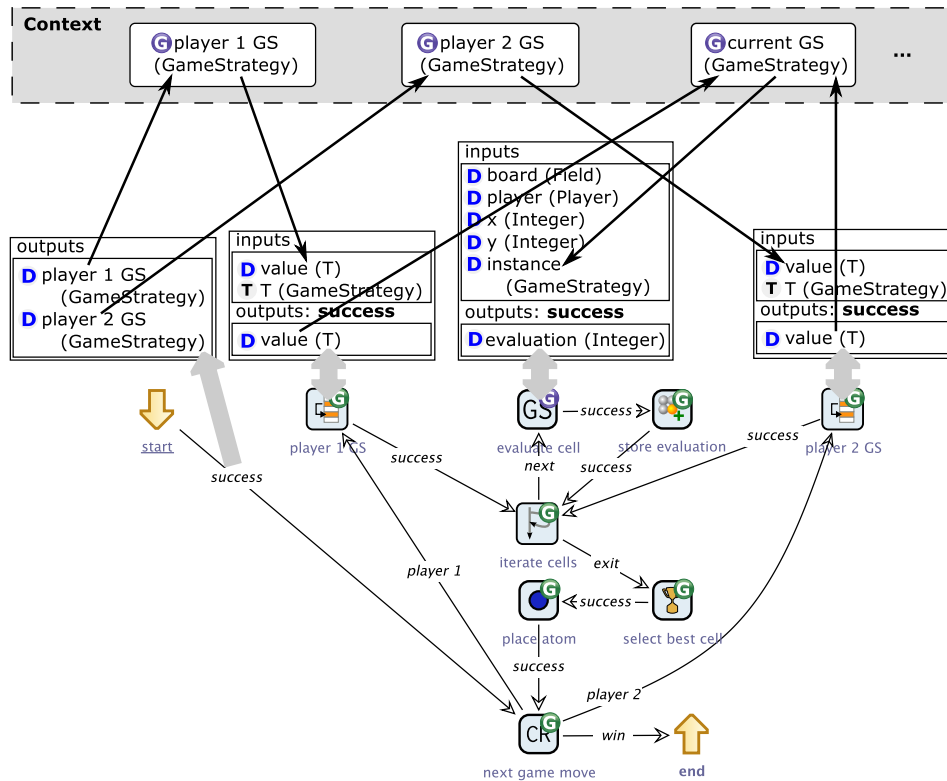


Figure 3.11: Service graph interacting between jABC4 and ChainReaction.

graph. They contain the process instances for both players representing the GSs in form of implementations of CellEvaluation . The SLG shown in Fig. 3.11 is a downgraded variant of the original process model, in that it has no check whether both GSs are set. In the original process an absent GS would result in setting a human opponent, so that the pupils can test their GS either by playing against it or letting it play against a competitor GS. Here we assume that both GSs are set.

The GSs are written to the context variables “player 1 GS” and “player 2 GS”. The first activity after the input SIB “next game move” executes the sub process handling the interaction with the game. For every game move it alternately returns with branch “player 1” or “player 2” until one of the oponents has won. Finally, “next game move” returns with the branch “win” meaning that the GS who did the last move has won.

The activities following the branches “player 1/2” reference the SLG PutToContext and write the respective cell evaluation process to the context variable “current GS”. In both cases the graph StartCR then iterates through the cells of the game board. For each cell the process in the variable “current GS” – which is interchangingly the GS configured for the first and second player respectively – is executed in the interface graph SIB “evaluate cell” and the evaluation is stored into a data structure.

Afterwards the cell with the best evaluation is selected and the actual move takes place in activity “place atom” and the game is queried for the next move. If the best cell is not definite, one of the best-rated is chosen randomly.

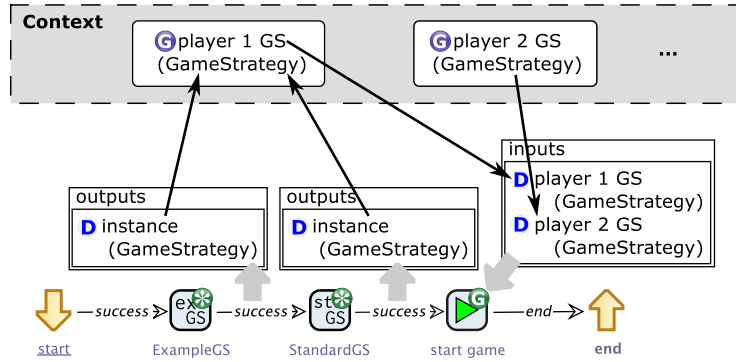


Figure 3.12: Starter graph for the instantiation of two GSs and execution of Chain-Reaction via the graph depicted in Fig. 3.11 in activity “start game”.

