

Modeling and Verification of Software (MOVES)
Department of Computer Science
RWTH Aachen University

Explicit-State Model Checking of an Architectural Design Language using SPIN

Diploma Thesis

from

Maximilian Reinhard Odenbrett

Supervised by

Priv.-Doz. Dr. Thomas Noll
Prof. Dr. Ir Joost-Pieter Katoen

Aachen, March 12, 2010

The work presented in this thesis emanates from the ESA-funded COMPASS project (ESA/ESTEC Contract 21171/07/NL/JD).

I would like to thank my supervisors, in particular Priv.-Doz. Dr. Thomas Noll and MSc. Viet Yen Nguyen, for their support and the patience they had with my nitpicking questions.

A special thanks goes to the student assistants Benedikt Brutsch and Christian Dehnert for the manifold support they provided to me when I reused the code that they developed for the COMPASS project.

The biggest debt of gratitude I owe to my parents who not only funded my studies but always attached great importance to my education.

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 12.03.2010

(gez. Maximilian R. Odenbrett)

Contents

1	Introduction	1
1.1	The COMPASS project	1
1.2	Contents of this Diploma Thesis	2
I	Preliminaries, Background	5
2	The Architectural Design Language SLIM	7
2.1	Component Type	8
2.2	Component Implementation	8
2.2.1	Nominal Modes	9
2.2.2	Control Subcomponents	10
2.2.3	Data Subcomponents	11
2.2.4	Data Port Connections and Flows	11
2.2.5	Event Port Connections	12
2.2.6	Transitions	12
2.3	Data Types and Expressions	14
2.4	Example	14
2.5	Modeling Errors	15
2.6	Defining Properties	19
3	The Model Checker SPIN	23
3.1	Input Language PROMELA	23
3.1.1	Syntactical Constructs	23
3.1.2	Operational Semantics	31
3.1.3	Atomic Execution Blocks	31
3.1.4	Embedded C-Code	33
3.2	Model Checking with SPIN	34
3.2.1	Assertion-based Verification	35
3.2.2	Never Claims	36
3.2.3	LTL Model Checking	37
3.3	Advanced Techniques to Improve Efficiency of Model Checking	38
3.3.1	Reduction of the State Space	38
3.3.2	Reduction of the State Vector Size	39

II	Model Checking SLIM Specifications with SPIN	41
4	Translation to PROMELA Programs	43
4.1	Proctypes for Components	43
4.2	Global Variables for Components	44
4.2.1	Integer Constants for Symbolic Identifiers	46
4.2.2	Handling of Reals by Embedded C-Code	47
4.2.3	Translation of Assignments	48
4.2.4	Translation of Expressions	49
4.3	Initialisation	50
4.4	Proactive Transitions	51
4.4.1	Ensuring Atomicity of Transitions	53
4.4.2	Resetting of Reactivated Data Subcomponents	54
4.4.3	Resetting of Disconnected Data Ports	54
4.4.4	Translation of the Effects	56
4.4.5	Adjustment of the Activation Status of Control Subcomponents	56
4.5	Multiway Event Communication	57
4.5.1	Implementation of Master Transitions	58
4.5.2	Implementation of Reactive Transitions	60
4.6	Data Port Updates	64
4.6.1	Fixpoint Iteration	67
4.7	Activation Handling	70
4.8	The Environment Process	73
4.9	The Whole Resulting PROMELA Program	73
4.10	Example	76
5	Translation of Properties to Never Claims	83
5.1	Defining Symbols for Atomic Propositions	84
5.2	Never Claims for Property Patterns	85
5.2.1	Propositional	85
5.2.2	Absence Global	86
5.2.3	Existence Global	86
5.2.4	Universality Global	86
5.2.5	Precedence Global	87
5.2.6	Response Global	89
5.3	Final Remarks	90
III	Slicing of SLIM specifications	91
6	The Basic Slicing Approach	93
6.1	Related Work	93
6.2	Component Instances	94
6.3	Identifying Interesting Parts	95
6.3.1	Interesting Data Elements	95
6.3.2	Interesting Events	96
6.3.3	Interesting Modes	96
6.4	The Basic Slicing Algorithm	96
6.5	The Basic Sliced Specification	98

7	Improvements of the Slicing Algorithm	99
7.1	Considering Reactivation of Control Subcomponents	99
7.2	Preserving Divergence Characteristics	101
7.3	Extensions for Hybrid Behaviour	102
7.3.1	No Adaption for Extended Models	103
7.4	Permanently active connections and flows do not make modes interesting	103
7.5	Weak Interesting Data Elements	104
7.6	The Improved Slicing Algorithm	105
7.7	The Improved Sliced Specification	105
7.7.1	Retransformation to SLIM code	109
7.8	Possible Further Refinements	110
IV	Evaluation and Conclusions	111
8	Implementation and Practical Issues	113
8.1	How to perform Model Checking	113
8.2	How to invoke Slicing	114
8.3	Other Lessons Learned	114
9	Experimental Results	117
9.1	Negate Random Bit	118
9.2	Integer Adder	119
9.2.1	Simulation of SLIM specifications	122
9.3	Redundant Battery System	124
9.4	Wind Turbine	127
9.4.1	Nominal Behaviour	127
9.4.2	Error Behaviour	130
9.5	Open Thermal Loop	132
10	Summary, Conclusions and Future Work	133
10.1	Summary	133
10.2	Conclusions	134
10.3	Future Work	134

Chapter 1

Introduction

Having seen a tremendous progress in technology, people depend on hardware systems more than ever. However, the more complex such systems become, the less can human beings cope with controlling them or the more errors they make. This development led to two possible solutions: On the one hand, the research field of Human Factors Engineering (cf. [43]), a sub-discipline of Psychology, came up trying to understand how interaction between humans and machines take place aiming at the derivation of design rules for engineers. On the other hand, controlling of systems was automated. In fact, to an increasingly extend “hardware systems” contain software components responsible for the correct interplay of all parts. For obvious reasons, this software should fulfil high requirements with respect to its reliability. This applies in particular to large, expensive and safety critical machines, such as rockets, space shuttles, satellites, space robots, etc. Consequently, the European Space Agency (ESA) definitely faces this challenge and thus funds the COMPASS project (ESA/ESTEC Contract 21171/07/NL/JD) led by the Modeling and Verification of Software group (MOVES) at RWTH Aachen University together with the subcontracts Fondazione Bruno Kessler, Italy, who provide the model checker NUSMV, and with Thales Alenia Space, France, who provide realistic case studies.

1.1 The COMPASS project

A good overview over the **C**orrectness, **M**odeling, and **P**erformability of **A**erospace **S**ystems (COMPASS) project is provided by [11, 1]. The project’s goal is to enable the coherent application of formal methods during the design of systems. The approach should allow for *system-software co-engineering*, that is, both, hardware and software components should be covered by it. To this end, a special modeling language named SLIM was developed as an extended subset of AADL. Furthermore, for the analysis and verification of SLIM specifications existing model checkers are reused. Therefore, a toolchain of translators between the existing pieces of software was developed as depicted in figure 1.1. First of all, SLIM specifications are translated to SMV, the input language of NUSMV (New Symbolic Model Checker, [16, 13]), a reimplementation of SMV (cf. [33]). NUSMV is used for the verification of functional correctness properties. Furthermore, it generates a transition system representation of the SLIM specification which is used as input for MRMC (Markov Reward Model Checker, [32]). The functionalities of MRMC are used for performance evaluation by applying probabilistic model checking techniques. The translation from NUSMV to MRMC is supported by an adapted version of SIGREF [44] which performs bisimulation minimisation on the transition system and transforms its symbolic representation, as created by NUSMV, to an explicit representation, as required by MRMC.

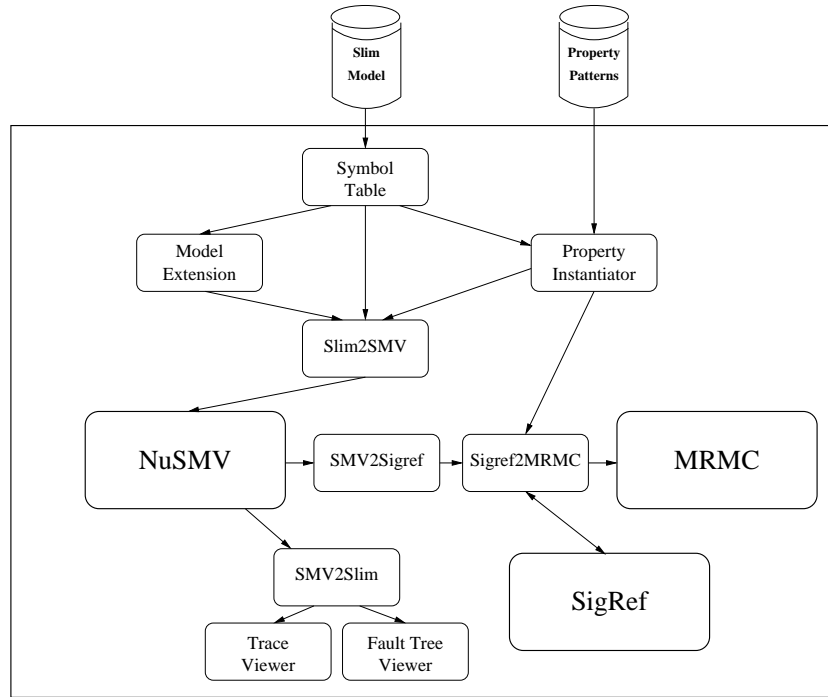


Figure 1.1: The COMPASS tool chain [11].

The outcome of the COMPASS project is a uniformed platform, that is, for the user one GUI is provided from which everything can be done with a mouse click and every result is presented in a coherent manner (cf. [7]). This eases the use of the toolset for system engineers by unburdening them from the need to manually invoke the different steps, to learn about every used software tool and especially from interpreting results in the model checker specific formats which are not directly linked to the objects used in SLIM specifications.

1.2 Contents of this Diploma Thesis

In the course of this diploma thesis, an alternative for model checking SLIM specifications with NuSMV was developed, namely model checking with the LTL-model checker SPIN. In analogy to translating SLIM to SMV, the basic task was to establish a translation of SLIM specifications to PROMELA programs, the input language of SPIN. Additionally, and totally independent of the translation to PROMELA, the idea of slicing was ported to SLIM specifications to tackle the peak state space size resulting from state space explosion. The thesis is organised in four parts:

- I. *Preliminaries*: Chapter 2 introduces syntax and semantics of the modeling language SLIM. The same is done for PROMELA, the input language of the SPIN model checker, in chapter 3, including some background on how model checking is done with SPIN.
- II. *Model Checking SLIM specifications using SPIN*: Chapter 4 formally defines the translation of SLIM models to PROMELA programs and explains the chosen approach. Similarly, properties have to be translated to PROMELA/SPIN as described in chapter 5.

- III. *Slicing SLIM specifications*: After providing a short review of existing slicing techniques, chapter 6 introduces the basics of the slicing algorithm for SLIM specifications. Thereafter, important but more involved extensions of the slicing algorithm are covered by chapter 7, including some optimisations.
- IV. *Results and Conclusions*: The implementation of the methods described in parts II and III is shortly considered in chapter 8. This implementation is used in chapter 9 to evaluate the translation and especially the reduction effects of slicing. Finally, chapter 10 summarises the thesis and gives its conclusions before an outlook on some possible future work is provided.

Part I

Preliminaries, Background

Chapter 2

The Architectural Design Language SLIM

This chapter introduces syntax and semantics of the SLIM (System-Level Integrated Modeling) language that was developed in the COMPASS project for the description of systems. The language is inspired by AADL (Architecture Analysis and Description Language, cf. [19]) and follows the same component-based paradigm. A SLIM specification defines a *hierarchy* of *control components*, collected in the set $CCmp^1$, under a topmost component, denoted $root \in CCmp$, which operate in parallel and can communicate with each other along different connections.

Each control component $c \in CCmp$ is given by its type, denoted $typ(c)$, and its implementation, denoted $imp(c)$. A *component type* describes the features of the component's interface as visible to other components: incoming and outgoing data and event *ports* for exchanging data instantaneously and event messages synchronously with other components. In contrast, a *component implementation* defines the internal behaviour of a component.

In the following sections, the formal syntax is sketched using a grammar-like notation with *NonTerminals* and *terminals*. Some “terminals” are variable, like *constant* values, *identifiers* or *names*. Identifiers, denoted *identifier*, id , id_i , are elements from the set

$$Ide := \{A, \dots, Z, a, \dots, z\} \times \{A, \dots, Z, a, \dots, z, 0, \dots, 9, -\}^*$$

However, SLIM keywords are forbidden as identifiers. Names, denoted *name*, consist of two identifiers separated by a dot:

$$Nam := Ide \times \{.\} \times Ide$$

If a variable terminal, like *identifier*, occurs several times within one syntax description it is meant to be the same one. Parentheses ($alt_1 \dots | alt_n$) are used to group alternative elements while square brackets [...] indicate optional parts. The presentation of the syntax concentrates on the aspects relevant for the scope of this thesis, that is, some details such as packages and physical bindings (e.g., which bus can be accessed by a component) are omitted. All details of the SLIM syntax can be found in [37]. At the same place, all syntactical restrictions, like the constraint that dependencies must always be acyclic, are listed.

The semantics of the SLIM language will be described textually in the subsequent sections. However, [35] gives a formal semantics in terms of a newly introduced automaton model called

¹The formal representation of SLIM specification is – except for some minor extensions – taken from [35].

event-data automaton that extends the concept of *hybrid automata* (cf. [22]) with the notion of data and event ports. The semantics defines for each control component an event-data automaton and a whole SLIM specification is reflected by a network of communicating event-data automata. In the end, the semantics of a SLIM specification yields a transition system whose states result from the product of all possible states from all control components, including the valuation of data elements. The transitions between such global states result from local transitions in one – or synchronously in several – control components, taking all possible ways of interleaving into account. For a more detailed description of the language, including a discussion of the similarities and extensions with respect to AADL, see also [8, 9, 10].

2.1 Component Type

Within a *component type* an arbitrary number of incoming and outgoing data and event ports can be declared. *Data ports* are used for the instantaneous exchange of data between components (cf. section 2.2.4) whereas *event ports* are needed for synchronous message communication involving transition triggers (cf. section 2.2.6). The syntax for a component type declaration is as follows:

```

ComponentCategory identifier
  features
    [ id1 : in data port DataType [ default constant1 ] ; ]*
    [ id2 : out data port DataType [ default constant2 ] ; ]*
    [ id3 : in event port ; ]*
    [ id4 : out event port ; ]*
  end identifier ;

```

Each component type is associated with a unique *identifier* which is later used to refer to it. The declaration of a component type contains also the choice of a *ComponentCategory*. Possible are **process**, **thread** and **thread group** for software components, **processor**, **memory**, **device** and **bus** for hardware components and finally **system** for composite components. However, since these categories do not have any effect on the semantics, they are irrelevant for the following discussions in this thesis and thus not further explained here.

For every data port its *DataType* (cf. section 2.3) must be given. Optionally, a *constant default value* of the respective type can be given. It is used – and thus required – for an incoming data port when it does not receive a value from outside the component and for an outgoing data port when the component does not explicitly provide an output value. As a consequence, for all incoming data ports of the **root** component default values must be given since no other component can provide inputs to the **root**.

For a control component $c \in CCmp$, the sets $IDPr(c)$, $ODPr(c)$, $IEPr(c)$, $OEPr(c)$ collect the incoming data ports, outgoing data ports, incoming event ports and outgoing event ports, respectively, of the component as defined in its component type $typ(c)$. Incoming and outgoing data ports for data and events are combined in the sets $DPr(c) := IDPr(c) \uplus ODPr(c)$ and $EPr(c) := IEPr(c) \uplus OEPr(c)$. The data type of a data port $dp \in DPr(c)$ is referred to as $typ(c, dp)$ and its default is given by $dfl(c, dp)$, which returns \perp if no default value exists.

2.2 Component Implementation

The behaviour of a component is defined by a component implementation in the following sense:

- A control component can contain other control components as subcomponents, reusing and combining their functionalities.

- Data subcomponents work like local variables for a control component.
- Port connections and flows are used to interconnect the ports of control components.
- Control components can operate in different modes. Together with transitions this allows an automata-like description of its behaviour. Furthermore, the activation of subcomponents and port connections can be restricted with respect to the mode of the component they belong to – giving allowance for *dynamic reconfiguration* of the whole system.

A component implementation declaration syntactically looks like this:

```

ComponentCategory implementation_name
  [ subcomponents
    ( ControlSubcomponent | DataSubcomponent )+ ]
  [ connections
    ( DataPortConnection | EventPortConnection )+ ]
  [ flows
    Flow+ ]
  [ modes
    StartingNominalMode
    NominalMode*
  [ transitions
    Transition+ ] ]
end_name;

```

The name is not only used for referring to the component implementation but also relates a component implementation to a component type: The first part of *name*, e.g., *typ* for *name* = *typ.imp*, must refer to a component type. The second part *imp* is used to differentiate between different implementations of the same component type. Every control component $c \in CCmp$ is an instance of one component implementation. However, there might be several control component instances of the same component implementation or even of different component implementations but of the same component type since multiple implementations for the same type are allowed.

2.2.1 Nominal Modes

The *nominal modes* of a control component $c \in CCmp$ are distinguished by their identifiers which are declared using the following syntax:

```
id : mode [ while invariant ] ;
```

The set $Mod(c)$ collects all modes of c . Furthermore, exactly one *starting mode* – either **initial** or **activation** – must be declared:

```
id : ( initial | activation ) mode [ while invariant ] ;
```

The starting mode of c is referred to as $stm(c)$. Iff the starting mode is of type **initial**, the component supports *mode history*, otherwise it does not. Mode history means, that a control component on reactivation continues operation from its state in which it was deactivated. A component without mode history is always reset on reactivation. Although suggested by the name, mode history does not only affect the mode information but also the values stored in data elements. Additionally, the absence of mode history propagates to all control subcomponents.

If a component implementation does not declare any mode at all, that is, $Mod(c) = \emptyset$, then it implicitly has a `_DefaultInitialMode` which is of type `initial`, thus a component without modes always supports “mode history” which is still relevant for data elements and control subcomponents. Formally, this results in $Mod(c) := \{_DefaultInitialMode\}$ and $stm(c) := _DefaultInitialMode$.

The optional *invariant* of a mode $m \in Mod(c)$ consists of subexpressions combined using `and`. Two kinds of subexpressions, serving different purposes, are possible:

- Boolean expressions, denoted $bexpr \in BExpr_c$, over the values of `clock` and `continuous` data subcomponents (see below). A component can be in a mode only when its invariant is fulfilled. On the one hand, this constraints whether a mode can be entered, and on the other hand, it limits whether it is possible to stay in the current mode.
- *Trajectory equations*, written as $\dot{d} = r$ for some `continuous` data subcomponent d and some $r \in \mathbb{R}$, defining r as the constant derivation for d while the component is in the respective mode.

An invariant can be perceived as a set of boolean expressions and trajectory equations and is referred to as $inv(c, m)$.

2.2.2 Control Subcomponents

Control components $c \in CCmp$ can be composed of other *control subcomponents*, reusing and combining their functionality. A control subcomponent is declared using the following syntax:

$$\underline{id} : ComponentCategory \underline{name} [\underline{in\ modes} (ListOfModes)] ;$$

The \underline{id} is a unique identifier of the subcomponent within the *supercomponent* in which it is defined. Those identifiers are collected in the set $CSub(c)$. The type and implementation of the subcomponent is given by \underline{name} which must be the name of a component implementation. It is possible to use several subcomponent instances of the same component implementation. However, it is forbidden to create recursive inclusion dependencies between component implementations. By giving an optional non-empty *ListOfModes*, the declared subcomponent can be restricted to be active only in the listed modes. The set $CSub(c, m)$ contains the identifiers of all control subcomponents $csc \in CSub(c)$ that are active in mode $m \in Mod(c)$. When no in modes information is given, the subcomponent is implicitly active in every mode.

Given one component implementation for the `root` component, the hierarchy of control components described by the SLIM specification implicitly results from the nesting of subcomponents. An important notion in the later descriptions is the so-called *access path*, used to refer to a certain control component *instance*. Formally, there is no difference between a control component and a control component instance – the wording “instance” is just sometimes used to emphasise that not component implementations are considered. Since several instances of the same component implementation can exist, it does in general not suffice to refer to a component implementation. The access path to a control component reflects the path from the `root` component through its nested control subcomponents to the control component in question. The access path to the `root` component is simply `root`. The access path to the control subcomponent $csc \in CSub(c)$ of a control component $c \in CCmp$ with access path $accessPath(c)$ is $accessPath(c).csc$. Formally, an access path is a non-empty, dot-separated concatenation of subcomponent identifiers, i.e., $APath \subset \{\text{root}\} \times (\cdot \times Ide)^*$. For simplicity, every control component $c \in CCmp$ is identified with its access path, i.e., $c = accessPath(c)$.

2.2.3 Data Subcomponents

Within a component implementation *data subcomponents*, comparable to local variables, can be declared:

$$\underline{id} : \underline{\text{data}} \text{ } \underline{\text{DataType}} [\underline{\text{default}} \text{ } \underline{\text{constant}}] [\underline{\text{in modes}} (\underline{\text{ListOfModes}})] ;$$

Like data ports, they are typed and can have a default value which is used as long as it is not overwritten. Like with control subcomponents, the availability of each data subcomponent can be restricted with respect to nominal modes of its supercomponent. In others than the listed modes the data subcomponent may not be used and on reactivation, including the first activation, it will be reset to its default value. Again, if no *in modes* information is present, the data component is implicitly always active.

For a control component $c \in CCmp$, the set $DSub(c, m)$ collects all identifiers of data subcomponents active in mode $m \in Mod(c)$. Consequently, $DSub(c) := \bigcup_{m \in Mod(c)} DSub(c, m)$ contains all data subcomponent identifiers used in c . Data subcomponents together with incoming and outgoing data ports are called *data elements*, represented by the sets $Dat(c, m) := DSub(c, m) \uplus DPrt(c)$ and $Dat(c) := DSub(c) \uplus DPrt(c)$. As with data ports, $typ(c, d)$ and $dfl(c, d)$ represent the data type and the default value (possibly \perp) of a data subcomponent $d \in DSub(c)$.

2.2.4 Data Port Connections and Flows

Data port connections are used for connecting different data ports of components to forward data so that the value of the *target port* \underline{tp} is instantaneously set to the value of the *source port* \underline{sp} whenever this changes. The syntax for establishing such connections is:

$$\underline{\text{data port}} \underline{sp} \underline{\rightarrow} \underline{tp} [\underline{\text{in modes}} (\underline{\text{ListOfModes}})] ;$$

Like control and data subcomponents, port connections can be restricted to be active in selected modes of the defining component only and again, when no *in modes* information is given they are implicitly always active. This way, not only the component hierarchy can dynamically be reconfigured but also the communication structure. The data types of source and target port must match. Furthermore, a port connection can be active only in modes in which both components, the one containing the source port and the one containing the target port, are active as well. Different kinds of port connections are possible inside a control component $c \in CCmp$:

- In-in: The value from an incoming data port $sp \in IDPrt(c)$ is forwarded to an incoming data port $tp' \in IDPrt(c.csc)$ of a control subcomponent $csc \in CSub(c)$, denoted $tp = csc.tp'$.
- Out-out: The value from an outgoing data port $sp' \in ODPrt(c.csc)$ of a control subcomponent $csc \in CSub(c)$, denoted $sp = csc.sp'$, is forwarded to an outgoing data port $tp \in ODPrt(c)$ of the component itself.
- Out-in: The value of an outgoing data port $sp' \in ODPrt(c.csc)$ of a control subcomponent $csc \in CSub(c)$, denoted $sp = csc.sp'$, is forwarded to the incoming data port $tp' \in IDPrt(c.csc')$ of a control subcomponent $csc' \in CSub(c)$, denoted $tp = csc'.tp'$. Indeed, the subcomponents can be the same one, i.e., $csc = csc'$ is possible.

A data flow, defined as

$$\underline{tp} := \underline{\text{expr}} [\underline{\text{in modes}} (\underline{\text{ListOfModes}})] ;$$

works in principle like a data port connection, but is more powerful: Firstly, not only the value of an data port can be forwarded but a new value can be calculated by an expression expr over the values of incoming data ports of the component itself and values of outgoing data ports of any control subcomponent. Secondly, flows allow additionally in-out connections, that is $tp \in ODPrt(c)$ and expr is/contains a $sp \in IDPrt(c)$. However, this introduces the risk of cyclic dependencies which are thus explicitly forbidden by syntactic restrictions: no data port may (transitively) depend on itself. Note carefully, that this does not exclude the forwarding of the value from an outgoing data port to an incoming data port of the same component. The restriction only forbids that this forwarding can influence the value of the outgoing data port in question. This situation of data dependencies cyclic on the component level and its consequences for the translation to PROMELA will be discussed in more detail in section 4.6.1.

If a data port connection or a data flow becomes deactivated then its target port is reset to its default value, if it exists. It must exist for all data ports that are targeted by data port connections and/or flows in some but not all modes. Fan-out is always possible whereas fan-in is not allowed.

Data port connections in a control component $c \in CCmp$ from sp to tp which are active in mode $m \in Mod(c)$ are collected in a set: $(sp, tp) \in DCon(c, m)$. Similarly, flows $d := expr$ are collected as $(d, expr) \in Flw(c, m)$.

2.2.5 Event Port Connections

In analogy to data port connections, event port connections transport message signals. The syntax of their declarations is nearly identical:

event port sp -> tp [in modes (list of Ide)] ;

The same connection topologies as with data port connections are possible, i.e., in-in, out-out and out-in connections. A corresponding counterpart for flows allowing in-out connections does not exist for events. Fan-out and – in opposite to data port connections – fan-in are possible, but it must be guaranteed that a component does not receive the same original event more than once. Furthermore, event connections must be acyclic on the component level, that is, a component can never receive an incoming event that came along an outgoing event port of the component somewhere before.

An event port connection defined in a control component $c \in CCmp$ from source port sp to target port tp active in mode $m \in Mod(c)$ is represented as $(tp, sp) \in ECon(c, m)$.

2.2.6 Transitions

Transitions between nominal modes allow to change the current mode of the component. A transition in a control component $c \in CCmp$, declared as

m_s -[[t] when g] then f] -> m_t ;

contains the following elements:

- A *source mode* $m_s \in Mod(c)$ and a *target mode* $m_t \in Mod(c)$.
- An optional *trigger* t which is either an event port of the component itself, i.e., $t \in EPrt(c)$, or an event port of a control subcomponent, that is, $t = csc.ep$ with $ep \in EPrt(c.csc)$ for some control subcomponent $csc \in CSub(c)$. Transitions without a trigger are called τ -*transitions*.

- The optional *guard* g is a boolean expression over data elements of the component which restricts the enabledness of a transition: A transition can be taken only when its guard evaluates to **true** in the current state.
- The *effect* f is a – possibly empty – list of assignments $d := expr$ to outgoing data ports and data subcomponents of c active in the target mode m_t , i.e., $d \in DSub(c, m_t) \uplus ODPrt(c)$. The new value can be calculated by an expression $expr$ over all data elements of c active in the source mode m_s , that is, over $d' \in Dat(c, m_s)$. Data ports of control subcomponents can neither be read nor set in transition effects. The right hand sides of all assignments are evaluated in the old state so that the effect $x := y; y := x$ indeed swaps the values of x and y .

Subject to the evaluation of the guard, a τ -transition can always be taken. In contrast, transitions with a trigger require *multiway event communication*: A transition with an outgoing event port of the component itself, i.e., $t \in OEPrt(c)$, or with an incoming event port of a control subcomponent, i.e., $t = csc.iep$, $csc \in CSub(c)$, $iep \in IEPrt(c.csc)$, as trigger – such transitions will be referred to as *master transitions* – can be taken only when synchronously in at least one other component $c' \in CSub \setminus \{c\}$ a transition can be taken that can “receive” the triggered event, that is, it has an incoming event port of its defining component, i.e., $t' \in IEPrt(c')$, or an outgoing event port of a control subcomponent, i.e., $t' = csc'.oep$, $csc' \in CSub(c')$, $oep \in OEPrt(c'.csc')$, as trigger – analogously, these transitions are referred to as *reactive transitions* – to which the originally emitted event is forwarded (transitively) along event port connections. Actually, during multiway event communication every component containing one or more enabled reactive transitions receiving a triggered event must take one of them. The other way round, a reactive transition can be taken only, when it is triggered by a master transition. Figure 2.1 sketches a typical situation for multiway event communication.

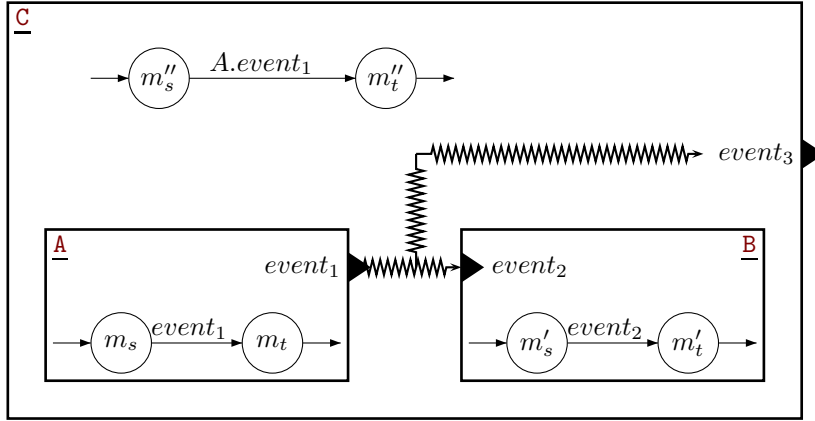


Figure 2.1: The master transition in component **A** can only be taken when at least one reactive transition in another component, like the one in **B**, to which the triggered event is forwarded along event port connections can be taken synchronously. The reactive transition in component **C** uses the special notation **A.event₁** for event ports of subcomponents.

A transition declaration as above in a control component $c \in CCmp$ is formally represented as $(m_s, t, g, f, m_t) \in MTr(c)$, sometimes written as $m_s \xrightarrow{t, g, f} m_t$.

2.3 Data Types and Expressions

SLIM supports the data types listed in table 2.1. The data types **clock** and **continuous**, which are used to model hybrid behaviour, are allowed for data subcomponents only, not for data ports. The derivation for **continuous** data subcomponents d can be mode dependent and is given by a trajectory equation $\dot{d} = a$, for some $a \in \mathbb{R}$, attached to mode invariants (cf. section 2.2.1). Data subcomponents of type **clock** – which can be seen as a special subtype of **continuous** – always change linearly over time, that is, with derivation 1, and can be reset to 0 only, not to other constant values. For a control component $c \in CCmp$ let $Enums(c)$ collect all symbolic identifiers that occur in an enumeration data type **enum**(id_1, \dots, id_n) defined in c . Furthermore, define $Enums := \bigcup_{c \in CCmp} Enums(c)$.

<i>DataType</i>	Possible Values
bool	true, false
int	$\mathbb{Z} = [-](0 \dots 9)^+$
real	$\mathbb{R} = [-](0 \dots 9)^+ [. (0 \dots 9)^+]$
enum (id_1, \dots, id_n)	Distinct symbolic values id_1, \dots, id_n for some $n \in \mathbb{N}_{>0}$
[$l..u$]	Integer Range: Like int , but restricted to $\{z \in \mathbb{Z} \mid l \leq z \leq u\}$ for some $l, u \in \mathbb{Z}$ with $l \leq u$
clock	$\mathbb{R}_{\geq 0}$, linearly increasing over time
continuous	\mathbb{R} , continuously changing according to a constant derivation

Table 2.1: SLIM data types and their possible values.

Expressions, which are always interpreted in the context of a control component $c \in CCmp$, are made up of constant values and reading accesses to data elements of the component $d \in Dat(c)$ or to outgoing data ports $d \in ODPrt(c.csc)$ of a control subcomponent $csc \in CSub(c)$, using the notation $csc.d$, combined by operators of different kinds:

- Arithmetic operators: **+**, **-** (unary and binary), *****, **/**, **mod**
- Relational operators: **=**, **!=**, **<**, **>**, **<=**, **>=**
- Boolean operators: **not**, **and**, **or**, **imp**, **iff**, **xor**, **xnor**

One more complex operator is the **case** operator using the following syntax for some $n \in \mathbb{N}_{>0}$:

case $b_1 : e_1 ; \dots ; b_n : e_n$ **otherwise** e_0 **end**

It returns the value of the subexpression e_i for the smallest $1 \leq i \leq n$ such that the boolean expression b_i evaluates to **true**. If no such i exists, the value of e_0 is returned.

For more details on the data types, the typing of expressions, the precedence of operators and syntactical restrictions see [37, section 4.2.1].

2.4 Example

Listing 2.1 gives an example SLIM specification modeling a system consisting of a **negator** component whose input **pos** is randomly generated using nondeterministic transitions in component **randomBit**. For gaining a better overview, a visualisation as in figure 2.2 is often useful. Formally, this specification is represented as follows:

- $CCmp := \{\text{root}, \text{root.randomBit}, \text{root.negator}, \text{root.aBus}\}$
- For $c = \text{root}$:
 - $typ(c) := \text{Root}, imp(c) := \text{Root.Impl}$
 - $IDPr(c) = ODP(c) = IEP(c) = OEP(c) := \emptyset$
 - $Mod(c) = \emptyset \Rightarrow Mod(c) := \{_DefaultInitialMode\}, stm(c) := _DefaultInitialMode$
 - $CSub(c, m) := \{\text{randomBit}, \text{negator}, \text{aBus}\}$ for every $m \in Mod(c)$
 - $DCon(c, m) := \{(\text{randomBit.value}, \text{negator.pos})\}$ for every $m \in Mod(c)$
 - $DSub(c, m) = ECon(c, m) = Flw(c, m) := \emptyset$ for every $m \in Mod(c)$
 - $MTr(c) := \emptyset$
- For $c = \text{root.randomBit}$:
 - $typ(c) := \text{RandomBit}, imp(c) := \text{RandomBit.Impl}$
 - $ODPr(c) := \{\text{value}\}, typ(c, \text{value}) := \text{bool}, dfl(c, \text{value}) := \text{false}$
 - $IDPr(c) = IEP(c) = OEP(c) := \emptyset$
 - $Mod(c) := \{m\}, stm(c) := m$
 - $CSub(c, m) = DSub(c, m) = DCon(c, m) = Flw(c, m) = ECon(c, m) := \emptyset$ for every $m \in Mod(c)$
 - $MTr(c) := \{(m, \tau, \epsilon, \text{value} := \text{true}, m), (m, \tau, \epsilon, \text{value} := \text{false}, m)\}$
- For $c = \text{root.negator}$:
 - $typ(c) := \text{Negator}, imp(c) := \text{Negator.Impl}$
 - $IDPr(c) := \{\text{pos}\}, typ(c, \text{pos}) := \text{bool}, dfl(c, \text{pos}) := \text{false}$
 - $ODPr(c) := \{\text{neg}\}, typ(c, \text{neg}) := \text{bool}, dfl(c, \text{neg}) := \text{false}$
 - $IEPr(c) = OEP(c) := \emptyset$
 - $Mod(c) = \emptyset \Rightarrow Mod(c) := \{_DefaultInitialMode\}, stm(c) := _DefaultInitialMode$
 - $Flw(c, m) := \{(\text{neg}, \text{!pos})\}$ for every $m \in Mod(c)$
 - $CSub(c, m) = DSub(c, m) = DCon(c, m) = ECon(c, m) := \emptyset$ for every $m \in Mod(c)$
 - $MTr(c) := \emptyset$
- For $c = \text{root.aBus}$: omitted

2.5 Modeling Errors

With the language constructs introduced so far, *nominal models* describing the normal system behavior are written. They can be complemented by *error models*, which express how and with which probability a component might fail, accompanied by *fault injections* describing the effects of failures, e.g., changes to the data elements. However, a process called *model extension* integrates error and normal behaviour to a new, almost normal nominal model. Thus, for the scope of this thesis it suffices to consider nominal models only. The principle idea of model extension is to add a new subcomponent called `_errorSubcomponent` to every nominal component which is associated with an error model. This new subcomponent is responsible for maintaining

```

system Root
end Root;

system implementation Root.Impl
  subcomponents
    randomBit: system RandomBit.Impl accesses aBus;
    negator:   system Negator.Impl  accesses aBus;
    aBus:     bus Bus.Impl;
  connections
    data port randomBit.value -> negator.pos;
end Root.Impl;

system RandomBit
  features
    value: out data port bool default false;
end RandomBit;

system implementation RandomBit.Impl
  modes
    m: initial mode;
  transitions
    m -[then value := true]-> m;
    m -[then value := false]-> m;
end RandomBit.Impl;

system Negator
  features
    pos: in data port bool default false;
    neg: out data port bool default false;
end Negator;

system implementation Negator.Impl
  flows
    neg := !pos;
end Negator.Impl;

bus Bus
end Bus;

bus implementation Bus.Impl
end Bus.Impl;

```

Listing 2.1: A system negating a random bit described as a SLIM specification.

