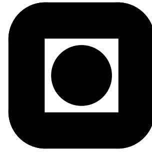
NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK OG ELEKTROTEKNIKK



HOVEDOPPGAVE

Kandidatens navn: Mads Henrik Stavang

Fag: Datateknikk

Oppgavens tittel (norsk):

Oppgavens tittel (engelsk): Object Oriented Concepts in Model Checking Tools

Oppgavens tekst:

A Model Checking Tool is tool designed for verification of finite-state machines specified in a model language. Primarily model-checking has been used for hardware verification. This thesis explores the capabilities of three leading-edge model checker with respect for their support of Object Oriented concepts.

For this purpose the available Model Checker's modeling languages are mapped and compared to imperative and Object Oriented Concepts.

Oppgaven gitt: 1. oktober 2003
Besvarelsen leveres innen: 1. april 2004
Besvarelsen levert: 31. mars 2004
Utført ved: Institutt for datateknikk og informasjonsvitenskap
Veileder: Elke Pulvermueller

Karlsruhe, 31. mars 2004

Reidar Conradi
Faglærer

Abstract

Model checkers or model checking tools are software that take models written in a language representing a finite-state machine as an input, and verify the correctness of the models using a given set of specifications.

To verify the correctness of a defined object oriented model, be it either UML or even a Java source program, a model checker could prove to be a useful tool. The problem is the semantic gap between an object oriented model and the capabilities of current model checking languages.

Before being able to close this gap, a set of model checking tools have to be examined. In this project SPIN, SMV and STeP are used. Each of these tools and their respective input languages are then analyzed, compared, and then implemented using a set of chosen object oriented. As suspected, not every object oriented concept can be implemented with the given model checking languages. However, several design proposals for mapping available model checking language concepts to object oriented constructs result from this thesis. Using an object oriented reference example these mappings are tested and successfully proven their functionality.

Preface

This graduate thesis was written at IDI - Department of computer and information science, NTNU. The thesis is a part of a project at the institute of Telematics, University of Karlsruhe.

I would like to thank Professor Reidar Conradi and Professor Gerhard Goos for making it possible for me to take part in this project at the University of Karlsruhe. I would also like to thank Dipl.Inf. Elke Pulvermueller for invaluable assistance and feedback throughout this thesis.

Karlsruhe, 31. mars 2004

Mads Henrik Stavang

Contents

1 Introduction **1**

1.1 Motivation and Problem Context 1

1.2 Project Goal 1

1.3 Project Process 2

1.4 Overview 2

2 Concepts **3**

2.1 Model checking 3

2.1.1 SPIN 4

2.1.2 SMV 5

2.1.3 STeP 5

2.2 Temporal languages 5

2.2.1 Computation tree logic 5

2.2.2 Linear-time temporal logic 12

2.2.3 CTL* 13

2.2.4 Fairness Constraints 13

2.2.5 Temporal logic in model checking tools 13

2.3 Model checking languages 15

2.3.1 Promela 15

2.3.2 SMV 15

2.3.3 SPL 15

2.4 Related Work 15

3 Analysis **17**

3.1 Overview . . . 17

3.2 Promela . . . 18

3.2.1 Composition of statements . . . 18

3.2.2 Conditional statements . . . 19

3.2.3 Looping constructs . . . 20

3.2.4 Data Representation . . . 21

3.2.5 Process activation . . . 23

3.3 SPL . . . 24

3.3.1 Composition of statements . . . 24

3.3.2 Conditional statements . . . 24

3.3.3 Looping constructs . . . 26

3.3.4 Data Representation . . . 26

3.3.5 Processes and procedures . . . 30

3.4 SMV . . . 31

3.4.1 Composition of statements . . . 31

3.4.2 Conditional statements and looping constructs . . . 31

3.4.3 Data representation . . . 32

3.4.4 SMV Modules . . . 35

3.5 Comparing the languages . . . 37

3.6 Case study . . . 37

3.6.1 Requirements specification . . . 38

3.6.2 The reference example . . . 38

3.6.2.1 Assigning an Advisor to the Customer . . . 40

3.6.2.2 Opening an Account for a Customer . . . 40

3.6.2.3 Depositing and withdrawing amounts from an Account 40

3.6.2.4 Transferring money between Accounts . . . 41

3.6.3 Requirements to the model implementation . . . 42

4 Design **45**
4.1 Promela design 45
4.1.1 Defining classes using a database table structure 46
4.1.2 Processes as class structure 46
4.1.3 Summary 50
4.2 SPL 50
4.3 SMV 52
4.3.1 The Class structure 53
4.3.2 The class methods 54
4.3.3 Sequential execution of statements 55
4.3.4 Summary 56

5 Implementation and tests **59**
5.1 Implementation 59
5.1.1 Promela 59
5.1.2 SPL 60
5.1.3 SMV 60
5.2 Testing 61
5.2.1 Promela tests 61
5.2.2 Promela results 63
5.3 Test conclusion 64

6 Summary and conclusion **67**

A Promela implementation part 1 **69**

B Promela implementation part 2 **75**

C SMV implementation **85**

Bibliography **91**

List of Figures

2.1 a) the Kripke structure. b) the computation tree 4
2.2 A system whose starting state satisfies EX ϕ 8
2.3 A system whose starting state satisfies EF ϕ 8
2.4 A system whose starting state satisfies EG ϕ 9
2.5 A system whose starting state satisfies E[ϕ_1 U ϕ_2] 9
2.6 A system whose starting state satisfies AX ϕ 10
2.7 A system whose starting state satisfies AF ϕ 10
2.8 A system whose starting state satisfies AG ϕ 11
2.9 A system whose starting state satisfies A[ϕ_1 U ϕ_2] 11

3.1 Syntax of Promela statements[Gert97] in Extended BNF 19
3.2 Syntax of Promela declaration in Extended BNF 22
3.3 Syntax of SPL statements in EBNF 25
3.4 Syntax of SPL variable declarations in EBNF 27
3.5 Syntax of SPL expressions in EBNF 28
3.6 SMV expressions in EBNF 32
3.7 SMV variable declaration and assignment in EBNF 33
3.8 SMV CTL specification in EBNF 33
3.9 SMV modules in EBNF 36
3.10 The UML diagram of the reference example 39
3.11 The sequence diagram for binding an Advisor to a Customer 40
3.12 The sequence diagram for opening an Account 41
3.13 The sequence diagram for transferring amounts between Accounts 41
3.14 The sequence diagram for depositing and withdrawing amounts from/to an Account 42

4.1 The Promela UML diagram 47

5.1 The last part of the program output after running the test case 2 64
5.2 Test Case 1: Initial values on the left-side and results on the right-side . . 65

List of Tables

2.1 LTL operators in SPIN syntax 14
2.2 CTL operators in SMV syntax 14
2.3 LTL operators in STeP syntax 14

3.1 Programming language concepts 37
3.2 The classes in the UML model 38
3.3 Concepts that are essential for the implementation of the reference example 44

4.1 Advantages and disadvantages to the different design issues for the Promela language 50
4.2 Advantages and disadvantages to the different design issues for the SMV language 57

5.1 The initial values in test case 1 62

1. Introduction

This thesis explores the various concepts of up-to-date model-checkers aiming at a comparison and mapping to current imperative and Object Oriented software concepts.

1.1 Motivation and Problem Context

Model checking is a technique for verifying the correctness of a computer system. The system is represented by a finite model $\mathscr{M}$ for an appropriate logic, and the specification is represented by a formula ϕ. The verification consists of computing if a model $\mathscr{M}$ satisfies ϕ, also written as $\mathscr{M} \models \phi$. The model $\mathscr{M}$ is a finite-state machine, and can be represented in a specific model checking language. The syntaxes of the different languages can vary, but they are all designed for easy conversion to a finite-state machine. The specification is coded in a temporal logic formula, often some form of CTL, LTL or CTL*.

The usage of model checking for verifying the correctness of a complex object oriented software is desirable, since a model checker can find paths in a program execution that fails to satisfy the defined specifications [HaDw01]. Before this can be done, one needs to bridge the semantic gap between object oriented software and the input languages of model checking tools [HaDw01].

There exists several model checking tools, each with its own model checking language. The model checking tools used for this assignment are SPIN, SMV and STeP, with their respective model checking languages, Promela, SMV language and SPL. These three tools have their advantages and disadvantages. Their suitability for bridging the gap between finite-state machines used in model checking and an object oriented software have to be explored with respect to detailed capabilities and restrictions of each model checking language, nd by applying their concepts to the Object Oriented model. The latter is the most challenging part [HaDw01].

1.2 Project Goal

The goal of this thesis is to explore the capabilities of the Model Checking Tools SPIN, SMV and STeP, mapping and comparing their modeling languages to imperative and Object Oriented Concepts.

1.3 Project Process

This project starts with the definition of the end goal of the project. This requires a prestudy, a necessary part to understand what model checking is, and how the input languages of the model checking tools differ from other known programming languages. The prestudy also includes a thorough analysis of the different languages, to categorize them and their syntax, and comparing them to imperative concepts which is a part of the goal. A case study is needed for the analysis, i.e. a reference example, and a set of object oriented concepts are chosen to be mapped from the modeling languages.

For every model checking language a design was made. Several possibilities in the design were presented for each language, where each design tried to map the modeling languages to the chosen object oriented concepts. A couple of these designs were used in the implementation. Some of the design issues needed to be manually tested, to prove their proper functionality. This is necessary since the possibilities and restrictions of the modeling languages are fairly unknown.

After the implementation some simple tests were run, and a conclusion of the work could be made.

The results of this thesis are the base for further improvements in the model checker CoV (Component Verifier). CoV is targeted to checking component software and high-level language concepts in particular. It has been developed at the Universitaet Karlsruhe in the context of the EU project EASYCOMP (IST 1999-14191) [Easy04].

1.4 Overview

The thesis is organized as follows: In chapter 1 an introduction to the goals, motivation and a summary of project process is presented. Chapter 2 explains the concepts of model checking tools, and also introduces the different tools used in this work. Chapter 3 contains the imperative concept analysis of each model language from the model checking tools, and defines object oriented concepts. Chapter 4 includes the design of the object oriented concepts defined in chapter 3. Chapter 5 discusses the implementation of the design and tests the implementation. Chapter 6 draws the conclusion of the work and suggests possible further work.

2. Concepts

In the following sections, the concepts of model checking and their tools will be presented. The first two sections explains unfamiliar concepts and are gone in greater depth. Section 2.1 introduces the reader to the concept of Model checking and the Model checking tools used in this project. Section 2.2 describes the fundamental concept of model checking, temporal logics. This section rely heavily on rules in logic. If the reader is well familiar with boolean operators and their logic, this section should be fairly understandable. Some of the concepts presented in this section are described in a very formal way, which is necessary to avoid ambiguity. To ease the reading figures and examples are therefore presented. The last section 2.3 introduces the programming languages used in the model checking tools, and are the basis for chapter 3, analysis.

2.1 Model checking

Sometimes being able to verify the correctness of a computer system can be of great help. Introducing some kind of verification techniques early on in the production cycle can be a great money saver, especially in commercial cases like mass-produced chips. The most obvious case where verifying the correctness is of great advantage is in safety-critical systems. Verification techniques can be summarized in three parts [HuRy00]

- A *framework for modeling systems*, typically a description language of some sort;
- A *specification language* for describing the properties to be verified;
- A *verification method* to establish whether the description of a system satisfies the specification.

There are at least two approaches for verification, proof-based and model-based. The proof-based approach tries to find a proof that $\Gamma \vdash \phi$. The Γ symbol represents a premise and ϕ a conclusion. Without going to deep into the semantic details of proof-based verification, the system description, Γ, is a set of formulas and the specification, ϕ, is another formula. Both are described in a suitable logic. Two appropriate logics are predicate logic and propositional logic[HuRy00].

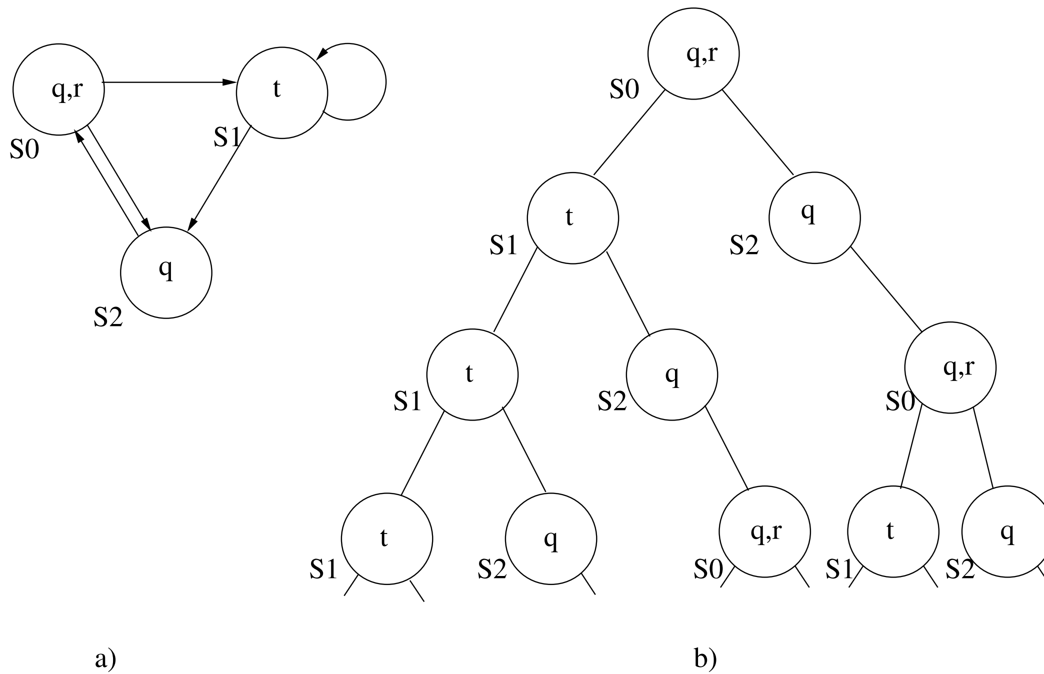


Figure 2.1: a) the Kripke structure. b) the computation tree

The Model-based approach tries to satisfy $\mathscr{M} \models \phi$. Here is $\mathscr{M}$ a model of a transition system and ϕ is a formula, both expressed in some form of temporal logic. The model $\mathscr{M}$ do not represent all of the real features of a computer system, rather an abstraction that is focused on the essential parts, namely those necessary for the checking of ϕ. ϕ represents the specification that the model $\mathscr{M}$ should satisfy. A typical example in a concurrent system is "There should not occur deadlocks". A transition system can be modeled as a directed graph. Each node in the graph represents a state and contains the values that are true in that state. A Kripke structure is such a graph, where each state has a set of boolean values that need to be true. This graph can be unwounded to an infinite computation tree.

In figure 2.1 the left graph is a model of a transition system. A transition system has one or more states. To understand states lets introduce a concept called *state transition*. Take the example x := x +1. Here a state transition from state s to state s' occurs, where state s' has an increased value of x, compared to s. If it is possible for a state s to reach state s' *in one computation step*, we write $s \rightarrow s'$.

In the sections 2.1.1-2.1.1 some model checking tools are introduced.