This example shows, that HOPE enables to pass process instances just like data, so that an activity (i.e., “evaluate cell”) even may execute a different process instance every time it is executed.

3.5.3 Constructor Activities and Interface Graphs

A constructor activity (cf. Sec. 3.3.5) creates an instance of a given service graph. If the service graph implements an interface, the instance may be written to a context variable typed with the corresponding interface graph and passed to a similarly typed input parameter of an abstraction activity. This way a modeler can steer which implementation is executed in the sub process just by parameterizing it with an instance created by a constructor activity.

Example 8: A starter graph for the ChainReaction scenario

The pupils were able to test their GSs, by creating a simple starter graph as depicted in Fig. 3.12. It contains a constructor activity for the GS to be tested and one for a competitor GS, which may be the same or any other GS. In this example the provided SLG GStandardGS is used. The instances are written to context variables and used as input parameters for the abstraction activity “start game”, which references the SLG GStartCR (cf. Example 7). Hence the pupils used higher-order process engineering seamlessly, as it hides the technical details of interacting with the game as well as handles the always same task of evaluating the complete game board. The pupils were able to simply create the evaluation for one cell in a divide-and-conquer manner, instantiate it together with a competitor cell evaluation process in a starter graph and pass it to the game process.

3.5.4 Configuration Interface Graphs

The feature of configuration graphs may be combined with interface graphs, in order to be able to provide preconfigured IO activities for all instances of a given interface

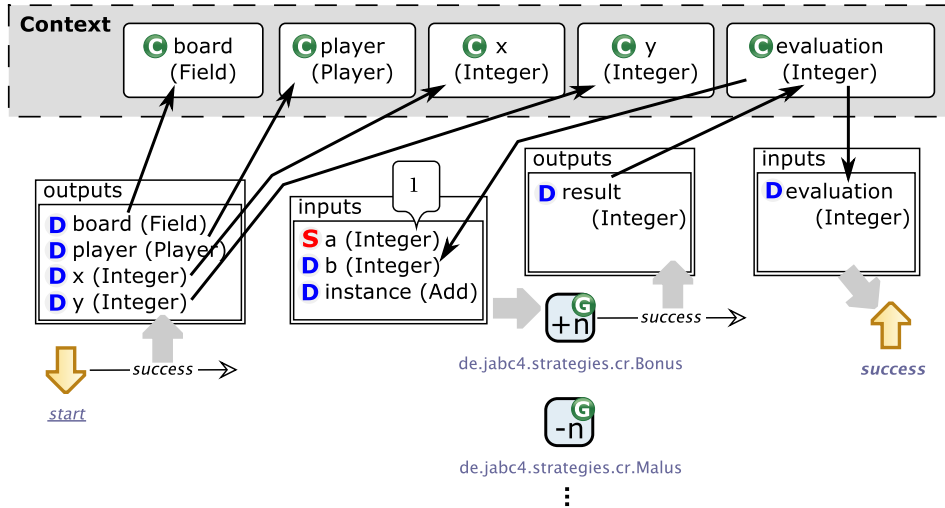


Figure 3.13: Example for a configuration interface graph combining the configuration graph shown in Fig. 3.8 and the interface graph of Fig. 3.10 for the ChainReaction scenario.

graph. This helps especially if there are a lot of implementations anticipated. But the telling argument for this hybrid form of SLGs is that a modeler may preconfigure to/from which context variables the input/output parameters of its start/end activities writes/reads.

In combination with arbitrary further preconfigured activities in the configuration interface graph, it is possible to predefine domain-specific activities determining not just the data-flow between the instantiated activities in every implementing service graph, but between the start/end activities and the execution context (and therefore for the other activities), too. Thus the data-flow may be preconfigured completely and consistent for all implementations of an interface graph.

A configuration interface graph is defined similar as an configuration graph (cf. Sec. 3.4.1), with the difference, that it may be used for both to create an IO browser like from a configuration graph and it may be applied to context variables and interface graph activities just like an interface graph (cf. Sec. 3.5.1).