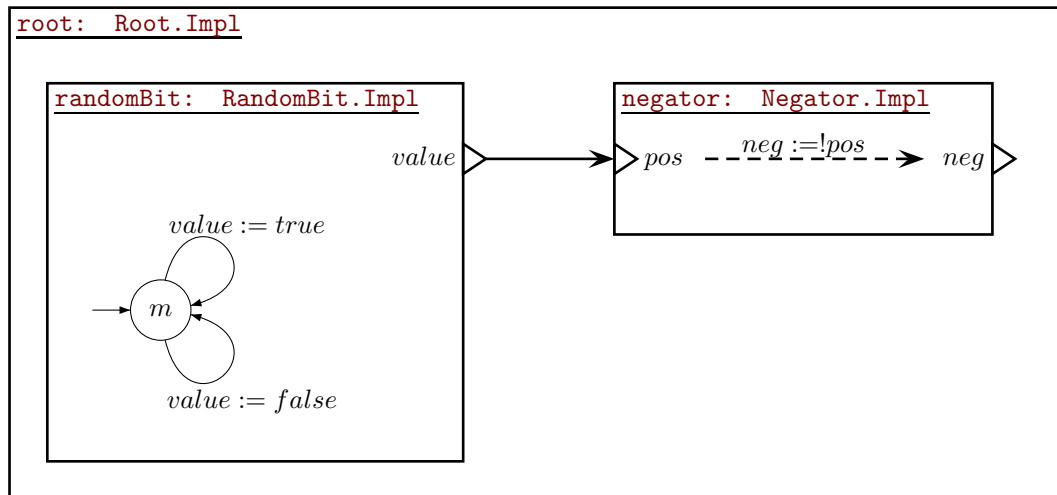


Figure 2.2: Visualisation of the SLIM specification Negate Random Bit from listing 2.1.

the current error state of the component. Furthermore, all transitions and flows of the original component are extended to consider the current error state and the fault effects that have to be applied in them. More on the syntax of error models can be found in [37, sections 4.4 & 4.5] while its semantics by model extension is described in [36, chapter 4], based on a method described in [12]. In the sense that the SLIM language is inspired by AADL, the concept of error models is taken from the *error model annex* of AADL (cf. [18]).

Listing 2.2 gives the error model **NegatorError.Impl**. The control component **root.negator** from the Negate Random Bit example in listing 2.1 can be associated with it together with the fault effect that in error state **Erroneous** its outgoing data port **neg** is constantly set to **false**. The SLIM specification generated by model extension from the nominal specification under this fault injection is given in listing 2.3.

```

error model NegatorError
  features
    OK: initial state;
    Erroneous: error state;
  end NegatorError;

error model implementation NegatorError.Impl
  events
    ErrorOccurs: error event;
  transitions
    OK -[ErrorOccurs]-> Erroneous;
  end NegatorError.Impl;

```

Listing 2.2: An error model containing two error states **OK** and **Erroneous**. The error state is changed when the error event **ErrorOccurs** spontaneously (here, no occurrence rate is given) happens.

```

system Extended__Root1
end Extended__Root1;

system implementation Extended__Root1.Impl
  subcomponents
    negator: system Extended_negator_Negator1.Impl accesses aBus;
    aBus: bus Extended_aBus_Bus1.Impl;
    randomBit: system Extended_randomBit_RandomBit1.Impl accesses aBus;
  connections
    data port randomBit.value -> negator.pos;
  modes
    _DefaultInitialMode : initial mode;
end Extended__Root1.Impl;

system Extended_randomBit_RandomBit1
  features
    value: out data port bool default false;
end Extended_randomBit_RandomBit1;

system implementation Extended_randomBit_RandomBit1.Impl
  modes
    m : initial mode;
  transitions
    m -[then value := true]-> m;
    m -[then value := false]-> m;
end Extended_randomBit_RandomBit1.Impl;

system Extended_negator_Negator1
  features
    pos: in data port bool default false;
    neg: out data port bool default false;
    _errorState: out data port enum(_Erroneous, _OK) default _OK;
end Extended_negator_Negator1;

system implementation Extended_negator_Negator1.Impl
  subcomponents
    _errorSubcomponent: system NegatorError1.Implementation;
  connections
    data port _errorSubcomponent._errorState -> _errorState;
  flows
    neg := case _errorSubcomponent._errorState=_Erroneous: false
           otherwise: not pos end in modes (_DefaultInitialMode);
  modes
    _DefaultInitialMode : initial mode;
  transitions
    _DefaultInitialMode -[_errorSubcomponent.#ErrorOccurs
      when _errorState = _OK]-> _DefaultInitialMode;
end Extended_negator_Negator1.Impl;

```

```

system NegatorError1
  features
    _errorState: out data port enum(_Erroneous, _OK) default _OK;
    _resetEvent: in event port;
    #ErrorOccurs: error event port;
end NegatorError1;

system implementation NegatorError1.Implementation
  modes
    Erroneous : error mode;
    OK : initial mode;
  transitions
    OK -[#ErrorOccurs then _errorState := _Erroneous]-> Erroneous;
    Erroneous -[_resetEvent]-> Erroneous;
    OK -[_resetEvent]-> OK;
end NegatorError1.Implementation;

bus Extended_aBus_Bus1
end Extended_aBus_Bus1;

bus implementation Extended_aBus_Bus1.Impl
  modes
    _DefaultInitialMode : initial mode;
end Extended_aBus_Bus1.Impl;

```

Listing 2.3: Extended model of the Negate Random Bit specification from listing 2.1.

2.6 Defining Properties

Within the COMPASS toolset, properties are defined based on *property patterns* (cf. [7, chapter 7]). These patterns consist of a *story* with *placeholders* (denoted P , S) for atomic propositions and correspond to a logic formula. Table 2.2 lists the LTL patterns used in the COMPASS toolset. Nearly the same patterns exist for CTL and others for CSL, dealing with probabilities and timing. However, since this thesis mainly targets at a translation of SLIM specifications to PROMELA, the input language of the LTL-model checker SPIN, those patterns are not discussed here.

Pattern	Story	LTL-Formula
Propositional	In the initial system state, P holds.	P
Absence Global	P never holds.	$\Box \neg P$
Existence Global	P eventually holds.	$\Diamond P$
Universality Global	P always holds.	$\Box P$
Precedence Global	P always precedes S	$(\neg S \cup P) \wedge \Box \neg S$
Response Global	Whenever P holds, eventually S holds.	$\Box (P \Rightarrow \Diamond S)$

Table 2.2: LTL property patterns used in COMPASS.

The wording “eventually” means “in the same or some subsequent state”. Note carefully, that LTL patterns implicitly argue always about all possible execution paths of a system. As one consequence, the patterns absence global and existence global are *not* dual: If P holds on

some but not all execution paths, none of them is valid. However, they are still mutual exclusive. Verifying that a property holds for some but not necessarily all paths is possible by falsifying that it is invalid for all paths. For example, “on some paths eventually P holds” is valid iff “on all paths never P holds” – which is an LTL property – is invalid. In this case, the counterexample is in fact a witness of the desired behaviour.

As *atomic propositions* to fill-in the placeholders in the property patterns, boolean expression over values of data elements, nominal modes and error states of control components occurring in the specification are used. As operators, all operators from the SLIM language can be used (boolean, relational, arithmetical). The wording “atomic” thus does not mean atomic with respect to boolean operators, i.e., only arithmetic and relational expressions, but indicates that no *temporal* operators can occur in atomic propositions. In order to refer to a data element of a certain control component, their identifier is prefixed with the access path to the respective component, e.g., the data element $d \in \text{Dat}(c)$ of a control component $c \in \text{CCmp}$ is addressed by $c.d$. The nominal and error mode of a control component $c \in \text{CCmp}$ is referred to by adding the suffix `.mode` or `.error`, respectively, to the access path of c . Numeric and boolean constant values can be used in the same way as in SLIM specifications. In contrast, identifiers of enumeration literals, nominal modes and error states must be prefixed with `enum:`, `mode:` or `error:`, respectively. The access paths used in properties actually omit the explicit naming of the root component, that is, instead of `root.d` or `root.sc.mode` just `d` and `sc.mode` are used. Possible properties for the SLIM specification from listing 2.1 are, for example:

- φ_1 : Universality global with $P = \text{root.negator.neg} = \text{not root.negator.pos}$, claiming that the negator works correctly (valid).
- φ_2 : Propositional with $P = \text{not root.negator.neg}$, stating that output `neg` of component `root.negator` is `false` in the initial state. Although $\text{df}(\text{root.negator}, \text{neg}) = \text{false}$ holds, this property is invalid since already in the initial state the component `root.randomBit` provides the `value false` which is forwarded to and negated by the `root.negator` control component resulting in `root.negator.neg` being `true` in the initial state.
- φ_3 : Absence global with $P = \text{root.randomBit.value}$, claiming that outgoing data port `value` of control component `root.randomBit` is never set to `true`. Obviously, this does not hold since the transition with effect `value := true` can always be taken.
- φ_4 : Existence global with $P = \text{root.randomBit.value}$, claiming that the outgoing data port `value` of control component `root.randomBit` will in every execution eventually be set to `true`. This property does not hold either since it is possible that `root.randomBit` only takes the transition with effect `value := false` ad infinitum.

An important limitation is that atomic propositions must not refer to deactive components, of both kinds, control and data. This can be achieved by using an additional implication with a premise about the modes of components. For example, consider a data subcomponent $d \in \text{DSub}(\text{root})$ of the `root` component that is active in the modes $\{m_0, \dots, m_n\} \subsetneq \text{Mod}(\text{root})$. Checking `d <= 10` might result in unexpected results since the value of deactive data sub-components is in general undefined. Instead, checking `(mode = mode:m0 or ... or mode = mode:mn) imp d <= 10` is safe. Unfortunately, SLIM does not offer predefined predicates about the activation status of components.

One or more properties can be stored in an XML-file which, besides the SLIM specification, later can be used as an input for the translator and slicer. Listing 2.4 gives an example of such an XML-file for the properties $\varphi_1, \dots, \varphi_4$.

```

<?xml version="1.0" encoding="UTF-8" ?>

<props:properties
  xmlns:props="http://compass.informatik.rwth-aachen.de/properties"
  xmlns:ltl="http://compass.informatik.rwth-aachen.de/ltl">

  <props:property>
    <props:name>Phi 1</props:name>
    <props:description>First Example Property</props:description>
    <props:ltlPatternInstance patternId="universalityGlobal">
      <ltl:stateFormulaInstance placeholderId="P">
        <ltl:atomicProposition>
          negator.neg = not negator.pos
        </ltl:atomicProposition>
      </ltl:stateFormulaInstance>
    </props:ltlPatternInstance>
  </props:property>

  <props:property>
    <props:name>Phi 2</props:name>
    <props:description>The Second Property</props:description>
    <props:ltlPatternInstance patternId="propositional">
      <ltl:stateFormulaInstance placeholderId="P">
        <ltl:atomicProposition>
          not negator.neg
        </ltl:atomicProposition>
      </ltl:stateFormulaInstance>
    </props:ltlPatternInstance>
  </props:property>

  <props:property>
    <props:name>Phi 3</props:name>
    <props:description>A third story</props:description>
    <props:ltlPatternInstance patternId="absenceGlobal">
      <ltl:stateFormulaInstance placeholderId="P">
        <ltl:atomicProposition>
          randomBit.value
        </ltl:atomicProposition>
      </ltl:stateFormulaInstance>
    </props:ltlPatternInstance>
  </props:property>

  <props:property>
    <props:name>Phi 4</props:name>
    <props:description>The last story</props:description>
    <props:ltlPatternInstance patternId="existenceGlobal">

```

```

    <ltl:stateFormulaInstance placeholderId="P">
      <ltl:atomicProposition>
        randomBit.value
      </ltl:atomicProposition>
    </ltl:stateFormulaInstance>
  </props:ltlPatternInstance>
</props:property>

</props:properties>

```

Listing 2.4: XML-file containing some properties.

Formally, a property φ is given by a *pattern*(φ) and a set of named atomic propositions $APs(\varphi) \subset \{(id, ap) \mid id \in Ide, ap \text{ an atomic proposition}\}$. For the example properties presented here, this yields:

- $pattern(\varphi_1) := \text{universalityGlobal}$,
 $APs(\varphi_1) := \{(P, \text{root.negator.neg} = \text{not root.negator.pos})\}$
- $pattern(\varphi_2) := \text{propositional}$, $APs(\varphi_2) := \{(P, \text{not root.negator.neg})\}$
- $pattern(\varphi_3) := \text{absenceGlobal}$, $APs(\varphi_3) := \{(P, \text{root.randomBit.value})\}$
- $pattern(\varphi_4) := \text{existenceGlobal}$, $APs(\varphi_4) := \{(P, \text{root.randomBit.value})\}$

Chapter 3

The Model Checker SPIN

SPIN is a widely used LTL-model checker that was mainly developed by Holzmann. In [27] he quotes Joseph Joubert with the words “The worst thing about new books is that they keep us from reading the old ones.” In this spirit, this chapter gives a rather short and informal introduction to SPIN, its input language PROMELA and the basic means of model checking with SPIN. It neither explains the theoretical approach to model checking nor does it give much of the details about implementation and usage of SPIN and PROMELA. Instead, for the former I refer to [41, 17, 5] and for the latter to [3, 23, 27, 6].

3.1 Input Language PROMELA

The system model for SPIN is not directly given as a transition system but as a PROMELA program, abbreviation for **Process Meta Language**, specifying a set of concurrent processes which can interact by communication via channels and global variables. This section introduces PROMELA with the focus on relevant parts as used in translation from SLIM to PROMELA. A complete description of PROMELA can be found in [3, 27].

3.1.1 Syntactical Constructs

Process Types *Processes are instances of a **proctype** (process type) declared in the PROMELA program defining the behaviour of the process. There are basically two ways to create processes from proctypes: If the keyword **active** is used in front of a proctype declaration, SPIN will automatically create a process instance from it on start-up. The other method is to use the **run** operator on a proctype name to start a new execution of that proctype from within an existing process (which might be of the same type). In the initial state at least one process must be active, thus at least one proctype must be declared **active** or the special notation for an **init**-process must be used: **init { ... }**. Every running process has a unique identifier, stored in the predefined local variable **_pid**. In theory a system can consist of an arbitrary (finite) number of processes, but SPIN limits it to 255 running processes. Listing 3.1 gives the obligatory first “Hello, World!” example in PROMELA code, using **printf** to show a message on screen.*

```

active proctype HelloWorld()
{
    printf("Hello , World!")
}

```

Listing 3.1: “Hello, World!” as a PROMELA program.

Variables, Assignments and Expressions Processes can keep and manipulate data stored in *variables*. Depending on the fact whether they are declared inside or outside of a proctype, their scope is either *global*, i.e., accessible from every process, or *local*, i.e., only accessible from the process they belong to, respectively. Every instance of a proctype has its own copy of local variables, independent from all other instances of the same proctype. Variables are declared in a C-like syntax, e.g., `int x = 5`, providing a name, a type and optionally an initial value. All variables store integer values - no floating-point numbers, no strings, etc. - but possibly from different domains and thus with different memory demands, as specified by the type: `bit` and `bool` ($\{0, 1\}$ in one bit), `byte` ($[0, 255]$ in one byte), `short` ($[-2^{15}, 2^{15} - 1] = [-32768, 32767]$ in two bytes) and `int` ($[-2^{31}, 2^{31} - 1]$ in four bytes). The type `unsigned` allows to specify how many bits of memory (n) should be allocated for a variable which then can store values from $[0, 2^n - 1]$. The type `mtype` can exactly once be declared with a list of literal symbols which are internally represented by integers - comparable to `enum` data types in C. Variables for which no initial value is provided are by default initialised to 0.

The value of variables can be changed by assignments to them, e.g., `x = 2*x`. The operators which can be used in expressions for calculations (`+`, `-`, `*`, `/` (integer division), `%` (modulo), etc.), for comparisons (`==`, `!=`, `<`, `>`, `<=`, `>=`) and for logical combinations (`!`, `&&`, `||`, etc.) are essentially the same as in C. One difference is that the conditional expression

$$\underline{cond} \rightarrow \underline{expr_1} : \underline{expr_2}$$

uses `->` in PROMELA instead of `?` in C. However, there is no difference in the meaning: The value of $\underline{expr_1}$ is returned iff \underline{cond} holds, otherwise the value of $\underline{expr_2}$ is used. In general, conditional expressions do hold when they evaluate to a value unequal zero. In particular, the constants `true` and `false` are just names for the values `1` and `0`, respectively. The results of comparisons and boolean operators are represented in the same way. Listing 3.2 gives an example PROMELA program using a local and a global variable.

```

int aGlobalVariable = 5;

active proctype MyFirstAutorunProcess()
{
    int aLocalVariable; // default: 0
    aLocalVariable = aLocalVariable + 2 * aGlobalVariable;
    printf("Result is: %d.\n", aLocalVariable)
    // Output: Result is: 10.
}

```

Listing 3.2: Local and global variables in PROMELA.

Channels for Message Passing As already mentioned, processes do not only share global variables but can communicate via *channels* as well. Channels have to be declared with a name, a *capacity* (non-negative integer constant) and the structure of messages that are to be transported using the following syntax:

chan name = [capacity] of { type (, type)* }

Messages are n -tuples $value_1, \dots, value_n$, where $n \geq 1$ is the number of occurrences of *types* in the channel declaration, such that $value_i$ is of type $type_i$ for $1 \leq i \leq n$. A message is send along a channel *name* using the !-notation:

name!expression₁, ..., expression _{n}

The message contains the values resulting from the evaluation of the given expression _{i} s. Analogously, a message can be received by:

name?placeholder₁, ..., placeholder _{n}

In both cases the number and type of parameters must exactly fit to the message structure defined in the channel declaration. For the placeholder _{i} s are typically variables used which store the values of the incoming message. Within one receive statement every variable may be used at most once. However, transmitted values that are not needed later on do not have to be stored in variables. Instead, the symbol `_` can be used as placeholder, even multiple times. Finally, constant values can be given as placeholders somehow allowing to filter incoming messages: only those messages can be received for which the value of the corresponding message field is equal to the constant value. The value of a variable cannot be used as a filter directly by just giving the variable name since that could syntactically not be distinguished from using the variable to store the received value. As a workaround, the predefined function `eval`, which returns the evaluation value of an expression or just a single variable, can be used. All three kind of placeholders can be mixed within one receive statement. Listing 3.3 gives a small example of data transfer between two processes whereas in listing 3.4 filtering is applied.

```
chan SimpleChannel = chan [0] of { int };

active proctype Sender()
{
    SimpleChannel!123;
}

active proctype Receiver()
{
    int x;
    SimpleChannel?x;
    printf("Received: %d.\n", x);
    // Output: Received: 123.
}
```

Listing 3.3: Synchronous transfer of 123 from `Sender` to `Receiver` which stores the received value in its local variable `x`.

```
mtype = { atA, atB, atC };
chan DeviceBus = chan [3] of { mtype, bool };

active proctype Controller()
{
```

```

    // Switch devices on
    DeviceBus!atA,true;
    DeviceBus!atB,true;
    DeviceBus!atC,true;
}

active proctype DeviceB()
{
    bool isOn = false;

    // Receive on/off message
    DeviceBus?atB,isOn;
}

```

Listing 3.4: Asynchronous communication with filtering: `DeviceB` will not react to messages whose first value is not `atB`.

Two types of channels can be distinguished: buffered (*capacity* > 0) and unbuffered (*capacity* = 0) ones. *Buffered channels* can store up to *capacity* many messages in a FIFO manner and thus allow *asynchronous* communication: a send statement is executable when the channel is not full, a receive statement is executable when the channel is not empty (and the first message fits the filter conditions, if present). *Unbuffered channels* cannot store message and thus require *synchronous* communication: a send statement in one process is only executable when in another process a receive statement for that message is executable at the same time. The PROMELA program in listing 3.5 contrasts asynchronous and synchronous communication. Firstly, `Sender` must write `1` to the buffered channel `channelA`. Secondly, `Sender` and `Receiver` can synchronously communicate along channel `channelS`. Thirdly, `Receiver` can receive the buffered message from channel `channelA`. If channel `channelA` is unbuffered as well, both process would stuck in the initial state.

```

chan channelA = [1] of { byte };
chan channelS = [0] of { byte };

active proctype Sender()
{
    channelA!1;    // (1.)
    channelS!2;   // (2. a)
}

active proctype Receiver()
{
    channelS?_;   // (2. b)
    channelA?_;  // (3.)
}

```

Listing 3.5: Asynchronous vs. synchronous communication.

Independent of buffered or unbuffered channels, communication is always binary, i.e., a message is always send by one process and (for buffered channels somewhen later) received by exactly one (for unbuffered channels: a different) process. Multiway communication from one sender to many receiver processes is not directly possible with PROMELA. Using angle brackets around the placeholders in a receive statement, e.g., `name?< placeholders>`, messages can be read from a

channel without removing them from it but this does not do any bookkeeping about which processes received the message. However, it might be that several processes are able to receive the same message at the same time. Then a nondeterministic choice between the possible receivers has to be made as the example in listing 3.6 demonstrates. It does not make any difference whether a buffered or an unbuffered channel is used.

```

chan channel = [1] of { byte };

active proctype Sender()
{
    channel!22;
}

active proctype Receiver1()
{
    byte x;
    channel?x;
}

active proctype Receiver2()
{
    byte x;
    channel?x;
}

```

Listing 3.6: Nondeterministic binary communication: Either `Receiver1` or `Receiver2` receives the message – but not both!

Using an unbuffered channel, a simple but powerful *request-acknowledgment-mechanism* between processes can be established as shown in listing 3.7. Note that, due to the fact that the used channel requires synchronous communication between different processes, it is impossible that either `Client` or `Provider` communicates with itself.

```

mtype = { ping };
chan channel = [0] of { mtype };

active proctype Client()
{
    channel!ping;
    // Wait for acknowledgment
    channel?ping;
}

active proctype Provider()
{
    channel?ping;
    // Do something ... Send acknowledgment
    channel!ping;
}

```

Listing 3.7: `Client` request something from `Provider` and then waits for the acknowledgment.

Scoping with channels is different to normal variables. A channel declaration actually does two things: Firstly, it creates a new channel with the given capacity for messages of the specified type. Each channel has a unique internal identifier. Secondly, it declares a variable of type `chan` that stores the identifier of the new channel. Like with normal variables the scope of the channel variable depends on where the declaration is placed. However, channels themselves are global objects and can be used by every process knowing the respective channel identifier. Indeed it is possible to transport channel identifiers along channels which allows dynamic communication structures or mobility. Like with proctypes the number of channels is restricted by SPIN to 255. The size of buffered channels should be chosen carefully since a linear increase in the capacity – or in general, in the amount of stored data – leads to an exponential increase of the induced state space size (one aspect of the so-called *state space explosion problem*).

Executability of Statements The notion of *executability* of statements is very important in PROMELA as this concept is the underlying principle of conditions. Statements are either *executable* or *blocked*. While assignments can always be executed, the executability of send and receive statements depends on the channel content or handshaking between processes as described above. In PROMELA, expressions are statements as well and they are “executable” – although without any side effect – iff they do not evaluate to zero. This allows to use both, typical conditional expressions and statements like send and receive, as conditions. Furthermore, a simple statement suffices to wait for a condition to come true, e.g., that a message could be sent/received or that variables have certain values. Active waiting loops like “while condition not fulfilled do nothing” are not necessary. With this method, simple synchronisation barriers based on global variables can be implemented as the example in listing 3.8 shows.

```

bool completedInitialisationOfA = false;
bool completedInitialisationOfB = false;

active proctype ProcessA()
{
    // Do some initialisation ...
    completedInitialisationOfA = true;
    // Wait until process B has finished its initialisation
    completedInitialisationOfB == true;
    // Continue ...
}

active proctype ProcessB()
{
    // Do some initialisation
    completedInitialisationOfB = true;
    // Wait until process A has finished its initialisation
    completedInitialisationOfA;
    // Continue ...
}

```

Listing 3.8: Global variables used as barriers.

Like `true`, the “empty statement” `skip` is just a symbolic name for `1`. Thus, both statements are always executable, whereas `false` as synonym for `0` is never executable.

Control Flow Like many other programming languages, PROMELA offers the possibility of *case selection* (**if**) and *repetitions* (**do**). However, unlike with typical programming languages, nondeterminism is possible.

The **if**-statement for case selection has the following syntax:

```

if
  :: guard1 -> statements1
  ...
  :: guardn -> statementsn
fi

```

After each **::** a new *branch* of possible control flow is declared. The first statement of each branch is called a *guard* since a branch can be chosen only iff its first statement is executable (or, in case of an expression, if it does not evaluate to zero). The different guards do not have to be mutually exclusive. When several branches have executable guards, one of them is nondeterministically chosen. The order of branches does not matter. If no guard is executable, the whole **if**-statement blocks until at least one guard is executable. For exactly one of the guards the condition **else** may be used. It is executable iff all other guards are not executable. The special separator **->** is just syntactical sugar to highlight the first statement. Syntactically, the guard is no special statement and the **->** could be replaced by the normal statement separator **;**. Actually, the first statement does not have to be a condition at all. This way unconditional nondeterminism can be modeled as in the following example containing a random number generator (between 1 and 3):

```

active proctype RandomNumberGenerator()
{
  int x;

  if
  :: x = 1;
  :: x = 2;
  :: x = 3;
  fi;

  printf("Randomly chose number: %d.\n", x);
  // Output: Randomly chose number: 1.
  //        or: Randomly chose number: 2.
  //        or: Randomly chose number: 3.
}

```

Listing 3.9: Random number generator using unconditional nondeterminism.

For repetitions the **do**-statement is used. Except for the keywords – **do** and **od** instead of **if** and **fi** – the syntax is the same as for the **if**-statement. In each iteration one of the branches with an executable guard is selected. As with **if**, a **do**-statement blocks when no guard is executable. Again, for one guard **else** can be used. A loop can be terminated using **break** anywhere in the branches. The following listing demonstrates how a factorial can iteratively be calculated:

```

active proctype CalculateFactorial()
{
  int x = 5;
  int factorial = 1;
}

```

```

do
  :: x > 0 -> factorial = factorial * x; x = x - 1
  :: x == 1 -> break
od;

printf("5! = %d.\n", factorial);
// Output: 5! = 120.
}

```

Listing 3.10: Iterative calculation of a factorial.

Finally, PROMELA allows to jump to program labels within one proctype using `goto name`. Labels can be attached to statements (*name: statement*) and must be unique within every proctype. `goto` can also be used to leave a loop.

Preprocessing Before SPIN interprets a PROMELA program, a preprocessor is invoked on it. It removes the functional irrelevant multi-line comments (`/* ... */`) and those which go to the end of the line (`// ...`). Using `#define ...` names for constant values, expressions and even parameterised macros can be declared as it is common practice in C. Every later occurrence of the name in the code will be replaced by the defined text, adapting the parameters accordingly, if given. In a similar way `inline`-functions are handled: They look like parameterised functions but indeed every call of them is replaced by their body, replacing the parameter names with the given parameters. Consequently, `inline`-functions must not directly or indirectly call themselves. The syntax is a bit more convenient than that of `#define`-macros, but the latter ones are more powerful as they can return a value (e.g., `#define MAX(a,b) a > b -> a : b`) and thus can be used in expressions as well. Be careful with the usage and declaration of variables inside inline functions and macros as they do not define a new scope.

Omitted Constructs Beyond the basic constructs of PROMELA introduced so far, many advanced ones exist. However, as they turned out to be not used by the translation of SLIM specifications to PROMELA programs, they are not discussed here. Some of those are:

- Parameters, priorities and `provided` clauses (which determine whether a process is active) for proctypes.
- Special statements for asynchronous channels, like testing the number of contained messages (`len`, `empty`, `nempty`, `full`, `nfull`), sorted send (`!!`), random receive (`??`) and polling (`?[...]`). Furthermore, channel assertions telling SPIN that a certain process has exclusive receiving (`xr`) or sending (`xs`) have no beneficial effects for unbuffered channels.
- Special statement `timeout` that becomes executable when the whole system blocks.
- The `unless`-block whose contained statements are executed – but not necessarily repeatedly – until an aborting condition becomes executable.
- Arrays and `typedef` for user-defined records.
- End-state and progress-state labels to mark valid end states (for deadlock checking) or states which are perceived as making progress (for identification of starvation/non-progress loops).

- Semantical details about the position of variable declarations and about the ordering in which processes die.

The interested reader finds all details about these constructs in the literature referenced at the beginning of this section.

3.1.2 Operational Semantics

The operational semantics of PROMELA defines for every PROMELA program a corresponding transition system. The states reflect every possible execution state of the system, that is, all combinations of values for program counters and local variables of every process and the content of global variables and buffered channels. In SPIN the data structure used to store a state is called *state vector* and its size, the memory required to store one state, mainly depends on the number and types of variables and channels used in the specification. In the initial state the program counters point to the first statement in the body of the corresponding processes, the variables are initialised (to 0 if no other value given in the declaration) and the channels are empty. From a state a transition goes out for each statement that is executable in any of the processes targeting at a state where the values of the program counters, variables and channels are adapted according to the effect of the statement. This way every possible interleaving of processes and every alternative in nondeterministic choices is considered. Except for handshaking alias synchronous communication, which involves two processes, a transitions always reflects the execution of exactly one executable statement in exactly one process.

To exemplify how transition systems for PROMELA programs look like, figures 3.1 and 3.2 sketch the transition systems for the PROMELA programs from listings 3.9 and 3.10, respectively. For more details on the operational semantics of PROMELA, see [27, chapter 7]. A formal semantics for a subset of PROMELA, called nanoPROMELA, can be found in [5].

3.1.3 Atomic Execution Blocks

Sometimes it is helpful when for a sequence of statements it can be guaranteed that their execution will never be interrupted and interleaved with the execution of another process, i.e., that their execution is *atomic*. PROMELA allows this with the `d_step { statements }` construct: the enclosed *statements* are executed as an undividable unit. This can be used for an easy implementation of semaphores, e.g., to safeguard mutual exclusive access as used later on in listing 3.13. However, several restrictions apply inside a `d_step`-block:

- Except the first one, none of the statements may block. This is reasonable since the only possibility that such a blocking statement becomes executable would be that another process changes some global variable or channel content. This contradicts the objective of atomicity.
- For the same reason, synchronous communication cannot be used. Sending and receiving with buffered channels is in principle possible, but must be used carefully: as these statements may not block as well, it must be assured that the channel is not full or empty, respectively.
- Jumps into or out of a `d_step`-block using either `goto` or `break` are not allowed.
- The *statements* are not only executed atomically but also deterministically. Thus, nondeterminism should not be used as SPIN will always resolve it by choosing the first executable alternative.

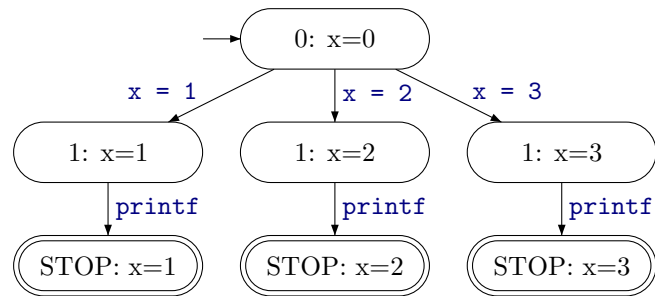


Figure 3.1: Transition system of the random number generator example from listing 3.9

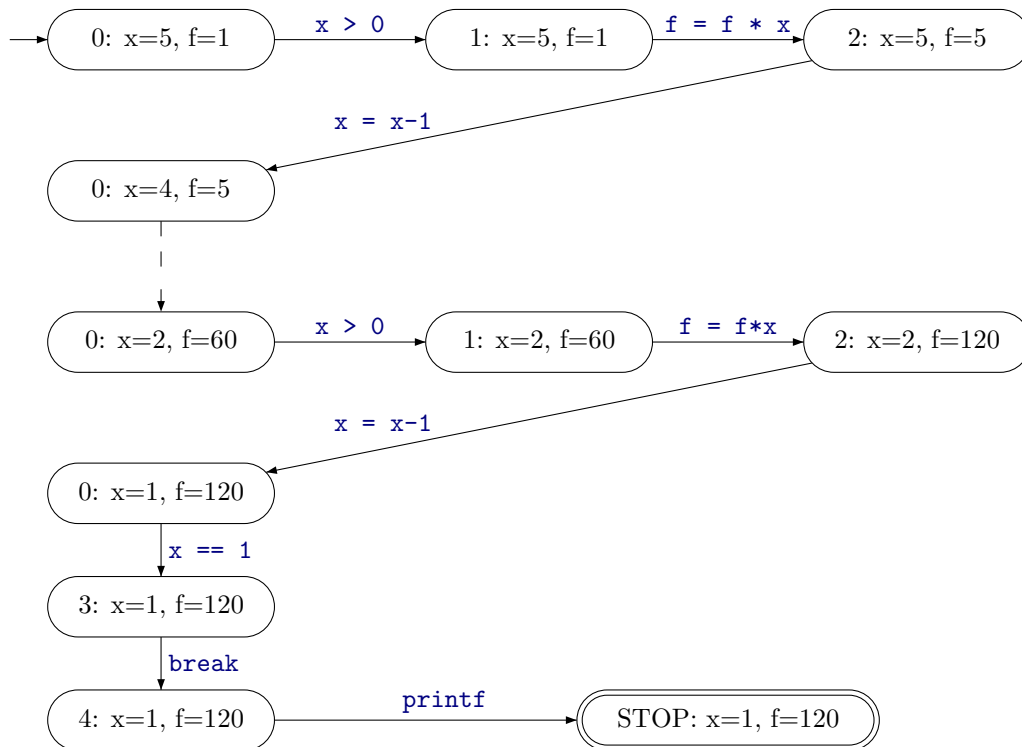


Figure 3.2: Transition system of the factorial example from listing 3.10.

Another advantageous effect of `d_step` is, that the intermediate states resulting from the stepwise execution of the contained statements are not generated. This way, the size of the state space in the resulting transition system and thus the complexity of model checking can be reduced. These missing intermediate states also explain why jumps are not possible. In general it is good practice to place sequences of assignments to variables into a `d_step`-block whenever the intermediate states are not relevant for checking the correctness of the system. Additionally, global variables which are used only locally inside `d_step`-blocks and whose value is irrelevant afterwards can be excluded from the state vector by declaring them as `hidden`, possibly reduce the state space – which results from the combination of every possible content for each variable and channel – further more. This is typically feasible for buffer variables like the one in listing 3.11.

```

hidden int buffer;

active proctype Swap()
{
    int x = 1;
    int y = 2;

    d_step
    {
        buffer = x;
        x = y;
        y = buffer;
    }
}

```

Listing 3.11: Swapping the values of two variables. For state space reduction this is done inside a `d_step`-block and using a `hidden` buffer variable.

Another construct similar to `d_step` exists in PROMELA: `atomic { statements }`. It does not impose the same restrictions as `d_step` but at the same time it is not that strong: atomicity is guaranteed only until the first blocking statement is encountered, then other processes may interleave. Consequently, the intermediate states must be generated and thus there is no beneficial effect on complexity.

3.1.4 Embedded C-Code

Arbitrary C-code can be embedded into PROMELA programs (cf. [27, 25, 28]) and thus more complex features, e.g., floating point variables, can be used. However, SPIN itself cannot handle embedded C-code directly and ignores it in simulations. Only in verification and simulation runs performed by PAN embedded C-code is executed. In fact, it is simply copied into the generated source code for PAN (see below).

Different premetives are used to embed C-code. First of all, a `c_code`-block can contain arbitrary C statements. All C statements inside are executed atomically once the block is reached and thus result semantically in one state transition in the transition system. By definition, a `c_code`-block is always executable. Variables declared in the PROMELA program can be used in the embedded C-code. However, the required notation depends on the scope of the variable: A global variable `x` can be referenced by `now.x`. In contrast, a variable `x` which is locally declared in a proctype called `Name` can be accessed from C-code embedded in that proctype by `PName->x` (note the prefix `P` before `Name`).

Inside `c_code`-blocks new C variables can be declared but they are not stored inside the state vector and thus cannot distinguish between different states. To include C variables into the state vector they must be declared inside a special `c_state` declaration and then accessed like global variables. For the declaration of new data types at the beginning of the resulting C code for PAN, `c_decl` should be used.

As C-code embedded in `c_code` is defined to be always executable it cannot be used for conditions over C variables. Instead, a `c_expr`-block must be used. It is executable (or “true”) iff the contained C-code returns a value different from 0. However, this must be testable without any changes to the current state, thus the C-code inside must be side-effect free. Listing 3.12 gives a simple example of how to use embedded C-code.

```

c_state "float fraction"

active proctype FloatingPointFraction()
{
  int x = 7;

  c_code { now.fraction = PFloatingPointFraction.x / 2.0; };
  // Result: fraction = 3.5

  if
  :: c_expr { now.fraction > 3.0 } -> printf("7/2 > 3\n")
  :: else -> printf("7/2 <= 3\n")
  fi
  // Output: 7/2 > 3
}

```

Listing 3.12: Using embedded C-code in PROMELA.