2.1.1 SPIN

SPIN is a model checker developed at *Bell Labs* in the beginning of the 1980. Since 1991 it has been available freely and was even awarded the prestigious *System Software Award* in April 2002. It is an efficient verification system for models of distributed software systems [Holz97]. Although its model checking abilities is perhaps its biggest strength it can also be used as a simulator and proof approximation system. The most efficient way to use SPIN is combined with the graphical user interface (GUI) Xspin. Xspin takes the advantage of the functionality of the SPIN tool and offers a window system with several dialogs for manipulating inputs. SPIN and Xspin may both be downloaded from [SPIN03].

2.1.2 SMV

The Symbolic Model Verifier (SMV) is model checking tool made at Carnegie Mellon University (CMU). It is especially designed for checking finite state systems, be it either completely synchronous or completely asynchronous. A common use of SMV is the verification of hardware. SMV can be downloaded from [SMV04].

2.1.3 STeP

The Stanford Temporal Prover (STeP) is a tool developed by the *REACT research group*. Even though STeP may be used as a model checker, it is just one of the many subsystems STeP provides. Among these are also theorem-proving and system verification. The GUI of STeP does not support a programming environment, only loading of already existing models. STep can be downloaded from [STeP03]

2.2 Temporal languages

In the following sections the different temporal languages, or temporal logics will be introduced. The sections 2.2.1-2.2.3 describe the three temporal logics CTL, LTL and CTL*. These represent the specification ϕ in the formula $\mathscr{M} \models \phi$. Section 2.2.4 describes Fairness Constraints, a concept that may be used together with any temporal language. In section 2.2.5 the usage of temporal logic in model checking tools are described.

2.2.1 Computation tree logic

Say we have a system. The model $\mathscr{M}$ of the system has three states, s_0, s_1, s_2. The only possible state transitions for this system are $s_0 \to s_1$, $s_0 \to s_2$, $s_1 \to s_1$, $s_1 \to s_2$, $s_2 \to s_0$. Each state contains a collection of atomic formulas that are true in that state. We denote this as L(s), which contains all atoms that are true in state s. If $L(s) \stackrel{def}{=} \{q, r, t\}$, then q, r, t are true if the system is in state s. Let $L(s_0) \stackrel{def}{=} \{q, r\}$, $L(s_1) \stackrel{def}{=} \{t\}$, $L(s_2) \stackrel{def}{=} \{q\}$. A representation for this model is on the left side in figure 2.1. Unwinding this model results in the infinite tree of all computation on the right side of the same figure. The computation tree logic, CTL, uses a temporal logic that is able to refer to one or more future states (or paths of states) of this model. To be able to use CTL, the model must satisfy some properties. Written formally: $\mathscr{M} = (S, \to, L)$, where every state $s \in S$ has some state s' where $s \to s'$, and the states are labeled such that

$L : S \to \mathscr{P}(Atoms)$

This means that there is a collection of states, S, where each state s in S has at least one next state, and each state also contains a set of atomic formulas, $L(s)$, that are true in state $s \in S$.

The CTL formulas written inductively via a Backus Naur form [1].

$\phi ::= \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \to \phi) \mid AX\phi \mid EX\phi \mid A[\phi \text{ U } \phi] \mid E[\phi \text{ U } \phi] \mid AG\phi \mid EG\phi \mid AF\phi \mid EF\phi$

Lets take a closer look on this definition. '$\bot$' and '$\top$' are CTL formulas which represents contradiction (always false) and tautology (always true). 'p' represents atomic formulas.

[1]Backus Naur form is a recursive representation of a programming syntax, often used to describe context-free grammars[HuRy00][Pars03][Seth96]

The rest are connectivities and modalities. For example is $(\neg\phi)$ a CTL formula if ϕ is one. The first four connectivities $\neg,\wedge,\vee,\rightarrow$ are similar to the boolean operators NOT, AND, OR, IF THEN (see [HuRy00] for more details). The modalities AX, EX, AG, EG, AU, EU, AF and EF on the other hand are in more unfamiliar territory. These are called *temporal modalities*[2]. Each temporal modality has a combination of two symbols. The first symbol is either A (along all paths) or E (along at least one path), also called path quantifiers. The second symbol is either X (next state), F (some future state), G (all future states, or globally), or U (until), usually referred as modalities. For example means $AG\phi$ that ϕ holds in all states, and $E[\phi_0 \text{ U } \phi_1]$ means that in at least one path ϕ_0 holds until ϕ_1.

If we combine a CTL model $\mathcal{M}$ (the system), and a CTL formula ϕ (the specifications) with a starting state s, we get:

$\mathcal{M}, s \models \phi$

This means that given any state $s \in S$ in the model $\mathcal{M}$, the formula ϕ holds. Using figure 2.1 as an example, $\mathcal{M}, s_0 \models r \vee t$ and $\mathcal{M}, s_1 \models t$ holds, but $\mathcal{M}, s_2 \models r \wedge q$ does not. The computation trees in Figures 2.2-2.9 show systems whose starting states satisfy the formulas EX ϕ, AX ϕ, EF ϕ, AF ϕ, EG ϕ, AG ϕ, EU ϕ, AU ϕ, respectively. Semantic definition[HuRy00]:

1. $\mathcal{M}, s \models \top$ and $\mathcal{M}, s \not\models \bot$ for all $s \in S$
2. $\mathcal{M}, s \models p$ iff $p \in L(s)$
3. $\mathcal{M}, s \models \neg\phi$ iff $\mathcal{M}, s \not\models \phi$
4. $\mathcal{M}, s \models \phi_1 \wedge \phi_2$ iff $\mathcal{M}, s \models \phi_1$ and $\mathcal{M}, s \models \phi_2$
5. $\mathcal{M}, s \models \phi_1 \vee \phi_2$ iff $\mathcal{M}, s \models \phi_1$ or $\mathcal{M}, s \models \phi_2$
6. $\mathcal{M}, s \models \phi_1 \rightarrow \phi_2$ iff $\mathcal{M}, s \not\models \phi_1$ or $\mathcal{M}, s \models \phi_2$
7. $\mathcal{M}, s \models$ AX ϕ iff *for all* s_1 such that $s \rightarrow s_1$ we have $\mathcal{M}, s \models \phi$. Thus AX says: 'in every next state'.
8. $\mathcal{M}, s \models$ EX ϕ iff *for some* s_1 such that $s \rightarrow s_1$ we have $\mathcal{M}, s \models \phi$. Thus EX says: 'in some next state'. E is dual to A - in exactly the same way that $\exists$ is dual to $\forall$ in predicate logic.
9. $\mathcal{M}, s \models$ AG ϕ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow ...$, where s_1 equals s, and all s_i along the path, we have $\mathcal{M}, s_i \models \phi$. Mnemonically: *for all* computation paths beginning in s the property ϕ holds *Globally*. Note that 'along the path' includes the path's initial state s.
10. $\mathcal{M}, s \models$ EG ϕ holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow ...$, where s_1 equals s, and for all s_1 along the path, we have $\mathcal{M}, s_i \models \phi$. Mnemonically: *there Exists* a path beginning in s such that ϕ holds *Globally* along the path.
11. $\mathcal{M}, s \models$ AF ϕ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow ...$, where s_1 equals s, there is some s_i such that $\mathcal{M}, s_i \models \phi$. Mnemonically: *for All* computation paths beginning in s there will be some *Future* state where ϕ holds.

[2]Temporal modalities [Lamp94] are sometimes referred as *temporal connectivities* [HuRy00]

12. $\mathcal{M}, s \models$ EF ϕ holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow ...$, where s_1 equals s, and for some s_i along the path, we have $\mathcal{M}, s_i \models \phi$. Mnemonically: *there Exists* a computation path beginning in s such that ϕ holds in some *Future* state;

13. $\mathcal{M}, s \models$ A[ϕ_1 U ϕ_2] holds iff for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow ...$, where s_1 equals s, that path satisfies ϕ_1 U ϕ_2, i.e. there is some s_i along the path, such that $\mathcal{M}, s_i \models \phi_2$, and, for each $j < i$, we have $\mathcal{M}, s_j \models \phi_1$. Mnemonically: *All* computation paths beginning in s satisfy that ϕ *Until* ψ holds on it.

14. $\mathcal{M}, s \models$ E[ϕ_1 U ϕ_2] holds iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow ...$, where s_1 equals s, that path satisfies ϕ_1 U ϕ_2 as specified in 13. Mnemonically: *there Exists* a computation path beginning in s such that ϕ *Until* ψ holds on it.

It is important to remember that the temporal connectivities and modalities are always pairs of A or E, and X, F, G, or U. As an example the following formulas are well-formed:

- AG p: p is true in every state
- E[r U t]: There exists a path, where r is always true, until an occurrence of t (which also exists in some future state).
- EF p $\rightarrow$ AX A[r U s]: If in some future state p is true, then in all its next states and the following states r is true until s.

The following formulas are not well-formed:

- E$\neg$X p
- EFG s
- AX [r U s]
- E [(t U u) $\wedge$ s]

Lets take a look on the third and the fourth formula. AX[r U s] is not well-formed since U is not paired with an A or an E. The E[(t U u)$\wedge$$s$] has a mismatched bracket. E[t U ($u \wedge s$)] and E[t U u] $\wedge$ s are two alternatives, both well-formed CTL formulas.

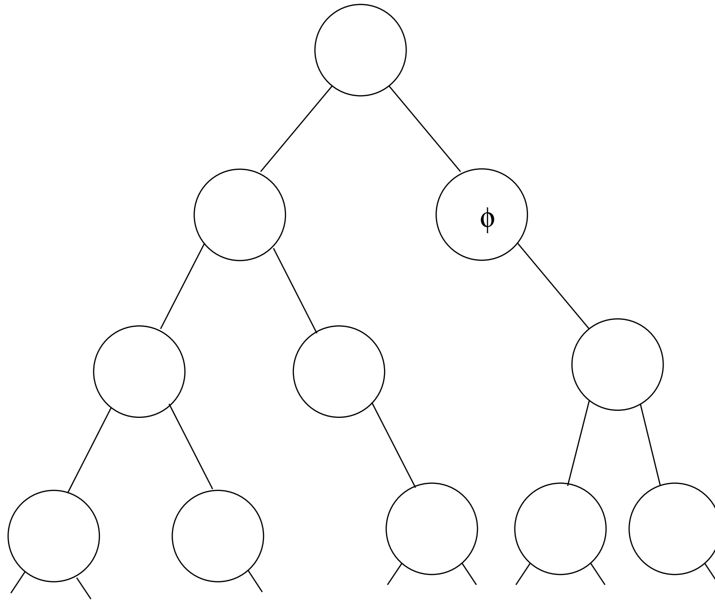


Figure 2.2: A system whose starting state satisfies EX ϕ

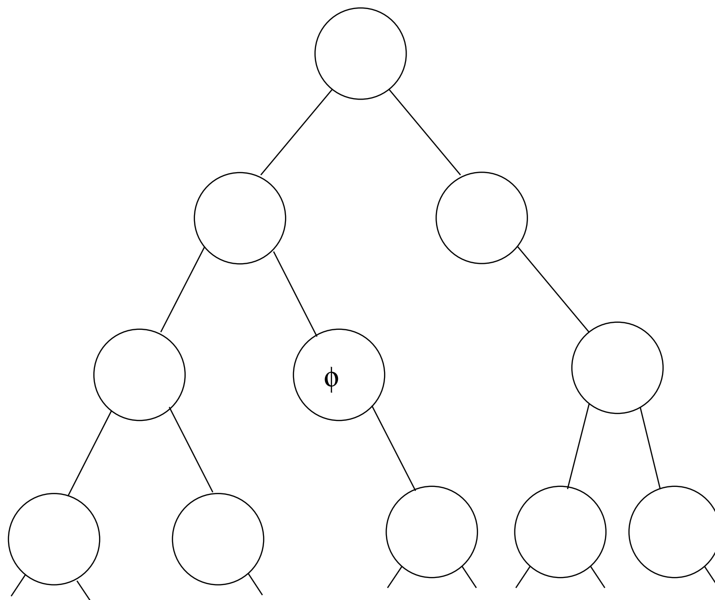


Figure 2.3: A system whose starting state satisfies EF ϕ

Figure 2.4: A system whose starting state satisfies EG ϕ

Figure 2.5: A system whose starting state satisfies E[ϕ_1 U ϕ_2]