Here the transitivity of auto-created context variables mentioned in Sec. 3.4.2 comes in handy. Experts of different domains may have prepared configuration graphs for their domain. A selection of template activities of these domains may be instantiated in the configuration interface graph and their configuration tailored to the needs of the current domain. This enables an even clearer form of *separation of concerns* (than mentioned before) as a modeler, who should implement the interface, gets a tailored set of activities borrowed from possibly different domains, which have been defined by separate domain experts, since the preparation of a configuration interface graph allows to reuse existing components from other configuration graphs.

Example 9: Configuration interface graph for the ChainReaction scenario
The configuration graph for the ChainReaction scenario (cf. Fig. 3.8) may be com-

bined with the interface graph (cf. Fig. 3.10) to a configuration interface graph. If a new implementation of this graph is created (i.e., a new cell evaluation SLG), the IO browser of the configuration interface graph contains the preconfigured input and output SIBs, which share the context variables with the other preconfigured activities like, e.g., “evaluation” which is used by activity “Bonus” and “Malus” and the end activity “success”. This way an GS may be created just by drag&drop of domain-specific preconfigured activities, and connecting them regarding the control-flow, without any need to deal with the data-flow. Still the data-flow information is available for validations and, e.g., boni may be adjusted via the constant parameters directly in the process model.

3.6 Inversion of Control

Sec. 3.3.1 mentions that global variables or scopes are not supported intentionally in favor of simplicity. Thus a consequent use of input and output parameters for the complete communication between components would bloat the parameterlists. This can be overcome via configuration graphs, already preconfiguring a lot of them. But it would aggravate the reuse of interfaces since there will always be at least ‘that’ parameter being necessary in one domain, but not in the other.

In the HOPE approach this issue is tackled via *inversion of control* (IoC) or more precisely dependency injection as Martin Fowler in 2004 [Fow04] called the software design principle of providing dependencies to an object at runtime via some kind of container. Dependency injection allows to mark context variables to be injected and at runtime a corresponding instance is provided by the execution environment so that it is not necessary to pass an instance through all hierarchy levels. This feature supports *separation of concerns* in that it hides (or encapsulates) dependencies of components. Furthermore by adapting the configuration of the environment a new level of runtime variability is attainable as different implementations can be injected into the process instances changing its behavior without touching any process model.

A prominent framework supporting dependency injection is contexts and dependency injection²⁰ (CDI), which is both part of the Java EE (enterprise edition) standard and has a Java SE (standard edition) implementation called “Weld”, so that it may be used in enterprise environments, standard desktop applications, Java web start, and applets. Hence, the CDI framework is employed.

The technical details for the interpreted case are described in Sec. 3.8. For the modeler it is simply a right-click followed by activating the chooser menu item “Inject instance”. Injection is possible for Java classes, Java interfaces, and interface graphs. Java types are injected via CDI which returns (injects) an instance of a corresponding *bean*²¹. If there is more than one bean type matching an inject declaration the CDI-container fails with an exception message “ambiguous dependency”. For interface graphs an implementing service graph is injected, accordingly. If more than one

²⁰<https://www.jcp.org/en/jsr/detail?id=299>

²¹The notion bean stands here for web or enterprise beans, which are container managed java instances being target of, e.g., dependency injection.

implementation is found in conformity with the behavior of the CDI-container an exception is thrown.

Example 10: IoC in the ChainReaction scenario

In the process models referenced by the abstraction activities labelled “next game move” and “place atom” in Fig. 3.11 the API of the game ChainReaction is invoked. As the game has a state during a match, an instance of the game object would have to be passed to the activities and stored in the execution context of **GStartCR**. Alternatively, both sub processes (namely **GCRMove** and **GPlaceAtom**) could mark the context variable holding the game instance as *injected*. In the spirit of *separation of concerns* the calling SLG **GStartCR** does not need a context variable for the game instance anymore, as it is encapsulated on the hierarchy level below.

3.6.1 On the Target Language Level

As described in [NS14] and [NWS14] the IoC capabilities have been used on the target language level (i.e., Java) for testing a web application. A context variable referenced the interface **IAdapter** which offers access mechanisms to the system under test (SUT). There are different implementations of this interface on the one hand regarding the access layer (i.e., web-frontend or business logic layer, cf. [NS14]) and system change in terms of system evolution as well as system migration [NWS14].