Finally, some warnings on the usage of embedded C-code in PROMELA programs: The compliance with the restrictions on executability and side-effect freeness as well as the proper declaration of variables are not checked by SPIN. Any error in those parts can lead to malfunction or wrong model checking results. Furthermore, the possibility to use complex data types in embedded C-code is tempting but the growth of state vector size and state space must be carefully considered.

3.2 Model Checking with SPIN

SPIN, the **S**imple **P**ROMELA **I**nterpreter, itself is no model checker but rather allows to simulate PROMELA programs. Simulations can help to understand what a PROMELA program does and they might find some errors. During simulations nondeterminism must be resolved. In a *random simulation* SPIN randomly chooses one alternative whereas in an *interactive simulation* the user is prompted to make a decision. Finally, a simulation can be *guided* by an error trail generated from an error that was found in a verification run. However, simulations do not suffice to verify the correctness of a system specification as they do not systematically explore all possible system states and execution paths. To this end, SPIN can generate C-code for PAN, the **P**rotocol **A**nalyzer, from the PROMELA program. This program, PAN, performs a complete search through the system’s state space (*nested depth-first search*, cf. [30]) for the sake of verification. If an error is found by PAN an *error trail*, describing the execution path to this error, is written to a file.

For verifying the correctness of systems, it is necessary to define which behaviour is desired and which behaviour is incorrect, or: what the possible errors are. Therefore, SPIN offers different possibilities which are explained in the following subsections: assertions, never claims and LTL-formulas.

3.2.1 Assertion-based Verification

The concept of assertions in PROMELA is the same as in other programming languages, like Java or Python. When the execution reaches a statement `assert(expression)`, it is checked whether the condition in *expression* holds. If so, the execution is continued. Otherwise, simulation (SPIN) stops with an error message and verification (PAN) produces an error trail. Listing 3.13 shows how it can be verified that the access of five identical processes to their critical sections is indeed mutual exclusive.

```

bool free = true;
byte inCS = 0;

active [5] proctype ProcessWithCriticalSection()
{
    // Enter Critical Section
    d_step { free; free = false; }
    inCS = inCS + 1;

    // Verify Mutual Exclusive Access
    assert(inCS == 1);

    // Leave Critical Section
    inCS = inCS - 1;
    free = true;
}

```

Listing 3.13: Verifying mutual exclusive access using `assert`.

The property “at any time at most one process is in its critical section” is a so-called *invariant* as its validity can be checked by considering each reachable system state in isolation. Invariant properties can be easily verified by adding a monitoring process to the PROMELA program which contains only the invariant as an assertion, as shown in listing 3.14. This is much easier than adding `asserts` after every statement in the PROMELA program that might affect the validity of the invariance in question. However, for simulations it is in general not guaranteed that the assertion in the monitor process is tested in the “right” moment, that is, when the invariant is violated. But for verifications using PAN this is no problem since every possible execution sequence, including those where the assertion is tested in a state where the invariant is violated, is checked.

```

bool free = true;
byte inCS = 0;
#define mutex inCS <= 1

active [5] proctype ProcessWithCriticalSection()
{

```

```

// Enter Critical Section
d_step { free; free = false; }
inCS = inCS + 1;

// ...

// Leave Critical Section
inCS = inCS - 1;
free = true;
}

active proctype Monitor()
{
  assert(mutex)
}

```

Listing 3.14: Invariant monitor process for mutual exclusive access.

3.2.2 Never Claims

The verification of a system can be seen as a game between two players observing the system behaviour: one player tries to prove the system’s correctness, the other tries to find a counterexample. For SPIN, this kind of a falsifier is called a *never claim*. It is an additional piece of PROMELA code that is executed intertwined with the PROMELA program and tries to prove the existence of an incorrect execution path. The basic rule is: If the never claim terminates before every reachable system state has been visited, it wins. Otherwise – especially when the never claim blocks – it loses, i.e., the system is correct. This suffices to check the validity of *safety properties*. They claim that “never something bad happens” and thus finite execution fragments which exhibit the unwanted behaviour suffice as counterexamples. In general, the “bad behaviour” can be temporally distributed across several states, like in “never B occurs after A happened”. *Invariants* are a special type of safety properties since for them the “bad behaviour” reduces to the reachability of “bad states”. Listing 3.15 shows the never claim for an invariant where `badstate` is a predicate defined in the PROMELA program which can identify bad states, e.g., `#define badstate inCS > 1`. The definition of boolean predicates is necessary since never claims cannot read the values of non-boolean variables from the PROMELA program.

```

never
{
  search:
    if
      :: badstate -> goto done
      :: true -> goto search
    fi;

  done:
    skip
}

```

Listing 3.15: Never claim for invariants.

The never claim can continue searching forever due to the `true`-branch. But, as soon as a bad state is visited, the never claim can terminate by jumping to the last dummy statement (necessary, since there is no default “end” label) and thus wins. For finding errors it indeed suffices that the never claim *can* win since every possible intertwined execution of PROMELA program and never claim is considered in verification runs. The expression `true` could be replaced by `!badstate` but that is not necessary!

For *liveness properties*, which say that “eventually something good happens”, the winning condition of first termination does not suffice. The reason for that is that counterexamples for liveness properties must be *infinite* executions on which never anything good happens. As the state space induced by PROMELA programs is finite, infinite executions must contain cycles. Due to this fact an additional winning condition is based on checking for cycles: The never claim wins as well for executions for which it infinitely often executes a statement whose label starts with `accept`. This is called an *acceptance cycle*. Listing 3.15 shows a correspondingly constructed never claim where `good` is again a predicate defined in the PROMELA program. As soon as something good happens, the never claim blocks. But, when an infinite execution is found where never something good happens, infinitely often `accept_search` is visited by the never claim and thus it wins.

```

never
{
  accept_search:
    if
      :: !good -> accept_search
    fi;
}

```

Listing 3.16: Never claim for a liveness property.

One detail concerning counterexamples for liveness properties must be considered: PROMELA allows to describe systems containing finite executions. When nothing good happens on those finite executions they should be perceived as a counterexample as well. However, they obviously cannot contain an acceptance cycle since they are finite. To solve this problem, the finite execution is extended to an infinite one by *stuttering*: the last state of the execution is repeated ad infinitum. The never claim must be constructed in a way that it can handle this artificial cycling in the last state, i.e., it must make sure that an `accept`-label is seen infinitely often.

3.2.3 LTL Model Checking

A more convenient way to define desired properties of a system is to specify them in *Linear Temporal Logic* (LTL). The automata theoretical approach, developed by Vardi and Wolper in [41], to check whether a system, given as a transition system model TS , satisfies a property, given as an LTL-formula φ , is to transform the negation of the property into an Büchi automaton $\mathcal{A}_{\neg\varphi}$, construct the synchronous product $TS \otimes \mathcal{A}_{\neg\varphi}$ and check whether its ω -language is empty, i.e., that no reachable accepting state lies on a cycle. SPIN does something similar: A given LTL-formula¹ is translated to a never claim, which is an implementation of the corresponding Büchi automaton, by a method described in [20]. Then PAN is generated to execute the PROMELA program and the never claim in parallel, like the synchronous product. However, PAN does model checking *on-the-fly*, that is, it does not generate the whole state space before the search but step by step during the search.

¹The LTL-operator Next is not allowed since it might destroy stutter-invariance of properties.

An important issue for model checking of distributed systems without any knowledge about the relative speeds of processes is *fairness*: Fairness assumptions basically say that none of the processes dominates and thereby they exclude the unrealistic but theoretically possible behaviour that among two processes which could execute a statement always the same process is chosen and never the other. By default, SPIN does not take fairness assumptions into consideration but it supports weak fairness.

3.3 Advanced Techniques to Improve Efficiency of Model Checking

For the applicability of model checking the two critical key figures are *time* and – even more important since it is a limited hardware resource – *memory* consumption. During a complete verification run, SPIN (precisely, the generated PAN) generates all states reachable from the initial state in a depth first search (cf. [30, 20]). Whenever a successor state is explored, SPIN checks whether it has been visited already and if so, it backtracks. This recognition obviously requires that every visited state is stored. For a transition system with $|S|$ many states and a state vector size, that is, the memory that is required to store all information about one single state, of $|StateVector|$, the memory requirement for a complete search is $|S| * |StateVector|$. To reduce the time which is needed for checking whether a state s has already been visited, PAN stores the states in a *hash table*. Then the state s has to be compared only to those states s' with $\text{hash}(s) = \text{hash}(s')$.

To increase the efficiency of model checking, the state space and the state vector size should be as small as possible. Besides a careful development of the model and its PROMELA implementation (e.g., ignoring as much irrelevant aspects as possible, choosing the smallest possible data types and channel capacities, etc.), some optimisations can be performed by SPIN itself (cf. [27, chapter 9]). However, in general there is a trade-off between a reduction of memory demands and an increase in run time.

3.3.1 Reduction of the State Space

The operational semantics of PROMELA states that every possible interleaving of processes is to be considered – with the consequence that many different execution paths exist in the resulting transition system. However, many of those paths have in principle the same effect since the order of so-called independent operations is irrelevant. For example, consider two processes P_1 and P_2 with an assignment to x and y , respectively. When the assignments do not read the other variable and when both variables are not compared in the property which is to be model checked then there is no data dependence between the variables. Hence, it is irrelevant which process performs its assignment first, the operations are independent. SPIN's algorithm for *partial order reduction* [29] tries to identify executions which differ only in the order of independent operations and to eliminate all but one of them.

The same idea is the basis of *statement merging* but this method is limited to local optimisations inside one proctype. In principle it adds `d_step`-blocks around sequences of statements that only affect the behaviour of that proctype, e.g., a sequence of assignments to local variables.

Both methods share the need to identify which operations are independent or local, respectively. Some of the more advanced language constructs (e.g., `provided` clauses and remote references) hinder this and are thus incompatible with those methods. Furthermore, the considered properties must be *stutter-invariant* and to guarantee that, the Next operator is not allowed in LTL-formulas. As long as none of these incompatible elements is used, both methods

are proven (cf. [15, 17]) to preserve model checking results and are thus enabled in SPIN by default.

3.3.2 Reduction of the State Vector Size

One approach is to compress the information stored in the state vector. To this end, typical subsequences in the state vector that do not highly correlate with other parts are identified. These subsequences are stored once together with a unique identification number which then replaces every occurrence of that sequence in any state vector. The sequences which contain the values of local variables for each process are typically low correlated. By sharing the stored sequences between state vectors *collapse compression* reduces the overall amount of required memory.

Another approach is to use a symbolic method for the storage of states, like OBDDs (*ordered binary decision diagrams*) which are used for symbolic model checking [33]. In SPIN a *minimized automaton* [31] can be used to represent the set of state vectors encoding states that have already been visited. A state vector is just a finite sequence of bytes, in other terms: a word over an alphabet with 256 different symbols. Thus, a (finite) set of state vectors is nothing else than a regular language which can be described by a unique minimal finite state automaton. All state vectors which lead from the initial to the accept state in this automaton belong to states which have already been visited, the others were not. Whenever PAN explores a new state that was not visited before it extends the automaton and minimizes it again so that the corresponding state vector will be accepted. The minimized automaton can exponentially reduce the memory requirements of model checking but unfortunately leads to an considerable increase of run-time as well.

Both methods presented for the reduction of the state vector size so far are *lossless*, that is, all information describing a state is exactly stored. Hence, the results of model checking stay unchanged. It even makes sense to combine both methods. However, due to the memory vs. run-time trade-off SPIN must be explicitly told to make use of them. Additionally, SPIN allows to use *lossy* reduction methods which store only partial information about the visited states. This can tremendously reduce the memory requirements without an increase of run-time. Though, a complete search of the state space cannot be guaranteed any more since a state that has not yet been visited but which has the same representation as an already visited state will not be explored. Worse, all successor states of it are possibly not visited as well when no other branch of the depth first search explores them. As a consequence, lossy methods do not preserve the model checking results but are rather a proof approximation: If a counterexample is found it will be correct (no false negatives), but if no counterexample is found nothing can be said about the validity of the property (false positives are possible).

As mentioned above, PAN stores the state vector of a state s in a hash table at slot $\text{hash}(s)$. As long as no hash collisions occur, it suffices to store one bit for every possible hash value which indicates whether a state hashing to it has been visited. This is what *Bitstate Hashing* [24], sometimes called *Supertrace*, does. However, as soon as two states s and s' are explored with $\text{hash}(s) = \text{hash}(s')$, that is, a hash collision occurs, this method becomes lossy. By using different hashing functions at the same time the possibility of collisions in all of them can be reduced. Another approach using hashing is *Hash Compact* [45]: It does not store bits for each possible hash value but only the hash values of visited states.

Part II

Model Checking SLIM Specifications with SPIN

Chapter 4

Translation to PROMELA Programs

Starting from the `root` component every control component instance $c \in CCmp$ is translated to exactly one `proctype` implementing the component's behaviour. By declaring it `active`, one process will be automatically instantiated from it by SPIN. The values of data elements are stored in global variables. Additionally, one globally declared unbuffered channel per component is used for synchronous communication between the component and its supercomponent. The different modes $m \in Mod(c)$ of a component are represented by program labels `mode_m`, each followed by a `do`-loop which implements all actions, such as taking a transition, and reactions, like updating outgoing data ports on receiving new values for the incoming data ports, that are possible in mode m . When nothing can or is to be done, the `do`-loop blocks, always ready to react on changes.

4.1 Proctypes for Components

For a control component $c \in CCmp$ the code in listing 4.1, later referred to as *Proctype(c)*, must be generated. The details, indicated by *placeholders* parameterised with component c and typically one of its modes $m \in Mod(c)$, are given in the following subsections.

To distinguish between variables and proctypes generated for different component instances $c \in CCmp$, their names contain the access path to c . Since the dot (`.`), used to separate identifiers in access paths, is not allowed in PROMELA identifiers, it is replaced by an underscore (`_`). However, this requires that underscores are eliminated in advance from all subcomponent identifiers occurring in the specification, that is, in the SLIM model and in the properties. The *underscore access path* of a component $c \in CCmp$ is in the following denoted as *UAccessPath(c)*.

For the proctype generated for a control component c it does not suffice to take simply *UAccessPath(c)* as its name since SPIN can handle only proctype names no longer than 55 characters. To guarantee unique names, every proctype is prefixed with a unique number followed by the underscore access path, which just serves readability and understandability of the generated code. The whole resulting name, denoted *ProctypeName(c)*, is then truncated to length 55.

```

GlobalVariables(c)

active proctype ProctypeName(c)
{
    Initialisation(c)

    For every mode  $m \in Mod(c)$ :
    mode_m:
        do
            ProactiveTransitions(c, m)
            MultwayEventCommunication(c, m)
            DataPortUpdates(c, m)
            ActivationHandling(c, m)
        od;
    }
}

```

Listing 4.1: Structure of the code generated by *Proctype(c)* for one control component $c \in CCmp$.

4.2 Global Variables for Components

The values of the component's data elements, that is, of incoming and outgoing data ports declared in the component type $typ(c)$ as well as of data subcomponents declared in the component implementation $imp(c)$, are stored in global variables. Since other components can have data elements with the same names ($Dat(c) \cap Dat(c') \neq \emptyset$ is possible for different $c, c' \in CCmp$), it is necessary to include the access path to the component which they belong to in the variable name. Furthermore, a prefix distinguishes between the different kinds of data elements: **idp**, **odp** and **dsc** for incoming data ports, outgoing data ports and data subcomponents, respectively. This helps to understand the resulting source code and its connection to the SLIM specification, but as $IDPrT(c)$, $ODPrT(c)$ and $DSub(c)$ are pairwise disjoint this distinction is not necessary. However, the difference in the prefixes **Store** and **New** is crucial since some data elements need a second variable for buffering new values. Buffering of outgoing data ports is needed to allow the fixpoint iteration used for updating data ports to compare new (buffered) values of an outgoing data port to its old (stored) one (cf. section 4.6). A data subcomponent $dsc \in DSub(c)$ needs to be buffered only when a transition effect exists that assigns to dsc before it is read the last time. The predicate $MustBeBuffered(c, dsc)$, as defined in section 4.4.4, provides the information whether this is the case. For simplification of following definitions, a further predicate indicates whether a buffer variable for a data element $d \in Dat(c)$ exists:

$$\begin{aligned}
 IsBuffered(c, d) := & d \in ODPrT(c) \vee \\
 & (d \in DSub(c) \wedge MustBeBuffered(c, d))
 \end{aligned}$$

Table 4.1 summarises the different kinds of global variables and the names used for them. Furthermore, it defines the placeholders $DataStoreVar(c, d)$ and $DataBufferVar(c, d)$ which stand for the name of the storage and buffer variable, respectively, corresponding to the data element $d \in Dat(c)$ in a component $c \in CCmp$.

During the generation of the PROMELA code for a control component $c \in CCmp$ not only the variables for data elements of that component are used but also those for incoming and

Data Element	<i>DataStoreVar(c, d)</i>	<i>DataBufferVar(c, d)</i>
Incoming Data Port	<i>idpStore_UAccessPath(c)_d</i>	-
Outgoing Data Port	<i>odpStore_UAccessPath(c)_d</i>	<i>odpNew_UAccessPath(c)_d</i> (exists always)
Data Subcomponent	<i>dscStore_UAccessPath(c)_d</i>	<i>dscNew_UAccessPath(c)_d</i> (iff <i>MustBeBuffered(c, d)</i>)

Table 4.1: Names of global storage and buffer variables for data elements $d \in \text{Dat}(c)$ of control components $c \in \text{CCmp}$.

outgoing data ports $dp \in \text{DPrt}(c.\text{csc}) \subset \text{Dat}(c.\text{csc})$ of its control subcomponents $\text{csc} \in \text{CSub}(c)$. To avoid additional case distinctions in the presented code the following notation is introduced for convenience:

$$\begin{aligned} \text{DataStoreVar}(c, \text{csc}.dp) &:= \text{DataStoreVar}(c.\text{csc}, dp) \\ \text{DataBufferVar}(c, \text{csc}.dp) &:= \text{DataBufferVar}(c.\text{csc}, dp) \end{aligned}$$

In the declaration of a variable not only its name but also its type must be given. To this end, SLIM data types must be matched to PROMELA data types which are at least as expressive as the SLIM ones. However, to keep memory demands as low as possible, the expressivity should be as close as possible to the SLIM data types. In the following, *PromelaType*($\text{typ}(c, d)$) stands for the PROMELA type corresponding to the SLIM type $\text{typ}(c, d)$ of a data element $d \in \text{Dat}(c)$ as stated in table 4.2. Literals used in enumeration data types are mapped to integer values. The exact integer data type used depends on the required expressivity, i.e., the number of different literals, as described in subsection 4.2.1. To abstract from this detail the symbol `enumIntType` is defined with the concrete type. Note that SLIM-`reals` are indeed translated to PROMELA-`ints` – the reason for that is discussed in subsection 4.2.2. Besides, note that SLIM’s data types `clock` and `continuous` could be translated like `reals` but since timed behaviour is not supported by the PROMELA translation, this is not done.

SLIM data type	PROMELA data type
<code>bool</code>	<code>bool</code>
<code>enum(id₁ , . . . , id_n)</code>	<code>enumIntType</code>
<code>int</code>	<code>int</code>
<code>real</code>	<code>int</code>
<code>[l..u]</code>	<code>int</code>
<code>clock</code>	-
<code>continuous</code>	-

Table 4.2: Mapping of SLIM data types to PROMELA types.

In addition to the variables for the data elements another global variable

$$\text{ModeVar}(c) := \text{nominalMode_UAccessPath}(c)$$

is used to store the current mode of the component. This is not necessary for the correct operation of the proctype, since that is based on the mode labels, but is needed later to allow reasoning about modes in properties (cf. section 5.1). Mode identifiers are represented using integers in the same way as it is done for enumeration data types but instead of `enumIntType` the self-defined type `modeIntType` is used.

Finally, for the synchronous communication between a control component $c \in CCmp$ and its supercomponent a global unbuffered channel variable

$$\mathit{Channel}(c) := \mathit{channel_UAccessPath}(c),$$

is declared. The messages contain two values: Firstly, one `mtype` describing the conveyed message itself and secondly an `int` for additional parameters, if needed. At the beginning of the PROMELA program (cf. section 4.9), the following message types are defined:

```
mtype = { update, invalidate, trymultiway,
          activate, deactivate, reset };
```

All together, for a component $c \in CCmp$, $\mathit{GlobalVariables}(c)$ generates the PROMELA variable declarations shown in listing 4.2.

```
chan  $\mathit{Channel}(c)$  = [0] of { mtype, int };
modeIntType  $\mathit{ModeVar}(c)$ ;
```

```
For every data element  $d \in \mathit{Dat}(c)$ :
 $\mathit{PromelaType}(\mathit{typ}(c, d))$   $\mathit{DataStoreVar}(c, d)$ ;
If  $\mathit{IsBuffered}(c, d)$  add:
 $\mathit{PromelaType}(\mathit{typ}(c, d))$   $\mathit{DataBufferVar}(c, d)$ ;
```

Listing 4.2: Variable declarations generated by $\mathit{GlobalVariables}(c)$.

4.2.1 Integer Constants for Symbolic Identifiers

Symbolic names occurring in the specification – such as identifiers used in enumeration data types, for modes and for event ports – are handled in the PROMELA code by mapping them to unique integer identifiers. To increase readability of the generated PROMELA code, symbols consisting of a prefix and the original symbolic name – `enum.id` for $id \in \mathit{Enums}$ and `modeID.m` for $m \in \mathit{Mod}$ – are defined using `#define` in the beginning of the program (cf. section 4.9).

In general, to allocate as little memory as possible for the representation of symbolic names collected in a set S , the concrete integer data type used depends on the cardinality $|S|$, i.e., the number of different names. For a cardinality $card \in \mathbb{N}_0$ the function $\mathit{IntType}(card)$ returns the appropriate PROMELA integer data type. The 32-bit `int` is the largest integer data type directly supported by SPIN. Since an upper bound of $2^{32} = 4,294,967,296$ different symbolic names is expected to suffice for most of the realistic SLIM specifications, no additional effort – e.g., combining several integer values – is made in the translation to extend it.

$$\mathit{IntType}(card) := \begin{cases} \mathit{bit} & , 0 \leq card \leq 2 = 2^1 \\ \mathit{byte} & , 3 \leq card \leq 256 = 2^8 \\ \mathit{short} & , 257 \leq card \leq 65,536 = 2^{16} \\ \mathit{int} & , 2^{16} + 1 \leq card \leq 2^{32} \\ \mathit{error} & , \text{otherwise} \end{cases}$$

For the mapping of elements from S to an integer value an arbitrary but fixed ordering – without loss of generality for the implementation the lexicographic order is chosen for event

ports while enumeration and mode identifiers are in the order of their first occurrence in the specification – of S is assumed so that every element $e \in S$ has a position $0 \leq \text{pos}(e, S) < |S|$. Basically, an element $e \in S$ is represented by the integer value $\text{pos}(e, S)$ resulting in a sequence $0, 1, 2, \dots$. However, with the PROMELA data types `short` and `int` negative values must be used when the largest positive number that is representable with the respective data type is reached. The function `MaxPos` returns this value for a set S by employing `IntType(|S|)` to differentiate between the integer data types.

$$\text{MaxPos}(S) := \begin{cases} 1 & , \text{IntType}(|S|) = \text{bit} \\ 255 & , \text{IntType}(|S|) = \text{byte} \\ 32,767 & , \text{IntType}(|S|) = \text{short} \\ 2^{31} - 1 & , \text{IntType}(|S|) = \text{int} \\ \text{error} & , \text{otherwise} \end{cases}$$

With those means, the function `IntValue`, mapping an element $e \in S$ to its integer representative, can be formally defined:

$$\text{IntValue}(e, S) := \begin{cases} \text{pos}(e, S) & , 0 \leq \text{pos}(e, S) \leq \text{MaxPos}(S) \neq \text{error} \\ -(\text{pos}(e, S) - \text{MaxPos}(S)) & , \text{error} \neq \text{MaxPos}(S) < \text{pos}(e, S) \\ \text{error} & , \text{otherwise} \end{cases}$$

The translation always distinguishes between enumeration identifiers and modes, i.e., there are different mappings. Consequently, different identifiers can have the same integer representation and vice versa, e.g., `enum_abc` is 1 but `modeID_abc` is 3. This explains – besides avoiding name clashes with variables and labels – why prefixing is necessary. The advantage of keeping the mappings separate is that in general the memory requirements can be reduced: Representing two modes and two different enumeration identifiers can be done with one `bit` each. Combining them results in four different names which requires a `byte`.

For event ports two specifics apply as far as the concrete implementation is concerned: First of all, no symbols are defined for them but the integer value is directly used in the messages (cf. section 4.5). Secondly, always the type `int` is used. This does not unnecessarily require memory in the state vector since integer representatives of events are never stored but only used as parameters in synchronous communication. The mapping of $event \in EPrt$ to a unique integer identifier is formally defined by:

$$\text{EventNumber}(event) := \begin{cases} \text{pos}(event, EPrt) & , 0 \leq \text{pos}(event, EPrt) \leq 2^{31} - 1 \\ -(\text{pos}(event, EPrt) - 2^{31} + 1) & , 2^{31} - 1 < \text{pos}(event, EPrt) \leq 2^{32} \\ \text{error} & , \text{otherwise} \end{cases}$$

4.2.2 Handling of Reals by Embedded C-Code

Since SPIN/PROMELA themselves do not support floating point values, data elements $d \in Dat(c)$ of type `real` are realised by embedding C-code into the PROMELA program using the language constructs described in section 3.1.4. However, instead of declaring the variables for such data elements as C-`floats` (4 byte) using the `c_state` primitive, a normal PROMELA variable of type `int` (4 byte) is declared. By using a typecasting trick, `float` values can be written to and read from such a variable. For a global PROMELA-`int` variable x this looks as follows inside the embedded C-code: `*((float *) &(now.x))`.

Note that typecasting on the pointer level is indeed necessary – explicit typecasting on the value level does not yield the desired result as it changes the value! Yet, why is this additional effort made instead of directly using C-floats? The advantage of declaring the variables as normal PROMELA ones is, that this allows to check for equality and to copy values of SLIM-reals as well as resetting them to 0 using normal PROMELA code, as it is done in the translation of transition effects (cf. section 4.4.4) and of data port updates (cf. section 4.6), both involving buffering. This is possible, since for equality checking and copying only the sequence of bits stored in a variable matters – but not their interpretation. Furthermore, the representation of 0 is identical for ints and floats: every bit is 0.

Floating point constants in C-code are by default interpreted as doubles (8 byte) which causes trouble when assigning them to a float variable (4 byte). Thus, for constants the desired type must explicitly stated by adding the suffix f, e.g., 3.5 becomes 3.5f. For integer constants – which are allowed as constants for SLIM-reals – this does not work. As a workaround for this case, the suffix .0f is added, e.g., 3 becomes 3.0f.

4.2.3 Translation of Assignments

Assignments $d := expr$ from a SLIM specification, e.g., transition effects and flows, can – except for assignments to real variables – simply be translated to PROMELA assignments: The left hand side of the assignment, identifying a data element $d \in Dat(c)$, is translated to the corresponding variable. The details of translating the expression $expr$ on the right hand side of the assignment are given in the following subsection.

During the translation of a control component $c \in CCmp$ to PROMELA, assignments to incoming and outgoing data ports of c , to data subcomponents of c and to incoming data ports of a control subcomponent $csc \in CSub(c)$ of c occur, that is, to a data element $d \in Dat(c) \cup_{csc \in CSub(c)} IDPrt(c.csc)$. In general, two kinds of assignments must be distinguished: *unbuffered* ones, writing to a storage variable, and *buffered* ones, writing to a buffer variable. Therefore, the following two placeholders are defined:

$$\begin{aligned} UnbufferedAssignment(c, d, expr) &:= DataStoreVar(c, d) = Expression(c, expr); \\ BufferedAssignment(c, d, expr) &:= DataBufferVar(c, d) = Expression(c, expr); \end{aligned}$$

Two special situations must be handled in a slightly different way: First, on assigning an integer value x to an data element d of the integer range data type $[l..u]$, the SLIM semantics requires a modulo calculation to adapt the value so that it fits into the given bounds: d is assigned the value of the expression $((x - l) \bmod (u - l + 1)) + l$. Second, assignments to data elements of type real are done inside a c_code-block, applying the typecasting trick introduced in the previous subsection and considering the different way of accessing variables from embedded C-code. For example, the buffered assignment doubling the value of a real data element d in a component c is realised as follows:

```
c_code {
    *( (float *) &(now.DataBufferVar(c,d)) ) =
        2.0f * (*( (float *) &(now.DataStoreVar(c,d)) ))
}
```

4.2.4 Translation of Expressions

For the translation of SLIM expressions $expr$, occurring in the context of a control component $c \in CCmp$, to PROMELA, the placeholder $Expression(c, expr)$ is used. Instead of giving a rather involved formal definition of this placeholder, this subsection textually describes all important aspects.

Translating an expression means in the first place adapting operators. Most of the SLIM operators do exist in PROMELA and C as well, some with a slightly different notation. However, for the boolean operators **iff**, **xnor**, **imp** and **xor** no adequate counterparts exist. Hence, they are translated to their defining boolean expression containing only the basic boolean operators. The details of translating operators are given in table 4.3.

SLIM	PROMELA/ C
+	+
-	-
*	*
/	/
mod	%
=	==
!=	!=
<	<
>	>
<=	<=
>=	>=
not	!
and	&&
or	
$x \text{ iff } y$	$(x \ \&\& \ y) \ \ (!x \ \&\& \ !y)$
$x \text{ xnor } y$	$(x \ \&\& \ y) \ \ (!x \ \&\& \ !y)$
$x \text{ imp } y$	$(y \ \ !x)$
$x \text{ xor } y$	$(x \ \&\& \ !y) \ \ (!x \ \&\& \ y)$
"case"	"->" / "?"

Table 4.3: Mapping of SLIM operators to PROMELA and C operators.

A different approach is necessary for the special SLIM operator **case**. The expression

```
case  $b_1:e_1; b_2:e_2; \dots; b_n:e_n$  otherwise  $e_0$  end
```

returns the value of expression e_i for the first, i.e., the smallest, $1 \leq i \leq n$ for which b_i is **true** – or of e_0 if no such i exists. It is translated to PROMELA using nested conditional expressions maintaining the correct order:

$$b_1 \rightarrow e_1 : (b_2 \rightarrow e_2 : (\dots (b_n \rightarrow e_n : e_0) \dots))$$

For a translation to C, \rightarrow simply has to be replaced by $?$.

Another issue in translating expressions is the handling of identifiers and constants occurring in it. Identifiers of data elements $id \in Dat(c)$ are transformed to reading accesses to the corresponding storage variable $DataStoreVar(c, id)$, whereas identifiers of enumeration data types $id \in Enums(c)$ are translated to the symbol `enum_id` which was defined for it. Which case applies

for an identifier is always determined since the syntactic restrictions D-5 and F-14 (cf. [37]) guarantee that the identifiers for data elements and the identifiers used in enumeration data types are disjoint within each component. Integer constants can directly be reused but for real constants a suffix as described in section 4.2.2 must be added.

If (a part of) an expression is translated to embedded C-code, any variable access must be prefixed with `now.` as stated in section 3.1.4 introducing embedded C-code. An expression on the right hand side of an assignment to a data element of type `real` must obviously be translated to embedded C-code. The translation of assignments takes care of this using `c_code` as seen in the previous subsection. However, even if no assignment to a `real` is considered, in particular when guards are checked, subparts of the expression might still require to be evaluated in embedded C-code. For example, a control component c might set its outgoing data port `isWorking` $\in ODPrt(c)$ of type `bool` depending on the power supply, modeled as an incoming data port `power` $\in IDPrt(c)$ of type `real`, and the status of a power switch, modeled as a data subcomponent `switch` $\in DSub(c)$ of type `bool`, by using a flow expression like `isWorking := power >= 3.5 and switch`. Instead of translating the whole assignment to embedded C-code, only the boolean subexpressions using `reals` are evaluated in embedded C-code using `c_expr`. For the mentioned example this yields:

```
DataBufferVar(c, isWorking) :=
  c_expr { now.DataStoreVar(c, power) >= 3.5f }
  && DataStoreVar(c, switch)
```

Finally, one rather technical remark: A transition guard g is nothing else than an expressions and thus is translated using `Expression(c, g)` (cf. section 4.4). However, a guard might be empty, restricting the enabledness of a transition not at all. In that case, `Expression(c, g)` simply returns `true`.

4.3 Initialisation

The variables storing the data elements must be initialised to their default values. If no default value is given, SPIN's default value is `0`. However, the initialisation does not take place when the process is instantiated, i.e., in the initial state of the PROMELA program. Instead it is postponed until the component is activated for the first time. Recall, that the activation of subcomponents can depend on the mode of their supercomponent. In order to activate a subcomponent, the supercomponent sends the message `activate` to the corresponding process (cf. section 4.7 on activation handling).

Upon receiving the activation message, the data elements are initialised. To reduce the state space, all these assignments are subsumed in one `d_step`-block. For the same reason the assignment to the mode variable is already included in this block as well. For outgoing data ports are not only the unbuffered variables initialised but also the buffered variables to cut short the first fixpoint iteration. Since data subcomponents are reset on reactivation anyway, only those variables corresponding to data subcomponents active in the component's start mode are initialised. As data ports are always active, `Dat(c, stm(c))` contains exactly the data elements which are considered during initialisation. The buffer variables for data subcomponents, unlike those for outgoing data ports, are not initialised since they are needed only locally inside `d_step`-blocks, realising transition effects, for buffering assignments of new values to data subcomponents whose old values are read somewhen later in the same effect (cf. `MustBeBuffered` as defined in section 4.4.4).

On activation of a component, not only its data elements must be initialised, but its control subcomponents which are active in the component's starting mode, i.e., `csc` $\in CSub(c, stm(c))$,

must be activated as well. This is done by sending `activate` messages to the corresponding processes. To guarantee that the subcomponent is activated an acknowledgment is awaited. This acknowledgment is again an `activate` message along the same channel but this time the roles of sender and receiver have changed. Due to the fact that the channel is unbuffered, i.e., communication is synchronous, the process cannot receive its own message as acknowledgment.

Finally, in the same manner as the control subcomponents do, an acknowledgment that the activation has been completed is sent to the supercomponent just before the process jumps to the mode label of its starting mode $stm(c)$.

Overall, $Initialisation(c)$ generates the code shown in listing 4.3. The whole initialisation code is prefixed with the label `FirstActivationOrReactivationWithoutModeHistory` which allows its reuse when a component is reset or when a component without mode history (starting mode is of type `activation`, not `initial`) is reactivated, cf. section 4.7 on activation handling. To ensure that no mode label with the same name exists, mode labels were prefixed by `mode_`.

`FirstActivationOrReactivationWithoutModeHistory:`

```

// Wait for activation
Channel(c)?activate, -;

// Initialise data elements active in start mode
d_step {
  For every  $d \in Dat(c, stm(c))$  with  $dfl(c, d) \neq \perp$ :
  [ UnbufferedAssignment(c, d, dfl(c, d));
    If  $d \in ODPrt(c)$ :
    [ BufferAssignment(c, d, dfl(c, d));
  ]
  ModeVar(c) = modeID_stm(c);
}

// Activate control subcomponents active in start mode
For  $csc \in CSub(c, stm(c))$ :
[ Channel(csc)!activate, 0;
  Channel(csc)?activate, -;
]

// Send activation acknowledgment
Channel(c)!activate, 0;

// Jump to start mode label
goto mode_stm(c);

```

Listing 4.3: Code generated by $Initialisation(c)$.

4.4 Proactive Transitions

At a first glance, the translation of transitions seems rather straight forward: Each transition starting from a mode $m \in Mod(c)$ becomes a branch in the `do`-loop under the mode label `mode_m`. Only transitions whose guard evaluates to true can be taken. This is realised by using

the guard expression as the first statement in each branch. That yields the desired effect since the executability of the first statement determines in PROMELA whether the branch can be chosen. The rest of the branch consists of assignments according to the transition's effects and finally a jump to the mode label of the transition's target mode. Nondeterminism in the transition relation does not require any extra effort since the `do`-loop in PROMELA is nondeterministic as well. However, the SLIM semantics contains some less obvious demands for adaptation in the course of taking a transition:

- Data subcomponents which are reactivated by the transition, i.e., they were not active in the source mode but are active in the target mode, have to be reset to their default value.
- Data ports which were targets of data port connections or flows in the source mode but are not in the target mode have to be reset to their default value as well.
- Assignments to data elements must be buffered whenever the data element occurs on the right hand side of a subsequent assignment in the same transition effect.
- The activation status of control subcomponents has to be adapted with respect to the target mode of the transition.

Additionally, the trigger of the transition must be considered as well. Three types of transitions can be distinguished by their trigger:

- No trigger, formally τ . Whether they can be taken depends only on their guard and thus we call them *proactive transitions*.
- An own outgoing event port or an incoming event port of a control subcomponent as trigger. Those transitions can be taken only when at least one other component can receive this event by synchronously taking a transition. Due to the fact that these transitions somehow start a request for multiway event communication we refer to them as *master transitions*.
- An own incoming event port or an outgoing event port of a control subcomponent as trigger. Those transitions can be taken only when the trigger is received from an other component. They are the counterpart of master transitions by reacting to the request for multiway event communication and are thus called *reactive transitions*.