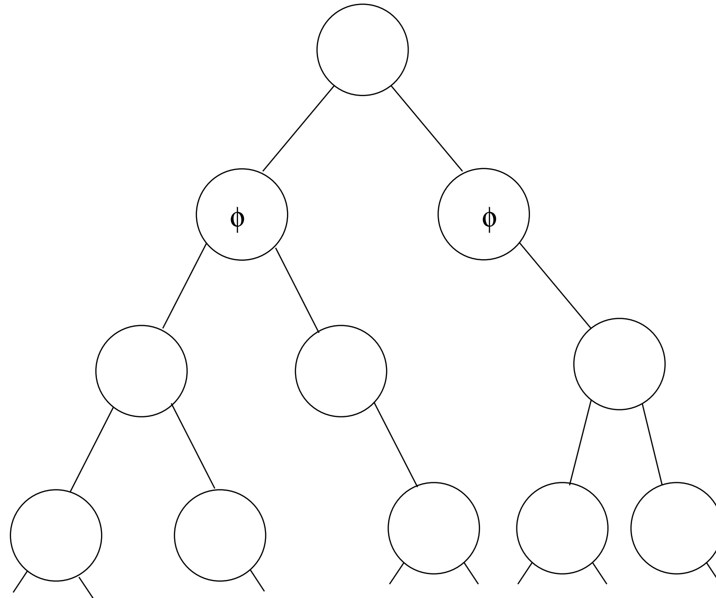


Figure 2.6: A system whose starting state satisfies AX ϕ

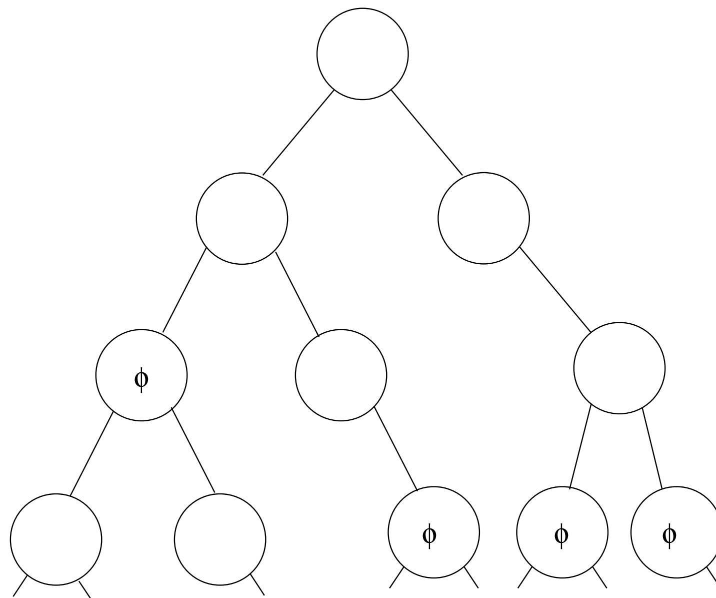


Figure 2.7: A system whose starting state satisfies AF ϕ

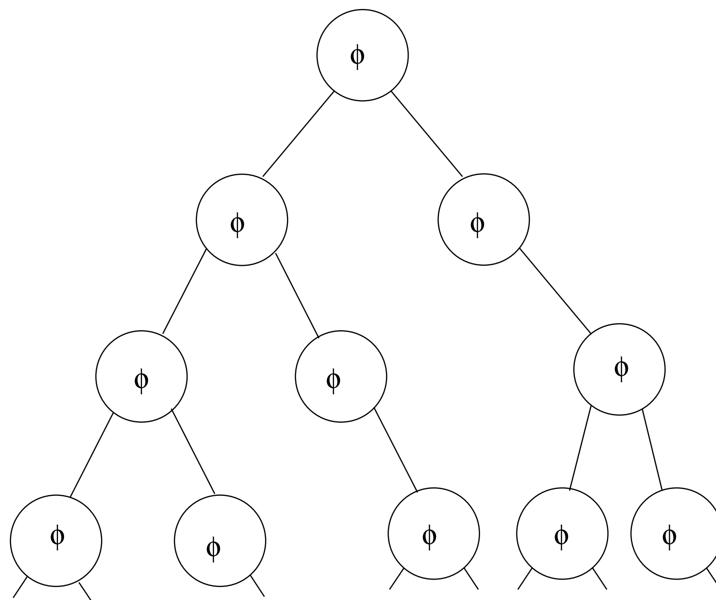


Figure 2.8: A system whose starting state satisfies AG ϕ

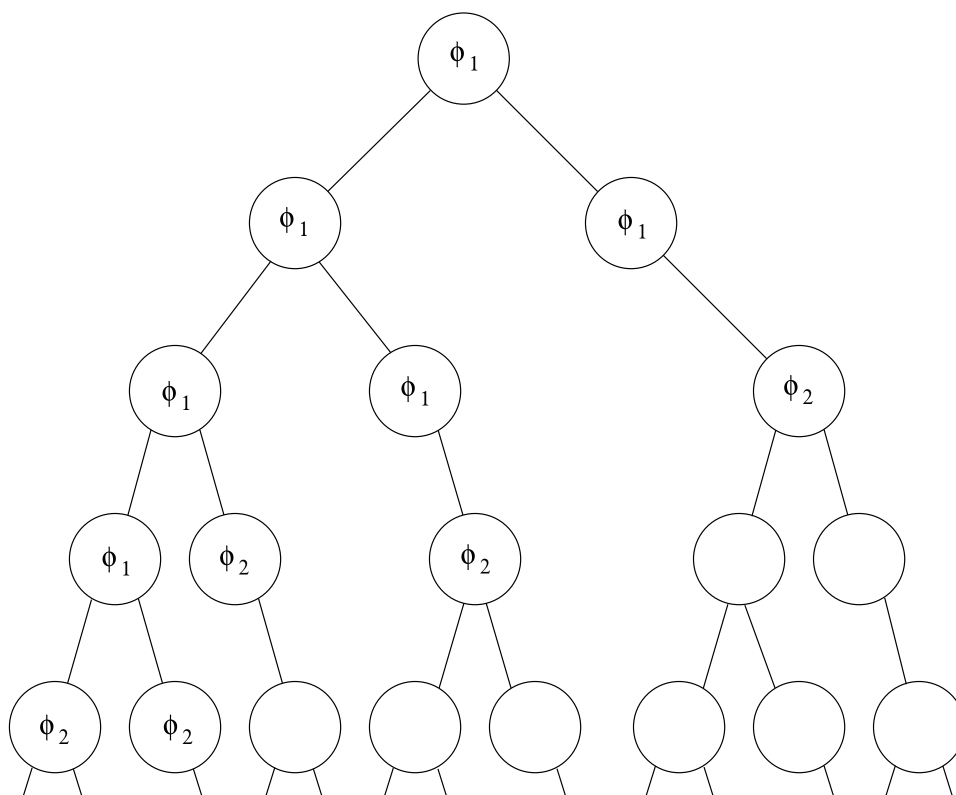


Figure 2.9: A system whose starting state satisfies A[ϕ_1 U ϕ_2]

2.2.2 Linear-time temporal logic

An alternative to CTL is linear-time temporal logic (LTL). The difference between CTL and LTL is that a LTL formula is evaluated on a single path, or a set of paths. This means that the quantifiers E and A are redundant. Excluding these quantifiers doesn't make LTL less expressive than CTL, since LTL is able to use nested boolean connectivities and modalities in a way not permitted by CTL. The LTL formulas written in Backus Naur form:

$\phi ::= \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (\phi \text{ U } \phi) \mid (\text{G } \phi) \mid (\text{F } \phi) \mid (\text{X } \phi)$

Given the path $\pi \stackrel{def}{=} s_1 \rightarrow s_2 \rightarrow ...$; let π^i be the suffix starting at s_i, then $\pi^i = s_i \rightarrow s_{i+1} \rightarrow ...$. If an LTL formula is evaluated on a set of path, the set satisfies ϕ if every path in the set satisfies ϕ.

Let $\mathscr{M} = (S, \rightarrow, L)$ be a model for LTL. Given the path $\pi \stackrel{def}{=} s_1 \rightarrow ...$, a LTL formula is satisfied when the following relations are true[HuRy00]:

1. $\pi \models \top$ and $\pi \not\models \bot$
2. $\pi \models p$ iff $p \in L(s)$
3. $\pi \models \neg\phi$ iff $\pi \not\models \phi$
4. $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$
5. $\pi \models \phi_1 \vee \phi_2$ iff $\pi \models \phi_1$ or $\pi \models \phi_2$
6. $\pi \models \phi_1 \rightarrow \phi_2$ iff $\pi \not\models \phi_1$ or $\pi \models \phi_2$
7. $\pi \models \text{X } \phi$ iff $\pi^2 \models \phi$
8. $\pi \models \text{G } \phi$ holds iff, for all $i \geq 1, \pi^i \models \phi$
9. $\pi \models \text{F } \phi$ holds iff, for some $i \geq 1, \pi^i \models \phi$
10. $\pi \models \phi \text{ U } \psi$ holds iff there is some $i \geq 1$ such that $\pi^i \models \psi$ and for all $j = 1, ..., i-1$ we have $\pi^j \models \phi$

Some example formulas written in LTL:

1. $\text{G}\neg(r \wedge s)$: r and s are both never true at the same time.
2. $\text{G}(p \rightarrow \text{F}q)$: Whenever p is true, q will eventually also be true.

2.2.3 CTL*

A combination of LTL's nested modalities and boolean connectivities, and CTL's path quantifiers E and A gives the logic CTL*. The syntax of this new logic can be divided in two classes[HuRy00]

- *state formulas*, which are evaluated in states:

  ϕ ::= $p \mid \top \mid \bot \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid \mathrm{A}[\alpha] \mid \mathrm{E}[\alpha]$,
  where p is any atomic formula and α any path formula; and

- *path formulas*, which are evaluated along paths:

  α ::= $\phi \mid (\neg\alpha) \mid (\alpha \wedge \alpha) \mid (\alpha \vee \alpha) \mid (\alpha \rightarrow \alpha) \mid (\alpha \mathrm{~U~} \alpha) \mid (\mathrm{G}\ \alpha) \mid (\mathrm{F}\ \alpha) \mid (\mathrm{X}\ \alpha)$,
  where ϕ is any state formula.

With CTL* we are able to construct formulas not possible with LTL or CTL[HuRy00]

- A [X $p \vee$ XX p]: along all paths, p is true in the next state, or the state after the next one.
- E [GF p]: there is a path along which p is infinitely often true.

2.2.4 Fairness Constraints

Sometimes the model $\mathcal{M}, s_0 \models \phi$ lacks some properties which lets its verification fail. This may occur because of some unrealistic behavior. Lets do an example. In figure 2.1 a), say we test the model with the CTL-formula AG (t →AF s). This means that for all states, if t is true, s will in all paths eventually be true. This CTL-formula fails though, because state s_1 (where t is true) may never go to state s_2. This is for most systems unlikely and we want to filter out these paths where s_1 never changes to another state. A way to do this is to introduce fairness constraints. A fairness constraint in this example is that whenever the system is in state s_1 it will eventually switch to state s_2. This is done by imposing the constraint !t infinitely often. By doing this the system will only evaluate paths where in every state, the value property !t will eventually be true. A *simple* fairness constraint is of the form:

Property ϕ is true infinitely often. [HuRy00]

2.2.5 Temporal logic in model checking tools

The SPIN tool uses LTL. The specifications can be directly written in a dialog window in XSpin. The symbols differ slightly from those used in section 2.2.2, as can be seen in table 2.1. For example, the LTL statement G¬($r \wedge s$) is the written in SPIN as *[]!(r && s)*. The symbols r and s are defined in the same dialog window in XSpin and specifies some properties of the model. The LTL statements are separately saved in an LTL specification file.

The SMV tool use CTL and the specifications are written directly in the model file. Only the boolean connectivities differ from the CTL syntax in section 2.2.1. The connectivies

LTL operator	SPIN syntax
G (*global*)	[]
F (*future*)	<>
X (*next*)	X
! (*negation*)	!
U (*until*)	U
∧ (*and*)	&&
∨ (*or*)	||
→ (*implication*)	->

Table 2.1: LTL operators in SPIN syntax

CTL operators	SMV syntax
EG/AG (*global*)	EG/AG
EF/AF (*future*)	EF/AF
EX/AX (*next*)	EX/AX
! (*negation*)	!
EU/AU (*until*)	EU/AU
∧ (*and*)	&
∨ (*or*)	|
→ (*implication*)	->

Table 2.2: CTL operators in SMV syntax

are listed in table 2.2. For example, the CTL statement EF p $\rightarrow$ AX A[r U s] is written in SMV as *EF p -> AX (r U s)*. The variables p, r and s are variables defined in the model language of SMV.

The STeP tool uses LTL and the specifications are written in a dialog window of the model checking tool. The connectivities used in STeP differ from those used in section 2.2.2, and are listed in table 2.3. Using the same example as in SPIN, the LTL statement $G\neg(r \wedge s)$ is in STeP written as *[] !(r ? s)*. The symbols r and s are specifications of the model, loaded separately with the model of the system. The LTL statements cannot be saved in a file.

LTL operator	STeP syntax
G (*global*)	[]
F (*future*)	<>
X (*next*)	()
! (*negation*)	!
U (*until*)	Until
∧ (*and*)	/\
∨ (*or*)	\/
→ (*implication*)	−−>

Table 2.3: LTL operators in STeP syntax

2.3 Model checking languages

The following sections introduce the model languages used in the model checking tools described in sections 2.1.1-2.1.3. A model checking language defines the model $\mathscr{M}$ in the formula $\mathscr{M} \models \phi$. A more detailed description of each language is done in chapter 3, where their syntax and semantics are closely examined.

2.3.1 Promela

The input language for the model checker SPIN is an imperative language is called Promela. Its syntax is very similar to the well-known programming language C. Coding Promela may be done with any text editor (e.g. emacs) or even with SPIN's own GUI, Xspin.

2.3.2 SMV

The Symbolic Model Verifier (SMV) defines its own language, often refered as SMV program. The SMV program syntax is designed for describing finite state machines, and incorperates CTL properties in the program itself, which means that the SMV tool and its input language are closely connected. The SMV language supports modules.

2.3.3 SPL

SPL is an imperative language similar to Pascal. The SPL origins from [MaPn95], and development team behind STeP has used [MaPn95] as basis for their SPL program. The syntax from [MaPn95] and [BBCF+98] differ slightly.

2.4 Related Work

Bandera is a tool that takes Java code as input and translates it to a finite state-model, suitable for model checking tools. Bandera is designed to be an open architecture, and can use several existing model checking tools to process the constructed finite-state model. Among these tools are SPIN and SMV. The biggest challenges in this finite-state model construction are the *state explosion problem*, the *model construction problem* and the *requirement specification problem* [HaDw01]. In our project, part of the *model construction problem* will be issued, which means the translation of an object oriented model into the input language of the model checking tools. Where the Bandera tool use Java as input language, we will directly implement an UML model, without using an object oriented programming language as a temporary stage towards a finite-state model.

ETAPS SC Workshop [Pulv02] is a project where the SMV structure is mapped to software components. The composition is at the SMV level and state-transitions symbolises method calls or control flow.

3. Analysis

To reach the goal of this project, a thorough analysis of the input languages Promela, SMV and SPL (see section 2.3) is needed to explore which programming concepts are available. Two of the programming languages are close in syntax of imperative languages, and we will therefore use the concepts found in imperative programming as basis for this analysis. This is a necessary step towards an object oriented program, since most of the basics in imperative languages, e.g. conditional statements and loops, are also found in object oriented languages. How each of the considered model checking languages fit in a object oriented system is discussed in chapter 4, Design.

In the first section 3.1 the imperative concepts which are to be analyzed are listed and explained. In the sections 3.2-3.3 each of the programming languages Promela, SMV and SPL are separately examined and compared to the concepts in section 3.1. In section 3.6 a case study is introduced. In this case study, a model is introduced which will serve as a reference example.

3.1 Overview

The different parts of imperative programming can be structured into the following sections[Seth96]:

1. Composition of statements

This section explains how a sequence of statements is represented.

2. Conditional statements

Here every conditional statement is listed. Typical conditional statements include *if-else* statements and *case-statements*.

3. Looping constructs

The most known looping constructs in imperative languages are:

while ’expression/condition’ **do** ’a set of statements’
repeat ’a set of statements’ **until** ’an expression/condition’ (similar to while-statements, except that the condition is evaluated after the sequence of statements is executed.
for ’initialization’ **to** ’an expression/condition’ **do** ’a set of statements’

4. Data representation

In this section the data types are listed, e.g. primitive types (integer, boolean), user defined types, arrays.

5. Procedure activation

How procedure calls work in the given language.

3.2 Promela

The Promela language use *processes* instead of procedures, so instead of *Procedure activation* section 3.2.5 is called *Process activation*. The difference will be explained in that section. Promela is the input language of the model checker SPIN.