The test blocks for interrogating the SUT are implemented with the HOPE approach. Technical SLGs access the API adapter for executing an action on the target system. These get the corresponding API adapter injected, so that the different actions can be provided as domain-specific activities to users with low technical experience without having to deal with retrieving the API adapter. Furthermore the service graphs representing test blocks (basing on the technical SLGs mentioned before) can be used to be executed on different access layers as well as for different system versions just by changing the environment configuration (e.g., the class path) without the need to bother the modeler.

3.6.2 On the Modeling Level

On the modeling level the IoC capabilities are used for creating different variants of the code generators for jABC4 process models [Neu14] – which are implemented via jABC4 process models itself – for the target language Java and Scala respectively. The dependency injection feature is implemented for the Java generator only, at the time of writing, so that both the Java and Scala generator process models are generated to Java.

The generators consist of some base process models for interrogating the structure of an SLG as well as language specific implementations for generating fragments of code for the respective target language. At any point where a language specific piece of code has to be generated for an element of the process model an interface graph SIB is used and the context variable for the instance parameter is configured as

“Injected instance”. The base process models are bundled into an SLG library. For each target language an SLG library is provided, too. The latter have a dependency to the base project and implement the interface graphs.

In the build management environment (i.e., apache maven) a concrete code generator for an SLG is selected via a maven plugin that references the SLG library for the concrete target language. The maven plugin invokes the base service graph for generating a process model and at the *injection points* the right service graph for the selected target language are executed since they are the only ones on the class path due to the maven configuration. If more than one SLG library for concrete target languages are loaded in one configuration an “ambiguous dependency” exception will be thrown as mentioned before. This runtime variability solution solves the same problem as addressed in [JLM⁺12] via a design-time variability approach using *variation points* and model weaving.

3.7 Transition from jABC3 to jABC4

Sec. 6 of [NNL⁺13] describes how a prototypic reference implementation has been created for the HOPE approach – the jABC4 plugin – on top of the existing version jABC3 [SM08] via its powerful plugin concept.

In jABC3 there is no differentiation neither between input and output parameters, nor between the output parameters of different branches. This is emulated by giving the parameters corresponding name prefixes like `input_` or `output_branchxy_`, which are processed in API facades to the jABC3 framework classes in order to present them nicely in the graphical jABC4 plugin components. The new type-aware, dynamic SIBs are realized via six “plain old” SIBs:

The service call SIB handles atomic activities. It stores information regarding the method it represents and uses the Java reflection API²² to execute the method dynamically.

The input SIB is used as the start activity of an SLG. It retrieves the input parameters from the parent context and stores them to the local context. For this purpose each input parameter is divided into two SIB parameter. One is a model parameter (cf. Chap. 2) with a label prefixed `input_` and retrieves the value from the parent context. The other is a normal SIB parameter with a label prefix `output_success_` responsible for storing the value to the local context. It has one sole branch “success”, which is followed right after the parameter transfer. This even works in case of a flat execution context, since UUIDs are used for the identifier of context variables (cf. Def. 3.3), so that every context variable has a unique name and therefore cannot interfere with any other.

The output SIB represents the end activities of an SLG. It retrieves the output parameters for the respective branch from the local context and stores them to the parent context. Analogously to input parameter of an SLG, each output

²²<http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html>

parameter is split into a normal SIB parameter prefixed `input_` for retrieving the value from the local context and a model parameter prefixed `output_branchxy_` for storing the value to the parent context. Each output SIB exports his only “success”-branch as a model branch (cf. Chap. 2) named like the label of the output SIB (i.e., `branchxy`).

The service graph SIB handles abstraction activities referencing a service graph and is a specialization of the *graph SIB* of jABC3. The SIB parameters are the model parameter defined by the input and output SIBs of the associated service graph, as this is the normal behavior of jABC3. As mentioned before these are prefixed `input_` for the input SIB and `output_branchxy_` for the output SIBs, so that the API facades of the plugin interpret them as the input and output parameter for the service graph SIB. Additionally, each model branch of the sub model is translated to a mutable branch of the service graph SIB according to the standard jABC3 semantics. At runtime it just executes the process instance in the context. A new instance will be created, if it does not exist.