All three types of transitions share the principle steps which are described in this section on proactive transitions. In the next section on multiway event communication the extended implementation details for master and reactive transitions are described.

Finally, one technical problem has to be solved: The semantics of SLIM transitions is – except for handshaking in multiway event communication – atomic, that is, two components cannot take a transition independent from each other at the same time. To this end, the macros `nt(guard)` and `ft`, as described in section 4.4.1, are used.

Before all implementation details are given, listing 4.4 provides an overview of the code generated by *ProactiveTransitions(c, m)*. To avoid the generation of intermediate states all assignments to variables are bundled in one `d_step`-block. The release of the semaphore by calling `ft` is not missing but hidden by sending the `invalidate` message to the supercomponent in order to notify it about possible changes in the values of the component's outgoing data ports. The details on that are described in section 4.6 on data port updates.

```

For  $(m_s, t, g, f, m_t) \in MTr$  with  $m_s = m$  and  $t = \tau$ :
:: nT(Expression(c, g)) ->
  d_step {
    ResetReactivatedDataSubcomponents(c, m_s, m_t)
    ResetDisconnectedDataPorts(c, m_s, m_t)
    Effects(c, f)
    ModeVar(c) = modeID_m_t;
  }
  AdjustActivationOfControlSubcomponents(c, m_s, m_t)
  Channel(c)!invalidate, 0;
  goto mode_m_t;

```

Listing 4.4: Code generated by *ProactiveTransitions(c, m)*.

4.4.1 Ensuring Atomicity of Transitions

A simple way to guarantee the atomicity of SLIM-transitions in the PROMELA translation would be to include all statements for taking a transition into one `d_step` or `atomic`-block. Unfortunately, `d_step` is too strong because it forbids synchronous communication and jumps while `atomic` is not strong enough since atomicity is not guaranteed once a blocking statement, here again the synchronous communication, is found. Alternatively, a semaphore, implemented as atomic test and set of the global variable `gflagGoNextTransition`, is used (cf. mutual exclusion as discussed in section 3.2.1). Requesting the semaphore must be the first statement in every `do`-branch corresponding to a transition. In particular, the guard must not be evaluated before the semaphore is retrieved: On the one hand, it must be guaranteed that no other transition changes the validity of the guard by changing some data between checking of the guard and retrieving the semaphore. On the other hand, artificial deadlocks which were not contained in the SLIM specification could be introduced this way. For example, consider two components c_1, c_2 and let the corresponding processes pass the checking of guards for one of their transitions. Now, assume c_1 retrieves the semaphore and tries to start synchronous communication with c_2 . In result, the whole system blocks since c_2 cannot participate in the synchronous communication as it still waits for the semaphore. One could draw the conclusion that the guard is to be checked simply after the semaphore is retrieved. To avoid blocking in case the guard does not evaluate to `true`, an additional `if` statement with `else` branch must be used. This works fine but leads to the generation of some unnecessary states. A better solution is to test the availability of the semaphore and to check the guard at the same time by using the boolean conjunction of them as the first statement in the `do`-branch of a transition. This yields the desired effect, that the branch can only be chosen for execution when both holds, the semaphore is available and the guard evaluates to true. Of course this combined test must be performed in one atomic step with the assignment to the flag variable. For a better understanding of the generated PROMELA code a macro `nT` (“next transition”), parameterised with the guard expression, is defined for that. It is executable iff the semaphore could be retrieved, i.e., `gflagGoNextTransition` was `false`, and the guard evaluates to `true`:

```

#define nT(guard) d_step { \
  guard && !gflagGoNextTransition; \
  gflagGoNextTransition = true }

```

The counterpart of `nT` is the macro `fT` (“free transition semaphore”) which simply sets the global flag variable to `false`:

```
#define fT gflagGoNextTransition = false
```

4.4.2 Resetting of Reactivated Data Subcomponents

Data subcomponents $dsc \in DSub(c)$ in a component $c \in CCmp$ which were not active in the source mode m_s of a transition, i.e., $dsc \notin DSub(c, m_s)$, but are active in the target mode m_t , i.e., $dsc \in DSub(c, m_t)$, are reset to their default value. The default value $dfl(c, dsc)$ of those data elements exists since this is syntactically required (cf. [37, page 16]). It is important that the assignments of the default values to reactivated data subcomponents are placed before the assignments originating from the transition effect to make sure that transition effects are not overwritten by the default values. Indeed the reset could be omitted for those data subcomponents to which an assignment exists in the transition effects. However, as all assignments take place inside on `d_step`-block this additional effort could not save any state and thus is not taken. The default value can always be assigned directly to the unbuffered variable `DataStoreVar(c, dsc)`, as it is done by `UnbufferedAssignment(c, dsc, expr)`, since the transition effects may never read values from data elements reactivated by the transition.

The placeholder `ResetReactivatedDataSubcomponents(c, m_s, m_t)` stands for the code shown in listing 4.5:

```
For  $dsc \in DSub(c, m_t) \setminus DSub(c, m_s)$ :
[ UnbufferedAssignment(c, dsc, dfl(c, dsc));
```

Listing 4.5: Code generated by `ResetReactivatedDataSubcomponents(c, m_s, m_t)`.

4.4.3 Resetting of Disconnected Data Ports

Similarly to the resetting of reactivated data subcomponents, data ports which were the target of a data port connection or a flow in the source mode m_s of a transition but are not in the target mode m_t are reset to their default value as well. In contrast to data subcomponents, data ports cannot be deactivated and thus this reset must take place as soon as the connection is disabled. The syntactic restrictions G-5 and G-6 (cf. [37, page 22]) guarantee that the default values for disconnected ports which are not target of another connection in the target mode of the transition exist. The translation however does not make the effort to distinguish between data ports that were indeed disconnected as described above and data ports for which the connection changed, i.e., they are targets of connections or flows in both, the source and the target mode of the transition, but of different ones. Instead, whenever a transition disables a data port connection or a flow, its target port is perceived as “disconnected” and thus reset to its default value, if it exists, that is, $dfl(c, dp) \neq \perp$. The check for the existence of the default value becomes necessary as the syntactic restrictions do not require default values for data ports which are target of connections or flows in every mode. Formally, the set of in that sense disconnected data ports is defined by:

$$DisconnectedDataPorts := \{tp \mid (sp, tp) \in (DCon(c, m_s) \setminus DCon(c, m_t))\} \cup \{d \mid (d, expr) \in (Flw(c, m_s) \setminus Flw(c, m_t))\}$$

Like with the unnecessary resets of reactivated data subcomponents whose default value is overwritten by an assignment in the transition effect, the unnecessary reset of data ports does not

introduce additional intermediate states since all assignments are contained in a `d_step`-block. If any of those data ports becomes the target of another connection or flow in the transition's target mode, the default value will be overwritten by the actual one later on, as described in section 4.6 about the realisation of data port updates. However, it is not true that simply every data port could be reset to its default value since then changes due to transition effects would get lost.

Among the disconnected data ports, two different types must be distinguished: First, outgoing data ports of the considered control component $c \in CCmp$ itself, that is $tp \in ODPrt(c)$. This is the case whenever a simple identifier is used as target port: $tp \in Ide$. In the other case, a name is used as target port: $tp \in Nam$ where $Nam = Ide \times \{.\} \times Ide$. Then, the name can be split up in two parts separated by the dot: $tp = csc.idp$. The first part refers to a control subcomponent $csc \in CSub(c)$ of the component while the second part must be an incoming data port of that subcomponent, i.e., $idp \in IDPrt(c.csc)$. This distinction is important because it determines whether a buffered or an unbuffered assignment has to be used. The new values for outgoing data ports of the component itself must be written to the buffer variable $DataBufferVar(c, tp)$ in an buffered assignment to allow fixpoint iteration to compare the new and the old value and thus determine whether something changed. Additionally, the old value must be preserved in case it is read on the right hand side of an assignment in the transition effect. In contrast, the new values for incoming data ports of subcomponents are directly assigned to the unbuffered variables since neither fixpoint iteration depends on them nor their old value can be read in transition effects (recall: transition effects are lists of assignments to data subcomponents or outgoing data ports and the right-hand side expression is over incoming and outgoing data ports and data subcomponents of the component which contains the transition).

Altogether, $ResetDisconnectedDataPorts(c, m_s, m_t)$ generates:

```

For  $tp \in DisconnectedDataPorts$ :
┌
│ If  $tp \in Ide$ :
│ ┌
│ │ If  $dfl(c, tp) \neq \perp$ :
│ │ ┌
│ │ │  $BufferedAssignment(c, tp, dfl(c, tp));$ 
│ │ └
│ │ Else, i.e.,  $tp = csc.idp \in Nam$ :
│ │ ┌
│ │ │ If  $dfl(c.csc, idp) \neq \perp$ :
│ │ │ ┌
│ │ │ │  $UnbufferedAssignment(c.csc, idp, dfl(c.csc, idp));$ 
│ │ │ └
│ │ └
│ └
└

```

Listing 4.6: Code generated by $ResetDisconnectedDataPorts(c, m_s, m_t)$.

One final remark on the position of $ResetDisconnectedDataPorts$ inside the code generated by $ProactiveTransitions$ (applies to $MultiwayEventCommunication$ as well): In an intermediate version of the SLIM language and its semantics it was allowed to set the value of data ports by data port connections or flows in some modes while in others assignments in transition effects could be used. For example, a transition could disable a data port connection to a data port and then directly assign a new value to that data port by its effect. In this situation, like with the resetting of reactivated data subcomponents, the assignment of the default value must be placed before the assignments resulting from the transition effect to make sure that the default value will not overwrite the effect. However, later on the syntactic restriction G-7 (cf. [37, page 22]) had to be introduced which forbids the mixture of data port connections and flows with transition effects: A data port which is target of a data port connection or flow may never occur on the left hand side of mode transition assignments in the same component implementation, and vice versa. Hence, the position of $ResetDisconnectedDataPorts$ is not crucial any more and in fact further optimisations became possible, as described in subsection 4.6.1 dealing with fixpoint iteration.

4.4.4 Translation of the Effects

As already mentioned in the introduction of this section, the SLIM assignments contained in a transition's effect f have to be translated to PROMELA assignments. However, assignments of new values to data elements – possible in transition effects are outgoing data ports and data subcomponents only – that occur on the right hand side of a subsequent assignment in the same effect must be buffered. For example, the effect $x := y; y := x$ must be translated to something like $x' := y; y := x; x := x'$. As assignments to outgoing data ports are buffered anyway for the fixpoint iteration, only for data subcomponents $d \in DSub(c)$ an analysis must be performed to find out whether they must be buffered when translating a transition effect f :

$$\text{RequiresBuffering}(d, f) :\Leftrightarrow f = \dots \quad d := expr; \dots; d' := expr'; \dots \\ \text{with } d \in expr'$$

Furthermore, a buffer variable $DataBufferVar(c, d)$ for a data subcomponent $d \in DSub(c)$ must be declared as soon as one transition with an effect exists that requires buffering:

$$\text{MustBeBuffered}(c, d) :\Leftrightarrow \exists(m_s, t, g, f, m_t) \in MTr(c) : \text{RequiresBuffering}(d, f)$$

Of course, in the end the buffered values must be copied to the corresponding storage variables. For outgoing data ports this is done by the fixpoint iteration over them. For data subcomponents, however, this must be done directly after the transition effects. To reduce the state space, the buffer variables of data subcomponents are reset to 0 afterwards because their value is not needed any more. Taking this into account, $Effects(c, f)$ generates the code shown in listing 4.7. Note that for copying and resetting the buffer variables PROMELA code is directly generated without the usage of *UnbufferedAssignment*. This also works for **reals**, as explained in section 4.2.2.

```

For every  $d := expr \in f$ :
┌ If  $d \in ODPrt(c)$  or  $(d \in DSub(c)$  with  $\text{RequiresBuffering}(d, f)$ ):
│    $\text{BufferedAssignment}(c, d, expr)$ ;
│ Else:
│    $\text{UnbufferedAssignment}(c, d, expr)$ ;
└

For every  $d := expr \in f$  with  $d \in DSub(c)$  and  $\text{RequiresBuffering}(d, f)$ :
┌  $\text{DataStoreVar}(c, d) = \text{DataBufferVar}(c, d)$ ;
│  $\text{DataBufferVar}(c, d) = 0$ ;
└

```

Listing 4.7: Code generated by $Effects(c, f)$.

4.4.5 Adjustment of the Activation Status of Control Subcomponents

Like with connections and flows, the activation status of control subcomponents can depend on the mode of their supercomponent. Consequently, when a component takes a transition changing the mode from source mode m_s to target mode m_t , the activation status of its control subcomponents must be adjusted: Components active in m_s but not in m_t must be deactivated, components not active in m_s but in m_t must be activated. For components which stay activated or deactivated nothing is to be done. To deactivate a subcomponent $csc \in CSub(c, m_s)$ the **deactivate** message is send by c to $c.csc$ along $Channel(c.csc)$. Analogously, to activate a subcomponent $csc \in CSub(c, m_t)$ the message **activate** is sent. The code for this is generated

by *AdjustActivationOfControlSubcomponents*(c, m_s, m_t) as shown in listing 4.8. The reaction of subcomponents on receiving those messages is described in section 4.7 about activation handling.

```

// Deactivation
For  $csc \in CSub(c, m_s) \setminus CSub(c, m_t)$ :
[ Channel( $c.csc$ )!deactivate, 0;
  Channel( $c.csc$ )?deactivate, _;

// (Re)Activation
For  $csc \in CSub(c, m_t) \setminus CSub(c, m_s)$ :
[ Channel( $c.csc$ )!activate, 0;
  Channel( $c.csc$ )?activate, _;

```

Listing 4.8: Code generated by *AdjustActivationOfControlSubcomponents*(c, m_s, m_t).

4.5 Multiway Event Communication

In the previous section, all steps necessary to implement explicit and implicit transition effects were described for *proactive transitions*, i.e., transitions without a trigger. In the present section, the focus lies on the realisation of handshaking between processes based on event ports used as transition triggers. A *master transition* in one component can only be taken if at least one *reactive transition* with a corresponding trigger in another component can synchronously be taken. The intricacy is that such event communication works in a *multiway* manner, that is, *every* component that contains one or more enabled transition(s) that can react to a master transition has to take one synchronously with the master transition. Additionally, the events used as triggers can be forwarded along event port connections whose activation can depend on the current mode of the component they belong to. Hence, in general the *global* system state must be known to identify potential communication partners.

For the translation to PROMELA, which has a local component-wise and not a global perspective on the system state, multiway event communication is realised based on a message protocol between components, their direct supercomponent and their direct control subcomponents only. Whenever a process, implementing one component, is in a mode from which a master transition with fulfilled guard goes out it can send a `trymultiway` message in order to ask for handshakings. The message is equipped with an integer value that indicates which event is to be used. To this end, *EventNumber*(*event*) injectively maps every $event \in EPrt$ to an unique integer identifier as described in section 4.2.1. The message's addressee depends on trigger t of the master transition: either a control subcomponent or the supercomponent receives it in the first place. Every component which receives a `trymultiway` message forwards it along its event port connections whose activation status can be determined locally since it only depends on the current mode of the component. Additionally, if any reactive transition with fulfilled guard exists in the receiving component, one of them is – nondeterministically, if several are possible – taken and the global variable `gflagTrymultiway`, that was initialised with `false`, is set to `true` which indicates that at least one partner was found for handshaking. Consequently, the master transition is taken if this variable is finally `true`. To make sure that the process with the master transition waits until every potential communication partner has received and processed the `trymultiway` message, every component that sends out a message waits for an acknowledgment by the receiver before

it continues. A beneficial side-effect of this is that all steps involved in multiway event communication are performed strictly sequential and hence there is no state space explosion effect for this part due to concurrency. Crucial for the correct functioning, especially termination, of the whole mechanism is, firstly, the circumstance that event port connections cannot be cyclic (this follows from the topological restrictions for event port connections, cf. section 2.2.5) and, secondly, the prohibition of fan-out to (different) incoming event ports of the same component (syntactic restriction G-11 in [37, page 22]). Together this guarantees that every component has to react to a `trymultiway` message at most once per multiway event communication.

In the following two subsections the generation of code for master and for reactive transitions will be treated separately. Together those two parts make up the code for multiway event communication:

$$\text{MultiwayEventCommunication}(c, m) := \begin{array}{l} \text{MasterTransitions}(c, m) \\ \text{ReactiveTransitions}(c, m) \end{array}$$

4.5.1 Implementation of Master Transitions

A transition defined in a control component $c \in CCmp$ is a *master transition* if its trigger t is either an outgoing event port of the component itself or an incoming event port of one of its control subcomponents. Since only active control components could possibly react by taking a reactive transition, the latter case is restricted further to control subcomponents that are active in the transition's source mode $m_s \in Mod(c)$, i.e., to $csc \in CSub(c, m_s)$. Thus, the translation has to consider those transitions which go out from mode m and whose trigger t is in the set

$$\text{MasterTriggers}(c, m) := \text{OEPr}(c) \cup \bigcup_{csc \in CSub(c, m)} \{csc.\} \times \text{IEPr}(c.csc)$$

As mentioned above, the addressee of the `trymultiway` message depends on the trigger. In more detail, this means:

- If an incoming event port iep of a control subcomponent $csc \in CSub(c)$, i.e., $iep \in \text{IEPr}(c.csc)$, is used as trigger, i.e., $t = csc.iep$, then the message is sent to this subcomponent (*downward trymultiway*).
- If an outgoing event port of component c itself is used, i.e., $t \in \text{OEPr}(c)$, the message must be sent to the supercomponent (*upward trymultiway*).

This case distinction is made by $\text{MasterTransitions}(c, m)$ as presented in listing 4.9. Independent of the concrete addressee, after receiving the acknowledgment from it the master transition is taken if `gflagTrymultiway` is executable, i.e., if it has been set to `true`. This part is identical to proactive transitions, including the `invalidate` message which starts the propagation of updated data port values and finally releases the semaphore which assures atomicity of transitions. Note that this semaphore has already been requested together with the evaluation of the guard by `nT` before the multiway event communication was started. In case no component could be found for handshaking, i.e., `gflagTrymultiway` is still `false`, this semaphore must be released by explicitly calling `fT` before the `do`-loop of the current mode is continued.

Of course `gflagTrymultiway` must be reset to `false` if it became `true` so that a following multiway event communication works correctly. Doing so directly after completing the multiway event communication – in contrast to a reset directly before the next master transition is tried – guarantees that outside this part `gflagTrymultiway` always has its initial value `false` and hence the possible doubling of the state space due to unimportant differences in the boolean value is avoided. However, this variable must not be excluded from the state vector by declaring it as `hidden` since its value obviously makes a difference in the course of multiway event communication.

```

For  $(m_s, t, g, f, m_t) \in MTr(c)$  with  $m_s = m$  and  $t \in MasterTriggers(c, m)$ :
:: nT(Expression(c, g)) ->

    // Send trymultiway request
    If  $t = csc.iEPrt$  with  $csc \in CSub(c)$  and  $iep \in IEPrt(c.csc)$ :
    [Channel(c.csc)!trymultiway, EventNumber(iep)];
    [Channel(c.csc)?trymultiway, -];
    Else, i.e.,  $t \in OEPrt(c)$ :
    [Channel(c)!trymultiway, EventNumber(t)];
    [Channel(c)?trymultiway, -];

    if

    // One or more communication partner found
    :: d_step { gflagTrymultiway ->
        gflagTrymultiway = false;
        ResetReactivatedDataSubcomponents(c, m_s, m_t)
        ResetDisconnectedDataPorts(c, m_s, m_t)
        Effects(c, f)
        ModeVar(c) = modeID_m_t;
    }
    AdjustsActivationOfControlControlSubcomponents(c, m_s, m_t)
    Channel(c)!invalidate, 0;
    goto mode_m_t

    // No communication partner found
    :: !gflagTrymultiway -> fT;

fi;

```

Listing 4.9: Code generated by *MasterTransitions(c, m)*.

4.5.2 Implementation of Reactive Transitions

A process receiving a `trymultiway` message must react by forwarding this message along event port connections and by taking a reactive transition, if possible. Two different ways of receiving the `trymultiway` message must be distinguished: Either it was sent by the supercomponent (*downward trymultiway*) or by one of the active control subcomponents (*upward trymultiway*). Both require in principle the same reactions but with subtle differences. Therefore, the simpler version for downward `trymultiway` is introduced first, before the differences for upward `trymultiway` are stressed. Together they cover all variants of reactive transitions:

$$\begin{aligned} \text{ReactiveTransitions}(c, m) &:= \text{ReactiveTransitions_Downward}(c, m) \\ &\quad \text{ReactiveTransitions_Upward}(c, m) \end{aligned}$$

Reactive Transitions for Downward Trymultiway

For downward `trymultiway` the `trymultiway` message is sent by the supercomponent of the considered control component $c \in CCmp$ and thus received along channel $Channel(c)$. The second message parameter indicates which of the component's incoming event ports $iep \in IEPr(c)$ is meant to be used in the ongoing multiway event communication. This integer value could be received into a variable which is then evaluated in a case distinction using `if`. Yet, this additional variable and the `if`-block can be saved by applying the filter mechanism of PROMELA receive statements (cf. section 3.1.1): For every $iep \in IEPr(c)$ one branch, beginning with the receiving of the message `trymultiway` equipped with the constant value $EventNumber(iep)$, is added to the mode's `do`-loop. Consequently, for every incoming event the correct branch becomes executable.

Inside the branch for incoming event port iep two things have to be done: forwarding the `trymultiway` message along event port connections and taking one reactive transition, if possible. Since the activation of event port connections can depend on the current mode $m \in Mod(c)$, forwarding is handled before a transition might change the mode.

In detail, forwarding works as follows: For every target port tp of an currently active event port connection $(sp, tp) \in ECon(c, m)$ that originates from the considered incoming event port, i.e., $sp = iep$, a new `trymultiway` message is sent to the respective component and a returning acknowledgment is awaited. Due to the topological restrictions that are imposed for event port connections (cf. section 2.2.5), it is known that the target port must be an incoming event port of a control subcomponent of c , that is, $tp = csc.iep'$ for some $csc \in CSub(c, m)$ and $iep' \in IEPr(c.csc)$ (in-in connection). Additionally, the syntactic restriction G-4 (cf. [37, page 22]) guarantees that a subcomponent is active whenever a connection to it is enabled and thus the synchronous communication cannot block. Finally, the topological restrictions together with the prohibition of fan-out to (different) event ports of the same component (G-11, *ibid.*) assure that every control subcomponent receives the `trymultiway` message at most once per multiway event communication. Due to this fact, the message can simply be forwarded in a sequential manner – without any further considerations about the order or the handling of nondeterminism. The second parameter of the new `trymultiway` message sent to csc is of course set to $EventNumber(iep')$, the integer identifier of the subcomponent's incoming event port, and not to the value that was originally received, identifying the component's own incoming event port iep .

The code generated for the reactive transitions itself is very similar to that of proactive transitions: For every transition outgoing from mode m whose trigger is the incoming event port for which the `trymultiway` was received, i.e., $t = iep$, a branch starting with the transition's guard expression g is added to an `if`-block which handles the possible nondeterminism when several reactive transitions are enabled. Inside those branches, the explicitly and the implicitly given transition effects are realised. Additionally, the global variable `gflagTrymultiway` is set to `true`

as indication for the master transition that at least one reactive transition has been taken. Finally, a `trymultiway` message is sent to the supercomponent – which originally sent the message to the considered component – as an acknowledgment of the fact that the component has finished its handling of the multiway event communication. An additional `invalidate` message is not necessary, the one that will be sent by the process containing the master transition suffices. For the case that no reactive transition is enabled, an `else`-branch is added which just sends the acknowledgment. If in c no reactive transition outgoing from mode m exists at all, the `if`-block contains only the `else`-branch. In this case the whole `if` can be replaced by just the sending of the acknowledgment (done in the implementation, but for the sake of readability omitted in the meta-code presented here). Note that, in contrast to proactive and master transitions, reactive transitions do not request the semaphore by executing `nt`. This is no contradiction to the atomicity of SLIM transitions since handshaking requires that at least one reactive transition is taken synchronously with the master transition. Besides that, reactive transitions could in fact never retrieve the semaphore since it is hold by the process containing the master transition during the whole multiway event communication. However, this still prevents that transitions that do not participate in the handshaking are taken independently while multiway event communication goes on.

All this is done with the PROMELA code generated by `ReactiveTransitions_Downward(c, m)` as shown in listing 4.10.

```

For  $iep \in IEPrt(c)$ :
  :: Channel(c)?trymultiway, EventNumber(iep) ->

    // Ask subcomponents via event port connections
    For  $(sp, tp) \in ECon(c, m)$  with  $sp = iep$ ,
    thus  $tp = csc.iep'$ ,  $csc \in CSub(c, m)$ ,  $iep' \in IEPrt(c.csc)$ :
      Channel(c.csc)!trymultiway, EventNumber(iep');
      Channel(c.csc)?trymultiway, -;

    // Take own reactive transition, if possible
    if
    For  $(m_s, t, g, f, m_t) \in MTr$  with  $m_s = m$  and  $t = iep$ 
      :: d_step { Expression(c, g) ->
          gflagTrymultiway = true;
          ResetReactivatedDataSubcomponents(c, m_s, m_t)
          ResetDisconnectedDataPorts(c, m_s, m_t)
          Effects(c, f)
          Mode Var(c) = modeID_m_t;
        }
          AdjustActivationOfControlSubcomponents(c, m_s, m_t)
          Channel(c)!trymultiway, 0;
          goto mode_m_t;
      :: else ->
          Channel(c)!trymultiway, 0;
    fi;

```

Listing 4.10: Code generated by `ReactiveTransitions_Downward(c, m)`.

Multiway event communication involving event ports of the `root` component is always possible – even when no master or reactive transition exists in the specification at all. In the translation this is realised by the `Environment` process, which can be perceived as a dummy supercomponent for `root`, as described in section 4.8.

Reactive Transitions for Upward Trymultiway

The handling of upward trymultiway is in principle identical to downward trymultiway. However, some subtle differences exist which make handling of upward trymultiway a little more involved. The following differences must be considered:

- First of all, the `trymultiway` message is not received from the supercomponent of c along `Channel(c)` but from an (active) control subcomponent $csc \in CSub(c, m)$ along `Channel(c.csc)`. Analogously, the acknowledgment is to be sent to this subcomponent. The second parameter of the `trymultiway` message must identify an outgoing event port $oep \in OEPrt(c.csc)$ of the respective subcomponent. Consequently, the `do`-loop for a mode $m \in Mod(c)$ contains for every control subcomponent $csc \in CSub(c, m)$ one branch starting with a receiving statement for any of its outgoing event ports $oep \in OEPrt(c.csc)$. In the meta-code this is reflected by the two nested `For`-blocks.
- In addition to forwarding the `trymultiway` message to the (other) subcomponents, forwarding is also possible to the supercomponent by a data port connection targeting at an outgoing event port of the considered component, i.e., $tp \in OEPrt(c)$. Since in-out connections are not allowed by the topological restrictions, forwarding to the parent was not possible with downward trymultiway. For the same reason in combination with G-1 (cf. [37, page 21]), which requires that source and target components of event port connections must be different, it is guaranteed that an upward trymultiway can never return as a downward trymultiway to components which already participated in the multiway event communication. In particular, the subcomponent which sent the original `trymultiway` message does not receive a forwarded one. Note that by G-11 (no fan-out to different ports of the same components) every subcomponent and the supercomponent retrieve at most one `trymultiway` message. All this is necessary to guarantee that no component can take two reactive transitions in succession within one multiway event communication.
- Finally, the least obvious but perhaps most critical difference in handling upward trymultiways is that the adjustment of the activation status of control subcomponents in the course of taking the reactive transitions must be postponed. Why is that necessary? Well, an upward trymultiway can only be received by a supercomponent of the component containing the master transition – the direct one or transitively at a higher level of the component hierarchy. If this supercomponent takes a reactive transition it might happen that the subcomponent containing the master transition becomes deactivated. Two problems arise when this is done immediately: First, the master transition – which is planned to be taken after the whole multiway event communication has finished – could not be taken any more. Second, the process realising the master transition awaits an acknowledgment of the `trymultiway` message and thus is not ready to receive an `deactivate` or `reset` message. To handle the situation, the adjustment of the activation status of control subcomponents in the supercomponent is postponed until the `invalidate` message has been received from the component from which the upward `trymultiway` message was received. The `invalidate` indicates that the master transition has been taken because it is sent by the corresponding process afterwards. Now, it is even safe to deactivate the subcomponent

containing the master transition. Afterwards, the caught `invalidate` message must be forwarded to the supercomponent. Note that this process can take place successively at different levels in the component hierarchy.

These necessary adaptations lead to the differences between the code for handling downward `trymultiway` (cf. listing 4.10) and the code generated by `ReactiveTransitions_Upward(c, m)` as shown in listing 4.11.

```

For csc ∈ CSub(c, m):
  For oep ∈ OEPrT(c.csc):
    :: Channel(c.csc)?trymultiway, EventNumber(oep) ->

        // Ask supercomponent via Event Port Connections
        For (sp, tp) ∈ ECon(c, m) with sp = csc.oep and tp ∈ OEPrT(c):
          Channel(c)!trymultiway, EventNumber(tp);
          Channel(c)?trymultiway, -;

        // Ask subcomponents via Event Port Connections
        For (sp, tp) ∈ ECon(c, m) with sp = csc.oep and
        tp = csc'.iep' for some csc' ∈ CSub(c, m), iep' ∈ IEPrT(c.csc'):
          Channel(c.csc')!trymultiway, EventNumber(iep');
          Channel(c.csc')?trymultiway, -;

        // Take own Reactive Transition, if possible
        if
        For (ms, t, g, f, mt) ∈ MTr with ms = m and t = csc.oep
          :: d_step { Expression(c, g) ->
              multiwayCounter = multiwayCounter + 1;
              ResetReactivatedDataSubcomponents(c, ms, mt)
              ResetDisconnectedDataPorts(c, ms, mt)
              Effects(c, f)
              ModeVar(c) = modeIDmt;
            }
          Channel(c.csc)!trymultiway, 0;
          Channel(c.csc)?invalidate, -; // Wait for master!
          AdjustActivationOfControlSubcomponents(c, ms, mt)
          Channel(c)!invalidate, 0; // Forward caught!
          goto modemt;
          :: else ->
            Channel(c.csc)!trymultiway, 0;
        fi;

```

Listing 4.11: Code generated by `ReactiveTransitions_upward(c, m)`.

4.6 Data Port Updates

Transitions can change the values of data ports by explicitly given effects as well as implicit impacts on data port connections and flows. The realisation of these local effects was described mainly in section 4.4 on proactive transitions. However, these changes can propagate along data port connections and flows through the whole component hierarchy. To this end, the translation to PROMELA establishes a mechanism named *invalidate-update-cycle* based on a communication protocol, comparable to the handling of multiway event communication with `trymultiway` messages (cf. section 4.5). After taking a transition, a process sends an `invalidate` message to its supercomponent indicating that the values of its outgoing data ports might have changed. Every component $c \in CCmp$ receiving an `invalidate` message from any of its currently active subcomponents $csc \in CSub(c, m)$ forwards it to its own supercomponent. The PROMELA code for that is generated by *ForwardInvalidate(c, m)* as shown in listing 4.12.

```

For  $csc \in CSub(c, m)$ :
[
  :: Channel(c.csc)?invalidate, _ ->
     Channel(c)!invalidate, 0;

```

Listing 4.12: Code generated by *ForwardInvalidate(c, m)*.

Finally, the `invalidate` message reaches the process realising the `root` component and is forwarded by it one last time, namely to the `Environment` process which serves somehow as a dummy supercomponent for `root`. The `Environment` process transforms the `invalidate` notification to an `update` directive which is send back to `root` and then spreads out to the whole system. Every component process $c \in CCmp$ receiving an `update` message from its supercomponent – thus, along *Channel(c)* – does basically two things:

1. Firstly, it initiates the update of its active control subcomponents $csc \in CSub(c, m)$. Therefore, their incoming data ports $idp \in IDPrT(c.csc)$ are updated according to active data port connections and flows (from outgoing data ports of other subcomponents or incoming data ports of the considered component c itself) before the `update` message is forwarded to each subcomponent along *Channel(c.csc)*. Again, the component process waits for an acknowledgment of every subcomponent before it continues its own update procedure. Forwarding the `update` directive may *not* be omitted for control subcomponents whose incoming data ports did not change their value since the last invalidate-update-cycle, including subcomponents which do not have any incoming data ports at all, since a transition taken in a deeper subcomponent still might change some data ports in the component hierarchy. The code for updating the control subcomponents is generated by *PropagateUpdateToActiveSubcomponents(c, m)* as shown in listing 4.13. In the implementation of the translation that piece of code is moved to an inline function two increase readability of the generated code, especially since it might occur twice.
2. After updating its active control subcomponents, the component can update its own outgoing data ports whose values can by data port connections and flows depend on outgoing data ports of its control subcomponents and on incoming data ports of the component itself (flows only). Finally, the component process must acknowledge that it completed its update procedure by sending an `update` message to its supercomponent.

The PROMELA code for the whole update procedure is generated by *HandleUpdate(c, m)* as defined in listing 4.14. For the termination of the invalidate-update-cycle the acyclicity of data

```

For  $csc \in CSub(c, m)$ :
  // Provide latest values for subcomponent's in data ports
  d_step{
    // Via active data port connections
    For  $(sp, tp) \in DCon(c, m)$  with  $tp = csc.idp$  for some  $idp \in IDPrT(c.csc)$ ,
    thus  $sp \in IDPrT(c)$  or
       $sp = csc'.odp$  for some  $csc' \in CSub(c, m)$  with  $odp \in ODPrt(c.csc')$ :
    [  $DataStoreVar(c.csc, idp) = DataStoreVar(c, sp)$ ;
    // Via active flows
    For  $(tp, expr) \in Flw(c, m)$  with  $tp = csc.idp$  for some  $idp \in IDPrT(c.csc)$ :
    [  $UnbufferedAssignment(c, csc.idp, expr)$ ;
  }

  // Send update message to subcomponent, await acknowledgment
  Channel(c.csc)!update, 0;
  Channel(c.csc)?update, -;

```

Listing 4.13: Code generated by *PropagateUpdateToActiveSubcomponents(c, m)*. If a subcomponent is not targeted by any forwarding data port connection or flow, the resulting empty `d_step`-block can be eliminated.

port dependencies, i.e., data port connections combined with flows, in every system state is crucial. Data port connections on their own are acyclic since in-out connections are not possible by topologic restrictions. With data flows this is not the case any more and thus syntactical restriction H-5 (cf. [37, page 24]) explicitly forbids cyclic dependencies between data ports. However, in contrast to event port connections, data dependencies can be cyclic on the component level, that means, the value of an outgoing data port *odp* of a component can be forwarded in multiple steps to an incoming data port *idp* of the same component. The only restriction is, that the value of this incoming data port *idp* must not change the value of *odp* again. In general, whenever in some component an outgoing data port changes whose value is forwarded to an incoming data port of an already updated component this receiving component must be updated again. This is why *HandleUpdate(c, m)* additionally contains a fixpoint iteration. The details of it are given in the following subsection. But before, some general final remarks:

- At three points in the generated PROMELA code, namely in *PropagateUpdateToActiveSubcomponents* and *HandleUpdate* where values are forwarded along data port connections as well as in *NonFixpointCopy* (see below) where values of the buffered variables are copied to the unbuffered ones, simple PROMELA assignments between variables are used instead of *UnbufferedAssignment*, which would generate embedded C-code for SLIM `reals`. Similarly, the unbuffered and the buffered variables are compared in *FixpointCheck* (see below) by the PROMELA operator `==`, again without embedded C-code. This can be done since the SLIM `reals` are stored in PROMELA `ints` and copying as well as equality checking do not depend on the data type but are just based on the sequence of bits, as discussed in section 4.2.2.
- During the whole invalidate-update-cycle, the semaphore for atomicity of transitions is hold by the process which sent the initial `invalidate` message. This guarantees that no other transition can take place in the meanwhile, even for multiway event communication

```

:: Channel(c)?update, _ ->

    // *** Update active subcomponents ***
    // First 'Iteration'
    PropagateUpdateToActiveSubcomponents(c, m)
    If FixpointIterationRequired(c, m):
    // Fixpoint Iteration
    do
    :: d_step { !( FixpointCheck(c, m) ) ->
        CopyBufferedSubcomponentsDataPortsToStore(c, m)
        }
        PropagateUpdateToActiveSubcomponents(c, m)
    :: d_step { ( FixpointCheck(c, m) ) ->
        CopyBufferedSubcomponentsDataPortsToStore(c, m)
        }
        break
    od;
    skip; // cannot jump into d_step
    Else:
    d_step {
        CopyBufferedSubcomponentsDataPortsToStore(c, m)
    }

    // *** Update own out data ports ***
    d_step {

        // Via active data port connections
        For (sp, tp) ∈ DCon(c, m) with tp ∈ ODPrT(c),
        thus sp = csc.odp for some csc ∈ CSub(c, m)
        with odp ∈ ODPrT(c.csc):
        [DataBuffer Var(c, tp) = DataStore Var(c.csc, odp);

        // Via active flows
        For (d, expr) ∈ Flw(c, m) with d ∈ ODPrT(c):
        [BufferedAssignment(c, d, expr)

    }

    // *** Acknowledge update ***
    Channel(c)!update, 0;

```

Listing 4.14: Code generated by *HandleUpdate(c, m)*. The `skip` is necessary, since jumps (here by `break` inside the `do`-loop) into a `d_step`-block are not possible. Neighbouring `d_step`-blocks, i.e., when no fixpoint iteration is necessary, can be joined while empty ones can always be omitted.

since the `invalidate` is sent after the master transition was taken, which happened after all reactive transitions had been taken. Of course, the semaphore must be released after the invalidate-update-cycle completed. As the component process containing the proactive or master transition does not know when this is the case, the `Environment` process will do so by calling `fT` after receiving the `update`-acknowledgment from `root` (cf. section 4.8).