3.2.1 Composition of statements

The control flow of a Promela program is evident from the syntactic structure of the program text. It follows sequentially through a sequence of statements. Example :

```
count = x; x = y; y = count+1
```

This sequence of statements can be grouped into a compound statement by enclosing it between the brackets { and }. A compound statement for the above sequence is:

```
{count = x; x = y; y = count+1}
```

As seen in the examples above, the separation of statements is used by a semicolon ’;’. In most cases this separator can be replaced with a two-character arrow symbol ’->’ without change of meaning. Using the examples above a sequence of statements can also be written like:

```
count = x; x = y-> y = count+1
```

The separator ’->’ is often used as an informal way to indicate an causal relation of two statements. This is more thorough described in 3.2.2

```
<stmnt>  ::= IF <options> FI
       | DO <options> OD
       | ATOMIC '{' <sequence> '}'
       | D_STEP '{' <sequence> '}'
       | '{' sequence '}'
       | <send>
       | <receive>
       | <assign>
       | ELSE
       | BREAK
       | GOTO <name>
       | <name> ':' <stmnt>
       | PRINT '(' <string> [ ',' <arg_lst> ] ')'
       | ASSERT <expr>
       | <expr>

<sequence>::= <step> { ';' <step> }

<step>  ::= <stmnt> [ UNLESS <stmnt> ]
       | <decl_lst>
       | XR <varref> {',' <varref> }
       | XS <varref> {',' <varref> }

<options> ::= ':' ':' <sequence> { ':' ':' <sequence> }
```

Figure 3.1: Syntax of Promela statements[Gert97] in Extended BNF

3.2.2 Conditional statements

The most simple conditional statement in Promela is

```
(a == b)
```

This means that until the value of variable 'a' equals the value of variable 'b', the will execution block and will not proceed to the next statement. An equivalent expression is:

```
/* wait for a == b */
while(a != b)
  skip /* Do nothing */
```

Like all other statements, the blocking statement can be a part of a sequence of statements

```
(a==b)-> count = x; x = y; y = count+1
```

This means that count, x and y get their values set only if a == b, and until then, are blocked. Notice that '->' is used as a separator instead of ';' after the conditional statement, for easier reading when doing code reviews. See figure 3.1 for a concise description of Promela statements in Extended BNF[1]

Like most other imperative languages, Promela also has IF statements. This conditional statement function a bit different than "normal" IF statements. Example:

[1]Extended Backus Naur Form (EBNF) is simply an extension of BNF, introducing some shortcuts[Seth96]

IF (a > 10) THEN a := 10
ELSE IF (a <= 10) THEN a := 0

The equivalent expression in Promela:

```
IF
 :: (a > 10) -> a = 10
 :: (a <= 10) -> a = 0
FI
```

The IF statement does not have an ELSE clause (see figure 3.1); it actually more resembles a CASE statements. Like the simple conditional statement (a == b), an IF statement can be blocked if none of the conditions preceding the double colons '::' are true. The first statement following'::' are normally referred as a *guard* [RaRH03]. Take a look at the following two IF statements, the second done in Promela syntax:

IF (a < 10) THEN a := 10
ELSE IF (a < 100) THEN a := 100

This roughly translates to:

```
IF
 :: (a < 10) -> a = 10
 :: (a < 100) -> a = 100
FI
```

These two statements are not equivalent. In the first example, if 'a' is less than 10, then 'a' gets the value 10. If its not less than 10, but less than 100, it gets the value 100. If its bigger than 100 the IF ELSE statement is skipped. In the second example, if 'a' is less than 10, then the IF statement has two conditions that are true. Promela however does not execute the first statement which is true. It evaluates all conditions, or guards. If more than one guard is true, then one of these conditions is selected nondeterministically. This means that one of the sequence of statements following the guards will be selected. 'a' will arbitrary be given either the value 10 or 100. If 'a' has some value between 10 and 99, 'a' gets the value 100 (like in the first example above). If its 100 or more, the IF statement in Promela blocks, until it is true.

CASE statements doesn't exist in Promela.

3.2.3 Looping constructs

Promela has only an indefinite looping construct, namely a DO statement. Its syntax and execution is almost identical to Promela's IF statement. The only difference is that the evaluation of the guards is repeated until the loop is terminated. This is normally done with a break statement. The following three examples are directly taken from [SPIN96]:

```
do
:: count = count + 1
:: count = count - 1
:: (count == 0) -> break
od
```

Here the DO statement terminates only if 'count' has the value 0. Since the first two guards are always true, it doesn't need to terminate when 'count == 0' is true. If we want it to terminate whenever 'count' reach the value 0, we can write:

```
do
:: (count != 0) ->
   if
   :: count = count + 1
   :: count = count - 1
   fi
:: (count == 0) -> break
od
```

Another way to terminate a DO statement, is to use the well known **goto** statement:

```
do
:: (x >  y) -> x = x - y
:: (x <  y) -> y = y - x
:: (x == y) -> goto done
od;
done:
skip
```

3.2.4 Data Representation

Promela has the following basic types:

bit, bool, byte, short, int

As we can see, Promela does not have floating point types. These types can be named, in a declaration of the form:

```
<typename> <name> = <any_expr>
```

Arrays are also possible:

```
<typename> <name> [<number>]
<typename> <name> [<number>] = <number>
```

Examples:

```
int n1;
byte car = 3;
short houses[3]; /* all values initialized with 0 */
int bricks[2] = 3 /* all values initialized with 3 */
```

```
<decl_lst> ::= <one_decl> { ';' <one_decl> }

<one_decl> ::= [ <visible> ] <typename> <ivar> {','<ivar> }

<typename> ::= bit | bool | byte | short | int | mtype | chan
        | <uname>

<ivar>  ::= <name> [ '[' <const> ']' ] [ '=' <any_expr>
        | '=' <ch_init> ]

<ch_init> ::= '[' <const> ']' OF '{' <typename>
          { ',' <typename> } '}'

<mtype>  ::= mtype [ '=' ] '{' <name> { ',' <name> } '}'

<utype>  ::= typedef <name> '{' <decl_lst> '}'
```

Figure 3.2: Syntax of Promela declaration in Extended BNF

Like in c, array indexes start with 0 and end with the length of the array minus one. Multi-dimensional arrays are not supported.

Constants can be defined with an mtype declaration. Example:

> mtype = some, different, names; mtype variable = different;

There can be only one mtype-definition declared in Promela.

User-defined types, also known as records [Seth96], are supported with the typedef command:

> typedef Vector
> int x;
> int y;
> int z;

A field *f* in a record *R* can be accessed by writing R.f:

```
Vector position;
position.x = 1;
position.y = 5;
position.z = -3
```

Promela does not have types like Set or even Pointers. What Promela does have is a type used for message exchange between procedures, namely **chan** (see also <ch_init> in figure 3.2)

```
chan msg = [3] of {int} /* Can store up to 3 messages of type int *
chan msgret = [1] of {int, chan} /* Can store only 1 message */
\\
msg!send1 /* putting the value of send1 into msg */
msg?recv1 /* putting first value from msg into recv1 */
msgret!recv1,msg /* putting both recv1 and msg into msgret */
```

Channels pass the messages in first-in-first-out order. Each message can have one or more values, where each value is a primitive type, a user-defined one or a channel. A 'msg!expr' will only execute if 'msg' is not full, else the statement will block. The same way 'msg?expr' will only execute when 'msg' is not empty. Trying to send or receive fewer or more parameters than the channel is declared for, results in an error message.

3.2.5 Process activation

The procedures in Promela are called Processes. The reason for this is that each process that is invoked runs concurrent, which means that the statements in different processes can interleave each other. All statements that are not global declarations, constants or user-defined types, need to be typed inside a process. The main process in Promela is called **init** and takes no arguments. The init process is first run by program start, and can call one or more processes if needed. Lets take a look on the elements of a Process:

```
<proctype> ::= [ <active> ] proctype <name> '('
           [ <decl_lst> ]')' [ <priority> ] [ <enabler> ]
           '{' <sequence> '}'

<active>   ::= active [ '[' <const> ']' ]

<init>   ::= init [ <priority> ] '{' <sequence> '}'
```

As seen here, an init process has some restriction compared to a normal process. The <sequence> is the process body, <decl_lst> is the Process parameter list and <name> is the name of the Process. The optional <enabler> and <priority> we will not explain in detail, where the first is some precondition for running the process, and the latter is a priority value for the process. A process can be named <active> if there exists no **init** process. The *active* process will then be the first to be run. Example:

```
init {
  int i=0;
  do
  :: i=i+1
  :: (i>10) -> break
  od
}
...
msg(bool send, chan b) {

  if
  :: (send == true) -> b!val /* val being some global value */
  :: (send == false) -> b?val
  fi
}
```

If more than one *active* processes are declared, or if both an *init* process and *active* processes are declared, they are all executed at program start, and run concurrently. To instantiate a process, we need to invoke its name and its parameters. This is done with a **run** statement

```
init {
  run a(); run b(2)
}

a() {
  printf(``Process a'')
}

b(int val) {
  printf(``Process b, value = \%d``, val)
}
```

A process can be invoked from all processes not only in *init*. A process can even invoke itself, making recursion possible.

Finally, all process parameters, except for **chan** are passed as call-by-value. **chan** is passed as call-by-reference.

3.3 SPL

The *Procedure activation* section has been changed to *Process and procedures*, since *Procedure activation* is inaccurate. This will be further explained in section 3.3.5. SPL is the is the input language of the model checker STeP.

3.3.1 Composition of statements

As in Promela SPL uses ';' as a separation of statements, *not* as a statement terminator. This means that there are no semicolon ';' after the last statement. Each statement is then executed in sequence. Other statement separators are '||' and 'OR'. These support *non deterministic selection* and interleaving of statements (see 3.3.2 and 3.3.5). In addition each statement must be labeled with an identifier in the program code. Example:

l0 : count = x; l1 : x = y; l2 : y = count+1

The same example can be grouped as a compound statement, where '[' and ']' are used to enclose the statement. Example:

l0 : [count = x; l1 : x = y; l2 : y = count+1]

As shown in the example, a compound statement may also be labeled.

3.3.2 Conditional statements

The simplest conditional statement in SPL is the **await** statement of the following form:

await c

```
<composite_stmt> ::= <composite_stmt> ';' <composite_stmt>
                 |  <composite_stmt> 'V' <composite_stmt>
                 |  <composite_stmt> OR <composite_stmt>
                 |  [<label>] <stmt>

<stmt>           ::= <basic_group_stmt>
                 |  REQUEST <variable>
                 |  RELEASE <variable>
                 |  NONCRITICAL
                 |  CRITICAL
                 |  CHOOSE <variable>
                 |  PRODUCE <variable>
                 |  CONSUME <variable>
                 |  GUARD <p_expn> DO <assignment>
                 |  IF <p_expn> THEN <composite_stmt>
                 |  IF <p_expn> THEN <composite_stmt>
                             ELSE <composite_stmt>
                 |  WHEN <p_expn> DO <composite_stmt>
                 |  WHILE <p_expn> DO <composite_stmt>
                 |  REPEAT <composite_stmt> UNTIL <p_expn>
                 |  LOOP FOREVER DO <composite_stmt>
                 |  FOR '('[<composite_stmt>], [<p_expn>],
                    [<composite_stmt>]')' <composite_stmt>
                 |  BREAK
                 |  RAISE <id>
                 |  <stmt> HANDLE <id>
                 |  <stmt> UNLESS <p_expn>
                 |  LOCK <p_epxn> <stmt>
                 |  [id :: ] '[' <procedure_body> ']'
                 |  '≪' <comp_group_stmt> '≫'
                 |  OR <binding> <composite_stmt>
                 |  'V' <binding> <composite_stmt>

<basic_group_stmt> ::= SKIP
                   |  <assignment>
                   |  await <p_expn>
                   |  <variable> ≤= <p_expn>
                   |  <variable> ==> <variable>
```

Figure 3.3: Syntax of SPL statements in EBNF

This statement does nothing until the boolean value 'c' is true. It then terminates and allows the subsequent statements to execute. This is useful for synchronisation.

The most common conditional statement is the IF-clause. Like in other imperative languages it can be either a *one-branch-conditional* statement or has a following ELSE-clause. Example:

if (a>b) then x:= 3 else b:=a

Another way to do a conditional selection is by using a *selection statement*. If we have the statements S_1, S_2, S_3 a *selection statement* is written as S_1 **or** S_2 **or** S_3. This means that one of the statements $S_1..S_3$ will be non deterministically chosen and executed. Only enabled statements are selected, and one way to chosen which are determined is using the *when*

statement(**when** c **do** $S \Leftrightarrow$ **await** c; S) and forming a *guarded command*. Assuming that c_1, c_2 and c_3 are the conditions to be evaluated for S_1, S_2, S_3, respectively. Then guarded command can be written as follows:

[**when** c_1 **do** S_1] **or** [**when** c_2 **do** S_2] **or** [**when** c_3 **do** S_3]

This is equivalent with Promela's IF-statement. An example written SPL:

```
l0 : when (a < 10) do
l1 : a := 10
     or
l2 : when (a < 100) do
l3 : a := 100
```

The outcome is identical to the corresponding IF-statement example in section 3.2.2.

3.3.3 Looping constructs

The SPL language allows the conventional looping constructs *while, repeat* and *for* (see figure 3.3 for syntax). A typical implementation looks as follows:

```
l0 : while (x < y) do [
l1 :   y := y/2;
l2 :   x := x+1
]
```

SPL also allows the *loop forever* statement (**loop forever do** S), which is an abbreviation of the statement (**while** true **do** S), where 'true' is a boolean value and S is a statement.

3.3.4 Data Representation

SPL has three basic types **bool, int, rat**[2]. When declaring a variable of a type, this is done at the start of the program. Example:

```
in a : int
local x, y : bool
out b : int
```

When declaring a variable, the programmer must define its *mode* [MaPn95]

in - Specifies the input variables to the program.

local - Specifies variables that are local to the program. These variables are used in the execution of the program, but are not visible outside the program.

[2]These types are not present in the SPL context-free grammar in [BBCF+98], but in the appendix they can be found.

```
<procedure_body> ::= { <decl> [;] } <composite_stmt> [; <label>]

<decl>          ::= <type_decl> | <datatype_decl>
                |   <value_decl> | <macro_decl>
                |   <system_var_decl>

<system_var_decl> ::= <mode> <ids> : <type> [ WHERE <p_expn> ]

<mode>          ::= IN | OUT | LOCAL

<ids>           ::= <id> {, <id>}

<id>            ::= <alpha> { <alpha> | <digit> | '_' }

<assignment>    ::= <left_val> ':=' <p_expn>

<left_val>      ::= <p_expn>
```

Figure 3.4: Syntax of SPL variable declarations in EBNF

out - Specifies the output variables of the program.

Its possible to specify constraints or initial values in the variable declaration. This is done with an optional **where** element. Example:

```
(* A is constrained to values between *)
(* 1 and 9 *)
in a : int where (a > 0 /\ a < 10)
local x, y : bool where x = false, y = true
out b : int where b = -1
```

Variables declared as **local** or **out** are restricted to initial value specification only, on the form *y := e*. In this example, variable 'a' can only have values from 1 and to 9, and the variables 'x', 'y' and 'z' have their initial values set to 'false', 'true' and '-1', respectively.

The variables can be declared to consist of several types. This is realised by concatenating the different types with the symbol '*'. In the following example we define a point with two integer values (x and y):

```
in point : int*int
```

User defined types are also allowed. They can consist of already existing types, or even define new ones. These are defined in the program with a **type** statement. Example:

```
type point = int*int
type system = {lefthand, righthand}
in position : point
local reference : system where reference = righthand
```

```
<p_expn>  ::= <bool_const>
           |  <int_const>
           |  <id>
           |  '≪' '≫'
           |  '≪' <p_expns> '≫'
           |  <id>'('<p_expns>')'
           |  <expr>'['<p_expns>']'
           |  '('<p_expn>, <p_expns>')'
           |  { <id> '=' <expn> { ';' <id> '=' <expn> } [';'] }
           |  '#' (<int_const> | <id>) <p_expn>
           |  '@' (<int_const> | <id>) '('<p_expn>, <p_expn>')'
           |  <p_expn> <infix> <p_expn>
           |  <prefix> <p_expn>
           |  IF <p_expn> THEN <p_expn> ELSE <p_expn>
           |  <bind> <ids> : <type>'.'<p_expn>

<p_expns> ::= <p_expn> | <p_expn> ',' <p_expns>

<infix>   ::= '/\' | '\/' | '—>' | '<—>'
           |  '+' | '*' | '−' | '/' | MOD | DIV
           |  '=' | '!=' | '<' | '>' | '≥' | '<' | '<>'

<prefix>  ::= '!' | '~' | '−'
           |  APPEND | HEAD | TAIL | LENGTH
           |  UPDATE

<bind>    ::= FORALL | EXISTS | EXISTS!

<bool_const> ::= TRUE | FALSE

<int_const> ::= <digit> {<digit>}
```

Figure 3.5: Syntax of SPL expressions in EBNF

Variable assignment can be done on a trivial one-to-one basis, or one can use multiple variable assignment. Assume that we have a list of variables $u_1, ..., u_k$ and a list of expressions $e_1, ..., e_k$ of corresponding types, then an *assignment statement* is on the form:

$$(u_1, ..., u_k) := (e_1, ..., e_k)$$

A special case is if $k = 1$. Then, the expression is simplified to $u := e$. An example of variable assignment:

```
(* Switch the values in x and y *)
(x,y) := (y,x)
```

If a variable has more than one value, the respective values are placed between two parentheses, separated with commas. Example:

```
(* A user defined type: system *)
(* with constants: lefthand and righthand *)
type system = {lefthand, righthand}
```

```
(* Declaring 2 points, xy and origo *)
local xy, origo : int*int
(* Declaring variable reference of type system *)
local reference : system

(* assigning point origo to be (0,0) *)
l0 : origo := (0,0);
(* Assigning point xy to be: (2,3), and *)
(* reference to get the value: lefthand *)
l1 : (xy, reference) := ((2,3), lefthand)
```

Other available types are **array** and **channel**[3]. Their declarations follow the form: [BBCF+98]

/* Multi dimensional arrays of type T */
array $[n_1..m_1, \ldots, n_k..m_k]$ of T
/* Channel for synchronized communication */
/* (Without queue) */
channel of *b*
/* Unbounded channel */
channel [1..] **of** *b*
/* Bounded channel */
channel $[n..m]$ **of** *b*

The difference between *range* and *ranges* above is that an **array** may be declared as multidimensional (hence the plural form), and a **channel** has only one *range*. This can be illustrated with some examples:

```
(* no negative range *)
in x,y : int where (x > 0 /\ y > 0)
(* 2-dimensional array of type integer *)
in m : array [1..x, 1..y] of int
(* channel store 10 messages of type integer *)
local : channel [1..10] of int

out n : array [1..x, 1..y] of int
```

Referring to an array element is done with '[]'. If *a* is a two-dimensional array then $a[2,3]$ is the value in position (2,3).