The constructor SIB is a graph SIB that references service graphs too but overrides the standard behavior in that it does neither use the input/output parameter nor the branches of the sub model. At runtime it just creates and stores a process instance of the sub model to the sole context variable output parameter “instance” of its only “success”-branch instead of executing the sub model.

The interface graph SIB handles abstraction activities referencing an interface graph and is defined analogously to service graph SIBs despite, that at runtime there has to be a process instance in the context or the variable has to be configured as an “Injected instance”. In the latter case the interface graph SIB searches for an implementing service graph and creates a corresponding new instance, stores it to the corresponding context variable and executes it.

These six “jABC3-SIBs” are sufficient to capture all the activities for the HOPE approach (cf Chap. 3). Each of these SIBs is represented as a SIB class with a SIB adapter. They use the concept of mutable parameters and branches instead of defining them statically in the SIB class. Mutable parameters and branches were originally used for graph SIBs, as they need to be able to adapt to the referenced sub model. This way one SIB class of the HOPE approach can represent a wide range of activity types and even switch between dynamic and static input parameters at the modeling level.

The type information and other meta-information is added as so called *user objects* dynamically to the SIBs and SLGs, which are persisted by the jABC framework automatically. An API facade of jABC4 offers an easy-to-use access to all this information.

The enhanced information can be viewed and manipulated in specially designed user interface components of the jABC4 plugin, which comprises hiding some com-

ponents of the plain old user interface. The meta-information is updated via model listeners, which are part of the plugin concept of jABC3.

3.8 The Interpreter

The jABC3 framework already offers an interpreter for SLGs via the tracer plugin with features like step-wise execution and breakpoints known from debugger as well as inspection of the current state of an SLG and many more. Since the dynamic SIBs are implemented via jABC3-SIBs, a jABC4-SLG is directly executable via the tracer. Even old and new SIBs can coexist in the same SLG maintaining its executability [NNL⁺13]). Furthermore it is possible to use the Genesys generators [Jö13] to generate executable code from it following the full-code generation paradigm.

Process instances are realized for stateful processes (cf. Sec. 3.3.3) and higher-order process modeling (cf. Sec. 3.5) via the class `GraphInstance`, which holds the unique identifier of the service graph it represents as well as an jABC3 execution context (altogether a mapping from string identifiers to arbitrary Java objects), representing its current state. Hence creating a new process instance of a given graph is just creating an empty `HashMap` and setting the identifier. If the instance should be executed, the interpreter looks up the corresponding SLG overwrites the context with the one of the process instance and invokes it.

But since all activity types are realized via the six plain old SIBs described in Sec. 3.7 and the type information is additive meta-information, there is no compile-time type-safety. Every call to a method is realized via a Java reflection call in the SIB adapter of the service call SIB. This holds even for the generated code of the Genesys framework as it calls the aforementioned generic SIB adapters.

The dependency injection feature described in Sec. 3.6 is realized via the aforementioned “Weld” container for Java SE. The tracer plugin is enhanced in that it creates a weld container for each new execution. In the adapter implementation of the service call, service graph, constructor and interface graph SIB the current weld container is retrieved and used to lookup the respective resources.

Some validations regarding the types are already realized in the jABC4 plugin on the modeling level, but since Java (and Scala) already have a sophisticated type system, a generator has been implemented following the full-code generation paradigm similar to Genesys, completely autonomous from the jABC3 framework. It translates an SLG into native Java (Scala) calls. The downside is, that the generator can handle the new SIBs only, but it produces, efficient code that can be type-checked by the compiler of the target language.

Example 11: Rapid prototyping of GSs in the ChainReaction scenario

The pupils may test their GS by executing a starter graph (cf. Fig. 3.12) via the interpreter in the jABC4 development environment. The configured GSs are passed to the service graph `StartCR` and invoked alternately. The own implementation for cell evaluation may be inspected via adding a *breakpoint* known from debuggers of