- Like with multiway event communication, due to the fact that every process waits until an acknowledgment arrives from the component to which it sent a message, the whole invalidate-update-cycle is performed sequential which is beneficial for size of the induced state space
- Data Subcomponents are not treated in the update procedure because they cannot be targets of data port connections or flows. Changes to them are only possible by transition effects (including deactivation and reset on reactivation) which are already handled by *Effects* (cf. section 4.4.4) and *ResetReactivatedDataSubcomponents* (cf. section 4.4.2) before the invalidate-update-cycle is started.

Altogether, *DataPortUpdates(c, m)* generates the whole PROMELA code necessary for handling invalidate-update-cycles as defined in listing 4.15:

```
DataPortUpdates(c, m) := ForwardInvalidate(c, m)
                          HandleUpdate(c, m)
```

Listing 4.15: Code generated by *DataPortUpdates(c, m)*.

4.6.1 Fixpoint Iteration

As mentioned above, in general it does not suffice to update every active control subcomponent just once. Instead, whenever a subcomponent $csc \in CSub(c, m)$ is updated and thus changes one of its outgoing data ports $odp \in ODPrT(csc)$ whose value is forwarded by a data port connection or a flow to an incoming data port $idp \in IDPrT(csc')$ of an already updated subcomponent $csc' \in CSub(c, m)$, then this subcomponent csc' must be updated again with the new input value. In particular, csc and csc' can be the same component¹ like in the SLIM specification given in listing 4.16 which is visualised in figure 4.1.

```
system Root
end Root;

system implementation Root.Impl
  subcomponents
    sc: system SC.Impl;
  connections
    data port sc.x -> sc.b;
    data port sc.y -> sc.c;
end Root.Impl;
```

¹In the latest version of the SLIM syntax this is not allowed any more. However, the principle problem stays unchanged since the value of an outgoing data port can be forwarded through a neighbour subcomponent to an incoming data port of the same component.

```

system SC
  features
    a: in data port int default 0;
    b: in data port int default 0;
    c: in data port int default 0;
    x: out data port int default 0;
    y: out data port int default 0;
    z: out data port int default 0;
  end SC;

system implementation SC.Impl
  flows
    x := a + 1;
    y := b + 1;
    z := c + 1;
  end SC.Impl;

```

Listing 4.16: SLIM specification requiring fixpoint iteration (cf. figure 4.1).

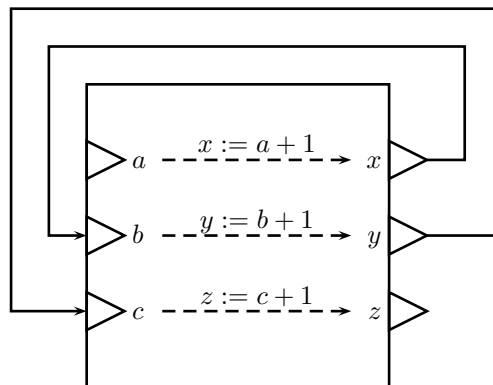


Figure 4.1: Visualisation of the SLIM specification from listing 4.16.

To handle these on the component level cyclic dependencies, the PROMELA program could check which out data ports changed their value on updating a subcomponent and mark all subcomponents to which this value is forwarded with an “needs update” flag comparable to typical worklist algorithms. However, this would complicate the generated code and even worse, it would require additional memory for the bookkeeping. Instead, in every iteration of the fixpoint iteration each control subcomponent receiving input values from other subcomponents is updated again. The iteration stops when no outgoing data port of any subcomponent whose value is forwarded to some other subcomponent changes any more, in other words, when a fixpoint is reached. To identify this situation, the new values of subcomponents’ outgoing data ports must be compared with their old ones. That is one reason – besides the requirement for buffering in sequencing transition effects – why a buffer (prefix `odpNew` as used by *DataBufferVar*) and a storage (prefix `odpStore` as used by *DataStoreVar*) variable exist for every outgoing data port. When a control subcomponent *c.csc* received an `update` and updated its own subcomponents in a nested fixpoint iteration it writes the new values for its own outgoing data ports to the respective buffer variable, allowing its supercomponent *c* to compare the new value with the old one still

stored in the storage variable. If the fixpoint is not reached yet, i.e., it exists an outgoing data port of some subcomponent whose value is forwarded to some subcomponent and for which the values stored in the buffer and in the storage variable differ, the new values from the buffer variables will be copied to the storage variables as the new old ones and another iteration is executed. The following sequence of intermediate states exemplifies the mechanism of fixpoint iteration for the first update (initially, all values are zero) of component `root.sc` from listing 4.16. The indices n and s distinguish between the new (buffered) and the old (stored) value, respectively, of the outgoing data ports. As the value of z is not forwarded to any neighbour subcomponent, the fact that it changed in the last iteration does not mean that no fixpoint is reached.

$$\begin{array}{l}
(a = 0, b = 0, c = 0 \ ; \ x_n = 0, x_s = 0, y_n = 0, y_s = 0, z_n = 0, z_s = 0) \\
\Downarrow \text{update: } (x, y, z)_n := (a, b, c) + 1 \\
(a = 0, b = 0, c = 0 \ ; \ \underbrace{x_n = 1, x_s = 0, y_n = 1, y_s = 0, z_n = 1, z_s = 0}_{\text{new} \neq \text{store}}) \\
\Downarrow \text{copy: } (x, y, z)_s := (x, y, z)_n \\
(a = 0, b = 0, c = 0 \ ; \ x_n = 1, x_s = 1, y_n = 1, y_s = 1, z_n = 1, z_s = 1) \\
\Downarrow \text{set inputs } (b := x, c := y) \\
(a = 0, b = 1, c = 1 \ ; \ x_n = 1, x_s = 1, y_n = 1, y_s = 1, z_n = 1, z_s = 1) \\
\Downarrow \text{update} \\
(a = 0, b = 1, c = 1 \ ; \ \underbrace{x_n = 1, x_s = 1}_{\text{new} = \text{store}}, \underbrace{y_n = 2, y_s = 1, z_n = 2, z_s = 1}_{\text{new} \neq \text{store}}) \\
\Downarrow \text{copy} \\
(a = 0, b = 1, c = 1 \ ; \ x_n = 1, x_s = 1, y_n = 2, y_s = 2, z_n = 2, z_s = 2) \\
\Downarrow \text{set inputs} \\
(a = 0, b = 1, c = 2 \ ; \ x_n = 1, x_s = 1, y_n = 2, y_s = 2, z_n = 2, z_s = 2) \\
\Downarrow \text{update} \\
(a = 0, b = 1, c = 2 \ ; \ \underbrace{x_n = 1, x_s = 1, y_n = 2, y_s = 2}_{\text{new} = \text{store} \Rightarrow \text{fixpoint reached}}, \underbrace{z_n = 3, z_s = 2}_{\text{new} \neq \text{store}}) \\
\Downarrow \text{final copy} \\
(a = 0, b = 1, c = 2 \ ; \ x_n = 1, x_s = 1, y_n = 2, y_s = 2, z_n = 3, z_s = 3)
\end{array}$$

To complete the PROMELA code generated by `HandleUpdate(c, m)`, the predicate and placeholders used in the context of fixpoint iteration can now be defined. First, a fixpoint iteration is necessary only if the value of at least one subcomponent's outgoing data port is forwarded by data port connections or flows to an incoming data port of an subcomponent:

$$\begin{aligned}
\text{ForwardedPorts}(c, m) &:= \\
&\bigcup_{csc \in CSub(c, m)} \{(csc, odp) \mid odp \in ODPrt(c.csc) : \\
&\quad (\exists (sp, tp) \in DCon(c, m) : tp \in Nam \wedge sp = csc.odp) \vee \\
&\quad (\exists (d, expr) \in Flw(c, m) : d \in Nam \wedge csc.odp \in expr) \}
\end{aligned}$$

$$\text{FixpointIterationRequired}(c, m) := \text{ForwardedPorts}(c, m) \neq \emptyset$$

The fixpoint check is then realised by:

$$\begin{aligned}
\text{FixpointCheck}(c, m) &:= \\
&\&_{(csc, odp) \in \text{ForwardedPorts}(c, m)} \text{DataStoreVar}(c.csc, odp) == \text{DataBufferVar}(c.csc, odp)
\end{aligned}$$

Finally, the copying of values from buffer to storage variables must be done for all outgoing data ports of ever subcomponent by:

```

CopyBufferedSubcomponentsDataPortsToStore(c, m) :=
  For csc ∈ CSub(c, m):
    For odp ∈ ODPrt(c.csc):
      DataStoreVar(c.csc, odp) = DataBufferVar(c.csc, odp);

```

If no fixpoint iteration is required at all, the only thing to do for a component after updating all its subcomponents but before setting its own outgoing data ports is to copy the values of the subcomponents' outgoing data ports from the buffer to the store variables, i.e., using *CopyBufferedSubcomponentsDataPortsToStore* once (cf. **Else** in listing 4.14). This is necessary due to the fact that the subcomponents always write the new values for their outgoing data ports to the buffer variables since they do not know whether their supercomponent will perform a fixpoint iteration or not. In the latter case, they could indeed directly write to the store variable. This could be achieved by using the optional parameter of the **update** message as an indication for the receiving subcomponent which variable is to be used.

Beside this, other optimisations are possible at compile time, based on a more fine grained static analysis of data port dependencies in the considered mode of a component:

- Those control subcomponents without changed input values since the last iteration, especially those without any incoming data ports at all, do not have to be updated in the following iteration. As an approximation to this, only those subcomponents should be reupdated which are in the current mode targeted by a data port connection or flow that has a subcomponent's outgoing data port as source.
- The fixpoint iteration also guarantees that the order in which subcomponents are updated does not matter. For example, when a data connection from a subcomponent csc_1 to another subcomponent csc_2 is active, fixpoint iteration guarantees that in the end the correct results are achieved even when csc_2 is updated first in every iteration. However, it would be better to analyse these dependencies and to update the subcomponents in a corresponding order. This could possibly reduce the number of required iterations when the new output values are directly reused for updating the following components.
- The retroactively introduced syntactic constraint that individual outgoing data ports are either set by data port connections and flows or by assignments in transition effects – but never mixed in different modes – allows the following optimisations:
 - Outgoing data ports which are only set by assignments in transition effects will not change during fixpoint iteration as the a transition effect is realised before the **invalidate** message is sent. Consequently, those data ports need not be buffered.
 - Similar to the buffers for data subcomponents, the buffer variables for outgoing data ports can be reset to 0 after copying their values to the storage variables.

However, all these optimisations are not considered in the implementation and thus perceived as future work.

4.7 Activation Handling

When the activation status of a component changes because the mode of its supercomponent changed the message **deactivate** or **activate**, respectively, is send to the component as described in section 4.4.5. On reception of an **deactivate** message, the receiving component

process forwards this message to all its active subcomponents – because they are deactivated together with their supercomponent – and then waits, in principle doing nothing else, until an `activate` message arrives. This message is again forwarded to all subcomponents that should be active in the current mode to reactivate them as well. Once more, as with the other communication protocols, acknowledgments are awaited for each communication.

In the course of activation handling, a distinction must be made between components that support mode history and those, that do not. A control component $c \in CComp$ maintains mode history iff its starting mode $stm(c)$ is of type `initial`. Contrariwise, if the starting mode is of type `activation` the component does not have mode history. Formally, the predicate *HasModeHistory* is defined as follows:

$$HasModeHistory(c) := stm(c) \text{ is of type } \textit{initial}$$

When a component with mode history is reactivated, it continues from the state it was in when it became deactivated. That means, neither the mode nor the values of any data elements do change. Thus, *DeactivateWithModeHistory(c, m)* generates code which simply waits for a reactivation after deactivation has been completed as shown in listing 4.17. Note that this protocol guarantees that an `activate` message is sent only to deactivated components which thus can receive it.

```

:: Channel(c)?deactivate, _ ->

// Deactivate control subcomponents active in current mode
For csc ∈ CSub(c, m):
  Channel(c.csc)!deactivate, 0;
  Channel(c.csc)?deactivate, _;

// Acknowledge deactivation, Wait for reactivation
Channel(c)!deactivate, 0;
Channel(c)?activate, _;

// Reactivate control subcomponents active in current mode
For csc ∈ CSub(c, m):
  Channel(c.csc)!activate, 0;
  Channel(c.csc)?activate, _;

// Acknowledge reactivation
Channel(c)!activate, 0;

```

Listing 4.17: Code generated by *DeactivateWithModeHistory(c, m)*.

In contrast, a component without mode history is restarted from its initial state on reactivation. That is, all data elements are reset to their default values and the mode is reset to the starting mode. For this reset on reactivation the initialisation code generated by *Initialisation(c)* (cf. listing 4.3 in section 4.3 on page 51) is reused which includes the receiving of the `activate` message. To this end, after completing the deactivation, the component process jumps to the `FirstActivationOrReactivationWithoutModeHistory` label. As for components with mode history, the subcomponents have to be deactivated as well. The difference is that any subcomponent is not just deactivated but reset with its supercomponent – no matter whether the

subcomponent itself does support mode history or not. To enforce this, a `reset` message is sent to the subcomponents instead of `deactivate`. In general, all subcomponents must be reset. For those active in the transition's source mode this is no problem, sending the `reset` message suffices. Deactive subcomponents without mode history are already reset. Only for subcomponents with mode history that are not active in the transition's source mode a special treatment is necessary: they have to be reactivated in order to be able to receive the `reset` message. The code for that, referred to as *DeactivateWithoutModeHistory(c, m)*, is presented in listing 4.18.

```

:: Channel(c)?deactivate, _ ->

    // Reset control subcomponents active in current mode
    For csc ∈ CSub(c, m):
    [ Channel(c.csc)!reset, 0;
      Channel(c.csc)?reset, _;

    // Reset control subcomponents with mode history deactive in
current mode
    For csc ∈ CSub(c) \ CSub(c, m) with HasModeHistory(c.csc):
    [ Channel(c.csc)!activate, 0;
      Channel(c.csc)?activate, _;
      Channel(c.csc)!reset, 0;
      Channel(c.csc)?reset, _;

    // Acknowledge deactivation
    Channel(c)!deactivate, 0;

    // Jump to initialisation code
    goto FirstActivationOrReactivationWithoutModeHistory;

```

Listing 4.18: Code generated by *DeactivateWithoutModeHistory(c, m)*.

Finally, the method for resetting a component is – independent of its support for mode history – identical to the one for deactivation of a component without mode history, except that the message name `deactivate` is replaced by `reset` as indicated by the substitution notation in listing 4.19 which gives the complete code for activation handling.

```

// React to deactivate, await activate
If HasModeHistory(c):
[ DeactivateWithModeHistory(c, m)
Else:
[ DeactivateWithoutModeHistory(c, m)

// React to reset, await activate
DeactivateWithoutModeHistory(c, m)[deactivate ↦ reset]

```

Listing 4.19: Code generated by *ActivationHandling(c, m)*.

4.8 The Environment Process

Besides the processes representing the different component instances, one special process named `Environment` is generated that can be perceived as a dummy supercomponent for the `root` component, which is actually translated in the same way as every other component. The special tasks of the `Environment` process are:

- On start-up it activates the root component and establishes the initial system state by invoking an update-cycle for the first time. Initialising the data elements to their default values does not suffice since already in the initial state data port connections and flows can be active and thus possibly overwrite the default values. The atomicity of the whole start-up code is guaranteed by the fact that the transition semaphore `gflagGoNextTransition` is initialised to `true`.
- Whenever an `invalidate` message is received from `root`, it is transformed into an `update` notification which is send back to the `root` process and then propagates through the whole component hierarchy.
- Multiway event communication is always possible when – directly or forwarded along event port connections – an outgoing event port of the root component is triggered. To this end, the environment process sets `gflagTrymultiway` to `true` and sends an acknowledgment on receiving a `trymultiway` message from the root (upward trymultiway).
- Similarly, incoming event ports of the `root` component can be triggered at any time. Therefor, the environment process mimics a master transition with corresponding trigger for every incoming event port of the `root` component by sending `trymultiway` messages to the `root`process (downward trymultiway).

The process type of the `Environment` process is shown in listing 4.20. Note, instead of using the placeholder `Channel(root)` the fixed name `channel_root` is used. The details of handling update-cycles are encapsulated in `Environment_PerformUpdate(root)` which is basically a simpler version of `HandleUpdate` (cf. listing 4.14 in section 4.6 on page 66). Additionally, it contains the call of the macro `fT` which releases the transition semaphore after completion of an invalidate-update-cycle. Values for incoming data ports of `root` are not set during updates since they always have their default value. However, changing input values from the environment could be modeled here. The code for updating the `root` component is given in listing 4.21.

4.9 The Whole Resulting PROMELA Program

To round the presentation of the translation from SLIM specifications to PROMELA programs off, listing 4.22 gives an overview over the complete PROMELA program resulting for a SLIM specification with root control component `root` $\in CCmp$ and a property φ . The translation of properties is covered in the next chapter – in particular, the placeholder `AtomicPropositions` is defined in section 5.1. The overall resulting PROMELA program contains all global declarations (`mtype`, `gflagGoNextTransition`, `gflagTrymultiway`) mentioned in the preceding sections as well as all definitions (`nT`, `fT`, integer representatives for symbolic identifiers).

```

active proctype Environment()
{
    // *** Activate root and establish initial state ***
    channel_root!activate, 0;
    channel_root?activate, _;
    Environment_PerformUpdate(root)

    // *** React to messages from root component ***
    do
    // React on invalidate with update
    :: channel_root?invalidate, _ ->
        Environment_PerformUpdate(root)
    // Upward trymultiway
    :: channel_root?trymultiway, _ ->
        gflagTrymultiway = true;
        channel_root!trymultiway, 0;
    // Downward trymultiway
    For idp ∈ IDPrT(root):
        :: nT(true) ->
            channel_root!trymultiway, EventNumber(idp);
            channel_root?trymultiway, _;
            if
            :: d_step { gflagTrymultiway ->
                gflagTrymultiway = false; }
                Environment_PerformUpdate(root)
            :: else -> fT
            fi;
        od;
    }
}

```

Listing 4.20: The type of the `Environment` process, generated by `Environment_PerformUpdate(root)`.

```

// Send update message, await acknowledgment
chan_root!update, 0;
chan_root?update, _;
// Copy buffered Out Data Ports of root
If |ODPrT(root)| > 0:
    d_step {
        For odp ∈ ODPrT(root):
            odpStore_root_odp = odpNew_root_odp;
        }
// Release Transition Semaphore
fT;

```

Listing 4.21: Controlling update-cycles, generated by `Environment_PerformUpdate(root)`.

```

// -- Message Types -----
mtype = { update, invalidate, trymultiway,
          activate, deactivate, reset };

// -- Macros for ensuring atomicity of transitions -----
bool gflagGoNextTransition = true;

#define nT(guard) d_step { \
    guard && !gflagGoNextTransition; \
    gflagGoNextTransition = true }

#define fT gflagGoNextTransition = false

// -- Flag for Multiway Event Communication -----
bool gflagTrymultiway = false;

// -- Define integer type & constants for Enums -----
#define enumIntType IntType(|Enums|)
For every enum  $\in$  Enums:
[#define enum_Enum IntValue(enum, Enums)]

// -- Define integer type & constants for Modes -----
#define modeIntType IntType(|Mod|)
For every mode  $\in$  Mod:
[#define modeID_mode IntValue(mode, Mod)]

// -- Define Atomic Propositions -----
AtomicPropositions( $\varphi$ )

// -- Global Variables for Data Elements -----
For every c  $\in$  CCmp:
[GlobalVariables(c)]

// -- Environment Process -----
Environment(root)

// -- Proctypes for Component Instances -----
For every c  $\in$  CCmp:
[Proctype(c)]

```

Listing 4.22: The whole resulting PROMELA program generated by *SLIM2Promela*(**root**, φ).

4.10 Example

The following PROMELA code was automatically generated – except for some comments and whitespaces that were removed afterwards – by the implemented translator for the Negate Random Bit example from listing 2.1 on page 16:

```
// ----- Message Types -----
mtype = { update, invalidate, trymultiway, activate, deactivate, reset };

// ----- MACROs for ensuring atomicity of transitions -----
bool gflagGoNextTransition = true;

#define nT(guard) d_step { \
    guard && !gflagGoNextTransition; \
    gflagGoNextTransition = true; }

#define fT d_step { gflagGoNextTransition = false; }

// ----- Flag for Multiway Event Communication -----
bool gflagTrymultiway = false;

// ----- DEFINE integer type & constants for enum data types -----
#define enumIntType bit

// ----- DEFINE integer type & constants for nominal modes -----
#define modeIntType bit
#define modeID__DefaultInitialMode 0
#define modeID_m 1

// ----- Global Variable Declarations for storing Components' Data -----

// *** root : __default__::Root.Impl ***
chan chan_root = [0] of { mtype, int };
modeIntType nominalMode_root;

// *** root_negator : __default__::Negator.Impl ***
chan chan_root_negator = [0] of { mtype, int };
modeIntType nominalMode_root_negator;
bool idpStore_root_negator_pos;
bool odpStore_root_negator_neg; bool odpNew_root_negator_neg;

// *** root_aBus : __default__::Bus.Impl ***
chan chan_root_aBus = [0] of { mtype, int };
modeIntType nominalMode_root_aBus;

// *** root_randomBit : __default__::RandomBit.Impl ***
chan chan_root_randomBit = [0] of { mtype, int };
modeIntType nominalMode_root_randomBit;
bool odpStore_root_randomBit_value; bool odpNew_root_randomBit_value;
```

```

// ----- Environment -----
active proctype Environment()
{
    // *** Activate & Update root component ***
    chan_root!activate, 0;
    chan_root?activate, _;
    chan_root!update, 0;
    chan_root?update, _;
    fT;

    // *** React to Messages from root component ***
    do

        // Invalidate: react with update
        :: chan_root?invalidate, _ ->
            chan_root!update, 0;
            chan_root?update, _;
            fT;

        // Upward Trymultiway:
        :: chan_root?trymultiway, _ ->
            gflagTrymultiway = true;
            chan_root!trymultiway, 0;

    od;
}

// ----- root : __default__::Root.Impl -----
inline inlineUpdateMode_DefaultInitialMode_root()
{
    // Update active subcomponent negator
    // 1.) Forward new inputs to subcomponents in data ports
    idpStore_root_negator_pos = odpStore_root_randomBit_value;
    // 2.) Send update message to subcomponent & receive acknowledgment
    chan_root_negator!update, 0;
    chan_root_negator?update, _;

    // Update active subcomponent aBus
    chan_root_aBus!update, 0;
    chan_root_aBus?update, _;

    // Update active subcomponent randomBit
    chan_root_randomBit!update, 0;
    chan_root_randomBit?update, _;
}

```

```

active proctype process1_root ()
{
    // *** Wait for first activation ***
    FirstActivationOrReactivationWithoutModeHistory:
    chan_root?activate, _;
    // Initialize Data Elements
    nominalMode_root = modeID__DefaultInitialMode;
    // Activate Non-Data Subcomponents (only those active in start mode)
    chan_root_negator!activate, 0;
    chan_root_negator?activate, _;
    chan_root_aBus!activate, 0;
    chan_root_aBus?activate, _;
    chan_root_randomBit!activate, 0;
    chan_root_randomBit?activate, _;
    // Activation completed
    chan_root!activate, 0;
    goto mode__DefaultInitialMode;

    // *** Run as automaton & React to messages ***
    mode__DefaultInitialMode:
    do

    // Incoming Message: update (from parent)
    :: chan_root?update, _ ->
        // Update active subcomponents: First Iteration
        inlineUpdateMode_DefaultInitialMode_root();
        // Update active subcomponents: Fixpoint Iteration
        do
        :: d_step {
            !(odpStore_root_randomBit_value == odpNew_root_randomBit_value) ->
                odpStore_root_negator_neg = odpNew_root_negator_neg;
                odpStore_root_randomBit_value = odpNew_root_randomBit_value; }
            inlineUpdateMode_DefaultInitialMode_root()
        :: d_step {
            (odpStore_root_randomBit_value == odpNew_root_randomBit_value) ->
                odpStore_root_negator_neg = odpNew_root_negator_neg;
                odpStore_root_randomBit_value = odpNew_root_randomBit_value; }
            break
        od;
        // Send update acknowledgment to parent
        chan_root!update, 0;

    // Incoming Message: invalidate (from subcomponents only)
    :: chan_root_negator?invalidate, _ ->
        chan_root!invalidate, 0;
    :: chan_root_aBus?invalidate, _ ->
        chan_root!invalidate, 0;
    :: chan_root_randomBit?invalidate, _ ->
        chan_root!invalidate, 0;

    // Incoming Message: deactivate (from parent) - keep mode history
    :: chan_root?deactivate, _ ->

```

```

    // deactivate active subcomponents
    chan_root_negator!deactivate, 0;
    chan_root_negator?deactivate, _;
    chan_root_aBus!deactivate, 0;
    chan_root_aBus?deactivate, _;
    chan_root_randomBit!deactivate, 0;
    chan_root_randomBit?deactivate, _;
    chan_root!deactivate, 0;
    // reactivate
    chan_root?activate, _;
    chan_root_negator!activate, 0;
    chan_root_negator?activate, _;
    chan_root_aBus!activate, 0;
    chan_root_aBus?activate, _;
    chan_root_randomBit!activate, 0;
    chan_root_randomBit?activate, _;
    chan_root!activate, 0;

    // Incoming Message: reset (from parent)
    :: chan_root?reset, _ ->
    // reset active subcomponents
    chan_root_negator!reset, 0;
    chan_root_negator?reset, _;
    chan_root_aBus!reset, 0;
    chan_root_aBus?reset, _;
    chan_root_randomBit!reset, 0;
    chan_root_randomBit?reset, _;
    // confirm reset
    chan_root!reset, 0;
    goto FirstActivationOrReactivationWithoutModeHistory;

    od;
}

// ----- root_negator : __default__ :: Negator.Impl -----

active proctype process2_root_negator()
{
    // *** Wait for first activation ***
    FirstActivationOrReactivationWithoutModeHistory:
    chan_root_negator?activate, _;
    // Initialize Data Elements
    d_step {
        idpStore_root_negator_pos = false;
        odpNew_root_negator_neg = false; odpStore_root_negator_neg = false;
        nominalMode_root_negator = modeID__DefaultInitialMode;
    }
    // Activation completed
    chan_root_negator!activate, 0;
    goto mode__DefaultInitialMode;
}

```

```

// *** Run as automaton & React to messages ***
mode__DefaultInitialMode:
do
// Incoming Message: update (from parent)
:: chan_root_negator?update, _ ->
// Update own out data ports
d_step {
odpNew_root_negator_neg = !(idpStore_root_negator_pos);
}
// Send update acknowledgment to parent
chan_root_negator!update, 0;

// Incoming Message: deactivate (from parent) - keep mode history
:: chan_root_negator?deactivate, _ ->
chan_root_negator!deactivate, 0;
chan_root_negator?activate, _;
chan_root_negator!activate, 0;

// Incoming Message: reset (from parent)
:: chan_root_negator?reset, _ ->
chan_root_negator!reset, 0;
goto FirstActivationOrReactivationWithoutModeHistory;

od;
}

// ----- root_aBus : __default__ :: Bus.Impl -----
active proctype process3_root_aBus()
{
// ... omitted ...
}

// ----- root_randomBit : __default__ :: RandomBit.Impl -----
active proctype process4_root_randomBit()
{
// *** Wait for first activation ***
FirstActivationOrReactivationWithoutModeHistory:
chan_root_randomBit?activate, _;
// Initialize Data Elements
d_step {
odpNew_root_randomBit_value = false;
odpStore_root_randomBit_value = false;
nominalMode_root_randomBit = modeID_m;
}
// Activation completed
chan_root_randomBit!activate, 0;
goto mode_m;
}

```

```

// *** Run as automaton & React to messages ***
mode_m:
do

// Proactive Transitions
:: nT(true) -> d_step {
  odpNew_root_randomBit_value = true;
  nominalMode_root_randomBit = modeID_m; }
  chan_root_randomBit!invalidate, 0; // Propagate changes, fT by root!
  goto mode_m;
:: nT(true) -> d_step {
  odpNew_root_randomBit_value = false;
  nominalMode_root_randomBit = modeID_m; }
  chan_root_randomBit!invalidate, 0; // Propagate changes, fT by root!
  goto mode_m;

// Incoming Message: update (from parent)
:: chan_root_randomBit?update, _ ->
  chan_root_randomBit!update, 0;

// Incoming Message: deactivate (from parent) - keep mode history
:: chan_root_randomBit?deactivate, _ ->
  chan_root_randomBit!deactivate, 0;
  chan_root_randomBit?activate, _;
  chan_root_randomBit!activate, 0;

// Incoming Message: reset (from parent)
:: chan_root_randomBit?reset, _ ->
  chan_root_randomBit!reset, 0;
  goto FirstActivationOrReactivationWithoutModeHistory;

od;
}

```

Listing 4.23: PROMELA code generated for the Negate Random Bit example from listing 2.1 on page 16.

Chapter 5

Translation of Properties to Never Claims

After having defined the translation of SLIM models to PROMELA programs in the previous chapter, this chapter focuses on the translation of SLIM properties, as introduced in section 2.6, to SPIN never claims, which were described in section 3.2.2. In principle, this could be done automatically: A SLIM property φ , consisting of a *pattern*(φ) and named atomic propositions $APs(\varphi)$, induces an LTL-formula which can be transformed to a never claim by SPIN (cf. section 3.2.3). However, this approach faces several problems:

- The transition system described by the PROMELA program has more states than the original transition system defined by the SLIM specification since it contains intermediate states which do not exist according to the SLIM semantics. For example, taking a transition in the SLIM semantics leads from one state directly to the successor state while the PROMELA program passes several intermediate states (evaluation of the guard, performing transition effects, adjusting activation of control subcomponents, sending invalidate, etc. – cf. section 4.4). Similarly, the communication protocols for multiway event communication and invalidate-update cycles (cf. sections 4.5 and 4.6, respectively) introduce several intermediate states. Let the term *stable states* refer to those states that exist in the transition system resulting from the SLIM semantics and the corresponding states of the PROMELA program, in opposite to *non-stable states* which only exist as intermediate states of the PROMELA program. This distinction is of utmost importance, since for model checking a property only the stable states are relevant while the non-stable states have to be “skipped”. Fortunately, stable states of the PROMELA program can be identified by the fact that the global flag `gflagGoNextTransition`, which was introduced in section 4.4.1 to guarantee the atomicity of transitions, has the value `false`. The LTL-formulas resulting from SLIM properties have to be augmented with this test for stable states: Instead of $\Box P$ for an universality global property, the formula $\Box(\neg\text{gflagGoNextTransition} \Rightarrow P)$ has to be used. Not quite analogously, for an existence global property $\Diamond(\neg\text{gflagGoNextTransition} \wedge P)$ replaces $\Diamond P$. Note that the formula $\Diamond(\neg\text{gflagGoNextTransition} \Rightarrow P)$, equivalent to $\Diamond(\text{gflagGoNextTransition} \vee P)$, is not correct, since this is already satisfied by the existence of a non-stable state.
- A rather technical problem arises from the fact that testing for stable states does not necessarily prevent that conditions are evaluated in non-stable states. The danger in that is that some conditions in non-stable states, in particular in those intermediate states

of the PROMELA program which occur before the initial stable state is reached, might crash. For example, consider the propositional property $100 / d > 2$ for some data port d . Let d have the default value 0 which is however already in the initial SLIM state overwritten with a positive value by a data port connection or flow. Having sequential evaluation as done by C – recall that SPIN generates a C program (PAN, cf. section 3.2) which performs the actual model checking – in mind, one could expect that for the LTL-formula $\text{gflagGoNextTransition} \cup (\neg \text{gflagGoNextTransition} \wedge \frac{100}{d} > 2)$ corresponding to the propositional property, the expression $\frac{100}{d} > 2$ is evaluated only in stable states. Unfortunately, SPIN sometimes changes the order of conditions in the generated never claim compared to the LTL-formula. Then, the expression $\frac{100}{d} > 2$ might be tested before the check on stable states take place. In the very first state of the PROMELA program where the data port updates have not yet been performed and thus d still has its default value 0, evaluating $\frac{100}{d}$ crashes due to division by zero. Consequently, it does not suffice to rely on sequential evaluation as the order of conditions in never claims cannot be controlled by the ordering of conditions in the LTL-formulas.

- Finally, the never claims automatically generated by SPIN from LTL-formulas are not optimal (cf. [6, p. 154]). This issue becomes worse by the fact that the LTL-formulas for properties – and with them the resulting never claims – became more difficult by introducing the test on stable states. Section 5.2.5 gives an example of an automatically generated never claim that unnecessarily requires cycle detection.

To solve or avoid these problems, for every LTL-property pattern used in COMPASS a never claim containing the corresponding placeholders was written by hand. Before these never claims are given in section 5.2, section 5.1 describes how the symbols used as placeholders for the atomic propositions are defined based on concrete atomic propositions.

5.1 Defining Symbols for Atomic Propositions

Since never claims can access variables from the PROMELA program of type boolean only (cf. section 3.2.2) while in SLIM-atomic propositions, in general, arbitrary data elements can be read, boolean symbols representing the atomic propositions – which themselves are boolean expressions – are defined in the PROMELA program. For every named atomic proposition $(id, prop) \in APs(\varphi)$ of a property φ , $AtomicPropositions(\varphi)$ defines a symbol $\text{atomicProposition}_{id}$ representing the evaluation of expression $prop$.

The atomic proposition $prop$ itself is translated to a PROMELA expression in a similar way as it is done by $Expression(c, expr)$ (cf. section 4.2.4) for normal SLIM expressions $expr$ in the context of a control component $c \in CCmp$. However, some differences must be taken into account:

- Atomic propositions are not evaluated in the context of a single control component but in the context of the whole system. From reading accesses to data elements $d \in Dat(c)$ of a control component $c \in CCmp$, denoted by prefixing d with the access path to c , i.e., $c.d$, not only the identifier d but also the context c must be extracted. The corresponding PROMELA variable is then formally given by $DataSotreVar(c, d)$. In the implementation however, the component hierarchy along the whole access path must be traversed to retrieve the instance of the control component c .
- Besides the values of data elements, the current nominal mode of control components $c \in CCmp$ can additionally be tested. A reading access $c.mode$ is translated to an access

of the variable *ModeVar(c)*. As error states have been translated to nominal modes of the specification resulting from model extension, these need not to be taken into account.

- The symbols defined for enumeration and mode identifiers (cf. section 4.2.1) must not be used in the definition of symbols for atomic propositions since the C preprocessor does not perform a textual replacement of previously defined symbols in later definitions of new symbols. Instead, the integer values used to represent enumeration and mode identifiers must be used directly.

5.2 Never Claims for Property Patterns

In section 3.2.2 the concept of never claims was introduced: They try to disprove the correctness by either terminating on finding a finite counterexample or by accepting infinite counterexamples containing acceptance cycle. This section presents the never claims that were developed for the COMPASS property patterns. To avoid linebreaks some abbreviations will be used: Instead of `atomicProposition_P` and `atomicProposition_S` the shorter versions `aP_P` and `aP_S`, respectively, will be used. Furthermore, `gflagGoNextTransition` is shortened to `gfgNT`.

5.2.1 Propositional

A property of the propositional pattern corresponds to the LTL-formula:

$$P$$

Augmented with the test for stable states the following LTL-formula results:

$$\text{gfgNT} \cup (\neg \text{gfgNT} \wedge P)$$

The manually written never claim presented in listing 5.1 just skips every non-stable state until the first, thus initial, stable state is reached. If *P* does not hold in this state, the property does not hold – otherwise it is valid.

```

never {
  go_on:
    if

    // *** As long as no stable state: go on ***
    :: gfgNT -> goto go_on

    // *** aP_P not fulfilled in first stable state: terminate ***
    :: !gfgNT && !(aP_P) -> goto stop

    // *** Otherwise: get stuck ***

    fi;

  stop:
    skip
}

```

Listing 5.1: Never claim for property pattern propositional.