A **channel** is processed in the same way as Promela's **chan** construct (see section 3.2.4), except for that channels in SPL may contain an unlimited number of messages. The syntax for sending and receiving messages are somewhat similar. Example:

```
a <== m  /* send: message 'm' is sent via channel 'a' */
a ==> m  /* receive: the first message in channel 'a' */
         /* is stored in message 'm' */
```

[3]As with the basic types, these types are also not present in the context-free grammar of SPL in [BBCF+98], but in its appendix.

Two other statements worth to mentioning are the **request** and the **release** statements. The **release** statement increments an integer value by 1. **request** decrements an integer by 1, but only if the value is positive. These are useful in semaphore operations

```
request t; /* t is decreased by 1 */
release t; /* t is increased by 1 */
```

3.3.5 Processes and procedures

An SPL program may consist of one or more processes, where the processes cooperate with each other in some way. The program first starts with the variable declarations, and continues then with one or more statements. The processes and the program itself may use labels, but this is optional. If two processes are to cooperate with each other, they are separated with a token '||'. This means that the statements in the two processes interleave, where every statement to be executed is nondeterministically chosen. A typical program with *cooperating statements* may be found in the following example:

```
in a : int
local i : int where i = 1
out k : int where k = a

P1 :: [
  l0 : request i;
  l1 : k = k + 1;
  l2 : release i
]
||
P2 :: [
  m0 : request i;
  m1 : k = k -1;
  m2 : release i
]
```

A *Cooperation statement* can also be declared inside a process, making them *internal processes*. In SPL one is also allowed to declare *procedures*. This is done with the **procedure** keyword. When the program calls the procedure, the compiler includes the body of the procedure. A procedure is written as follows:

procedure *id*(*[mode] id:type,...,[mode] id:type*)[*stmt*]

The *mode* of the parameter arguments are restricted to **in** and **out**. All **local** declarations of variables inside a procedure are treated as global values. Calling the procedure is done by calling its name and enclose its parameters with parentheses, separating the arguments with commas. Example

```
in a : int
out b : int
```

```
procedure P(in i : int, out j : int) [
l0 : j := i +1
]

l1 : P(a,b)
```

3.4 SMV

SMV does not have any procedures, so a *Procedure activation* section is omitted. Instead, *SMV modules* are included, which do not have any similar concept in imperative languages. SMV is the input language of the model checker with the same name.

3.4.1 Composition of statements

Unlike Promela, SMV uses the semi-colon ';' as a statement terminator, hence there is a ';' after each statement ending. The statement flow looks then similar to most imperative languages. Some statements come under certain labels, which will be further discussed in the following chapters. There are a few exception to the ';' rule. Some statements are restricted to only one expression, and only under certain labels, and need therefore no ';' terminator, since its syntax restricts itself. These exceptions will also be further explained.

3.4.2 Conditional statements and looping constructs

Since the SMV input language is designed for finite-state machines, the syntax and its execution differ somewhat from most imperative languages. Actually, its only conditional statement is a CASE statement. Each CASE statement have one or more pairs of expressions, unlike the typical CASE statements e.g. in C, where the pairs consist of a constant and a statement. The SMV CASE statement is similar to an IF ELSE clause, since the order of the expression-expression pairs effect the outcome. Example [McKi00]:

```
case
  state = ready & request : busy;
  1 : {ready, busy}
esac;
```

The expressions left of the colon ':' are those to be evaluated, starting with the first expression. In this case 'state = ready & request'. If 'state' equals 'ready' is true, and the boolean value 'request' is true, then the result is the expression 'busy'. If not, the next pair is executed. This time it is '1' which is the boolean value for true, hence always true. The resulting expression is either 'ready' or 'busy', which is randomly chosen in run-time.

SMV does not have any looping constructs. However, since it is a finite-state machine language each program is in a infinite iteration, where every iteration computes its next state transition. The states and its transitions are defined in its variables (chapter 3.4.3).

```
<expr> ::= <atom> /* a symbolic constant */
     |   <number>
     |   <id>     /* a variable identifier */
     |   '!' <expr>
     |   <expr> '&' <expr>
     |   <expr> '|' <expr>
     |   <expr> '->' <expr>
     |   <expr> '<->' <expr>
     |   <expr> '=' <expr>
     |   <expr> '<' <expr>
     |   <expr> '>' <expr>
     |   <expr> '≤' <expr>
     |   <expr> '≥' <expr>
     |   <expr> '+' <expr>
     |   <expr> '-' <expr>
     |   <expr> '*' <expr>
     |   <expr> '/' <expr>
     |   <expr> 'mod' <expr>
     |   NEXT '(' <id> ')'
     |   <set_expr>
     |   <case_expr>

<case_expr> ::= CASE <expr> ':' <expr> ';'
             { <expr> ':' <expr> ';' }
             ESAC;

<set_expr> ::= '{' <val> { ',' <val> } '}'
         |   <expr> 'in' <expr>
         |   <expr> 'union' <expr>

<val>    ::= <atom> | <number>
```

Figure 3.6: SMV expressions in EBNF

3.4.3 Data representation

SMV's only primitive type is **boolean**. The boolean value is false for 0 and true for 1. The declaration of variables is done using the keyword **VAR**. Example:

```
VAR
  busy : boolean;
  readay : boolean;
```

Other variable types (see figure 3.7) are user defined types, arrays and even modules, where the latter will be better explained in chapter 3.4.4. The user-defined types are a list of values which the variable can take. The array is defined by its upper and lower bound. Example:

```
VAR
  state : {busy, ready};
  buttons : array 0 .. 4 of boolean;
  windows : array 1 .. 3 of {open, close};
```

```
<decl_var> ::= VAR
        <atom> : <type> ';'
        {<atom> : <type> ';'}

<type>   ::= <boolean>
        |  '{' <val> {',' <val>} '}'
        |  ARRAY <expr> '..' <expr> OF <type>
        |  <atom> [ '(' <expr> {',' <expr>} ')' ]
        |  PROCESS <atom> [ '(' <expr> {',' <expr>} ')' ]

<decl_assign> ::= ASSIGN
        <dest> ':=' <expr> ';'
        {<dest> ':=' <expr> ';'}

<dest>   ::= <atom>
        | INIT '(' <atom> ')'
        | NEXT '(' <atom> ')'

<decl_init> ::= INIT <expr>

<decl_trans> ::= TRANS <expr>

<decl_define> ::= DEFINE <atom> ':=' <expr> ';'
                  {<atom> ':=' <expr> ';'}
```

Figure 3.7: SMV variable declaration and assignment in EBNF

```
<decl_spec> ::= SPEC <ctlform>

<ctlform> ::= <expr>
        |  '!' <ctlform>
        |  <ctlform> '&' <ctlform>
        |  <ctlform> '|' <ctlform>
        |  <ctlform> '->' <ctlform>
        |  <ctlform> '<->' <ctlform>
        |  E <pathform>
        |  A <pathform>

<pathform> ::= X <ctlform>
        |  F <ctlform>
        |  G <ctlform>
        |  <ctlform> U <ctlform>
```

Figure 3.8: SMV CTL specification in EBNF

The assignment of values of declared variables is done under the keyword **ASSIGN**. Each variable can be either assigned a single value (which means that the left side is simply equal to the right side) or a set of two values, one **init** value and one **next**. **init** is the value the variable gets at initial state. **next** is value the variable gets by each state transition (or iteration). The right side of a variable assignment can be a list of possible values. If this is the case, one of the values listed is randomly selected. Example:

```
ASSIGN
  init(state) : ready;
```

```
  next(state) : {ready, busy};
```

Here the initial value of 'state' is 'ready' and in every next state its value can be either 'ready' or 'busy'. Case statements are also often used

```
ASSIGN
  next(state) :
              case
                state = ready & request : busy;
                1 : {ready, busy}
              esac;
```

Here the initial state is not defined, so the program will at runtime decide which value it gets. Another possibility for state transitions are with the **INIT** and the **TRANS** keyword. Under each label there can be only one expression. If two or more variables are to be assigned, multiple 'INIT' and 'TRANS' labels are needed. Some examples [vHen03], which describes the behavior of a traffic light:

```
ASSIGN
  init(signal) := red;
  next(signal) :=
    case
      signal=red : green;
      signal=green : {green, yellow}
      1 : red;
    esac;

..which have the same transitions as in:

INIT
  signal = red;
TRANS
   (signal=red & next(signal)=green)
  |(signal=green & next(signal)={green, yellow)
  |(signal=yellow & next(signal)=red)
```

Using ASSIGN instead of INIT and TRANS is recommended, because of logical absurdities that can occur in these declarations [McKi00]. Another way of declaring and assigning variables values is with a DEFINE label. The declarations following this label are a list of symbols and their assigned expressions. An example from [McKi00]:

```
MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := 0;
  next(value) := value + carry_in mod 2;
```

```
DEFINE
  carry_out := value & carry_in;

..can also be implemented as

MODULE counter_cell(carry_in)
VAR
  value : boolean;
  carry_out : boolean;
ASSIGN
  init(value) := 0;
  next(value) := value + carry_in mod 2;
  carry_out := value & carry_in;
```

One of the reasons for using **DEFINE** instead of **VAR** and **ASSIGN**, is that defined symbols are dynamically typed, and "normal" variables are not. Forward references [4] are allowed in define declarations. Example:

```
MODULE main
VAR
sys : System
DEFINE
sys.active := 1;

MODULE System
VAR idle : boolean;
ASSIGN
init(idle) := 1;
next(idle) := active;
```

In the example above, the main module defines the System module's active variable. This is not possible with **VAR** and **ASSIGN**. Modules are explained further in section 3.4.4.

Unlike Promela, the SMV programming language implements the temporal logic specifications in the program itself, namely under the label 'SPEC' (see also 3.8)

```
SPEC
  AG(request -> AF state = busy)
```

The meaning of the CTL operators like AG or AF are described in 2.2.1.

3.4.4 SMV Modules

A SMV program is divided into modules. Like in the C programming language, the module named 'main' has a special meaning. Example [McKi00]:

[4]Forward reference means using an identifier before it is defined or (in SMV's case) declared [Bott03]

```
<program> ::= {<module>}

<module> ::= MODULE <atom> [ '(' <atom> {',' <atom>} ')' ]
              <decl_list>

<decl_list> ::= {<decl>}

<decl>   ::= <decl_var>
           |  <decl_assign>
           |  <decl_trans>
           |  <decl_init>
           |  <decl_invar>
           |  <decl_spec>
           |  <decl_fairness>
           |  <decl_print>
           |  <decl_define>
```

Figure 3.9: SMV modules in EBNF

```
MODULE main
VAR
  request : boolean;
  state : {ready, busy};
ASSIGN
  init(state) := ready;
  next(status)
```

Modules can have parameters, which are defined after the module-name inside a pair of parentheses. The values used in the parameters can be of any type. A module is declared below the **VAR** keyword. Example:

```
MODULE main
VAR
  status : {ready, busy}
  node : server(status)
...
MODULE server(state)
ASSIGN
  state := ready;
```

As seen in the example above, two modules are defined, 'main' and 'server'. Main instantiates a 'server' module via the variable 'node'. The module parameters are passed as call-by-reference. In the example above, the variable 'status' in the main module gets the value 'ready'.

The module variables can be accessed through a module instantiation using a '.' symbol, similar to other languages (Java, C++). Extending the previous example, the module 'main' can access a variable in module 'server':

```
MODULE main
```

```
VAR
  status : {ready, busy}
  node : server(status)
  serverstate : {ready, busy}
ASSIGN
  serverstate := node.state;
...
MODULE server(state)
ASSIGN
  state := ready;
```

All instantiated module are executed in parallel. However, if a module is instantiated with the keyword **process**, it becomes a process module. In each iteration step, the program will non-deterministically choose a process and execute all of the assignment statements within that process in parallel.

3.5 Comparing the languages

In table 3.1 the different imperative concepts discussed in the sections 3.2 - 3.3 are listed for each language. Concepts like *process*, *module* and *channels* are also included in the list. *Non-deterministic* conditional statements means that the conditional statements are executed like a guarded command.

Language concept	**Promela**	**SPL**	**SMV**
Call-by-value	only for **chan**	N/A	yes
Process	yes	yes	yes
Modules	no	no	yes
Primitive types	boolean, integer	boolean, integer, float	boolean
Array types	array	array	array
User defined types	yes	yes	yes
Channels	yes	yes	no
Looping constructs	yes	yes	no
Conditional statements	non deterministic	if-else, non-deterministic	case

Table 3.1: Programming language concepts

3.6 Case study

Before designing an object oriented model using the languages in the sections 3.2 - 3.3, a reference example is needed. This model will be a simple component structure, where the different components interact with each other. For this purpose an UML class diagram and sequence diagram will be used.

For this reference example we use a Customer-Bank scenario. The Customer wants to utilize all services the Bank offers, but within certain limits. This model will be implemented in each of the programming languages listed in the sections 3.2, 3.4 and 3.3.

3.6.1 Requirements specification

In the Customer-Bank scenario a couple of functional requirements are specified. These are very basic and are not necessarily realistic in the real world. Many simplifications are used to get a model suited for this project. For this purpose some functional requirements and restrictions are invented. These are not meant to work in a more realistic environment using a real Bank and Customer specifications.

The requirements

1. A Customer can open an Account at a Bank
2. A Customer can get one Advisor at a Branch
3. A Customer can deposit and withdraw amounts to/from his/her Account
4. A Customer can transfer amounts to other Accounts

The restrictions

1. A Customer can only open an Account if s/he has an Advisor
2. A Customer can not have negative credits on his/her Account
3. A Customer's Advisor has to work at Branch located at the same city as the Customer

3.6.2 The reference example

As explained in section 3.6.1 the Customer-Bank scenario is imaginary and most likely does not correspond to a similar real world scenario.