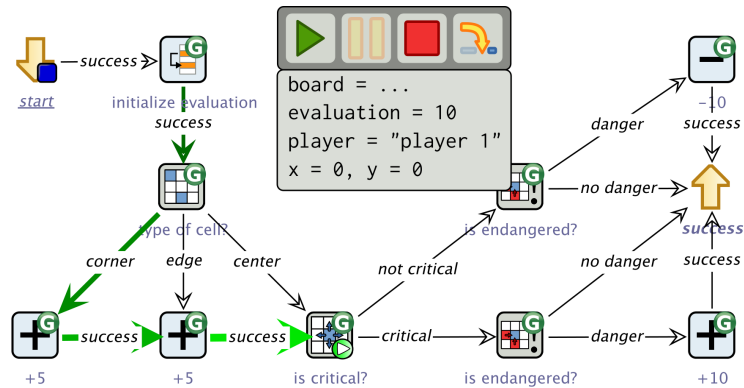


Figure 3.14: The example GS (cf. Fig. 3.2) in a step-by-step execution via the tracer plugin.

programming languages. This is done by right-clicking on an activity and choosing “Add breakpoint”. A breakpoint is visualized via a small purple rectangle in the bottom right of the activities icon. As soon as the execution reaches an activity with a breakpoint, the corresponding SLG is opened and the user may use the debugging window for *step-by-step* execution of the SLG and view the current content of the execution context. In Fig. 3.14 a breakpoint has been added to the activity “start” and the execution has been inspected step-by-step until the activity “is critical?” (cf. the little green “play”-icon in the bottom right of its icon) for the top left cell of the game board for the first player (cf. `player = "player 1"`, `x = 0`, and `y = 0` in the execution context). The last steps of execution may be tracked via the green highlighting of the edges.

Via breakpoints the rapid prototyping of higher-order processes is easy to use, as the interpreter stops right in the own process models so that the circumstances in which the SLG has been executed are hidden from the modeler.

4 Conclusion

In this document the HOPE approach has been described in detail, which partitions the development process in different layers in order to support *separation of concerns*. The basis are standard APIs in a target language, i.e. Java [Blo08] or Scala [OSV10] in the reference implementation. On top of that a layer of technical process models is situated, that directly binds methods of the target language dynamically to activities (cf. Sec. 3.2). The underlying target language provides well-defined semantics guaranteeing executability. Technical details are encapsulated via hierarchical modeling (cf. Sec. 3.3). Moreover, the HOPE approach is component-based, it enforces sophisticated input/output parameterization supporting parametric polymorphism¹ of process models. Together with the already available type system of the underlying target language this enables to explicitly model the (type-aware) data-flow information of all components in addition to the control-flow information. The complexity being a consequence thereof has been tackled via preconfiguration of activities and their input/output parameterization as well as the use of inversion of control (IoC) to inject components into processes (cf. Sec. 3.4 and 3.6). The process models get flexible and stay comprehensible by adding a higher-order flavor: services and process instances are treated as *first-class citizens*. Implementations of an activity may be exchanged at runtime (cf. Sec. 3.5). These layers may be iteratively applied by diverse participants in the development process to create a complete hierarchy of domain-specific, preconfigured components, and make them available in libraries to be reused as activities on the respective next layer of abstraction.

¹Realizations of parametric polymorphism are often referred to as “generics” in the corresponding target languages [NW06].

Bibliography

- [AAA⁺07] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guzar, N. Kartha, C.K. Liu, R. Khalaf, Dieter Koenig, M. Marin, V. Mehta, S. Thatte, D. Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0 (OASIS Standard). WS-BPEL TC OASIS, <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>, 2007.
- [Ang87] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
- [BA04] K. Beck and C. Andres. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.
- [Blo08] J. Bloch. *Effective Java*. Java Series. Pearson Education, 2008.
- [DS12] Markus Doedt and Bernhard Steffen. An Evaluation of Service Integration Approaches of Business Process Management Systems. In *Proc. of the 35th Annual IEEE Software Engineering Workshop (SEW 2012)*. IEEE, 2012.
- [Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern, Jan 2004. <http://www.martinfowler.com/articles/injection.html>.
- [FR14] Michael Felderer and Rudolf Ramlar. A multiple case study on risk-based testing in industry. *STTT-RBT*, 2014. Under Review.
- [GT02] Paul Gerrard and Neil Thompson. *Risk Based E-Business Testing*. Artech House, Aug 2002.
- [JLM⁺12] Sven Jörges, Anna-Lena Lamprecht, Tiziana Margaria, Ina Schaefer, and Bernhard Steffen. A Constraint-based Variability Modeling Framework. *International Journal on Software Tools for Technology Transfer (STTT)*, 14(5):511–530, 2012.
- [JS12] Sven Jörges and Bernhard Steffen. Exploiting Ecore’s Reflexivity for Bootstrapping Domain-Specific Code-Generators. In *Proc. of 35th Software Engineering Workshop (SEW 2012)*, pages 72–81. IEEE, 2012.
- [Jö13] Sven Jörges. *Construction and Evolution of Code Generators - A Model-Driven and Service-Oriented Approach*, volume 7747 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Germany, 2013.