5.2.2 Absence Global

A property of the absence global pattern corresponds to the LTL-formula:

$$\Box \neg P$$

Augmented with the test for stable states the following LTL-formula results:

$$\Box \neg (\neg \text{gfGNT} \wedge P)$$

The manually written never claim presented in listing 5.2 can terminate and thus wins as soon as it observes a stable state in which P holds.

```

never {
  go_on:
    if

    // *** If stable state found where aP_P holds: terminate ***
    :: !gfGNT && (aP_P) -> goto stop

    // *** "Otherwise": go on ***
    :: goto go_on

    fi;

  stop:
    skip
}

```

Listing 5.2: Never claim for property pattern absence global.

5.2.3 Existence Global

A property of the existence global pattern corresponds to the LTL-formula:

$$\Diamond P$$

Augmented with the test for stable states the following LTL-formula results:

$$\Diamond (\neg \text{gfGNT} \wedge P)$$

The manually written never claim presented in listing 5.3 tries to find an eventually cycling infinite execution path on which for every stable state P does not hold. In such an execution, P never holds in any stable state.

5.2.4 Universality Global

A property of the universality global pattern corresponds to the LTL-formula:

$$\Box P$$

Augmented with the test for stable states the following LTL-formula results:

$$\Box (\neg \text{gfGNT} \Rightarrow P)$$

```

never {
  accept_go_on:
    if

    // *** While no stable state found where aP_P holds: go on ***
    :: gfGNT || !(aP_P) -> goto accept_go_on

    // *** Otherwise: get stuck ***

    fi;
}

```

Listing 5.3: Never claim for property pattern existence global.

The manually written never claim presented in listing 5.4 terminates as soon as it observes a stable state where P does not hold, which violates the property.

```

never {
  go_on:
    if

    // *** Stable state found where aP_P does not hold ***
    :: !gfGNT && !(aP_P) -> goto stop

    // *** "Otherwise": go on ***
    :: goto go_on

    fi;

  stop:
    skip
}

```

Listing 5.4: Never claim for property pattern universality global.

5.2.5 Precedence Global

A property of the precedence global pattern corresponds to the LTL-formula:

$$(\neg S \cup P) \vee \square \neg S$$

Augmented with the test for stable states the following LTL-formula results:

$$((\neg \mathbf{gfGNT} \Rightarrow \neg S) \cup (\neg \mathbf{gfGNT} \wedge P)) \vee \square (\neg \mathbf{gfGNT} \Rightarrow \neg S)$$

The manually written never claim presented in listing 5.5 follows the consideration that the only possibility for the never claim to disprove the property is to find a stable state in which S holds before a stable state is seen where P holds.

```

never {
go_on:
  if

  // *** No stable state: go on ***
  :: gfGNT -> goto go_on

  // *** Stable state where neither P nor S hold: go on ***
  :: !gfGNT && !(aP_P) && !(aP_S) -> goto go_on

  // *** Stable state found where S holds but P does not ***
  :: !gfGNT && (aP_S) && !(aP_P) -> goto stop

  // *** Stable state found where P (and may be S) hold ***
  // get stuck

  fi;

stop:
  skip
}

```

Listing 5.5: Never claim for property pattern precedence global.

Comparison with the automatically generated never claim

As mentioned in the introduction of this chapter, one benefit of manually writing the never claims is to avoid unnecessarily difficult never claims resulting from the automatic transformation of LTL-formulas by SPIN. Listing 5.6 gives the automatically generated never claim for precedence global properties. It is not only much more difficult to understand but due to the until required in the LTL-representation of precedence global properties it requires cycle detection. As listing 5.5 of the manually generated never claim shows, this cycle detection – which increases complexity of model checking – can be avoided.

```

never {
/* !( [ ] (! gfGNT -> ! (aP_S)) || (( ! gfGNT -> ! aP_S) U (! gfGNT &&& aP_P))) */

T0_init:
  if
  :: (! ((aP_P)) && ! ((gfGNT)) && (aP_S)) -> goto accept_S3
  :: (! ((aP_P)) && ! ((gfGNT)) && (aP_S)) -> goto accept_all
  :: (((! ((aP_P))) || ((gfGNT)))) -> goto T0_init
  :: (! ((aP_P)) && ! ((gfGNT)) && (aP_S)) -> goto accept_S37
  fi;

accept_S3:
  if
  :: (((! ((aP_P))) || ((gfGNT)))) -> goto accept_S3
  :: (! ((aP_P)) && ! ((gfGNT)) && (aP_S)) -> goto accept_all
  fi;

```

```

accept_S37:
  if
  :: (! ((gfGNT)) && (aP_S)) -> goto accept_all
  :: (1) -> goto T0_S37
  fi;

T0_S37:
  if
  :: (! ((gfGNT)) && (aP_S)) -> goto accept_all
  :: (1) -> goto T0_S37
  fi;

accept_all:
  skip
}

```

Listing 5.6: More difficult never claim automatically generated by SPIN for properties of the pattern precedence global.

5.2.6 Response Global

A property of the response global pattern corresponds to the LTL-formula:

$$\Box(P \Rightarrow \Diamond S)$$

Augmented with the test for stable states the following LTL-formula results:

$$\Box((\neg \text{gfGNT} \wedge P) \Rightarrow \Diamond(\neg \text{gfGNT} \wedge S))$$

The manually written never claim presented in listing 5.7 uses nondeterminism to disprove the property: Whenever it observes a stable state where P but not directly S holds, it tries to find an infinite continuation of the execution so that S never holds in any successive stable state.

```

never {
go_on:
  if

  // *** Stable state where P but not S holds ***
  // *** => candidate for violation ***
  :: !gfGNT && (aP_P) && !(aP_S) -> goto accept_violation

  // *** "Otherwise": go on ***
  :: goto go_on

  fi;
}

```

```

accept_violation:
  if

  // *** No stable state OR S does not hold ***
  // *** => violation still possible ***
  :: gfGNT || !(aP_S) -> goto accept_violation

  // *** Otherwise (i.e., stable state found where S holds) ***
  // get stuck

  fi;
}

```

Listing 5.7: Never claim for property pattern response global.

5.3 Final Remarks

The storage of data elements in global PROMELA variables (cf. section 4.2) is tailored to the need of never claims which must be able to access them (via defined global symbols). Although the usage of global variables hinders statement merging (cf. section 3.3.1), it is preferable to the usage of local variables. The reason for that is twofold: First, the negative effects on statement merging can be reduced by explicitly enclosing statements that have only local effects in `d_step`-blocks. Second, if local variables are used, so-called *remote references* `name:var` would be necessary to access a local variable `var` inside the process instance of proctype `name` for the evaluation of properties reasoning about a data element stored in that variable. However, remote references are not only incompatible with statement merging but, worse, also with partial order reduction whose effects cannot be mimicked by adding `d_step`-blocks around statements in individual proctypes since partial order reduction considers sequences of global states, possibly changed by different processes. The evaluation of a first translation of SLIM specifications to PROMELA programs that made use of local variables clearly showed that the translation using global variables is more efficient with respect to model checking.

A similar reasoning applies to remote references of the form `name@label`, returning `true` when the program counter from the process instance of proctype `name` is currently at label `label`. Consequently, these remote references are not used to find out in which mode a component $c \in CCmp$ – respectively, the corresponding process – currently is. Instead, the mode information is stored in a separate variable `ModeVar(c)` (cf. section 4.2).

Part III

Slicing of SLIM specifications

Chapter 6

The Basic Slicing Approach

The practicability of model checking larger systems is restricted by the size of the induced state space. To combat the so-called *state-space explosion problem*, abstraction techniques must be employed. However, for some of the available abstraction techniques, like bisimulation minimisation, the whole state space of the system must be generated before it can be reduced by applying the abstraction. This might speed-up model checking but it does not reduce the peak memory requirements – a major bottleneck for the feasibility of model checking. Furthermore, such approaches annul the benefits of on-the-fly model checking as it is done with SPIN. Consequently, methods that operate on the system model before constructing its state space are preferable to those that try to minimise the resulting transition system. This chapter introduces the basics of such a method: a slicing approach for SLIM specifications. Afterwards, the subsequent chapter discusses necessary extensions and some optimisations.

6.1 Related Work

The term “slicing” has been coined by Weiser [42] who empirically observed it as a debugging technique used by experienced programmers to identify those parts of a program that affect a certain behaviour under consideration and to abstract from the irrelevant parts with respect to this behaviour. The initial approach for sequential programs was extended later on in several ways by many authors (cf. [39]). The most important work related to the approach presented here is the extension of slicing to concurrent programs by Cheng [14] and the application of slicing to software model checking including formal notions of correctness by Hatcliff, Dwyer and Zheng [21]. However, none of the methods presented there is directly applicable to the nested structures supporting dynamic reconfiguration and the various kinds of interaction between components as they are possible within SLIM specifications.

The principal idea of slicing is to remove all parts of a program, typically variables and statements, that do not influence the behaviour of interest, typically the values of some variables at some statements, described by a slicing criterion. To determine which parts are relevant, the transitive backward closure of the slicing criterion along different kinds of dependences, typically data and control dependences, is computed. However, finding a minimal sliced program is in general unsolvable since the halting problem can be reduced to it (cf. [42]).

Since version 3.4.0, released in 2000, SPIN contains a slicing algorithm [34, 26] for PROMELA programs with respect to a property, given as a LTL-formula or directly as a never claim. However, besides the fact that this slicing algorithm only makes suggestions about which parts of a PROMELA program could be removed but does not adapt the code automatically, SPIN’s slicing

algorithm turned out to be not really as effective as expected on PROMELA programs resulting from a SLIM specification. This is due to the fact that the fixpoint iteration generated by the translation for handling data port updates of control subcomponents (cf. section 4.6) introduces control dependences between PROMELA variables that do not exist between the data ports in the original SLIM specification. For example, consider a component hierarchy as outlined in figure 6.1 and a property only reasoning about `root.A.x`. Updating all subcomponents obviously could change the value of `root.A.x`, e.g., by a data flow from an incoming data port or from a subcomponent (not depicted). Since the value of `root.B.y` is forwarded by a data port connection to another subcomponent `C`, the translation creates a fixpoint iteration which repeatedly updates all subcomponents until the value of `root.B.y` does not change any more. Due to this fact, updating the subcomponents and thus possibly changing the interesting data element `root.A.x` becomes control dependent on `root.B.y`. Consequently, SPIN's slicing algorithm will not eliminate `root.B.y`. However, for the considered property only reasoning about `root.A.x`, the other subcomponents `B` and `C` are completely irrelevant and can indeed be sliced away. This unsatisfactory circumstance led to the efforts of developing a slicing algorithm which directly works on the SLIM specification.

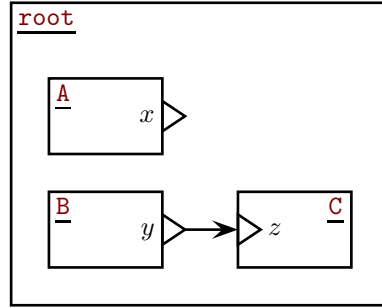


Figure 6.1: The fixpoint iteration makes the forwarded `root.B.y` relevant for SPIN's slicing algorithm – even for properties that reason only about `root.A.x`.

6.2 Component Instances

Note that, in analogy to distinguishing calling environments of procedures, slicing of SLIM specifications is done for individual component instances $c \in CCmp$ and not for their implementation $imp(c)$ or type $typ(c)$. This is more effective since different instances of the same component implementation or type might be sliced differently. Therefore, identical parts of different component instances are distinguished by associating the component instance with them. For example, in the following a mode m is perceived as a tuple (c, m') with $c \in CCmp$ and $m' \in Mod(c)$ a mode of the component instance c . Formally, the following sets are used:

- $Mod(c) := \{(c, m) \mid m \in Mod(c)\}$, $Mod := \bigcup_{c \in CCmp} Mod(c)$
- $IDPrT := \bigcup_{c \in CCmp} \{(c, idp) \mid idp \in IDPrT(c)\}$
- $DSub((c, m)) := \{(c, d) \mid d \in DSub(c, m)\}$
- $Dat(c) := \{(c, d) \mid d \in Dat(c)\}$, $Dat := \bigcup_{c \in CCmp} Dat(c)$

- $\mathcal{EPrt}(c) := \{(c, ep) \mid ep \in EPrt(c)\}$, $\mathcal{EPrt} := \bigcup_{c \in CCmp} \mathcal{EPrt}(c)$
- $\mathcal{CSub}((c, m)) := \mathcal{CSub}(c, m)$
- $\mathcal{MTr} := \bigcup_{c \in CCmp} \{((c, m_s), (c, t), g', f', (c, m_t)) \mid (m_s, t, g, f, m_t) \in MTr(c)\}$,
where g', f' result from g, f by extending every data element d occurring in them to (c, d)
- $\mathcal{Flw}((c, m)) := \{(c, d) := a' \mid (d, a) \in Flw(c, m)\}$, $\mathcal{Flw} := \bigcup_{m \in Mod} \mathcal{Flw}(m)$,
where a' results from a by extending every data element d occurring in it to (c, d)
- $\mathcal{DCon}((c, m)) := \{((c, sp) \rightarrow (c, tp)) \mid (sp, tp) \in DCon(c, m)\}$, $\mathcal{DCon} := \bigcup_{m \in Mod} \mathcal{DCon}(m)$
- $\mathcal{ECon}((c, m)) := \{((c, sp) \rightsquigarrow (c, tp)) \mid (sp, tp) \in ECon(c, m)\}$, $\mathcal{ECon} := \bigcup_{m \in Mod} \mathcal{ECon}(m)$
- $inv((c, m)) := inv(c, m)$, where the data elements d are extended to (c, d)

For simplicity the notation $(c, csc.x)$ for some $c \in CCmp$, $csc \in \mathcal{CSub}(c)$ and $x \in DPrt(c.csc) \cup EPrt(c.csc)$ is identified with $(c.csc, x)$. Similarly, no difference is made between $(c, d) \in \mathcal{Dat}$ and $(c, m) \in \mathcal{Mod}$, as used here, and $c.d$ or $c.mode:m$, respectively, as it occurs in properties.

6.3 Identifying Interesting Parts

Used as an abstraction technique before model checking a SLIM specification, slicing aims at removing those parts from a given specification S that are irrelevant for checking whether it fulfils a CTL* property φ . The resulting specification S_{sliced}^φ should be smaller than but still equivalent to S with respect to the property, that is, $S \models \varphi$ iff $S_{sliced}^\varphi \models \varphi$ (cf. [21]). Consequently, comparable to a slicing criterion, the property defines the initially interesting parts that must not be sliced away: data elements and modes used in φ (events are not allowed in SLIM properties but could be added). Subsequently, the transitive backward closure of the set of interesting parts, i.e., all other aspects that have an (indirect) influence on them and thus on the property, e.g., data elements read on the right hand side of assignments to an interesting data element, is calculated in a fixpoint iteration. Obviously this iteration always terminates but in the worst case all parts of the specification become interesting.

In the following three subsections the closure rules for adding data elements, events and modes to the set of interesting parts are described before the actual slicing algorithm is presented in pseudo-code.

6.3.1 Interesting Data Elements

Like with data flow dependence for classic program slicing, all data elements used to calculate a new value for an interesting data element are interesting, too. Here, this affects the right hand sides of assignments to an interesting data element, either in transition effects or by data flows. Similarly, the source ports of data port connections and flows targeting at interesting data ports become interesting as well. Furthermore, comparable to control flow dependence, all data elements used in guards on interesting transitions (see below) must be kept in the sliced specification as the evaluation of the guard at runtime determines whether the transition can indeed be taken.

6.3.2 Interesting Events

The main difference of SLIM specifications compared to sequential programs is that they describe a hierarchy of components that can synchronously communicate by sending and receiving events. Comparable to synchronisation and communication dependences (cf. [14]), all events used as triggers on interesting transitions are important. As events can be forwarded by event port connections, all events connected to an interesting event in any direction are interesting as well.

6.3.3 Interesting Modes

Similarly to the program location in classical slicing, the slicing algorithm for SLIM specifications does *not* treat the mode information as a data element which is either interesting or not at all, but tries to eliminate uninteresting modes. The difficulty is that the questions whether a mode, a data element or an event is interesting are related to each other since all those elements can be combined in the transition relation: On the one hand, transitions are (partially) interesting when they change an interesting data element, have an interesting trigger or their source or target mode is interesting. On the other hand, triggers, guards, source modes and possibly some data elements read in effects of those transitions are interesting. However, transitions themselves are not considered as elements of interest in the fixpoint iteration. Instead, modes are made interesting and with them implicitly all incoming and outgoing transitions. More concretely, besides the modes used in the property the following modes are interesting as well:

- Source modes of transitions changing an interesting data element. This obviously applies to transitions with assignments to interesting data elements in their effects but also to transitions reactivating an interesting data subcomponent, that is, it is active in the target mode but not in the source mode, since it will be reset to its default value in this case.
- All modes in which a data flow or a data port connection to an interesting data port or in which an event port connection to/from an interesting event port is active. This also guarantees that all transitions that deactivate a data flow to an interesting data element and thus reset it to its default value are included in the sliced specification.
- Source modes of transitions with interesting events as triggers because of their relevance for synchronous event communication which can only take place when a sent event can be received by at least one enabled transition in another component.

Moreover, the reachability of interesting modes from the initial mode matters. Thus, every predecessor of an interesting mode, that is, the source modes of transitions to interesting target modes, is interesting as well.

6.4 The Basic Slicing Algorithm

The pseudo-code description of the basic slicing algorithm is given in listing 6.1 on the next page. The algorithm computes the sets of interesting data elements (\mathfrak{D}), interesting event ports (\mathfrak{E}) and interesting modes (\mathfrak{M}).

```

/* Initialisation */
 $\mathcal{D} := \{d \in \mathcal{Dat} \mid d \text{ occurs in } \varphi\};$ 
 $\mathcal{E} := \emptyset;$ 
 $\mathfrak{M} := \{m \in \mathcal{Mod} \mid m \text{ occurs in } \varphi\};$ 

/* Fixpoint Iteration */
repeat

  /* Transitions that update/reactivate interesting data elements
  or have interesting triggers */
  for all  $m_s \xrightarrow{t,g,f} m_t \in \mathcal{MTr}$  with  $\exists d \in \mathcal{D} : f \text{ updates } d$ 
  or  $\exists d \in \mathcal{D} : d \in \mathcal{DSub}(m_t) \setminus \mathcal{DSub}(m_s)$ 
  or  $t \in \mathcal{E}$ 

  do
     $\mathfrak{M} := \mathfrak{M} \cup \{m_s\};$ 

  /* Transitions from/to interesting modes */
  for all  $m_s \xrightarrow{t,g,f} m_t \in \mathcal{MTr}$  with  $m_s \in \mathfrak{M}$  or  $m_t \in \mathfrak{M}$  do
     $\mathcal{D} := \mathcal{D} \cup \{d \in \mathcal{Dat} \mid g \text{ reads } d\}$ 
     $\cup \{d \in \mathcal{Dat} \mid f \text{ updates some } d' \in \mathcal{D} \text{ reading } d\};$ 
     $\mathcal{E} := \mathcal{E} \cup \{t\};$ 
     $\mathfrak{M} := \mathfrak{M} \cup \{m_s\};$ 

  /* Data flows to interesting data ports */
  for all  $d := a \in \mathcal{Flw}$  with  $d \in \mathcal{D}$  do
     $\mathcal{D} := \mathcal{D} \cup \{d' \in \mathcal{Dat} \mid a \text{ reads } d'\};$ 
     $\mathfrak{M} := \mathfrak{M} \cup \{m \in \mathcal{Mod} \mid d := a \in \mathcal{Flw}(m)\};$ 

  /* Data port connections to interesting data ports */
  for all  $d \rightarrow d' \in \mathcal{DCon}$  with  $d' \in \mathcal{D}$  do
     $\mathcal{D} := \mathcal{D} \cup \{d\};$ 
     $\mathfrak{M} := \mathfrak{M} \cup \{m \in \mathcal{Mod} \mid e \rightarrow e' \in \mathcal{DCon}(m)\};$ 

  /* Event port connections involving interesting event ports */
  for all  $e \rightsquigarrow e' \in \mathcal{ECon}$  with  $e \in \mathcal{E}$  or  $e' \in \mathcal{E}$  do
     $\mathcal{E} := \mathcal{E} \cup \{e, e'\};$ 
     $\mathfrak{M} := \mathfrak{M} \cup \{m \in \mathcal{Mod} \mid e \rightsquigarrow e' \in \mathcal{ECon}(m)\};$ 

until nothing changes;

```

Listing 6.1: The basic slicing algorithm.

6.5 The Basic Sliced Specification

After calculating the fixpoint, the sliced specification S_{sliced}^{φ} can be generated. Essentially, it contains only the interesting data elements (\mathcal{D}), the interesting event ports (\mathcal{E}) and the interesting modes (\mathcal{M}) plus data flows and connections to them, i.e., $d := a \in \mathcal{F}lw$ with $d \in \mathcal{D}$, $d \rightarrow d' \in \mathcal{D}Con$ with $d' \in \mathcal{D}$ and $e \rightsquigarrow e' \in \mathcal{E}Con$ where $e, e' \in \mathcal{E}$. The default values of data elements stay the same.

The sliced transition relation contains all transitions $m_s \xrightarrow{t,g,f} m_t \in \mathcal{M}Tr$ leaving an interesting mode $m_s \in \mathcal{M}$ with slight modifications: If the target mode is not interesting ($m_t \notin \mathcal{M}$), it is replaced by a dummy sink mode `_SlicedSinkMode` $\notin Mod$ which is added to every component that had uninteresting modes. Furthermore, only those transition effects $d := expr$ in f are retained that assign to an interesting data element, i.e., $d \in \mathcal{D}$. From the lists of modes in which connections (including flows) and subcomponents (both, control and data) are active all uninteresting modes are deleted. However, for subcomponents the `_SlicedSinkMode` must be added if the subcomponent was originally active in an uninteresting mode. This is necessary, since not all modes in which subcomponents are active become interesting but only those from which they are reactivated. Finally, all “empty” components, i.e., those that neither have interesting data elements, interesting event ports, interesting modes nor non-empty subcomponents, are completely removed in a bottom-up manner.

Since the next chapter introduces some relevant additional details and some optimisations for the slicing algorithm, a formal definition of the resulting sliced specification is postponed to section 7.7.

Chapter 7

Improvements of the Slicing Algorithm

After introducing the basic slicing algorithm for SLIM specifications in the previous chapter, the present chapter deals with improvements of it. Some of them are necessary extensions for the general correctness of the slicing algorithm:

- Considering reactivation of control components not supporting mode history but containing interesting data elements or modes.
- Preserving divergence characteristics for liveness properties.
- Extensions to language constructs for hybrid behaviour.

Other improvements are optimisations to yield smaller sliced specifications by a more fine-grained analysis on which elements should become interesting:

- Permanently active connections and flows do not need to make all modes interesting.
- Data elements interesting inside one component only are in fact “weak interesting”.

Each improvement is described in one of the following subsections, starting with the necessary ones, before the complete improved version of the slicing algorithm is given in pseudo-code.

7.1 Considering Reactivation of Control Subcomponents

In the basic slicing algorithm, transitions – more precisely: their source modes – changing an interesting data element become interesting. There, not only explicit changes by the transition’s effect but also implicit ones by reactivating data subcomponents – which are reset to their default value in this case – are considered. Actually, there is another form of implicit changes by transitions similar to the reactivation of data subcomponents: the reactivation of control subcomponents. Those that do not support mode history are reset in this case and with them all their data elements, that is, its incoming and outgoing data ports and its data subcomponents. The severity is that not only direct but also indirect control subcomponents, that means, transitively the subcomponents of subcomponents and so forth, must be considered. Furthermore, it does not suffice to consider whether such a transitive control subcomponent itself maintains mode history but the support for mode history of all subcomponents in between must be taken into

account. Figure 7.1 illustrates a situation where this is necessary: On reactivation of component **B** its transitive subcomponents **C** and **D** are reactivated as well. However, component **C** does not support mode history, thus all its transitive subcomponents, here **D**, are reset – no matter whether they support mode history or not. Obviously, this can change the values of incoming data ports, like x , data subcomponents, like y , and outgoing data ports, like z , of **D**.

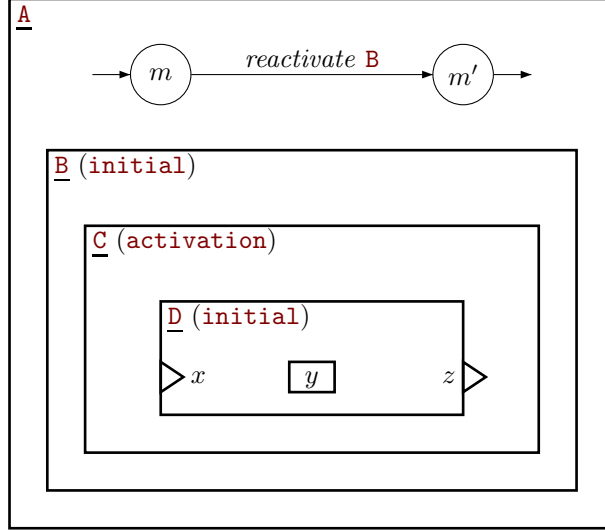


Figure 7.1: Reactivation of **B** induces reset of transitive control subcomponents **C** and **D**.

This reasoning extends to modes since on the reactivation of a transitive control subcomponent without transitive mode history its mode is reset to the start mode and this can affect the reachability of interesting modes. In contrast, interesting events are handled indirectly: If there is a transition using them as trigger or a forwarding connection not active in every mode of a transitive subcomponent, at least one mode of it became interesting. This interesting mode guarantees that reactivations of the transitive subcomponent are considered during slicing. If an interesting event is never received by a transition and always forwarded in the same way, if forwarded at all, then resets of the subcomponent do not matter.

For a formal characterisation, first of all the set of *transitive control subcomponents* of a control component $c \in CCmp$, denoted as $CSub^+(c)$, has to be defined. Instead of a recursive definition the prefix operator is applied: $c \sqsubseteq c'$ for $c, c' \in CCmp$ means c is a prefix of c' , i.e., $c' = c.c''$. Following the semantics of access paths, this states that c' is a direct or indirect control subcomponent of c .

$$CSub^+(c) := \{c' \in CCmp \mid c \sqsubseteq c'\}$$

Data elements and modes are reset in those transitive control subcomponents that do not have *transitive mode history*, that is, either they themselves do not support mode history or one of the transitive subcomponents occurring in between does not have mode history. The set $CSub_{-MH}^+(c)$ of transitive control subcomponents of a control component $c \in CCmp$ which do not have transitive mode history is defined as follows:

$$CSub_{-MH}^+(c) := \{c' \in CSub^+(c) \mid \exists c'' \in CCmp : c \sqsubseteq c'' \sqsubseteq c' \wedge \neg HasModeHistory(c'')\}$$

However, not every reactivation of such components is important. Reactivation of a transitive control subcomponent is relevant only, if this component contains a (strong) interesting data element $d \in \mathcal{D}$ or an interesting mode $m \in \mathfrak{M}$. Consequently, the considered set is limited to:

$$CSub_{-MH}^+|_{\mathcal{D}\mathfrak{M}}(c) := \{c' \in CSub_{-MH}^+(c) \mid \text{Dat}(c') \cap \mathcal{D} \neq \emptyset \vee \text{Mod}(c') \cap \mathfrak{M} \neq \emptyset\}$$

Now, a transition from mode m_s to $m_t \in \text{Mod}(c)$ in a control component $c \in CCmp$ might reset a (strong) interesting data element or mode if a control subcomponent is reactivated that is or contains a transitive control subcomponent of c with a (strong) interesting data element or mode but without transitive mode history:

$$\text{RelevantReactivation}(m_s, m_t) := \exists c' \in CSub_{-MH}^+|_{\mathcal{D}\mathfrak{M}}(c) : \text{first}_c(c') \in CSub(m_t) \setminus CSub(m_s)$$

For the formal definition the auxiliary function $\text{first}_c(c') := csc \in CSub(c)$, so that $c.csc \sqsubseteq c'$, is used which returns for a transitive control subcomponent c' of c the control subcomponent csc of c in which it is contained. For example, $\text{first}_A(A.B.C.D, A)$ is $A.B$ for the system depicted in figure 7.1.

Finally, a remark on reactivation of data subcomponents: Data subcomponents are reset when they are reactivated due to a transition within the control component which they belong to. However, when a control component with (transitive) mode history is reactivated as a whole, its data subcomponents are not reset but keep their previous value!

7.2 Preserving Divergence Characteristics

Since the semantics of SLIM does not impose any fairness assumptions, divergence characteristics of specifications must be considered. For safety properties (“nothing bad happens”), which can be falsified by finite execution path fragments, this is irrelevant, but for liveness properties (“eventually something good happens”) the possibility of divergence matters as the example in figure 7.2 demonstrates: The property “eventually component A reaches mode m' ” is valid only if component B is not present.

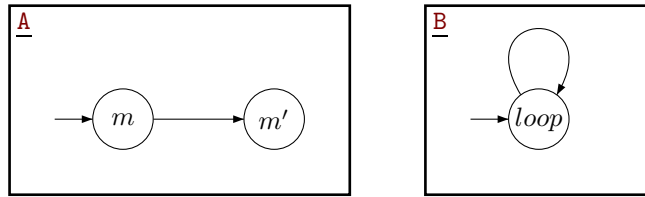


Figure 7.2: Component B can block component A by cycling ad infinitum in $loop$ and thus prevent that mode m' is reached by A .

To preserve divergence characteristics under slicing for a liveness property, every mode on a syntactical cycle is made interesting. This can be done in the initialisation of \mathfrak{M} by extending it to:

$$\mathfrak{M} := \{m \in \text{Mod} \mid m \text{ occurs in } \varphi \vee m \text{ lies on a syntactical cycle}\}$$

A mode $m \in \mathcal{Mod}(c)$ lies on a syntactical cycle when it can be reached from itself by transitively taking one or more transitions, i.e., $(m, m) \in \{(m_s, m_t) \mid (m_s, t, g, f, m_t) \in \mathcal{MTr}(c)\}^+$, where R^+ denotes the transitive closure of a binary relation R . Such cycles can be determined using “Tarjan’s Algorithm” (cf. [38]). The wording “syntactical” indicates that triggers and guards as well as mode invariants of intermediate modes are not considered. In fact, a mode on a syntactical cycle might turn out to be unreachable from itself according to the SLIM semantics, depending on triggers, guards, and invariants. In the simplest case, a self-loop with guard **false** is a syntactical cycle which does not result in a semantical one. Of course, a more sophisticated analysis could identify this and some more complex situations. However, in general it is undecidable whether a mode can be reached from itself since the halting problem can be reduced to this question (cf. undecidability of finding minimal slices as described in [42]).

7.3 Extensions for Hybrid Behaviour

Two syntactic constructs of the SLIM language that are used for describing hybrid behaviour (cf. section 2.2.1) were not considered in the slicing algorithm described so far: First of all, boolean conditions over the values of **clock** and **continuous** data subcomponents attached to modes as *mode invariants* constrain, on the one hand, whether a certain mode can be entered and, on the other hand, whether it is possible to stay in the current mode. The data subcomponents read in such mode invariants of interesting modes become interesting. Actually, it suffices to make them weak interesting (set \mathfrak{W}) as described shortly after in section 7.5. This guarantees that the referred data subcomponents exist in the sliced specification as well and that they have the same value as they would have in the unsliced specification – at least as far as interesting modes are considered. Second, *trajectory equations* added to mode invariants describe how data subcomponents d of type **continuous** change over time while the component is in the respective mode by giving a constant value for the derivation, i.e., $\dot{d} = r$ for some $r \in \mathbb{R}$.

Altogether, the following lines have to be added to the fixpoint iteration of the slicing algorithm:

```

/* Data subcomponents read in mode invariants */
for all  $m \in \mathfrak{M}$  do
   $\mathfrak{W} := \mathfrak{W} \cup \{d \in \mathcal{Dat} \mid \text{inv}(m) \text{ reads } d\}$ 

/* Modes changing strong interesting data subcomponents
   by trajectory equation */
for all  $d \in \mathcal{D}$  do
   $\mathfrak{M} := \mathfrak{M} \cup \{m \in \mathcal{Mod} \mid \dot{d} = r \in \text{inv}(m) \text{ for some } r \in \mathbb{R}\}$ 

```

In the end, the sliced specification contains only the interesting modes whose invariants are adapted as follows: The conditions stay unchanged but only those trajectory equations are included that talk about (strongly or weak) interesting data elements, that is:

$$\text{inv}_{\text{sliced}}(m) := \text{inv}(m) \setminus \{\dot{d} = r \in \text{inv}(m) \mid d \notin D \cup W\}.$$

Finally, the mode invariants $\text{inv}(m_t)$ of uninteresting modes $m_t \in \mathcal{Mod} \setminus \mathfrak{M}$ targeted by transitions from interesting modes $m_s \in \mathfrak{M}$ need a special treatment: They cannot simply be removed together with the uninteresting mode since they still constrain whether an interesting transition from m_s to m_t can be taken. One solution is to make the target modes, and not only the source modes, of interesting transitions interesting as well. To avoid these additional interesting modes, the boolean conditions $b \in \text{inv}(m_t) \cap \text{BExpr}$ from the invariant of a target

mode m_t can be added to the guard g of every transition leading to this mode, instead. This way, the possibilities of taking the transition are constrained as well. However, changes by the transition's effect f to the data elements occurring in the boolean conditions must be anticipated by a corresponding substitution, denoted $b[f]$. All this can be done in a preprocessing step so that the slicing algorithm itself needs no further extensions:

$$\begin{aligned} \mathcal{MTr} &:= \{(m_s, t, g', f, m_t) \mid (m_s, t, g, f, m_t) \in \mathcal{MTr}\}, \\ \text{where } g' &:= (g) \wedge \bigwedge_{b \in \text{inv}(m_t) \cap BExpr} (b[f]) \end{aligned}$$

7.3.1 No Adaption for Extended Models

In analogy to the extensions for hybrid behaviour, one could expect that the additional language constructs used for model extension (cf. section 2.5) require additional treatment by the slicing algorithm – but that is not the case. Error event ports are handled like normal outgoing event ports. The only difference is that the distribution parameter has to be preserved, but that does not change the slicing approach. Every other language construct used in an extended model is valid for nominal models without error models as well. In particular, the `reset` trigger of components with error model is transformed to `_errorSubcomponent.reset` for a new subcomponent `_errorSubcomponent` having an incoming event port `reset`.

7.4 Permanently active connections and flows do not make modes interesting

Since connections and flows targeting at interesting ports are important, all modes in which they are active are made interesting by the basic slicing algorithm. This guarantees, that their activation status is preserved in the sliced specification. However, when a connection or flow is active in every mode of the component it belongs to, all of them become interesting. This is excessive since the information that the connection or flow is always active would suffice.

This consideration leads to the following two changes in the slicing algorithm: Firstly, the modes in which a connection or flow to an interesting port is active are made interesting only when it is not active in every mode. For the formal definition the auxiliary function `RealModeSubset` is used to filter out those sets containing all modes of a component $c \in CCmp$. For a set of modes $M = \{(c, m_0), \dots, (c, m_n)\} \subseteq Mod(c)$ it is defined as follows:

$$\text{RealModeSubset}(M) := \begin{cases} M & , M \subsetneq Mod(c) \\ \emptyset & , \text{otherwise} \end{cases}$$

The basic slicing algorithm is changed by applying this filter whenever the modes in which interesting connections or flows are active are made interesting. That means, instead of lines like

$$\mathfrak{M} := \mathfrak{M} \cup \{m \in Mod \mid \text{connection/flow is active in } m\}$$

an instruction like

$$\mathfrak{M} := \mathfrak{M} \cup \text{RealModeSubset}(\{m \in Mod \mid \text{connection/flow is active in } m\})$$

is used.

Secondly, it is necessary to adapt the original list of modes in which a connection or flow is active for the sliced specification as follows: All interesting modes are kept while the uninteresting

modes are removed from the list. If at least one uninteresting mode was contained in the list – in that case, all modes must have been in there – the sink mode is added. This yields the desired effect – for both, connections and flows that are not, and those that are always active.

7.5 Weak Interesting Data Elements

Data and control dependencies make parts of the specification interesting which were not interesting in the first place. Sometimes, these make again other aspects interesting as described in the fixpoint iteration of the basic slicing algorithm. However, this is not always necessary. Consider the example shown in figure 7.3 with a property reasoning about the reachability of mode m_1 . Obviously the value of the data subcomponent ok must be known in mode m_0 because it occurs in the guard of the transition from m_0 to m_1 . Due to the fact that the value of ok is changed by the transition from m_2 to m_3 , the mode m_2 and all data elements read in $expr$ become interesting as well. But actually, this change of ok is irrelevant for the reachability of mode m_1 and consequently too many things were unnecessarily made interesting.