The reference example consists of six classes:

Class name	**Purpose**
Bank	The financial institute. Has Customers, Branches and Accounts. Allows credit transfers between Accounts
Customer	Related to one Bank. May also have Accounts and an Advisor
Account	A Bank Account. The balance is kept here and may change due to money transfers
Branch	A physical part of the Bank. Is located in a city and contains Advisors
Advisor	The work force of a Branch. Is necessary for Customers to open Accounts
Person	An abstract class. Inherited by Advisor and Customer. Contains personal information like name and city

Table 3.2: The classes in the UML model

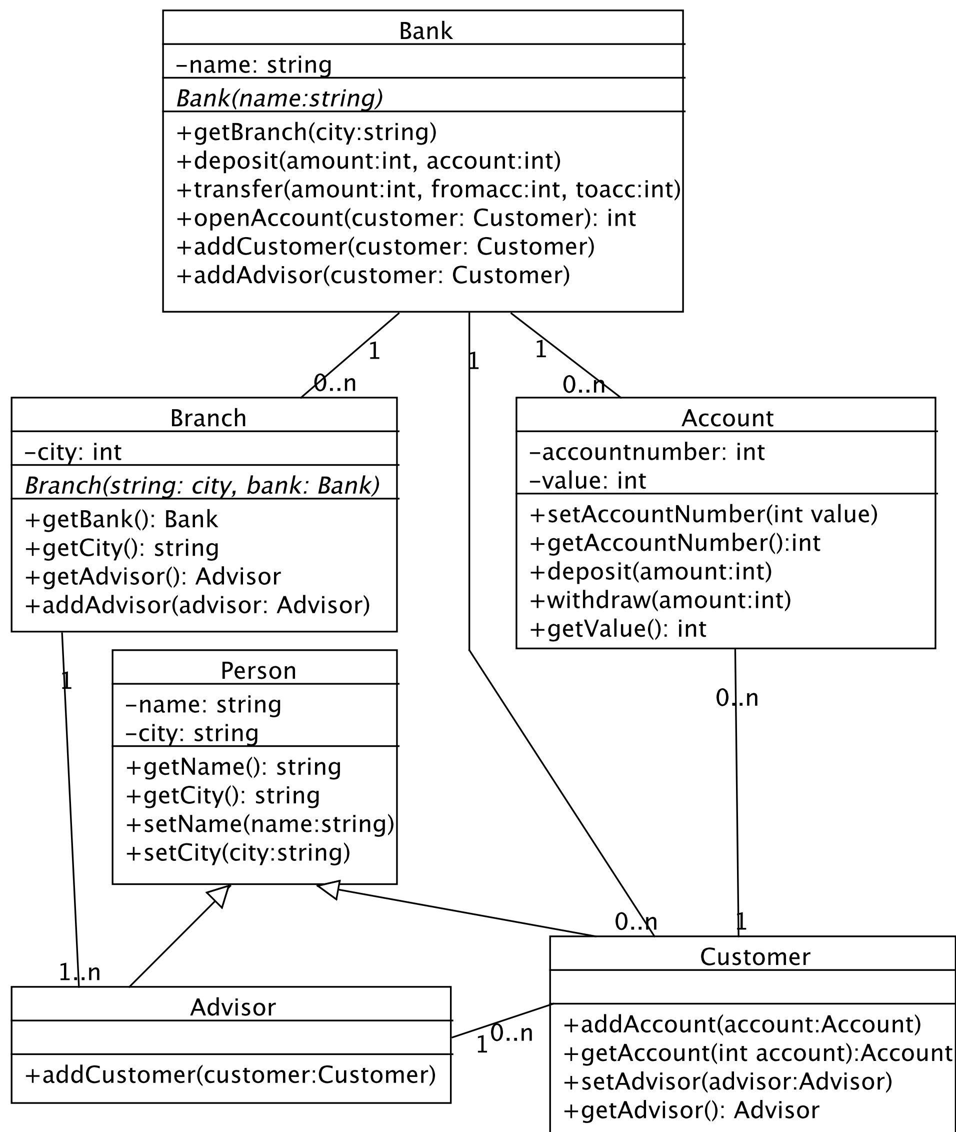


Figure 3.10: The UML diagram of the reference example

A UML class diagram of the classes listed in table 3.2 is depicted in figure 3.10. The core of this model are the Bank, Customer and Account classes and how they interact with each other. Initially a Bank has a set of Customers, a set of Branches with their respective Advisors and zero Accounts. An Account is first created when a Customer wants to open one. The Person class generalizes the Advisor class and the Customer class, for the purpose of introducing inheritance into the model.

In the following subsections, four sequence diagrams will be introduced. Each diagram depicts different interactions between the classes in this model.

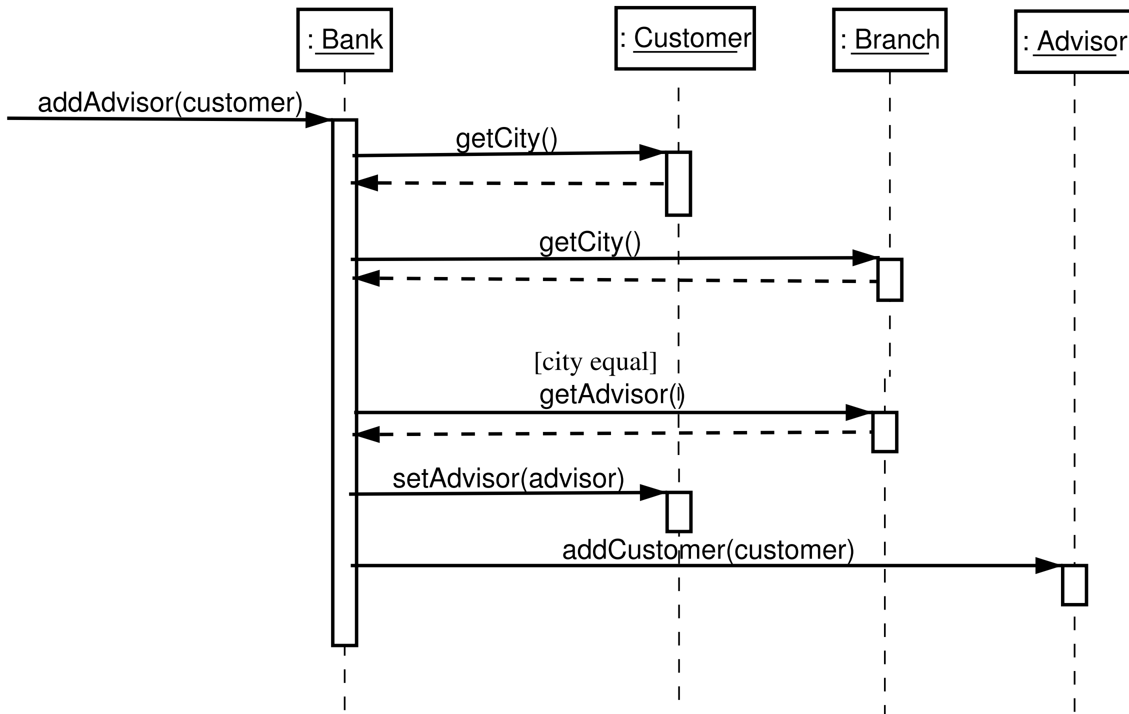


Figure 3.11: The sequence diagram for binding an Advisor to a Customer

3.6.2.1 Assigning an Advisor to the Customer

The first step for a Customer of a Bank to open an Account is getting an Advisor. Since the Customer cannot open the Account directly himself, s/he needs some authorized person. Because of the restrictions in this model (see section 3.6.1) a Customer may only open Accounts if the Bank has a Branch in the same city as where s/he lives.

To assign an Advisor to a Customer the Bank's *addAdvisor(Customer: Customer)* needs to be called, with the Customer as parameter value. The Bank compares the Customer's home city with the location of its Branches. If this comparison has a match, an Advisor is assigned to the Customer. This is depicted in the sequence diagram in figure 3.11.

3.6.2.2 Opening an Account for a Customer

After the step in section 3.6.2.1, a new Account can now be opened. After opening an Account, the Customer may finally store some money for safekeeping.

To open an Account, the Bank's *openAccount(Customer: Customer)* has to be called, using the Customer as a parameter value. If the Customer has an Advisor the Bank instantiates a new Account object, giving it an unique Account number and adding it to the Customer. If successful a valid Account number is returned for future reference. This is depicted in the sequence diagram in figure 3.12.

3.6.2.3 Depositing and withdrawing amounts from an Account

Finally, the Customer has an Account. Hopefully s/he remembered to write down the Account number returned when s/he opened his/her Account. s/he may now deposit amounts to his/her Account for safekeeping and withdrawing amounts when needed. Though s/he may not withdraw more than his/her Account balance (see restrictions in section 3.6.1).

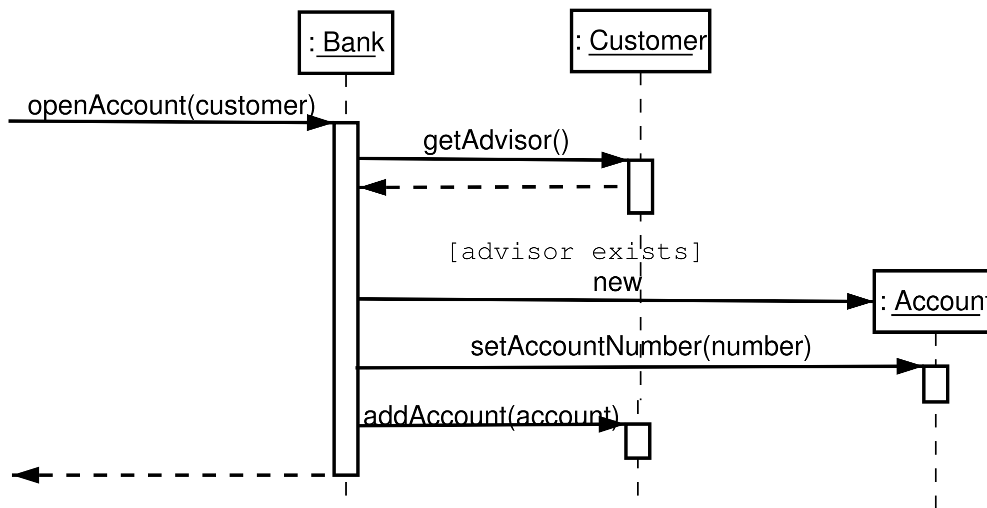


Figure 3.12: The sequence diagram for opening an Account

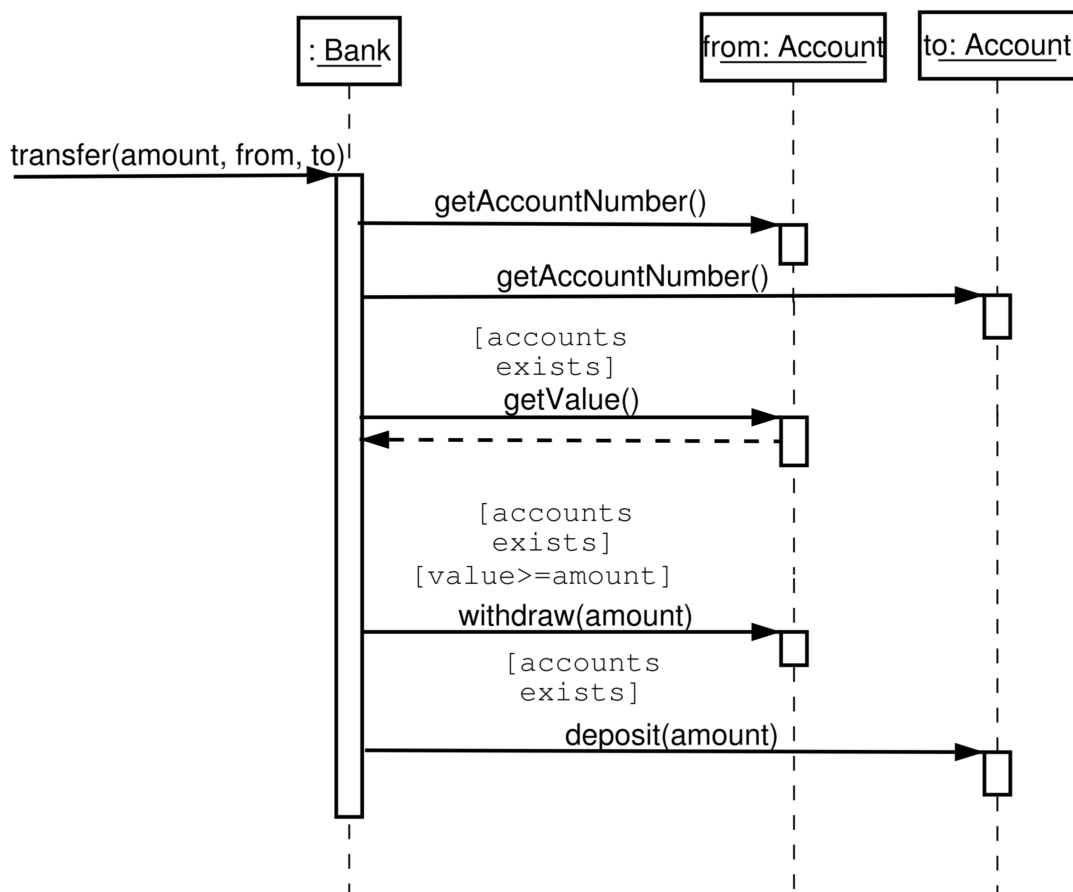


Figure 3.13: The sequence diagram for transferring amounts between Accounts

To make a deposit to one of the Customer's Accounts, the *getAccount(Account:int)* must be called to get a reference to the Account. Then, any deposit or withdrawal necessary can then be carried out. This is depicted in the sequence diagram in figure 3.14.

3.6.2.4 Transferring money between Accounts

Not only may a Customer deposit or withdraw amounts from his/her own Account, s/he may even transfer money between Accounts, if s/he has the Account numbers. A money

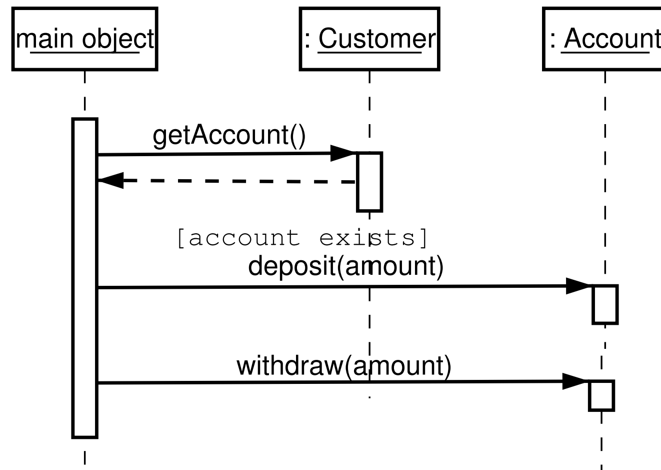


Figure 3.14: The sequence diagram for depositing and withdrawing amounts from/to an Account

transfer is only successful if the Account the is money transfered from has a sufficient balance.

To transfer amounts between two Accounts, the Bank's *transfer(amount:int, fromacc:int, toacc:int)* needs to be called. The execution first gets the reference to the source and destination Accounts. If they both exist, a balance check of the source Account is necessary to check if a withdrawal from that Account is possible. If so, then a transfer is possible. The source Account withdraws the amount from its balance and the destination Account adds the same amount to its balance. This is depicted in the sequence diagram in figure 3.13.