- [KA90] Setrag Khoshafian and Razmik Abnous. *Object Orientation: Concepts, Languages, Databases, User Interfaces*. John Wiley & Sons, Inc., New York, NY, USA, 1990.
- [MS04] Tiziana Margaria and Bernhard Steffen. Lightweight coarse-grained coordination: a scalable system-level approach. *Software Tools for Technology Transfer*, 5(2-3):107–123, 2004.
- [MS06] T. Margaria and B. Steffen. Service engineering: Linking business and it. *Computer*, 39(10):45–55, Oct 2006.
- [MS09a] Tiziana Margaria and Bernhard Steffen. Agile IT: Thinking in User-Centric Models. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *Communications in Computer and Information Science*, pages 490–502. Springer Berlin/Heidelberg, 2009.
- [MS09b] Tiziana Margaria and Bernhard Steffen. Business Process Modelling in the jABC: The One-Thing-Approach. In Jorge Cardoso and Wil van der Aalst, editors, *Handbook of Research on Business Process Modeling*. IGI Global, 2009.
- [MS11] T. Margaria and B. Steffen. Special session on "simplification through change of perspective". In *Software Engineering Workshop (SEW), 2011 34th IEEE*, pages 67–68. IEEE, Jun 2011.
- [MS12] Tiziana Margaria and Bernhard Steffen. Service-orientation: Conquering complexity with xmdd. In Mike Hinchey and Lorcan Koyle, editors, *Conquering Complexity*. Springer, 2012.
- [MSHM11] Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next Generation LearnLib. In *Proc. of 17th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011), Saarbrücken, Germany*, pages 220–223. Springer, 2011.
- [MSR05] Tiziana Margaria, Bernhard Steffen, and Manfred Reitenspieß. Service-Oriented Design: The Roots. In *Proc. of the 3rd Int. Conf. on Service-Oriented Computing (ICSOC 2005), Amsterdam, The Netherlands*, volume 3826 of *LNCS*, pages 450–464. Springer, 2005.
- [MSS99] Markus Müller-Olm, David Schmidt, and Bernhard Steffen. Model-Checking - A Tutorial Introduction. In *Proceedings of the 6th International Symposium on Static Analysis (SAS '99)*, pages 330–354. Springer, 1999.
- [Neu14] Johannes Neubauer. *Higher-Order Process Engineering*. Phd thesis, TU Dortmund, May 2014.
- [NMS13] Johannes Neubauer, Tiziana Margaria, and Bernhard Steffen. Design for Verifiability: The OCS Case Study. In *Formal Methods for Industrial*

-
- Critical Systems: A Survey of Applications*, chapter 8, pages 153–178. Wiley-IEEE Computer Society Press, Mar 2013.
- [NNL⁺13] Stefan Naujokat, Johannes Neubauer, Anna-Lena Lamprecht, Bernhard Steffen, Sven Jörges, and Tiziana Margaria. Simplicity-First Model-Based Plug-In Development. In *Software: Practice and Experience*. John Wiley & Sons, Ltd., 2013. first published online.
- [NS13a] Johannes Neubauer and Bernhard Steffen. Plug-and-Play Higher-Order Process Integration. *IEEE Computer*, 46(11):56–62, August 2013.
- [NS13b] Johannes Neubauer and Bernhard Steffen. Second-Order Servification. In Georg Herzwurm and Tiziana Margaria, editors, *Software Business. From Physical Products to Software Services and Solutions*, volume 150 of *Lecture Notes in Business Information Processing*, pages 13–25. Springer Berlin Heidelberg, 2013.
- [NS14] Johannes Neubauer and Bernhard Steffen. Learning-Based Cross-Platform Conformance Testing. In *STVR*. John Wiley & Sons, Ltd., 2014. in submission.
- [NSB⁺12] Johannes Neubauer, Bernhard Steffen, Oliver Bauer, Stephan Windmüller, Maik Merten, Tiziana Margaria, and Falk Howar. Automated continuous quality assurance. In *Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012*, pages 37–43. Springer, 2012.
- [NSM13] Johannes Neubauer, Bernhard Steffen, and Tiziana Margaria. Higher-Order Process Modeling: Product-Lining, Variability Modeling and Beyond. *Electronic Proceedings in Theoretical Computer Science*, 129:259–283, 2013.
- [NW06] M. Naftalin and P. Wadler. *Java Generics and Collections*. O’Reilly Media, 2006.
- [NWS14] Johannes Neubauer, Stephan Windmüller, and Bernhard Steffen. Risk-Based Testing via Active Continuous Quality Control. In *Special Issue: Risk-Based Testing*, Software Tools for Technology Transfer. Springer, 2014. to appear.
- [OMG11] OMG. Business Process Model and Notation (BPMN) Version 2.0, 2011. <http://www.omg.org/spec/BPMN/2.0/>.
- [OSV10] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Series. Artima, Incorporated, 2010.
- [PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. *SIGPLAN Not.*, 26(11):146–161, Nov 1991.