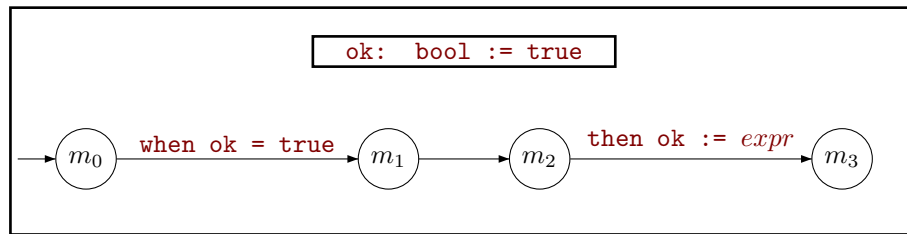


Figure 7.3: The reachability of mode m_1 makes data subcomponent ok weak interesting, not (strong) interesting.

To avoid such situations, a distinction is made between (*strong*) *interesting* and *weak interesting* data elements. Weak interesting data elements are data elements which are needed to evaluate guards – like ok in the previous example – or right hand sides of assignments within a component as long as it is inside an interesting mode – like m_1 and its predecessor m_0 in the mentioned example. In uninteresting modes, changes to weak interesting data elements – as by the transition from m_2 to m_3 in the given example – do not matter and thus no other elements are made interesting by such changes. Due to the differentiation between interesting and uninteresting modes of the control component to which a data element belongs, the notion weak interesting works only within a component. Consequently, only data subcomponents and outgoing data ports – as long as they are not forwarded and read by another component – can be weak interesting. Contrariwise, incoming data ports can only be (strong) interesting but not weak interesting since their value is provided from outside the component.

This improvement is implemented in the slicing algorithm by the introduction of a new set \mathfrak{W} collecting the weak interesting data elements while \mathfrak{D} denotes the set of (strong) interesting data elements. Furthermore, the inclusions of data elements read in guards and right hand sides of assignments to the set \mathfrak{D} are changed to inclusions to the set \mathfrak{W} . However, every weak interesting incoming data port is automatically (strong) interesting since the notion of weak interesting is restricted to single components. For the same reason, all source ports of data port connections and flows to weak interesting (outgoing) data ports are made (strong) interesting, not weak interesting. In opposite to data port connections or flows to (strong) interesting data ports, the modes in which such data port connections or flows are active do not become interesting.

The resulting sliced specification contains all (strong) interesting and all weak interesting data elements. However, for the weak interesting data elements only the assignments to them taking place in the interesting part of the modes is preserved while all others are removed together with the uninteresting modes themselves. Finally, the list of modes in which data port connections or flows to only weak interesting outgoing data ports as well as the list of modes in which only weak interesting data subcomponents – or – control subcomponents transitively containing weak interesting but neither (strong) interesting data elements nor modes are active have to be adapted by removing all uninteresting modes. Again, this can be done since the changes that would take place in uninteresting modes are irrelevant.

7.6 The Improved Slicing Algorithm

Listing 7.1 shows the initialisation and listing 7.2 the fixpoint iteration of the complete improved slicing algorithm.

```

/* Initialisation */
 $\mathcal{D} := \{d \in \mathcal{Dat} \mid d \text{ occurs in } \varphi\};$ 
 $\mathcal{W} := \emptyset;$ 
 $\mathcal{E} := \emptyset;$ 
 $\mathcal{M} := \{m \in \mathcal{Mod} \mid m \text{ occurs in } \varphi \vee m \text{ lies on a syntactical cycle}\};$ 

/* Fixpoint Iteration */
repeat
  ... see listing 7.2 ...
until nothing changes;

```

Listing 7.1: Initialisation of the improved slicing algorithm.

7.7 The Improved Sliced Specification

After the informal description of the basic resulting sliced specification in section 6.5, this section gives a formal definition of S_{sliced}^φ , including all discussed extensions and improvements.

Control Components First of all, all control components with strong interesting data elements¹, interesting events or interesting modes are interesting and thus contained in the sliced specification.

$$CCmp_{sliced} := \{c \in CCmp \mid \mathcal{D}(c) \cup \mathcal{E}(c) \cup \mathcal{M}(c) \neq \emptyset\},$$

where $\mathcal{D}(c) := \mathcal{D} \cap \mathcal{Dat}(c)$
 $\mathcal{E}(c) := \mathcal{E} \cap \mathcal{EPrt}(c)$
 $\mathcal{M}(c) := \mathcal{M} \cap \mathcal{Mod}(c)$

Of course, this propagates bottom-up to the supercomponents:

$$c.csc \in CCmp_{sliced} \Rightarrow c \in CCmp_{sliced}$$

¹Weak interesting data elements do not have to be considered because they can exist only if at least some (strong) interesting element exists in the component.

```

/* Transitions that update/reactivate strong interesting data elements,
   perform a relevant reactivation or have interesting triggers */
for all  $m_s \xrightarrow{t,g,f} m_t \in \mathcal{MTr}$  with  $\exists d \in \mathcal{D} : f$  updates  $d$ 
   or  $\exists d \in \mathcal{D} : d \in \mathcal{DSub}(m_t) \setminus \mathcal{DSub}(m_s)$ 
   or RelevantReactivation( $m_s, m_t$ )
   or  $t \in \mathcal{E}$ 
do  $\mathfrak{M} := \mathfrak{M} \cup \{m_s\}$ ;

/* Modes changing strong interesting data subc. by trajectory equation */
for all  $d \in \mathcal{D}$  do
    $\mathfrak{M} := \mathfrak{M} \cup \{m \in \mathcal{Mod} \mid \dot{d} = r \in \text{inv}(m) \text{ for some } r \in \mathbb{R}\}$ 

/* Transitions from/to interesting modes */
for all  $m_s \xrightarrow{t,g,f} m_t \in \mathcal{MTr}$  with  $m_s \in \mathfrak{M}$  or  $m_t \in \mathfrak{M}$  do
    $\mathfrak{W} := \mathfrak{W} \cup \{d \in \mathcal{Dat} \mid g \text{ reads } d\}$ 
    $\cup \{d \in \mathcal{Dat} \mid f \text{ updates some } d' \in (\mathcal{D} \cup \mathfrak{W}) \text{ reading } d\}$ ;
    $\mathcal{E} := \mathcal{E} \cup \{t\}$ ;
    $\mathfrak{M} := \mathfrak{M} \cup \{m_s\}$ ;

/* Data subcomponents read in mode invariants */
for all  $m \in \mathfrak{M}$  do
    $\mathfrak{W} := \mathfrak{W} \cup \{d \in \mathcal{Dat} \mid \text{inv}(m) \text{ reads } d\}$ 

/* Weak interesting incoming data ports are strong interesting */
 $\mathcal{D} := \mathcal{D} \cup (\mathcal{IDPr} \cap \mathfrak{W})$ ;

/* Data flows to strong interesting data ports */
for all  $d := a \in \mathcal{Flw}$  with  $d \in \mathcal{D}$  do
    $\mathcal{D} := \mathcal{D} \cup \{d' \in \mathcal{Dat} \mid a \text{ reads } d'\}$ ;
    $\mathfrak{M} := \mathfrak{M} \cup \text{RealModeSubset}(\{m \in \mathcal{Mod} \mid d := a \in \mathcal{Flw}(m)\})$ ;

/* Data flows to weak interesting data ports */
for all  $d := a \in \mathcal{Flw}$  with  $d \in \mathfrak{W}$  and  $\exists m \in \mathfrak{M} : d := a \in \mathcal{Flw}(m)$  do
    $\mathcal{D} := \mathcal{D} \cup \{d' \in \mathcal{Dat} \mid a \text{ reads } d'\}$ ;

/* Data port connections to strong interesting data ports */
for all  $d \rightarrow d' \in \mathcal{DCon}$  with  $d' \in \mathcal{D}$  do
    $\mathcal{D} := \mathcal{D} \cup \{d\}$ ;
    $\mathfrak{M} := \mathfrak{M} \cup \text{RealModeSubset}(\{m \in \mathcal{Mod} \mid d \rightarrow d' \in \mathcal{DCon}(m)\})$ ;

/* Data port connections to weak interesting data ports */
for all  $d \rightarrow d' \in \mathcal{DCon}$  with  $d' \in \mathfrak{W}$  and  $\exists m \in \mathfrak{M} : d \rightarrow d' \in \mathcal{DCon}(m)$  do
    $\mathcal{D} := \mathcal{D} \cup \{d\}$ ;

/* Event port connections involving interesting event ports */
for all  $e \rightsquigarrow e' \in \mathcal{ECon}$  with  $e \in \mathcal{E}$  or  $e' \in \mathcal{E}$  do
    $\mathcal{E} := \mathcal{E} \cup \{e, e'\}$ ;
    $\mathfrak{M} := \mathfrak{M} \cup \text{RealModeSubset}(\{m \in \mathcal{Mod} \mid e \rightsquigarrow e' \in \mathcal{ECon}(m)\})$ ;

```

Listing 7.2: Fixpoint iteration of the improved slicing algorithm.

Modes A sliced control component $c \in CCmp_{sliced}$ basically contains all its interesting modes and a dummy sink mode `_SlicedSinkMode` $\notin Mod$ collectively representing all uninteresting modes if such one(s) exist(s):

$$Mod_{sliced}(c) := \begin{cases} \{m \in Mod(c) \mid (c, m) \in \mathfrak{M}\} \\ \cup \begin{cases} \text{_SlicedSinkMode} & , \exists m \in Mod(c) : (c, m) \notin \mathfrak{M} \\ \emptyset & , \text{otherwise} \end{cases} \end{cases}$$

The starting mode stays unchanged, in particular its type, i.e., `initial` or `activation`. Only if the original starting mode is not interesting – in this case, no mode at all is interesting since the starting mode is a (transitive) predecessor of every mode and would thus be included by the fixpoint iteration as soon as some mode is interesting – the new mode `_SlicedSinkMode` is set as starting mode. Its type is copied from the original starting mode.

$$stm_{sliced}(c) := \begin{cases} stm(c) & , (c, stm(c)) \in \mathfrak{M} \\ \text{_SlicedSinkMode} & , \text{otherwise} \end{cases}$$

The boolean conditions in invariants of interesting modes $m \in Mod_{sliced}(c) \setminus \{\text{_SlicedSinkMode}\}$ are completely preserved while only those trajectory equations are included that talk about strong or at least weak interesting data elements:

$$inv_{sliced}(c, m) := (inv(c, m) \cap BExpr) \cup \{d = r \in inv(c, m) \mid (c, d) \in \mathfrak{D} \cup \mathfrak{W}\}$$

The invariant of the new mode `_SlicedSinkMode` is empty, i.e., always fulfilled:

$$inv_{sliced}(c, \text{_SlicedSinkMode}) := \emptyset$$

Data Elements All strong and weak interesting data elements are included in a sliced control component $c \in CCmp_{sliced}$. While incoming data ports are always strong interesting as soon as they become weak interesting, outgoing data ports and data subcomponents can indeed be only weak but not strong interesting, requiring to consider the union of \mathfrak{D} and \mathfrak{W} for them.

$$\begin{aligned} DSub_{sliced}(c, m) &:= \{dsc \in DSub(c, m) \mid (c, dsc) \in \mathfrak{D} \cup \mathfrak{W}\} \\ DSub_{sliced}(c, \text{_SlicedSinkMode}) &:= \{dsc \in DSub(c) \mid \\ &\quad (c, dsc) \in \mathfrak{D} \wedge \exists m \in Mod(c) : ((c, m) \notin \mathfrak{M} \wedge dsc \in DSub(c, m))\} \\ IDPrt_{sliced}(c) &:= \{idp \in IDPrt(c) \mid (c, idp) \in \mathfrak{D}\} \\ ODPrt_{sliced}(c) &:= \{odp \in ODPrt(c) \mid (c, odp) \in \mathfrak{D} \cup \mathfrak{W}\} \end{aligned}$$

For data subcomponents, the modes in which they are active must be adapted: Both, strong and weak interesting data subcomponents are obviously active in every interesting original mode $m \in Mod_{sliced}(c) \setminus \{\text{_SlicedSinkMode}\}$ in which they were active in the unsliced specification. Furthermore, strong interesting data subcomponents have to be active in the new `_SlicedSinkMode` if they were active in some uninteresting mode of the original specification. This results from the fact that not all modes in which strong interesting data elements are active become interesting by the fixpoint iteration but only those from which a strong interesting data element is changed, either by explicit transition effect, by the mode's invariant or by an outgoing transition reactivating the data element. Consequently, a strong interesting data element can indeed be active in an uninteresting mode. However, it is guaranteed that this data element is never changed again as soon as an uninteresting mode is entered. In particular, no assignment to and no reset by reactivation of strong interesting data elements can occur. The only change

that might happen to a strong interesting data element in uninteresting modes is that it becomes deactivated. But that can be ignored as properties must not reason about deactivated elements. If the property considers the mode of components to find out whether a certain data subcomponent is active, these modes are guaranteed to be contained in the sliced specification since every mode occurring in the property is made interesting in the initialisation of the slicing algorithm. Hence, it is safe to make the data element active in every uninteresting mode, i.e., in `_SlicedSinkMode`. Weak interesting data elements – whose values are interesting only while their defining component is in interesting modes – do not have to be active in `_SlicedSinkMode`.

Event Ports Analogously to data ports, all interesting event ports are included in the sliced component:

$$\begin{aligned} IEPr_{sliced}(c) &:= \{iep \in IEPr(c) \mid (c, iep) \in \mathfrak{E}\} \\ OEPr_{sliced}(c) &:= \{oep \in OEPr(c) \mid (c, oep) \in \mathfrak{E}\} \end{aligned}$$

Control Subcomponents The set of control subcomponents for a sliced control component $c \in CCmp_{sliced}$ can be derived from the set of all sliced control components. However, the modes in which such control subcomponents are active must additionally be given. For interesting modes $m \in Mod_{sliced}(c) \setminus \{_SlicedSinkMode\}$ the original activation status is preserved:

$$CSub_{sliced}(c, m) := \{csc \in CSub(c, m) \mid c.csc \in CCmp_{sliced}\}$$

In the sliced sink mode, all control subcomponents having or being a transitive control subcomponent with an interesting data element, an interesting mode or with an interesting event – or simply: control subcomponents contained in the sliced specification – that were active in at least one uninteresting original mode are active. On the one hand, this guarantees that components are not disabled too soon and that the possible behaviour is not restricted too much. On the other hand, possible later deactivations of control subcomponents do not matter – comparable to deactivations of strong interesting data elements in uninteresting modes. Furthermore, as discussed in section 7.1, no relevant reactivation of control subcomponents can take place in uninteresting modes. In particular it is important, that for control subcomponents which are not active in every original mode no (data and event) communication between it and its supercomponent can take place in uninteresting modes of the supercomponent.

$$\begin{aligned} CSub_{sliced}(c, _SlicedSinkMode) &:= \{csc \in CSub(c) \mid \\ &\quad c.csc \in CCmp_{sliced} \wedge \exists m \in Mod(c) : ((c, m) \notin \mathfrak{M} \wedge csc \in CSub(c, m))\} \end{aligned}$$

Transitions The sliced transition relation of a sliced control component $c \in CCmp_{sliced}$ contains all transitions $m_s \xrightarrow{t, g, f} m_t \in MTr$ leaving an interesting mode $(c, m_s) \in \mathfrak{M}$ with slight modifications: If the target mode is not interesting, i.e., $(c, m_t) \notin \mathfrak{M}$, it is replaced by `_SlicedSinkMode`. Furthermore, only those transition effects $d := expr$ in f are retained that assign to a strong or weak interesting data element, i.e., $(c, d) \in \mathfrak{D} \cup \mathfrak{W}$. In fact, for transitions to uninteresting target modes only assignments to strong interesting data elements are needed.

$$MTr_{sliced}(c) := \{(m_s, t, g, \text{adaptEffect}_c(f, m_t), \text{adaptTargetMode}_c(m_t)) \mid (m_s, t, g, f, m_t) \in MTr(c) \text{ with } (c, m_s) \in \mathfrak{M}\},$$

where

$$\begin{aligned} \text{adaptEffect}_c(f, m_t) &:= \begin{cases} \{d := expr \in f \mid (c, d) \in \mathfrak{D} \cup \mathfrak{W}\} & , (c, m_t) \in \mathfrak{M} \\ \{d := expr \in f \mid (c, d) \in \mathfrak{D}\} & , (c, m_t) \notin \mathfrak{M} \end{cases} \\ \text{adaptTargetMode}_c(m_t) &:= \begin{cases} m_t & , (c, m_t) \in \mathfrak{M} \\ _SlicedSinkMode & , \text{otherwise} \end{cases} \end{aligned}$$

The `_SlicedSinkMode` indeed does not need any outgoing transition as it is impossible to change an interesting data element, to send or receive an interesting event or to reach an interesting mode as soon as an uninteresting mode – which is represented by `_SlicedSinkMode` – has been entered by the original component.

Flows & Connections For every interesting mode $m \in Mod_{sliced}(c) \setminus \{_SlicedSinkMode\}$ the active flows to strong or weak interesting data elements are kept in the sliced specification:

$$Flw_{sliced}(c, m) := \{(d, a) \in Flw(c, m) \mid (c, d) \in \mathfrak{D} \cup \mathfrak{W}\}$$

Furthermore, the flows to strong interesting data elements that are active in every original mode are kept active in the sliced sink mode:

$$Flw_{sliced}(c, _SlicedSinkMode) := \{(d, a) \in Flw(c) \mid (c, d) \in \mathfrak{D} \wedge \exists m \in Mod(c) : ((c, m) \notin \mathfrak{M} \wedge (a, d) \in Flw(c, m))\}$$

The same applies for data port connections:

$$\begin{aligned} DCon_{sliced}(c, m) &:= \{(sp, tp) \in DCon(c, m) \mid (c, tp) \in \mathfrak{D} \cup \mathfrak{W}\} \\ DCon_{sliced}(c, _SlicedSinkMode) &:= \{(sp, tp) \in DCon(c) \mid (c, tp) \in \mathfrak{D} \wedge \exists m \in Mod(c) : ((c, m) \notin \mathfrak{M} \wedge (sp, tp) \in DCon(c, m))\} \end{aligned}$$

Analogously, event port connections are handled:

$$\begin{aligned} ECon_{sliced}(c, m) &:= \{(sp, tp) \in ECon(c, m) \mid (c, tp) \in \mathfrak{E}\} \\ ECon_{sliced}(c, _SlicedSinkMode) &:= \{(sp, tp) \in ECon(c) \mid (c, tp) \in \mathfrak{E} \wedge \exists m \in Mod(c) : ((c, m) \notin \mathfrak{M} \wedge (sp, tp) \in ECon(c, m))\} \end{aligned}$$

7.7.1 Retransformation to SLIM code

The resulting specification S_{sliced}^φ is again a valid SLIM specification (except the fact that only syntactically required buses which do not have any semantical effect are missing). In particular, every object referenced in it is indeed declared as it was included in the fixpoint iteration, e.g., the data elements used in the guards of interesting transitions. Thus, S_{sliced}^φ can be retransformed to SLIM code. However, for each component instance an own component type and implementation must be created since component instances which were of the same type or implementation in the original specification might differ after slicing (cf. section 6.2). This might result in a longer SLIM code than for the unsliced specification, but that does not have any negative effects on the induced state vector and state space sizes. To achieve that, the original component and implementation names are overwritten:

$$\begin{aligned} cat_{sliced}(c) &:= UAccessPath(c) \\ imp_{sliced}(c) &:= UAccessPath(c).Impl \\ typ_{sliced}(c) &:= typ(c) \end{aligned}$$

7.8 Possible Further Refinements

Beyond the optimisations presented so far, one can think of further refinements. Three of them are briefly discussed here:

- The mode information of a component could be treated as a data element that is either interesting or not – but it was not. Instead, uninteresting modes were identified and sliced away. This approach could be extended to the values of data elements to realise data abstraction: The type of a data element determines which values it theoretically can have. However, using data flow analysis, the slicing algorithm should be able to determine which values a data element possibly can have – and which ones not. For example, the data ports `x`, `y` and `sum` of the `IntegerAdder` component presented in section 9.2 are of type `int` but can only contain the values $0, \dots, 30$ or $0, \dots, 60$, respectively. This information could be used to change the data type to a smaller one and thus to reduce the memory required to store an individual state. However, the overall state space size is not reduced by this. Therefore it would be necessary to combine different values with somehow the same meanings, that is, to partition the value domain into equivalence classes.
- In the presented versions of the slicing algorithm, every predecessor of an interesting mode is made interesting. However, perhaps it is possible to join some intermediate modes, which do not affect the validity of a property, that occur between really interesting modes.
- Instead of making all modes interesting in which an interesting connection or flow is active in the original specification, this is done in the improved slicing algorithm only for connections and flows that are not active in every mode which avoids that unnecessarily every mode becomes interesting. This idea can be ported to trajectory equations: For those (strong) interesting data elements to which the same derivation is assigned in every mode, no mode has to be made interesting based on the attached trajectory equation. Instead, a corresponding trajectory equation must be added to the `._SlicedSinkMode`.

Part IV

Evaluation and Conclusions

Chapter 8

Implementation and Practical Issues

The translation of SLIM specifications to PROMELA programs, as described in part II, and the slicing algorithm for SLIM specifications, that was introduced in part III, were implemented in Python [2], the programming language used for the COMPASS project. This gave allowance to reuse the SLIM parser and the code for model extension that were developed within the COMPASS project. About 5,900 lines of new code were written by the author. A lot of them are dealing with uninteresting practical issues such as the elimination of underscores in the names occurring in the component hierarchy, necessary because the underscore is needed in the translation to separate the identifiers on an access path (cf. *UAccessPath(c)* in section 4.1), or filling up of empty `in modes` lists which implicitly contain every mode.

This chapter gives a short overview of the implementation and how to use it and concludes with some other lessons learned about working in projects.

8.1 How to perform Model Checking

The translation of SLIM specifications to PROMELA programs is implemented in the module `translation.py` in the COMPASS-repository in the path `./Prototype/Code/trunk/tools/-slim_compiler/src/promela`. It contains the convenience function `doTranslation` which, if it is provided with the `root`'s component implementation and a list of named atomic propositions, returns the PROMELA code.

In order to perform model checking of SLIM specifications with SPIN in an easy way, based on this translation to PROMELA, the Python program `script.py`, which can be found in the same location, controls all necessary steps. It can be run with the following main parameters:

```
python script.py (name of text file containing SLIM declarations)+  
  [ -p name of XML file containing properties ]  
  [ -f name of XML file describing fault injections ]
```

Additional parameters are:

- For Fault Injection / Model Extension:
 - `--storefaultinjectionresult`: The extended SLIM specification is stored in a file.

- For Slicing:
 - `--noslicing`: Disables slicing.
 - `--cycleslicing=(on|off|auto)`: Determines whether modes on syntactical cycles are always (`on`), never (`off`) or automatically whenever needed, depending on the property (`auto`, default), included in the sliced specification.
 - `--storeslicingresult`: If given, the SLIM specification resulting from slicing is written to a file.
 - `--verboseslicing`: Prints out verbose information about the slicing process to screen.
- For Memory Management (forwarded to SPIN):
 - `--collapse`: Activates SPIN's state vector compression.
 - `--ma=int`: Tells SPIN to use the minimized automaton representation for the state space; needs upper bound *int* for the sizes of the state vectors.
 - `--memlim=int`: Limits the memory that should be used by SPIN to *int* megabytes.
- Miscellaneous:
 - `-r root`: Name of the root's component implementation. Necessary only if no unique component implementation that is never used as a subcomponent is contained in the specification.
 - `-v`: Prints out verbose information to screen.
 - `--simulate=length`: Adds `printfs` to the generated PROMELA code and performs simulation with SPIN, bounded to the given *length*.
 - `--sdstart=,--sdfactor=,--sdadd=,--sdmax=int`: Used to iteratively try out increasing search depth bounds for SPIN.
 - `--nospin`: Disables invocation of SPIN.

In order to make `script.py` work correctly, SPIN must be installed in a path contained in the search path.

8.2 How to invoke Slicing

The slicing of SLIM specifications with respect to a list of atomic propositions is implemented in the module `slicing.py` in the COMPASS-repository under `./Prototype/Code/trunk/tools/-slim_compiler/src/slicing`. It contains the convenience function `doSlicing` which, if it is provided with the `root`'s component implementation and a list of atomic propositions, returns the sliced SLIM specification.

8.3 Other Lessons Learned

The reader might perceive the work presented in this thesis as not very complex and difficult. Indeed, a lot of the handling time went into implementation and debugging. Although I learned a lot – especially on coordination issues – while being affiliated to the international COMPASS project team with developers in Aachen, Germany and Trento, Italy, the fact that the project was still ongoing made the implementation more difficult. First of all, the source code that I reused

was not-yet documented. With Python, in which variables and parameters are not declared to be of a certain type, this meant a lot of try-and-error combined with introspection to find out how to use it. Second, the sources were still subjects to changes whenever a new demand or a conceptual problem arised in the project. Due to the fact, that my work was only affiliated to but not part of the official COMPASS project, I had to follow these changes nearly without any influence on them. Third, I had to do some kind of beta-testing: By reusing the COMPASS code, some of the bugs in there became bugs of my own program. Similarly, by comparing the model checking results from SPIN with those produced by the COMPASS toolset, some questions arised which had to be discussed in the team – a process which itself takes time. The following lists of tickets that I opened illustrates this situation:

- #230 “Strange Model-Checking Results”, which led to the inclusion of deadlock checking in the COMPASS GUI.
- #299 “Not checked: In Data Ports of root must have default value (G-5)”, a missing check in the parser.
- #300 “Default value for root’s in data ports is not used”, which revealed that the translation from SLIM to SMV did not consider those values.
- #313 “Flow of integer comparison to boolean out data port breaks simulation/deadlock checking/model checking”, pointing to a bug in the SLIM parser’s typing system.
- #320 “Missing checkbox “model extended by Fault Injections” for Deadlock Checking”, triggering a discussion about masking as well as introducing new deadlocks by model extension.
- #324 “Model Extension breaks Flows”, elaborating a blocking error in the concept of model extension which turned out to be in general more difficult with flows.
- #377 “BDD Model Checking immediately says unknown”, leading to the discovery of a bug in the translation of properties to formulas.
- #381 “Problem with precedenceGlobal pattern”, again about a bug in the translation of properties to formulas.
- #390 “Correct propositional properties with SAT Model Checking”, provoking a discussion about bounded SAT-based model checking.
- #14 “Support for more operators in the SLIM language”, reopened because the unary minus was not completely implemented.

Many other issues were discusses directly with the team members in Aachen, bypassing the official ticket system. The most important ones were:

- “Multiple Receiving of the same Event by one Component”, describing a problem when fan-out is used with event port connections targeting at different event ports of the same component. As a result, the syntactic restriction (G-11) was added to [37].
- “Fault Injection makes unaffected Properties false, possibly be breaking flows”, a discussion leading to ticket #324.
- “Case Expressions in Fault Injections”, revealing some problems with that.

- “Probabilistic Transitions with effects in Syntax and Compiler”, which led to the removal of probabilistic transitions from the SLIM syntax. They were initially planned to be used for representation of probabilistic error occurrences after model extension, but became superfluous as model extension changed.
- “Reset on Deactivation vs. Reset on Reactivation”, dealing with an inconsistency between the SLIM semantics, which originally defined that data subcomponents are reset to their default value when they become deactivated, and the translation to SMV, which resets data subcomponents on their reactivation.

The point in those lists is, that again and again issues came up that were not within the scope of my work but hindered or even blocked it. This makes an important difference to a rather “stand-alone” purely theoretical thesis.

Finally, I would like to mention that a minor undocumented limitation of SPIN parsing PROMELA code was identified by testing model checking of SLIM specifications with SPIN: names of process types must not exceed length 55. Presented to Gerard Holzmann, he decided to increase the allowed length in the next release of SPIN.

Chapter 9

Experimental Results

Having described the translation of SLIM specifications to PROMELA programs in part II and a slicing approach for SLIM specifications in part III, the question arises whether these are correct. Based on the formal semantics of SLIM it is expected that formal correctness proofs can be established. For the correctness of slicing it should suffice to show that the original and the sliced specification, both containing only the atomic propositions from the property under consideration as labels, are stutter bisimilar since stuttering bisimulation is known to preserve LTL without the Next operator. For a correctness proof of the translation the problem arises that no precise, formal semantics of PROMELA is defined (except by the SPIN source code). However, formal correctness proofs are not within the scope of this thesis. Instead, the implementation described in the previous chapter was used to run tests and case studies as a sanity check. For all tests and examples that were run, the model checking results using SPIN agreed with the intuition on the validity of the considered properties and with the results obtained from the COMPASS toolset. Furthermore, the results of model checking without and with slicing never differed.

Beyond correctness, another important question is how resource demanding model checking SLIM specifications with SPIN is. In particular, the effectiveness of slicing is reflected by differences in the resource demands for model checking unsliced and sliced specifications. In each of the following subsections an example SLIM specification together with properties and sometimes extended with error behaviour is introduced before the corresponding resource demands collected by running SPIN on those examples are discussed. The taken measures are:

- *State Vector Size* ($|SV|$): The amount of bytes needed by SPIN to store all information about one single state.
- *State Space Size* ($\#States$): The number of states generated by SPIN during model checking. Note that often only a part of the whole state space is generated by SPIN, e.g., for invalid invariants model checking can terminate as soon as a counterexample is found.
- *Memory*: The amount of megabytes needed by SPIN to store all generated states.
- *Time*: The duration of model checking in seconds.
- *Hash Collisions* ($\#HC$): For one example, also the number of hash collisions is given. SPIN stores the generated states in a hash table. Whenever two different states have the same hash value this is called a hash collision. Those collisions are resolved as usual by storing a list of states under the hash value in question. However, resolving hash conflicts consumes time as well so that avoiding hash collisions is beneficial.

9.1 Negate Random Bit

The introductory example from listing 2.1 (page 16) models a component (`root.negator`, instance of `Negator.Impl`) negating a boolean value which is provided as input from a random bit generator (`root.randomBit`, instance of `RandomBit.Impl`). The following properties were introduced and discussed for this SLIM specification in section 2.6:

- $\varphi_1 := \Box(\text{root.negator.neg} = \neg\text{root.negator.pos})$
- $\varphi_2 := \neg\text{root.negator.neg}$
- $\varphi_3 := \Box\neg(\text{root.randomBit.value})$
- $\varphi_4 := \Diamond(\text{root.randomBit.value})$

Table 9.1 shows the results of model checking the specification for these properties. Here, the non-duality of the property patterns absence global (φ_3) and existence global (φ_4), as discussed in section 2.6, becomes obvious. Furthermore, the table shows the resource demands for performing model checking on the original (unsliced) and on sliced specifications. The different numbers of states analysed for unsliced specifications result from the following facts: Property φ_1 requires analysing the full state space (105 states), since it is a valid invariant. For property φ_2 of pattern propositional, only the first stable state, which is reached after 26 non-stable states, has to be analyzed. Both properties φ_3 and φ_4 do not require a search through the full state space because they are violated and thus, as soon as a counterexample is found, the model checker can terminate. However, for all properties the state vector of the unsliced specification has – in principle – the same size. This is different with the sliced specifications: For properties φ_1 and φ_2 , the state vector size is already reduced to 68 byte (81% of its original size) although nearly everything of the specification is interesting for it. The only element that could be sliced away is the syntactically required but semantically irrelevant empty bus. As properties φ_3 and φ_4 do only reason about `root.randomBit`, for them the component `root.negator` can be sliced away as well. Consequently, the state vector size reduces even more, namely to 52 byte (61%). For the same reasons, a similar reduction is found with respect to the amount of states (about 83% for φ_1 and φ_2 , 45% for φ_3 and φ_4). This gives a first impression of the beneficial effects of slicing while preserving the model checking result. For this very small example, memory and time consumptions do not differ (baseline memory requirement of 2.195 MB and time consumption is reported by SPIN with either “0.00” or “0.01” seconds) and are thus not listed in the table.

Property	Sliced?	Result	SV	#States
φ_1	unsliced	true	84	105
	sliced	true	68	88
φ_2	unsliced	false	84	27
	sliced	false	68	22
φ_3	unsliced	false	84	54
	sliced	false	52	24
φ_4	unsliced	false	88	69
	sliced	false	52	32

Table 9.1: Comparison of results and resource demands for model checking the unsliced vs. the sliced specification of Negate Random Bit.

Property	Sliced?	Result	$ SV $	$\#States$
φ_1	unsliced	false	104	188
	sliced	false	84	165
φ_2	unsliced	false	104	33
	sliced	false	84	28
φ_3	unsliced	false	104	66
	sliced	false	52	24
φ_4	unsliced	false	104	84
	sliced	false	88	59

Table 9.2: Model checking results and resource demands for the extended Negate Random Bit model.

Beyond the nominal specification, error behaviour for the negator component was described in section 2.5. This changes two things: First of all, the property φ_1 is not valid any more since the output `root.negator.neg` is permanently set to false by the fault effect. Secondly, the extended model is more complex which leads to bigger state vector sizes for all properties and a bigger state space at all. Table 9.2 gives the model checking results and resource demands for the extended model. Again, state vector and state space size can be reduced by slicing to average 72% of the their original size. The best slicing result is obtained for property φ_3 with a reduction to 50% of the state vector size and to 36% of the states.

9.2 Integer Adder

Listing 9.1 gives the SLIM specification of an component adding two integers (`IntAdder.Impl`). This component receives two integer values between 0 and 30 as inputs, randomly generated by `RandomIntValue.Impl` exploiting the possibility of nondeterminism for the transition relation. Figure 9.1 visualises this specification.

```

----- root: IntAdderFrame.Impl -----

system IntAdderFrame
end IntAdderFrame;

system implementation IntAdderFrame.Impl
  subcomponents
    intValue1: system RandomIntValue.Impl accesses aBus;
    intValue2: system RandomIntValue.Impl accesses aBus;
    intAdder: system IntAdder.Impl accesses aBus;
    aBus: bus Bus.Impl;
  connections
    data port intValue1.value -> intAdder.x;
    data port intValue2.value -> intAdder.y;
  modes
    chooseIntValue1: activation mode;
    chooseIntValue2: mode;
  transitions
    chooseIntValue1 -[intValue1.update]-> chooseIntValue2;
    chooseIntValue2 -[intValue2.update]-> chooseIntValue1;

```

```

end IntAdderFrame.Impl;

----- root.intValue[1/2]: RandomIntValue.Impl -----

system RandomIntValue
  features
    value: out data port int default 2;
    update: in event port;
end RandomIntValue;

system implementation RandomIntValue.Impl
  modes
    cycle: activation mode;
  transitions
    cycle -[update then value := 0]-> cycle;
    cycle -[update then value := 1]-> cycle;
    -- and so on with value := 2,...,29
    cycle -[update then value := 30]-> cycle;
end RandomIntValue.Impl;

----- root.intAdder: IntAdder.Impl -----

system IntAdder
  features
    x: in data port int;
    y: in data port int;
    sum: out data port int;
end IntAdder;

system implementation IntAdder.Impl
  flows
    sum := x + y;
end IntAdder.Impl;

----- root.aBus: Bus.Impl -----

bus Bus
end Bus;

bus implementation Bus.Impl
end Bus.Impl;

```

Listing 9.1: Integer Adder in SLIM

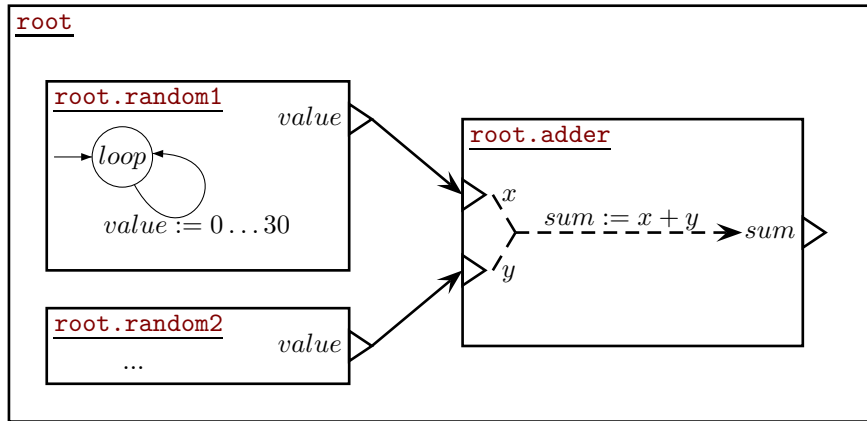


Figure 9.1: Visualisation of Integer Adder specification.

For this specification, the following properties were defined:

- $\varphi_1 := \square (0 \leq \text{intAdder.sum} \wedge \text{intAdder.sum} \leq 60)$
- $\varphi_2 := \square (0 \leq \text{intValue1.value} \wedge \text{intValue1.value} \leq 30 \wedge 0 \leq \text{intValue2.value} \wedge \text{intValue2.value} \leq 30)$
- $\varphi_3 := \square (0 \leq \text{intValue1.value} \wedge \text{intValue1.value} \leq 30)$
- $\varphi_4 := \diamond (\text{intValue1.value} \neq \text{intValue2.value})$
- $\varphi_5 := \square \neg (\text{intValue1.value} \neq \text{intValue2.value})$

The properties $\varphi_1, \dots, \varphi_3$ claiming correct bounds are valid invariants and thus require a full state space search. Both properties φ_4 and φ_5 are invalid invariants – again demonstrating the non-duality of existence global and absence global – and thus model checking can be aborted as soon as a violating state is found instead of analysing the whole state space. Table 9.3 shows the results and resource demands of model checking for the unsliced Integer Adder specification whereas table 9.4 shows the corresponding values for the sliced specifications. The differences in the resource demands when sliced for $\varphi_1, \dots, \varphi_3$ results from the differences in the corresponding sets of interesting data elements: While for φ_1 every data port is needed, for φ_2 the whole `root.adder` component can be removed and for φ_3 only `intValue1.value` is interesting. The removal of the empty bus component accounts for the reduction from the unsliced specification to the one sliced for φ_1 . The state vector size could be reduced to 88% for φ_1 and to average 58% for the other properties. The overall reduction of the state space size and memory requirements are even more impressive: from average 49% for φ_3 and φ_4 down to 2.23% for φ_2 and to 0.07% for φ_3 .