3.6.3 Requirements to the model implementation

The most important part of this project is a correct object oriented implementation of the model. The definition of verification and validation taken from [NPAR03] is as follows:

Validation

> *The process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model. (AIAA G-077-1998)*

Verification

> *The process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model. (AIAA G-077-1998)*

The process *validation* has a very low priority (as already stated in section 3.6.1 and section 3.6.2). A more important aspect of this project is the *verification*. Since the implementation should be as close to object oriented thinking [Seth96] as possible, some functionality are more essential than others. In table 3.3, some of these features are listed.

The most important of these concepts in this project is without any doubt the *class structure* and *class association*. Should it be impossible to find a satisfying implementation, the rest of the concepts listed in table 3.3 are more or less meaningless. Without a proper *class structure*, we will not have a structure that resembles the UML model classes (figure 3.10). Class *association* is necessary for the relatiship and interaction between the classes; without it, the objects cannot communicate. The next in the list are class *methods*. These implements the object behavior and are necessary for *information hiding*. A sufficient set of *variable types* are also necessary for being able to correctly implement both *class structure* and *methods.Inheritance* has the lowest priority of the already mentioned concepts, but is one of the most powerful tools in object oriented programming.

Concept	Description
class structure	A clearly defined class structure of and the possibility to instantiate objects of the defined class is essential in object oriented programming.
association	Class association should be able to implement, so that one object instance can refer to another object instance.
method	The class should have methods which are only accessible through an instantiated object of that class.
information hiding	The variables of an object should be hidden, only accessible through its methods.
association	Class association should be able to implement, so that one object instance can refer to another object instance.
variable types	Variable types that closely match the ones given in the model are important to express the attributes of an object.
inheritance	One of features specific to object oriented programming is the ability to implement subclasses.

Table 3.3: Concepts that are essential for the implementation of the reference example

4. Design

In this chapter an object oriented design for each of the model checking languages SPL, Promela and SMV will be discussed. In section 3.6 a reference example was defined. Each of the model checking languages will in this chapter try to approach the model defined in this reference example, using the concepts available for each language (see sections 3.2 - 3.3). If several approaches for the same object oriented concepts in table 3.3 are possible, then all of them are listed.

4.1 Promela design

The Promela language has several limitations that makes it difficult, but not impossible, to design a model in the light of the requirements stated in section 3.6.3

- Lack of Class/Module structure
- No character/string types available
- The processes (methods/procedures) run as threads (they interleave each other)
- Variable assignments[1] is only allowed for the primitive types and channels (user-defined types create error when reassigned)
- All processes are global
- The processes use call-by-value (except when **chan** is parameter)

The sections 4.1.1 and 4.1.2 show two different models of a class structure in Promela, and section 4.1.3 summarizes the different approaches made in these two sections.

[1]Must not be confused with variable declaration, which is allowed for all types.

4.1.1 Defining classes using a database table structure

The user defined types replaces parts of the class structure in the UML model, but due to of the global processes, the class methods are absent. Another problem is that variable assignments are only allowed for primitive types, e.g. an Account type (user defined) can not be assigned to refer to an already existing Account type. To cope with this a program structure similar to database tables is used. Each user defined type needs a unique primary key (e.g. name for Customer) and a foreign key for object association. The result can be seen in 4.1. The class generalization is designed using redundancy, which means the variables and methods presented in the super class are instead put into all of the subclasses. These tables are stored as global values. Since all the values are public, get and set methods are unnecessary. The rest of the methods are converted to global procedures. The new global procedures got these signatures

assignAdvisor(mtype incustomer; chan intoken)
deposit(int amount; int account; chan intoken)
withdraw(int amount; int account; chan intoken)
transfer(int amount; int fromaccount; int toaccount; chan intoken)
openAccount(mtype incustomer; chan intoken)

One problem with Promela procedures is that they run concurrently and may overlap each other. To force them to run sequential, a token is introduced. If a process (our procedure) is started, a token is given as a parameter. The statements following a process call cannot execute until the process has released its token, e.g. at process ending.

The biggest drawback with this method is the lack of class methods and information-hiding. All the variables, in the sense of UML terms[FoSc00], are *public*, and all processes are global, which means that the method calls in the sequence diagrams in figures 3.11-3.14 are all from the same "object". In addition the **mtype**-type is not optimal since it can only use values that are already predefined. A solution for the latter problem is to use an array of integers and mapping each letter in a given *string* to a value in an ASCII-table. This is on the syntax-level closer to the original, but much more difficult to read on a higher abstraction-level. A solution to the first problem is more complex as we will see in section 4.1.2.

4.1.2 Processes as class structure

Instead of using *defined types* to construct a class, a *process* can be used instead. First, it must be possible to instantiate an object of a class. This can be done by **run** *process name*. Second, the *object* must be persistent, at least as long as it is referred to. This can be done with a *loop* inside the process, and finally we need an *object reference*. Before getting into details, let us just say that this is done by putting a **chan** type in the process parameter (our class) and use this as our reference. The class private variables are simply the process' local variables

```
/* our main method */
init {
  chan a = [1] of {chan};
  run Account(a);
```

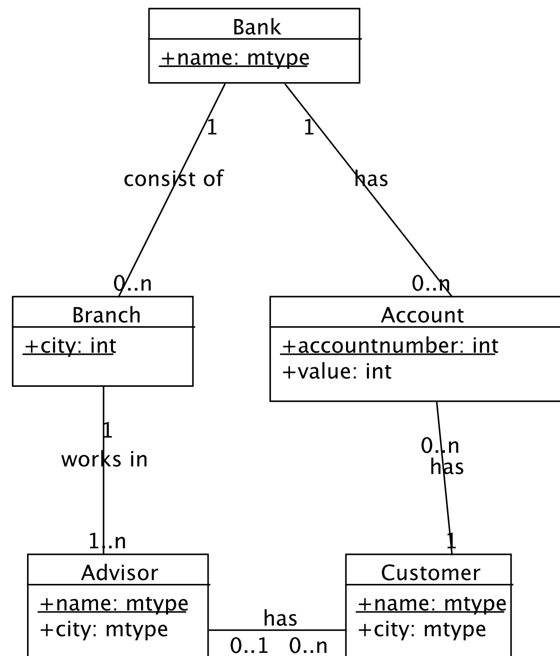


Figure 4.1: The Promela UML diagram

```
}

proctype Account(chan a) {
  int value;
  do
  :: skip /* dummy statement */
  od;
}
```

So far, so good. Now we need to access the private variables through get and set methods. The methods are also constructed with processes. The naming convention used in this example is <*class_name*>'_ '<*method_name*>

```
proctype Account_setValue(chan a) {
```

```
    skip /* dummy statement */
  }
```

To access this method, simply call it by using the *object reference* as parameter. The value which we want to replace with Account's **value** must be appended to the object reference, which also functions as the communication channel to the object

```
init {
  ...
  /* Need to encapsule the value 10 inside a channel */
  chan param = [1] of {int};
  param!10;
  /* Adding param to a's tail */
  a!z;
  /* Executing Account's setValue method */
  run Account_setValue(a);
  ...
}
```

Our desired effect is that **value** now has the value 10. So far, we have now three processes that share one common **chan**, our main method **init**, our class **Account**, and our method **Account_setValue**. To avoid that the method **Account_setValue** is called with the wrong object reference (e.g. any other **chan**), our object reference has to contain a class identifier. If this identifier is not presented when calling **Account_setValue**, the method does nothing. This identifier is set as we "instantiate" our object from our class. If the identifier is correct, a method identifier is appended to the object reference. The object itself will then do the necessary operations. Finally a token is introduced, so interleaving statement execution will not be a problem.

```
init {
  /* The token */
  chan token = [1] of {int};
  /* Initializing an object of the Account class */
  chan a = [3] of {chan};
  token!0;
  run Account(a, token);
  (len(token) == 0);
  /* Need to encapsule the value 10 inside a channel */
  chan param = [1] of {int};
  param!10;
  /* Adding param to a's tail */
  a!param;
  /* Executing Account's setValue method */
  token!0;
  run Account_setValue(a);
  (len(token) == 0);
  /* Terminating the object */
  chan b =[1] of {chan}
```

```
  a?b;

proctype Account(chan a; chan token) {
  /* Class variables */
  int value;
  /* Method and class identifiers */
  int setValue = 3221;
  int classId = 45521;
  /* inserting classid into object reference */
  chan z = [1] of {int}
  z!classId;
  a!z;

  /* Creating channels for receiving  method calls
and their parameters */
  chan methodcall = [1] of {int};
  int methodvalue;
  chan parameterSetValue = [1] of {int}

  token?0; /* Object construction complete */

  /* Waiting for method calls */
  do
  :: (len(a) == 3) -> a?methodcall; methodcall?methodvalue;
      /* Method call */
      if
      :: methodcall?[setValue] -> a?parameterSetValue;
         parameterSetValue?value; token?0
      fi;
  :: (len(a) == 0) -> break /* Object termination */
  od;
}

proctype Account_setValue(chan a) {
  int classId = 45521;
  int inClassId;
  chan methodId = [1] of {int}
  methodId!3221;
  /* Getting classid out of object reference */
  chan inClass = [1] of {int};
  a?inClass;
  inClass?inClassId;

  chan parameter = [1] of {int}

  /* If correct classId, then the object reference consist of:
      <head> methodId | <tail> parameter | <tail> classId */
  if
  :: (inClassId == classId) -> a!methodId; a?parameter;
```

```
a!parameter; a!inClass
  :: (inClassId != classId) -> skip
  fi;
}
```

In this example terminates the object Account at the end of the main method (init). In a more realistic example with multiple reference to the same object, a more sophisticated garbage collection would be necessary. This kind of garbage collection would not be impossible, but requires more time and effort than meant for such a project as this one, and is therefore not taken into the design.

4.1.3 Summary

The different solutions to the requirements in section 3.6.3 are listed in table 4.1

Using typedef as class construct	
Advantages	*Disadvantages*
Well defined class structure	No class methods (global procedures)
Class association with primary/ foreign keys	No information hiding
Easy to implement	No inheritance
Using process as class construct	
Advantages	*Disadvantages*
Well defined class structure	No inheritance
Class association through channels	Very primitive garbage collection
Allows class methods	Complicated overheads
Allows information hiding	
Using mtype for string/char	
Advantages	*Disadvantages*
Easy to implement	Restricted to the defined variables only
Using integer array for string/char	
Advantages	*Disadvantages*
A good mapping from characters to values	Complex to implement

Table 4.1: Advantages and disadvantages to the different design issues for the Promela language

4.2 SPL

The severe limitations of *SPL* make it virtually impossible to design a program that satisfies the reference example. The problems are:

- Lack of Class/Module structure
- No character/string types available

- All variables are global, even those declared inside procedures
- The processes run only where they are defined and cannot be called

SPL actually fails on every point in the requirements specified in section 3.6.3. The biggest problem is the lack of any encapsulating constructs, be it either a class or a procedure. The procedures used in SPL are *artificial*. The variables defined inside are global, which means that the procedure do not introduce anything new, except for a way to reduce code typing. The processes could be of interest, but they are not possible to be started by a call. The user defined types cannot be used to construct the classes. The reason for this is that the syntax does not allow to extract a single primitive value out of the defined type, like in Promela. Example:

Promela code

```
/* defining a type */
typedef Account {
int accountNumber;
int value:
}

/* Declaring an int variable */
int temp;

init {
/* Declaring a variable for the type
and assigning its values */
Account acc;
acc.accountNumber = 1;
acc.value = 100;

/* Accessing only accountNumber */
temp = acc.accountNumber;

}
```

In the Promela code, the types in the user defined types can be accessed separately

SPL code

```
(* defining a type *)
type Account = int*int
(* Declaring a variable for the type *)
local acc : Account
(* Declaring an int variable *)
local temp : int
```

```
(* Assigning value to the user defined type *)
l0: acc := (1, 100);
(* Accessing only accountNumber *)
```

As seen in this SPL example the code where 'accountNumber' is to be accessed is not there. The syntax for such an operation is absent from the SPL specification [BBCF+98], and cannot be constructed.

Since the other languages offers more than SPL can offer, more time and effort had to be invested used on these instead.

4.3 SMV

The SMV language is quite different from both Promela and SPL. Even though SPL has its shortcomings, it still offers the some of the functionality that we are used to from imperative languages, like loops and sequential execution of statements. SMV may execute all its statements in parallel, if the conditional statements allow it. Some of the conflicts are listed:

- No character/string type available
- No integer/float type available
- No procedures/methods available
- No looping constructs available
- Every variable assignment can only appear once in a code.

The last point in this list may be a bit confusing. The best way to illustrate this is by an example:

```
MODULE Account
VAR
/* Value and accountnumber may contain the numbers 0->9 */
value : {0,1,2,3,4,5,6,7,8,9};
accountNumber : {0,1,2,3,4,5,6,7,8,9};

ASSIGN
/* Initial value of value is 0 */
init(value) := 0;
next(value) := 2;

MODULE Customer
VAR
/* Using predefined constants as string values */
name : {Mads};
city : {Trondheim, Karlsruhe};
/* A reference to an account */
```

```
account : Account;

ASSIGN
/* Assigning 5 to Account's value variable */
next(account.value) := 5;
```

This example will produce a compilation error, because both Account and Customer define the next value of Account's value. If either *next(value)* in Account or *next(account.value)* in Customer is removed, the program compiles without the error message *line 22: redefining account.value*.

4.3.1 The Class structure

The class structure and its associations can be constructed using *SMV modules*. Each class is then represented as a module and a class association is constructed using variables as in traditional object oriented languages. The variable types are a quite different matter. The only primitive type available in SMV is *boolean*. The only way to extend the set of types is to this is with user defined types:

```
VAR
/* An integer with max 3 digits */
integerVal : array 0 .. 2 of {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
/* A character array with max 11 characters */
stringVal : array 0 .. 10 of {a,b,c,d,e,f,g,h,i,j,k,l,
m,n,o,p,q,r,s,t,u,v,w,x,y,z};
/* A user defined type */
name : {Mads, Elke, Reidar};
```

As one can see, this not is a satisfying substitute. One reason is the complexity involved with such types, since looping constructs do not exist in SMV, making Array traversal impossible. Using an array instead of multiple variables does not give any advantage at all. Unfortunately this is the only way to construct other types than *boolean*. Of course, a module can be defined, but then one will have the same problem defining the values inside the module. Example:

```
MODULE string(l0,l1,l2,l3,l4,l5,l6,l7,l8,l9,l10)
VAR
stringVal : array 0 .. 10 of {a,b,c,d,e,f,g,h,i,j,k,l,
m,n,o,p,q,r,s,t,u,v,w,x,y,z};
ASSIGN
stringVal[0] := l0;
stringVal[1] := l1;
stringVal[2] := l2;
stringVal[3] := l3;
stringVal[4] := l4;
stringVal[5] := l5;
stringVal[6] := l6;
stringVal[7] := l7;
```

```
stringVal[8] := l8;
stringVal[9] := l9;
stringVal[10] := l10;


MODULE Customer
VAR name : string(m,y,n,a,m,e,i,s,p,e,r}
```

It is also possible define an entire string value without arrays, but then the value is restricted for only the values it is defined for.