- [RSBM09] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. LearnLib: a framework for extrapolating behavioral models. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(5):393–407, 2009.
- [RSM08] Harald Raffelt, Bernhard Steffen, and Tiziana Margaria. Dynamic Testing Via Automata Learning. In *Proc. of the Haifa Verification Conference 2007 (HVC '07)*, volume 4899 of *LNCS*, pages 136–152. Springer, 2008.
- [Ses12] Peter Sestoft. Higher-order functions. In *Programming Language Concepts*, volume 50 of *Undergraduate Topics in Computer Science*, pages 77–91. Springer London, 2012.
- [SHM11] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to Active Automata Learning from a Practical Perspective. In Marco Bernardo and Valérie Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer Berlin Heidelberg, 2011.
- [SM99] Bernhard Steffen and Tiziana Margaria. Metaframe in practice: Design of intelligent network services. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 390–415. Springer Berlin Heidelberg, 1999.
- [SM08] B. Steffen and T. Margaria. Business process modelling in the jabc: The one-thing approach. In *Handbook of Research on Business Process Modelling*. IGI Global, 2008.
- [SMBK97] Bernhard Steffen, Tiziana Margaria, Volkar Braun, and Nina Kalt. Hierarchical Service Definition. *Annual Review of Communications of the ACM*, 51:847–856, 1997.
- [SMC⁺96] Bernhard Steffen, Tiziana Margaria, Andreas Claßen, Volker Braun, Manfred Reitenspieß, and Helmut Wendler. Service Creation: Formal Verification and Abstract Views, 1996.
- [SMCB96] Bernhard Steffen, Tiziana Margaria, Andreas Claßen, and Volker Braun. Incremental Formalization: A Key to Industrial Success. *Software - Concepts and Tools*, 17(2):78–95, 1996.
- [SMN⁺06] Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges, and Christian Kubczak. *Model-Driven Development with the jABC*, volume 4383 of *LNCS*, pages 92–108. Springer Berlin/Heidelberg, 2006.
- [SN07] Bernhard Steffen and Prakash Narayan. Full Life-Cycle Support for End-to-End Processes. *IEEE Computer*, 40(11):64–73, 2007.
- [SS06] August-Wilhelm Scheer and Kristof Schneider. Aris — architecture of integrated information systems. In Peter Bernus, Kai Mertins, and Günter Schmidt, editors, *Handbook on Architectures of Information Systems*,

- pages 605–623. Springer Berlin Heidelberg, 2006. 10.1007/3-540-26661-5_25.
- [STA05] August-Wilhelm Scheer, Oliver Thomas, and Otmar Adam. *Process Modeling using Event-Driven Process Chains*, pages 119–145. John Wiley & Sons, Inc., 2005.
- [VG07] Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *Proceedings of the 11th International Software Product Line Conference, SPLC '07*, pages 233–242. IEEE Computer Society, Washington, DC, USA, 2007.
- [WNS⁺13] Stephan Windmüller, Johannes Neubauer, Bernhard Steffen, Falk Howar, and Oliver Bauer. Active Continuous Quality Control. In *16th International ACM SIGSOFT Symposium on Component-Based Software Engineering, CBSE '13*, pages 111–120. ACM SIGSOFT, New York, NY, USA, 2013.
- [WS73] W. Wulf and Mary Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2):28–34, feb 1973.