Property	Result	SV	#States	Memory	Time
$\varphi_1, \varphi_2, \varphi_3$	true	128	1,207,046	171.873	3.87
φ_4	false	132	196	8.299	0.01
φ_5	false	128	65	8.299	0.00

Table 9.3: Model checking results for the unsliced Integer Adder specification.

Property	Result	$ SV $	$\#States$	Memory	Time
φ_1	true	112	1,306,985	165.037	3.67
φ_2	true	76	26,925	11.424	0.10
φ_3	true	72	885	8.397	0.00
φ_4	false	76	93	8.299	0.00
φ_5	false	76	33	8.299	0.00

Table 9.4: Model checking results for the sliced Integer Adder specification.

For property φ_1 a non-beneficial anomaly of slicing becomes apparent which leads to an increase in the state space size to 108%. However, the actual problem is not with slicing itself but with the ordering of subcomponents in the PROMELA code. The implemented translation handles subcomponents in the order of their storage in the data structure provided by the parser. This can be done since neither the SLIM semantics nor the concepts used in the translation depends on the ordering of subcomponents. Actually, the state space of the resulting PROMELA program should be independent from that order. For example, during multiway event communication (cf. section 4.5) it does not matter in which order the message is forwarded to the subcomponents. Since changes to outgoing data ports are buffered, this holds for invalidate-update-cycles, as described in section 4.6, as well. For the same reason, the implementation of slicing does not take care of preserving the order of subcomponents within the parser data structure. However, for SPIN's optimisation techniques, in particular for partial order reduction (cf. section 3.3.1), this order of communication with processes implementing subcomponents is crucial. The anomaly for slicing the Integer Adder specification with respect to property φ_1 results exactly from the fact that the slicing algorithm changed the order of subcomponents and had the misfortune that a worse ordering for SPIN's optimisation techniques resulted. This argument is supported by the fact that slicing can reduce the state space size from 1,654,872 to 1,420,383 (86%) when SPIN's optimisation techniques are disabled. The only sensible conclusion from this is that the ordering of communication with processes should be optimised for SPIN's optimisation techniques but this seems to be a non-trivial challenge.

9.2.1 Simulation of SLIM specifications

In addition to model checking SLIM specifications, they can be simulated by SPIN as well based on the translation to PROMELA. This is realised by adding `printf` statements to the PROMELA code and invoking SPIN with the parameters for simulation, e.g., a bound for the length. The simulation contains the most important intermediate steps, e.g., taking a transition or receiving update requests, associated to the corresponding SLIM control component. Whenever a stable state is reached, this, i.e., mode information and data values of all components, is printed out. However, for modes only their integer representative (cf. section 4.2.1) is given but not retranslated to the mode name. Listing 9.2 gives the initial fragment of the a simulation for the Integer Adder example where the following integer representation of modes is used: `chooseIntValue1` \mapsto 0, `chooseIntValue2` \mapsto 1, `DefaultInitialMode` \mapsto 2, `cycle` \mapsto 3.

```
-- Omitted: Communication for activation & initialisation
--           of component hierarchy.
```

```
NEW STABLE STATE:
  root.intAdder.x=2, root.intAdder.y=2, root.intAdder.mode=2,
  root.intValue1.value=2, root.intAdder.sum=4, root.mode=0,
```

```

    root.intValue1.mode=3, root.aBus.mode=2, root.intValue2.mode=3,
    root.intValue2.value=2

root: Got semaphore to take its next transition!
root_intValue1: receives trymultiway request for 'update'
root_intValue1: takes reactive transition:
    cycle -[update then value := 26]-> cycle
root_intValue1: enters state 'cycle'
root: takes master transition:
    chooseIntValue1 -[intValue1.update]-> chooseIntValue2
root: enters state 'chooseIntValue2'
root: update with new input values:
root_intAdder: update with new input values: y=2, x=2
root_intAdder: update finally lead to new output values: sum=4
root_aBus: update with new input values:
root_aBus: update finally lead to new output values:
root_intValue1: update with new input values:
root_intValue1: update finally lead to new output values: value=26
root_intValue2: update with new input values:
root_intValue2: update finally lead to new output values: value=2
root_intAdder: update with new input values: y=2, x=26
root_intAdder: update finally lead to new output values: sum=28
root_aBus: update with new input values:
root_aBus: update finally lead to new output values:
root_intValue1: update with new input values:
root_intValue1: update finally lead to new output values: value=26
root_intValue2: update with new input values:
root_intValue2: update finally lead to new output values: value=2
root: update finally lead to new output values:
Environment: Returned transition semaphore!

NEW STABLE STATE:
    root.intAdder.x=26, root.intAdder.y=2, root.intAdder.mode=2,
    root.intValue1.value=26, root.intAdder.sum=28, root.mode=1,
    root.intValue1.mode=3, root.aBus.mode=2, root.intValue2.mode=3,
    root.intValue2.value=2

-- Omitted: Communication for root taking master transition
--           from chooseIntValue2 to chooseIntValue1.

NEW STABLE STATE:
    root.intAdder.x=26, root.intAdder.y=15, root.intAdder.mode=2,
    root.intValue1.value=26, root.intAdder.sum=41, root.mode=0,
    root.intValue1.mode=3, root.aBus.mode=2, root.intValue2.mode=3,
    root.intValue2.value=15

-- And so on ...

```

Listing 9.2: Simulation fragment for the Integer Adder specification.

Simulation of specifications containing `reals` cannot be done directly with SPIN, because SPIN itself does not execute the embedded C-code which is used for handling `reals` (cf. section 4.2.2). In this case, the C-program PAN generated by SPIN can be used to some extent for simulation. In principle, the only thing to do is to add `printfs` inside embedded C-code so that they are copied to PAN (PROMELA `printfs` get lost). An example for this kind of simulation is given in the next section.

9.3 Redundant Battery System

To avoid a too low or total loss of power supply, a safety critical system should make use of two redundant batteries. Whenever the power provided by one battery does not suffice any more the system switches to the other one. Listing 9.3 gives the SLIM specification¹ of such a system. The energy of a battery is handled as a `real` typed data subcomponent. The hybrid behaviour of degeneration of energy over time is mimicked by a self-looping transition stepwise reducing the value of `energy`. When the energy of a battery is depleted the battery emits the signal `empty` to which the root component must react. Note that the component representing the battery which is currently not used is deactive. As `Battery.Imp` does not support mode history, on reactivation of that battery its energy is reset to 100.0, that is, the battery is somehow recharged.

```
-- root: Power.Imp -----
system Power
  features
    voltage: out data port real;
end Power;

system implementation Power.Imp
  subcomponents
    batt1: system Battery.Imp in modes (primary);
    batt2: system Battery.Imp in modes (backup);
  connections
    data port batt1.voltage -> voltage in modes (primary);
    data port batt2.voltage -> voltage in modes (backup);
  modes
    primary: initial mode;
    backup: mode;
  transitions
    primary -[batt1.empty]-> backup;
    backup -[batt2.empty]-> primary;
end Power.Imp;

-- root.batt[1/2]: Battery.Imp -----
system Battery
  features
    empty: out event port;
    voltage: out data port real default 6.0;
end Battery;
```

¹This specification is the non-hybrid version of an example presented by Thomas Noll at the second D-CON (German Chapter CONCUR) meeting in Berlin, 2009.

```

system implementation Battery.Imp
  subcomponents
    energy: data real default 100.0;
  modes
    charged: activation mode;
    depleted: mode;
  transitions
    charged -[when 0.999 * energy >= 20.0
              then energy := 0.999 * energy;
              voltage := 6.0 * (0.999 * energy) / 100.0
            ]-> charged;
    charged -[empty when 0.999 * energy < 20.0
              then energy := 2.0; voltage := 0.1]-> depleted;
end Battery.Imp;

```

Listing 9.3: SLIM specification of a system containing two redundant batteries.

The following properties are considered for this example:

- $\varphi_1 := \square(\text{root.mode} = \text{mode:primary} \Rightarrow (\text{root.batt1.mode} = \text{mode:charged} \wedge \text{root.batt1.energy} \geq 20.0))$
- $\varphi_2 := \square(\text{root.mode} = \text{mode:backup} \Rightarrow (\text{root.batt2.mode} = \text{mode:charged} \wedge \text{root.batt2.energy} \geq 20.0))$
- $\varphi_3 := \square(\text{root.voltage} \geq 1.2)$

Property φ_1 claims that `root.batt1` is in mode `charged` and has at least `energy` of 20.0 when it is used, that is, when the `root` component is in mode `primary`. Property φ_2 states the same for `root.batt2` when `root` is in mode `backup`. Finally, property φ_3 checks that the voltage provided by the `Power` system is always at least 1.2. Table 9.5 lists the model checking results and resource demands for these properties on the original and on the sliced specifications. For properties φ_1 and φ_2 the overall average reduction is to 73% but for property φ_3 slicing does not help at all since for the verification of this property the whole specification is interesting.

Property	Sliced?	Result	SV	#States	Memory	Time
φ_1, φ_2	unsliced	true	112	62,779	43.162	0.14
	sliced	true	80	43,471	37.889	0.09
φ_3	unsliced	true	112	62,779	43.162	0.15
	sliced	true	112	62,779	43.162	0.15

Table 9.5: Comparison of results and resource demands for model checking the unsliced vs. the sliced nominal specification of Redundant Battery System.

A typical error situation which leads to the loss of power supply is a loose contact. This is modeled by a fault injection associating the `Battery` component `root.batt1` of the nominal specification with the error model `BatteryError.Impl` which is given in listing 9.4. Furthermore, in error state `dead` as error effect `voltage := 0` is used. The `energy` of a “dead” battery, however, is not affected. Consequently, property φ_3 becomes invalid while the others stay valid.

```

error model BatteryError
  features
    ok : initial state;
    dead: error state;
end BatteryError;

error model implementation BatteryError.Impl
  events
    batteryDies: error event;
  transitions
    ok -[batteryDies]-> dead;
end BatteryError.Impl;

```

Listing 9.4: Error model for batteries.

Results and resource demands of model checking for the extended model are given in table 9.6. In comparison with the values from table 9.5, the state space explosion resulting from model extension becomes obvious: The nominal model on its own induces 62,779 states whereas the extended model yields 222,080 states – making slicing even more important. For the first two properties slicing the extended model reduces state vector size, state space size and overall memory requirement to about 78% and the time requirement even down to 47%.

Property	Sliced?	Result	$ SV $	$\#States$	Memory	Time
φ_1, φ_2	unsliced	true	128	222,080	67.772	1.20
	sliced	true	96	177,036	54.100	0.49
φ_3	unsliced	false	128	47	32.713	“0.00”
	sliced	false	128	47	32.713	“0.00”

Table 9.6: Comparison of results and resource demands for model checking the unsliced vs. the sliced extended model of Redundant Battery System.

As mentioned at the end of the previous section, simulation of SLIM specifications using **reals** can be with PAN. Listing 9.5 shows a fragment of such a simulation where 0 represents the mode **primary** and 1 mode **backup**, respectively.

```

NEW STABLE STATE: root.mode=0, root.voltage=6.000000, ...
NEW STABLE STATE: root.mode=0, root.voltage=5.880000, ...
NEW STABLE STATE: root.mode=0, root.voltage=5.762400, ...
...
NEW STABLE STATE: root.mode=0, root.voltage=1.241040, ...
NEW STABLE STATE: root.mode=0, root.voltage=1.216219, ...
NEW STABLE STATE: root.mode=1, root.voltage=6.000000, ...
NEW STABLE STATE: root.mode=1, root.voltage=5.880000, ...
...

```

Listing 9.5: Simulation fragment for the Redundant Battery System specification.

9.4 Wind Turbine

9.4.1 Nominal Behaviour

Two important aspects in the controlling of wind turbines are, on the one hand, adjusting the rotating head according to the wind position and, on the other hand, disabling the turbine by moving the blades into a position with lowest possible wind resistance when the wind speed becomes too high. A SLIM specification modeling a `WindTurbine` with a `WindDirectionControl` and a `WindSpeedControl` subcomponent is shown in listing 9.6. Different speeds and directions of wind are simulated by `randomWind`.

```

----- root: WindTurbineInEnvironment -----

system WindTurbineInEnvironment
end WindTurbineInEnvironment;

system implementation WindTurbineInEnvironment.Impl
  subcomponents
    windTurbine: system WindTurbine.Impl accesses environmentBus;
    randomWind: system randomWind.Impl accesses environmentBus;
    environmentBus: bus Bus.Impl;
  connections
    data port randomWind.direction -> windTurbine.windDirection;
    data port randomWind.speed -> windTurbine.windSpeed;
end WindTurbineInEnvironment.Impl;

----- root.windTurbine: WindTurbine.Impl -----

system WindTurbine
  features
    windDirection: in data port enum(wdN, wdNE, ..., wdW, wdNW);
    windSpeed: in data port int;
    turbinePosition: out data port enum(wdN, wdNE, ..., wdNW);
    turbineBladesEnabled: out data port bool;
end WindTurbine;

system implementation WindTurbine.Impl
  subcomponents
    windDirectionControl: system WindDirectionControl.Impl
                          accesses turbineBus;
    windSpeedControl: system WindSpeedControl.Impl
                      accesses turbineBus;
    turbineBus: bus Bus.Impl;
  connections
    data port windDirection -> windDirectionControl.windDirection;
    data port windDirectionControl.turbinePosition
              -> turbinePosition;
    data port windSpeed -> windSpeedControl.windSpeed;

```

```

    data port windSpeedControl.turbineBladesEnabled
        -> turbineBladesEnabled;
end WindTurbine.Impl;

----- *.windDirectionControl: WindDirectionControl.Impl -----
system WindDirectionControl
    features
        windDirection: in data port enum(wdN, wdNE, ..., wdW, wdNW);
        turbinePosition: out data port enum(wdN,...) default wdS;
end WindDirectionControl;

system implementation WindDirectionControl.Impl
    flows
        turbinePosition := windDirection;
end WindDirectionControl.Impl;

----- *.windSpeedControl: WindSpeedControl.Impl -----
system WindSpeedControl
    features
        windSpeed: in data port int;
        turbineBladesEnabled: out data port bool default false;
end WindSpeedControl;

system implementation WindSpeedControl.Impl
    flows
        turbineBladesEnabled := windSpeed < 10;
end WindSpeedControl.Impl;

----- root.randomWind: randomWind.Impl -----
system randomWind
    features
        direction: out data port enum(wdN, ..., wdNW) default wdN;
        speed: out data port int default 5;
end randomWind;

system implementation randomWind.Impl
    modes
        cycle: initial mode;
    transitions
        -- Change nothing

```

```

cycle -[]-> cycle;
-- Change speed
cycle -[then speed := 0]-> cycle;
-- and so on, for speed := 1 ... 11
cycle -[then speed := 12]-> cycle;
-- Change direction
cycle -[then direction := wdN]-> cycle;
-- and so on, for direction := wdNE .. wdW
cycle -[then direction := wdNW]-> cycle;
-- Change direction and speed (just a few examples)
cycle -[then speed := 0; direction := wdN]-> cycle;
cycle -[then speed := 1; direction := wdNE]-> cycle;
cycle -[then speed := 2; direction := wdE]-> cycle;
cycle -[then speed := 5; direction := wdSE]-> cycle;
cycle -[then speed := 6; direction := wdS]-> cycle;
cycle -[then speed := 9; direction := wdSW]-> cycle;
cycle -[then speed := 10; direction := wdW]-> cycle;
cycle -[then speed := 12; direction := wdNW]-> cycle;
end randomWind.Impl;

----- *.environmentBus, *.turbineBus: Bus.Impl -----

bus Bus
end Bus;

bus implementation Bus.Impl
end Bus.Impl;

```

Listing 9.6: Nominal model of a Wind Turbine described in SLIM. The complete lists of enumeration identifiers for wind directions (`wdN`, `wdNE`, `wdE`, `wdSE`, `wdS`, `wdSW`, `wdW`, `wdNW`) are abbreviated to avoid line breaks.

The following properties are considered for which table 9.7 gives the model checking results and resource demands:

- $\varphi_1 := \square(\text{windTurbine.turbinePosition} = \text{randomWind.direction})$, asserting that the wind turbine always faces the wind.
- $\varphi_2 := \square(\text{windTurbine.turbineBladesEnabled} = \text{randomWind.speed} < 10)$, stating that the blades are disabled if and only if the wind speed is 10 or higher.
- $\varphi_3 := \square(\text{randomWind.speed} \geq 10 \Rightarrow \neg \text{windTurbine.turbineBladesEnabled})$, claiming that the blades are disabled if the wind speed is 10 or higher. In contrast to φ_2 it is not required that the blades are enabled at lower wind speeds.

For all properties, slicing the nominal specification can reduce state vector size to 60%, the amount of states to average 2.7%, the overall memory requirement to about 16% and the run time to average 4%. The memory reduction could be expected to be about 2% ($60\% \cdot 2.7\% \approx 2\%$), but for this small example the baseline memory requirements of SPIN for storing the empty hash table comes into effect.

Specification	Result	$ SV $	$\#States$	Memory	Time	$\#HC$
unsliced	true	160	111,119	21.666	0.46	750
sliced for φ_1	true	92	1,901	3.306	0.01	0
sliced for φ_2, φ_3	true	100	4,111	3.502	0.03	0

Table 9.7: Comparison of resource demands for model checking unsliced vs. sliced nominal Wind Turbine specification.

9.4.2 Error Behaviour

Errors can – among others – happen independently in the wind direction control and in the wind speed control subcomponent. To model this, the error models from listing 9.7 are used.

```

error model WindDirectionControlError
  features
    OK: initial state;
    FixedToE: error state;
  end WindDirectionControlError;

error model implementation WindDirectionControlError.Impl
  events
    eeFixedToE: error event;
  transitions
    OK -[eeFixedToE]-> FixedToE;
  end WindDirectionControlError.Impl;

error model WindSpeedControlError
  features
    OK: initial state;
    DisableTooEarly: error state;
    DisableTooLate: error state;
  end WindSpeedControlError;

error model implementation WindSpeedControlError.Impl
  events
    eeDisableTooEarly: error event;
    eeDisableTooLate: error event;
  transitions
    OK -[eeDisableTooEarly]-> DisableTooEarly;
    OK -[eeDisableTooLate]-> DisableTooLate;
  end WindSpeedControlError.Impl;

```

Listing 9.7: Error models for the Wind Turbine specification.

With those error models, three different possible errors were considered independently:

1. *Blades Disabled Too Early*: The control component `windTurbine.windSpeedControl` is associated with the error model `WindSpeedControlError.Impl` together with the fault

effect `turbineBladesEnabled := windSpeed < 9` in error state `DisableTooEarly`. The other error state has no effect. Table 9.8 shows the results.

2. *Blades Disabled Too Late*: The control component `windTurbine.windSpeedControl` is associated with the error model `WindSpeedControlError.Impl` together with the fault effect `turbineBladesEnabled := windSpeed < 11` in error state `DisableTooLate`. The other error state has no effect. Table 9.9 shows the results.
3. *Position Fixed To East*: The control component `windTurbine.windDirectionControl` is associated with the error model `WindDirectionControlError.Impl` together with the fault effect `turbinePosition := wdE` in error state `FixedToE`. Table 9.10 shows the results.

Of course, all fault injections can be applied at the same time. This would result in an even bigger state space. However, when all three fault injections are applied, then all three properties become false and thus never the whole state space would be analysed.

The taken measures show that slicing can half the state vector size and reduce the state space to less than 1% (property φ_1 for Blades Disabled Too Early). Furthermore, beneficial effect of slicing for a reduction of the number of hash conflicts can be concluded from the results.

Property	Sliced?	Result	SV	#States	Memory	Time	#HC
φ_1	unsliced	true	180	373,013	71.861	1.86	18,889
	sliced	true	92	1,901	3.306	0.01	0
φ_2	unsliced	false	180	2,191	3.502	0.04	0
	sliced	false	120	1,471	3.306	0.02	0
φ_3	unsliced	true	180	373,013	71.861	1.93	18,889
	sliced	true	120	14,610	4.966	0.04	3

Table 9.8: Comparison of results and resource demands for model checking the unsliced vs. the sliced extended model of Wind Turbine resulting from fault injection Blades Disabled Too Early.

Property	Sliced?	Result	SV	#States	Memory	Time	#HC
φ_1	unsliced	true	180	373,013	71.861	2.45	19,462
	sliced	true	92	1,901	3.306	0.01	0
φ_2	unsliced	false	180	124,974	26.353	0.65	1,002
	sliced	false	120	6,457	3.892	0.07	0
φ_3	unsliced	false	180	124,974	26.353	0.82	1,002
	sliced	false	120	6,457	3.892	0.05	0

Table 9.9: Comparison of results and resource demands for model checking the unsliced vs. the sliced extended model of Wind Turbine resulting from fault injection Blades Disabled Too Late.

Property	Sliced?	Result	$ SV $	$\#States$	Memory	Time	$\#HC$
φ_1	unsliced	false	180	93	3.111	0.01	0
	sliced	false	112	66	3.111	0.01	0
φ_2	unsliced	true	180	240,371	47.545	1.30	5,893
	sliced	true	100	4,111	3.502	0.01	0
φ_3	unsliced	true	180	240,371	47.545	1.30	5,893
	sliced	true	100	4,111	3.502	0.01	0

Table 9.10: Comparison of results and resource demands for model checking the unsliced vs. the sliced extended model of Wind Turbine resulting from fault injection Position Fixed To East.

9.5 Open Thermal Loop

This section summarises some of the most important insights gained from model checking a corrected and to syntactical changes adapted version of a SLIM specification modeling the temperature dependent control of redundant heaters and the monitoring of possible errors as part of a satellite’s thermal control². For the considered properties the specification induces a state vector of 452 bytes and 4,783,136 states. A full state space search requires 2,134 MB of memory but on the test system the memory was limited to 2,048 MB. Extending memory by using virtual memory does not really work with SPIN. Slicing could help by reducing the state vector size to 420 bytes (93%), the state space size to 4,504,872 (94%) and the overall memory requirement to 1,866 MB (87%). Model checking the slices specification took 59 seconds. For some other properties, slicing did not suffice to conform with the resource restrictions. In those cases, two possible options of SPIN can help to reduce the memory requirements (cf. section 3.3.2): First of all, the state vectors can be compressed. Then, only 300 MB without slicing or 267 MB after slicing are needed to store the state space – a reduction to 14%! The time for model checking did not increase for this example. Another method, which is even more memory efficient, is to use a minimized automaton representation for the state space. Then, only 92 MB or 84 MB, respectively, are needed. However, the run time of model checking increases to 457 or 425 seconds, respectively, that is, more than the septuple. Of course, both methods – collapse and minimized automaton – can be combined to enable model checking of even bigger specifications but always with the trade-off between time and memory consumption.

²The original specification was developed by Xavier Olive (Thales Alenia Space, France) for a first evaluation of the COMPASS toolset.

Chapter 10

Summary, Conclusions and Future Work

10.1 Summary

The COMPASS project funded by ESA aims for a coherent application of formal methods in the design of large safety-critical systems (cf. chapter 1). To this end, the modeling language SLIM was developed within the project as an extended subset of AADL supporting system-software co-engineering, including the description of possible failures. Furthermore, a tool chain of translators from SLIM to different model checkers was established in order to provide a possibility of performing diverse analysis and verifications of SLIM specifications.

Within the scope of this thesis, a translation of SLIM specifications to PROMELA, the input language of the LTL-model checker SPIN, was established. Therefor, chapter 2 introduced the SLIM language which allows to describe hierarchies of concurrently executed components that can communicate by instantaneously exchanging data and by synchronous event communication. The behaviour of each component can be described by a kind of hybrid automaton. Afterwards, chapter 3 gave an overview of SPIN: its input language PROMELA, the approach to model checking and some optimisations to improve the efficiency of model checking. On this basis, the translation of SLIM specifications to PROMELA programs was presented in part II. The basic idea is to transform SLIM components to PROMELA proctypes which implement all possible behaviour of the component. Values of data elements are stored in global variables. Propagation of data values through the component hierarchy, event communication between multiple components as well as de- and reactivation of whole subtrees in the component hierarchy were realised by communication protocols on PROMELA channels. In particular, the necessity of generating a fixpoint iteration was discussed for the propagation of data values. Properties reasoning about a SLIM model have to be translated to PROMELA in analogy to the translation of the model itself. Due to different problems described in chapter 5, the mechanism of SPIN to handle LTL-formulas automatically was not used. Instead, never claims corresponding to the property patterns as they can occur in COMPASS were manually designed.

A typical problem affecting the feasibility of model checking for large and complex systems is the so-called state space explosion problem. To reduce the state space induced by a model, abstraction techniques should be used. However, methods that do not need to generate the whole state space before its reduction are preferable since they do not only speed-up model checking but also reduce the peak memory consumption. In part III the idea of slicing was ported to SLIM

specifications: All parts of a model that are irrelevant for model checking a given property are removed so that a smaller model with smaller state space results which, however, preserves the model checking result. Chapter 6 introduced the basic slicing algorithm which, starting from the elements occurring in the property, employs a fixpoint iteration to calculate the transitive backward closure of all parts of a model that directly or indirectly can influence the validity of the property. Additionally, more involved aspects of slicing SLIM specifications, such as preserving of divergence characteristics for liveness properties, were discussed in chapter 7.

To evaluate both, the translation and the slicing, they were implemented. The examples presented in chapter 9 showed that, on the one hand, model checking SLIM specifications with PROMELA is practicable and, on the other hand, that slicing can drastically reduce the induced state space size and thus the complexity of model checking.

10.2 Conclusions

The two most important conclusions were already drawn in the summary: Model checking of SLIM specifications with SPIN is feasible and slicing can considerably reduce the state space size. For the translation some not that straight forward constructs are needed, for example the fixpoint iteration for data port updates or the usage of embedded C-code for data elements of type `real`. Additionally, the translation tries to keep the induced state space as small as possible by using only unbuffered channels, by enclosing sequences of statements which have only local effects in `d_step`-blocks, by resetting buffer variables as soon as they are not needed any more and by limiting itself to the usage of PROMELA constructs that are compatible with partial order reduction. Compared with the translation to NUSMV and MRMC developed in the COMPASS project, model checking SLIM specifications with SPIN lacks the support for CTL properties and timed probabilistic behaviour – both not (immediately) possible with SPIN. However, the translation to PROMELA also exhibits some advantages: First of all, model checking of specifications containing deadlocks still yields correct results – possible by the stuttering extension of finite executions to infinite ones performed by SPIN. Furthermore, the SLIM data type `real` is completely supported (as a 4 byte `C_float`) without any limitations. In particular, neither is model checking restricted to bounded model checking nor are only linear expressions over `reals` allowed – as it is the case with NUSMV which falls back to SAT-based model checking as soon as `reals` are contained in a specification. As a practical consequence, the SLIM specification of a Redundant Battery System, as presented without fault injection in section 9.3, could easily be model checked with SPIN in about 0.2 seconds while NUSMV after about 45 minutes and 280 MB of memory consumption only reached bound 347 and not yet a final result. Nevertheless, there is still open work left for the future as indicated in the next section.

10.3 Future Work

This section gives some ideas for future work that is beyond the scope of this thesis. First of all, the optimisations for the fixpoint iteration handling data port updates mentioned in section 4.6.1 can be implemented. In contrast, the ideas for further refinements of the slicing algorithm discussed in section 7.8 need some more theoretical development. This holds as well for establishing the formal correctness proofs for the translation and for the slicing algorithm.

Beyond that, a variety of other possible improvements of the generated PROMELA code exists:

- Instead of using variables of type `short` or `int` to store integer representatives of symbolic identifiers used in the SLIM specification, the data type `unsigned` should be used as it allows to define – using a special syntax – exactly how many bits are needed.
- Buffer variables for data subcomponents could be declared as `hidden` to exclude them from the state vector and thus save memory since they are only used locally inside `d_step`-blocks. However, this causes some trouble when those variables have to be accessed from embedded C-code and is thus not yet implemented.
- The sequence of assignments in transition effects – their order does not have any semantical effect as they are performed “in parallel” – could be rearranged to minimise the amount of required buffering. For example, instead of translating `x := 2; y := x`, which requires buffering, `y := x; x := 2`, requiring no buffering, should be used.
- In fact, in most of the cases, never all buffer variables are needed at the same time. To reduce the state vector size, the buffer variables should be shared between data elements. When at most *max* data elements are to be buffered at the same time, *max* buffer variables should suffice. By the way, the outgoing data ports of the `root` component do not have to be buffered at all.
- It is tempting to omit sending the `invalidate` message when a transition with empty effect is taken. However, due to implicit transition effects and the possibility of multiway event communication, this is in general not possible. However, for proactive transitions with an empty effect and no implicit effects, such as de- or reactivating subcomponents or connections, this should be possible.
- The process generated for the `root` component could be extended with the special functionality of the `Environment` process. Then, the `Environment` process could be removed.
- As the anomaly discussed for the Integer Adder specification in section 9.2 revealed, the order in which processes communicate with other processes – although it is irrelevant for the functional correctness of the translation – can influence the reduction effects achieved by partial order reduction. If one can find out which order is the best one with respect to partial order reduction, this order should be chosen by the translation. Alternatively, a new predicate could be introduced in PROMELA telling SPIN that for a enclosed sequence of statements the order of their execution does not matter and that SPIN should choose the most efficient one.

Three alternatives for the translation to SPIN could be evaluated:

- Instead of creating one process for every control component of a SLIM specification, one process could suffice: In local variables this process could store the current valuation of all data elements and the mode information of all components – similar as it is done with the global variables in the translation that was described. In a complex `do`-loop this process could manipulate the values as the SLIM components could do it. This could be perceived as a direct transformation of the formal flat semantics of SLIM to PROMELA.
- To handle probabilistic transitions, the translation that was presented could be extended to target PRO[B]MELA, a probabilistic extension of PROMELA (cf. [4]). However, the

problem is that in SLIM error models do not contain “simple” probabilities but occurrence rates. Furthermore, PRO[B]MELA does not allow the nondeterministic choice between probabilistic and non-probabilistic transitions as it can occur in SLIM models resulting from model extension.

- To allow model checking of SLIM specifications containing timed behaviour, RT-SPIN (cf. [40]) instead of SPIN could be used. However, the development of RT-SPIN seems to be discontinued so that optimisations that were later on included in SPIN are not contained.

Finally, after having dealt with a translation to PROMELA in order to reuse the model checker SPIN, one should ask the question how sensible that is. Obviously, the translation generates some overhead, for example the intermediate, non-stable states. Instead of translating to PROMELA, one could think of splitting SPIN into two parts: One language-specific frontend that can generate all successor states of a given system state and one language-independent backend that performs model-checking.

Bibliography

- [1] The COMPASS Project Web Site. <http://compass.informatik.rwth-aachen.de/>.
- [2] Python Programming Language – Official Website. <http://www.python.org/>.
- [3] The SPIN Web Site. <http://spinroot.com/>.
- [4] Christel Baier, Frank Ciesinski, and Marcus Größer. PROBMELA: a modeling language for communicating probabilistic processes. In *MEMOCODE*, pages 57–66. IEEE, 2004.
- [5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.
- [6] Mordechai Ben-Ari. *Principles of the SPIN Model Checker*. Springer, 2008.
- [7] B. Bittner, Friedrich Gretz, Viet Yen Nguyen, Thomas Noll, and D. Tomasoni. Integrated Platform User Manual. Technical report, COMPASS Project, 2010.
- [8] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. Codesign of Dependable Systems: A Component-Based Modelling Language. In *7th Int. Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 121–130. IEEE CS Press, 2009.
- [9] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. Model-Based Codesign of Critical Embedded Systems. In *Proc. 2nd Int. Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB 2009)*, volume 507 of *CEUR Workshop Proceedings*, pages 87–91. Sun SITE Central Europe, 2009. <http://ftp1.de.freebsd.org/Publications/CEUR-WS/Vol1-507/>.
- [10] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems. In *28th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP 2009)*, volume 5775, pages 173–186. Springer, 2009.
- [11] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. Safety, Dependability, and Performance Analysis of Extended AADL Models. *The Computer Journal*, 2010. To be published.
- [12] Marco Bozzano and A. Villafiorita. The FSAP/NuSMV-SA Safety Analysis Platform. *International Journal on Software Tools for Technology Transfer*, 9(1):5–24, 2007.
- [13] Robert Cavada, Alessandro Cimatti, Gavin Keighren, Emanuele Olivetti, Marco Pistore, and Marco Roveri. NuSMV 2.2 Tutorial. Technical report, ITC-irst, Povo, Trento, Italy.

- [14] Jingde Cheng. Slicing Concurrent Programs - A Graph-Theoretical Approach. In Peter Fritzson, editor, *AADEBUG*, volume 749 of *Lecture Notes in Computer Science*, pages 223–240. Springer, 1993.
- [15] Ching-Tsun Chou and Doron Peled. Formal Verification of a Partial-Order Reduction Technique for Model Checking. *J. Autom. Reasoning*, 23(3-4):265–298, 1999.
- [16] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A New Symbolic Model Checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.
- [17] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
- [18] Peter Feiler and Ana Rugina. Dependability Modeling with the Architecture Analysis & Design Language (AADL). Technical report, Software Engineering Institute, 2007.
- [19] Peter H. Feiler, David P. Gluch, and John J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, Software Engineering Institute, 2006.
- [20] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In Piotr Dembinski and Marek Sredniawa, editors, *PSTV*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1995.
- [21] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing Software for Model Construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.
- [22] Thomas A. Henzinger. The Theory of Hybrid Automata. In *LICS*, pages 278–292, 1996.
- [23] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
- [24] Gerard J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13(3):289–307, 1998.
- [25] Gerard J. Holzmann. Logic Verification of ANSI-C Code with SPIN. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2000.
- [26] Gerard J. Holzmann. Software Model Checking. In *Computer and System Sciences*, volume 180, pages 309–355. IOS Press, 2000. Course notes, NATO Summer School, Marktobendorf, Germany.
- [27] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
- [28] Gerard J. Holzmann and Rajeev Joshi. Model-Driven Software Verification. In Susanne Graf and Laurent Mounier, editors, *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2004.
- [29] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In Dieter Hogrefe and Stefan Leue, editors, *FORTE*, volume 6 of *IFIP Conference Proceedings*, pages 197–211. Chapman & Hall, 1994.

- [30] Gerard J. Holzmann, Doron Peled, and Mihalis Yannakakis. On Nested Depth First Search. In *The Spin Verification System*, pages 23–32. American Mathematical Society, 1996.
- [31] Gerard J. Holzmann and Anuj Puri. A Minimized Automaton Representation of Reachable States. *International Journal on Software Tools for Technology Transfer*, 2(3):270–278, 1999.
- [32] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov Reward Model Checker. In *Quantitative Evaluation of Systems (QEST)*, pages 243–244. IEEE Computer Society, 2005.
- [33] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [34] Lynette I. Millett and Tim Teitelbaum. Issues in Slicing PROMELA and Its Applications to Model Checking, Protocol Understanding, and Simulation. *STTT*, 2(4):343–349, 2000.
- [35] Thomas Noll. Flat Semantics of the SLIM Language. Technical report, COMPASS Project, 2009.
- [36] Thomas Noll. Hierarchical Semantics of the SLIM Language. Technical report, COMPASS Project, 2009.
- [37] Thomas Noll. Model-Based Analysis and Verification: Potential Solutions. Technical note, COMPASS Project, 2010.
- [38] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
- [39] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.
- [40] Stavros Tripakis and Costas Courcoubetis. Extending Promela and Spin for Real Time. In Tiziana Margaria and Bernhard Steffen, editors, *TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 329–348. Springer, 1996.
- [41] Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In *LICS*, pages 332–344. IEEE Computer Society, 1986.
- [42] Mark Weiser. Program Slicing. In *ICSE*, pages 439–449, 1981.
- [43] Christopher D. Wickens, Justin G. Hollands, and Justin Hollands. *Engineering Psychology and Human Performance*. Addison-Wesley, 1999.
- [44] Ralf Wimmer, Marc Herbstritt, Holger Hermanns, Kelley Strampp, and Bernd Becker. Sigref – A Symbolic Bisimulation Tool Box. In Susanne Graf and Wenhui Zhang, editors, *ATVA*, volume 4218 of *Lecture Notes in Computer Science*, pages 477–492. Springer, 2006.
- [45] Pierre Wolper and Denis Leroy. Reliable Hashing without Collision Detection. In Costas Courcoubetis, editor, *CAV*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer, 1993.