4.3.2 The class methods

The abilities of the class, the procedures or methods that can change the state of a class, making it dynamic, is totally absent. Procedures does not exist in the SMV language. Trying to find a substitute is a tough challenge. The problem lies in SMV's nature, its structure is very static. Variables can be assigned to other values, even with *forward reference* but only under one ASSIGN statement. If one module type is declared to change the value of a variable, no other modules may do this. This can prove to be useful if constructing the methods as modules, since only the modules will be able to change the values of its class variables. The problem occurs when the same method is called from twice. Example:

```
MODULE main
VAR
account : Account;
bank : Bank(account);
/* Running the setValue method of Account */
setAccount : Account_setValue(account,7);

MODULE Account
VAR
value : {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
ASSIGN
init(value) = 0;

/* Defining the setValue method of Account */
MODULE Account_setValue(account,variable)
ASSIGN
next(account.value) := variable;

/* A bank with only one Account */
MODULE Bank(singleAccount)
/* Running the setValue method of Account */
VAR
accountMethod : Account_setValue(singleAccount,5);
```

This gave an error message *line 6: redefining account.value*.

A method construct proves to be impossible and the attempt has been abandoned. An effort to instead construct a centralized method is instead put to trial in section 4.3.3.

4.3.3 Sequential execution of statements

As explained earlier, all statements in a SMV program may execute at the same time, if the conditional statements are set correct or totally absent. Some sort of control of which statements that are allowed to execute is needed. One effort is to construct a statement holder module, which has all the necessary variables for the execution of one statement. Declaring several variables of this module type, it would be possible to iterate through a series of statements, in the sense of an imperative language[Seth96]. The name of the module, *Customeraction*, refer to the point of view in which the statements are executed. Example:

```
MODULE main
VAR
/* The set of statements that are to be executed
sequentially */
customeractions : array 0 .. 10 of Customeractions(5);
/* The current statement */
currentAction : Customeraction(1);
/* A nullpointer */
nullcustomeraction : Customeraction(0);

MODULE Customeraction(null)
VAR
type : {idle, openaccount, transfer, deposit, withdraw,
getadvisor};
val2 : {0, 1, 2, 3, 4, 5, 6, 7, 8 ,9};
val1 : {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
fromaccount : {1, 2, 3, 4};
toaccount : {1, 2, 3, 4};
DEFINE
nullpointer := null;
```

Every variable that potentially changes its value, has to check the *current statement* for each iteration. Example:

```
MODULE main

/* Simple variable assignments */
next(bank.val2) := case currentAction.type in {deposit,
withdraw, transfer} : currentaction.val2;
1: bank.val2;
esac;
next(bank.val1) := case currentAction.type in {deposit,
withdraw, transfer} : currentaction.val1;
1: bank.val1;
esac;

MODULE Bank
```

```
VAR
/* Class association */
accounts : array 0 .. 3 of Account(1);
branches : array 0 .. 1 of Branch;
customers : array 0 .. 3 of Customer(0);
/* Variables necessary for Bank operations */
newcustomer : Customer(0);
currentcustomer : Customer(0);
val2 : {0, 1, 2, 3, 4, 5, 6, 7, 8 ,9};
val1 : {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Finally, a *statement* switcher in each iteration is needed (or else the same *statement* will be executed in each iteration. Example:

```
next(currentaction) := case currentaction =
nullcustomeraction : customeractions[0];
currentaction = customeractions[0] : customeractions[1];
currentaction = customeractions[1] : customeractions[2];
1 : customeractions[3];
esac;
```

4.3.4 Summary

The SMV design is summarized in table 4.2.

Using module as class construct	
Advantages	*Disadvantages*
Well defined class structure	No information hiding
Class association through variable reference	No inheritance
Easy to implement	
Using modules as methods	
Advantages	*Disadvantages*
Allows information hiding	Can only be accessed from one module, but not multiple instances of one module
Using array of predefined types for integer/string	
Advantages	*Disadvantages*
Allows greater freedom	Very complex to implement
Using predefined types for integer/string	
Advantages	*Disadvantages*
Easy to implement	Restricts the number of values allowed.
Using main module as only procedure/method	
Advantages	*Disadvantages*
Perhaps the only implementation possible for a method	No information hiding
	No class methods

Table 4.2: Advantages and disadvantages to the different design issues for the SMV language

5. Implementation and tests

In this chapter, the implementation of the reference model in section 3.6 will be discussed and some test cases will be defined.

5.1 Implementation

The design in chapter 4 offers several possibilities how each programming language may implement the reference model. In the following sections the Promela and the SMV implementation will be discussed. SPL is not considered in this section for the given reasons of section 4.2.

5.1.1 Promela

The Promela design in chapter 4 offered two ways to implement the class structure. Both of them are used in the implementation, making it two different codes with the same language. Both implementations are discussed in the following subsections.

First implementation

Using the list summarized in section 4.1.3, the following concepts have been used in the implementation:

- **Using typedef as class construct**
- **Using mtype for string/char**

The implementation went without any significant problems.

The second implementation

The following concepts taken from the list in section 4.1.3 are implemented:

- **Using process as class construct**
- **Using mtype for string/char**

Because of lack of time available and the complexity involved with such a model, only a small part of the model using processes as class-objects is implemented, namely Account and Customer. A list of the methods implemented and their respective signatures in Promela follows:

- **Account**
  - setAccountNumber(int value) -> Account_setAccountNumber(chan a)
  - getAccountNumber():int -> Account_getAccountNumber(chan a)
  - deposit(amount:int) -> Account_deposit(chan a)
  - withdraw(amount:int) -> Account_withdraw(chan a)
  - getValue():int -> Account_getValue(chan a)
- **Customer**
  - addAccount(account:Account) -> Customer_addAccount(chan a)
  - getAccount(int account):Account -> Customer_getAccount(chan a)
  - setName(string name) -> Customer_setName(chan a)
  - getName():string -> Customer_getName(chan a)

The methods used in the list above corresponds well to both the UML class diagram in 3.10 and the method execution follow the sequence diagram in figure 3.14, if one overlooks the overhead used to make the implementation of the methods possible.

5.1.2 SPL

Because of the problems explained in section 4.2, no implementation in the SPL language exists.

5.1.3 SMV

Using the list summarized in section 4.3.4, the following concepts are used:

- **Using module as class construct**
- **Using array of predefined types for integer**
- **Using predefined types for string**
- **Using main module as only procedure/method**

The first three concepts are implemented without any significant problems. The fourth concept provides big problems. As this implementation is in progress and tested, the *SMV* compiler becomes very slow. It stops in the middle of its execution and does not make any progress. The only thing that seems to help is changing the iteration of the *Customeraction* array (see section 4.3.3). The source which relults in this problem:

```
next(currentaction) := case currentaction =
nullcustomeraction : customeractions[0];
currentaction = customeractions[0] : customeractions[1];
currentaction = customeractions[1] : customeractions[2];
1 : customeractions[3];
esac;
```

This solves the problem:

```
next(currentaction) := case currentaction =
nullcustomeraction : customeractions[0];
currentaction = customeractions[0] : customeractions[1];
1 : customeractions[3];
esac;
```

As one can see, only one line is taken away. The source of this error is unclear, except for perhaps a bug in the SMV system. Even before getting this far, the SMV language proved to be too cumbersome and static for the requirements listed in 3.6.3. Due to these problems, SMV does not allow the full implementation of the reference example.

5.2 Testing

This section will explain how the implementation is tested. Unfortunately the SMV implementation is not possible to finish, so its implementation is not tested. The SPL language is not implemented at all. Only Promela has implementations that can be tested.

5.2.1 Promela tests

Promela had two implementations of the model. The first implementation uses **typedef** as class structure and the second implementation uses processes to represent the class structure (see section 5.1.1). Only the first implementation uses the entire model, so a seperate test case is used for both.

Test case 1

In this section, the test case for the first implementation will be defined. The initial values for the model are listed in table 5.1. The basis for this test case are the sequence diagrams in figures 3.11 - 3.14.

The tests run as follows:

1. Assigning an Advisor to Customer[0]
2. Assigning an Advisor to Customer[1]
3. Assigning an Advisor to Customer[2]
4. Opening an Account for Customer[0]
5. Opening an Account for Customer[1]

6. Opening an Account for Customer[2]
7. Depositing 50 to Customer[0]'s Account
8. Withdrawing 30 from Customer[0]'s Account
9. Withdrawing 30 from Customer[0]'s Account
10. Transfering 20 from Customer[0]'s Account to Customer[1]'s Account

Following the list above, item 3. will not succeed, since no Branches are in his home city. This implies that item 6. also fails, since Customer[2] has no Advisor. Item 9. will also fail, because the balance does not allow negative values. At the end, Customer[0] has a zero balance and Customer[1] has 20 on her account.

Class : Bank			
name			
Deutsche Bank			
Class : Customer			
No	*name*	*city*	*bank*
1	Mads	Karlsruhe	Deutsche Bank
2	Elke	Heidelberg	Deutsche Bank
3	Matthias	Mannheim	Deutsche Bank
Class : Advisor			
No	*name*	*city*	*branch*
1	Ole	Karlsruhe	Karlsruhe
2	Tore	Heidelberg	Heidelberg
3	Miriam	Mannheim	Heidelberg
Class : Branch			
No	*city*	*bank*	
1	Karlsruhe	Deutsche Bank	
2	Heidelberg	Deutsche Bank	
Class : Account			
No	*accountNumber*	*bank*	
1	1000	Deutsche Bank	
2	1001	Deutsche Bank	
3	1002	Deutsche Bank	
4	1003	Deutsche Bank	

Table 5.1: The initial values in test case 1

Test case 2

This test case is much smaller since only the Customer class and the Advisor class are implemented. This test will focus on the methods in both classes, and initialization of objects. The basis for this test case is the sequence diagram in figure 3.14.

The test runs as follows:

1. Create two Accounts, Account[0] and Account[1]

2. Set their account numbers to 1 and 2, respectively
3. Check if the first account number is correctly set
4. Deposit 50 and withdraw 30 from Account[0]
5. Check if value is 20
6. Withdraw 30 from Account[1]
7. Check if value is 0
8. Create a Customer with the name Mads
9. Check if name is correctly set
10. Add both Accounts to the Customer
11. Get Account[1] from Customer and check its balance
12. Get an Account number that do not exist and check the return value

The items 3, 5, 7, 9, 11 and 12 are tested by using Promela's function *assert(expression)*. If an **assert** function fails, the program is terminated. If e.g. item 3 fails, the output could be on the following form:

spin: text of failed assertion: assert((returnValue==1))

A succesful test could produce the following output, without the error above:

531: proc 3 (Customer) terminates
531: proc 2 (Account) terminates
532: proc 1 (Account) terminates
532: proc 0 (:init:) terminates
22 processes created

This output is generated by running the program without the *assertions*.

5.2.2 Promela results

Both test cases runs as anticipated. As one can see in figure 5.2, the results of the first test case match the values set in section 5.2.1, except for the Bank name which is DB (short for Deutsche Bank). Test case 2 ran successful without any premature program termination, as can be seen in the program output in figure 5.1.

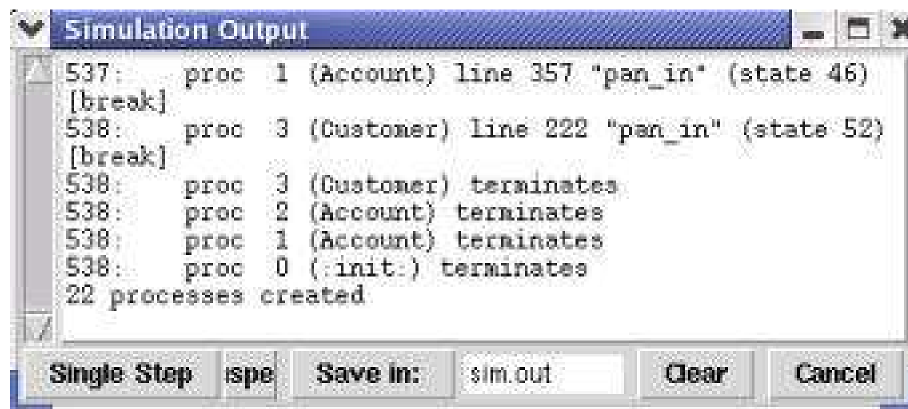


Figure 5.1: The last part of the program output after running the test case 2

5.3 Test conclusion

Of all the implementations, only those of the Promela language are tested. The SPL did not have any implementation since the design is found lacking. The SMV language did not have any tests, since the implementation is not completed, and therefore difficult to test.

Instead, Promela had two different implementations, both which are tested using the case study in section 3.6 as reference and their respective designs in chapter 4. Both tests are successful, showing that the design is correctly implemented.

The program efficiency is totally ignored except for SMV, where the program execution slows to a halt halfway through the implementation. One suspects a bug in the SMV tool, but this can not be verified.

Data Values

```
accounts[0].accountnumber = 1000
accounts[0].bank = DB
accounts[0].credit = 0
accounts[0].customer = 0
accounts[1].accountnumber = 1001
accounts[1].bank = DB
accounts[1].credit = 0
accounts[1].customer = 0
accounts[2].accountnumber = 1002
accounts[2].bank = DB
accounts[2].credit = 0
accounts[2].customer = 0
accounts[3].accountnumber = 1003
accounts[3].bank = DB
accounts[3].credit = 0
accounts[3].customer = 0
advisors[0].branch = Karlsruhe
advisors[0].city = Karlsruhe
advisors[0].name = Ole
advisors[1].branch = Heidelberg
advisors[1].city = Heidelberg
advisors[1].name = Tore
advisors[2].branch = Heidelberg
advisors[2].city = Mannheim
advisors[2].name = Miriam
bank.banknumber = 123
bank.name = DB
branches[0].bank = DB
branches[0].city = Karlsruhe
branches[1].bank = DB
branches[1].city = Heidelberg
customers[0].advisor = 0
customers[0].bank = DB
customers[0].branch = 0
customers[0].city = Karlsruhe
customers[0].name = Mads
customers[1].advisor = 0
customers[1].bank = DB
customers[1].branch = 0
customers[1].city = Heidelberg
customers[1].name = Elke
customers[2].advisor = 0
customers[2].bank = DB
customers[2].branch = 0
customers[2].city = Mannheim
customers[2].name = Matthias
```

Save in: var.out Clear Close

Data Values

```
accounts[0].accountnumber = 1000
accounts[0].bank = DB
accounts[0].credit = 0
accounts[0].customer = Mads
accounts[1].accountnumber = 1001
accounts[1].bank = DB
accounts[1].credit = 20
accounts[1].customer = Elke
accounts[2].accountnumber = 1002
accounts[2].bank = DB
accounts[2].credit = 0
accounts[2].customer = 0
accounts[3].accountnumber = 1003
accounts[3].bank = DB
accounts[3].credit = 0
accounts[3].customer = 0
advisors[0].branch = Karlsruhe
advisors[0].city = Karlsruhe
advisors[0].name = Ole
advisors[1].branch = Heidelberg
advisors[1].city = Heidelberg
advisors[1].name = Tore
advisors[2].branch = Heidelberg
advisors[2].city = Mannheim
advisors[2].name = Miriam
bank.banknumber = 123
bank.name = DB
branches[0].bank = DB
branches[0].city = Karlsruhe
branches[1].bank = DB
branches[1].city = Heidelberg
customers[0].advisor = Ole
customers[0].bank = DB
customers[0].branch = 0
customers[0].city = Karlsruhe
customers[0].name = Mads
customers[1].advisor = Tore
customers[1].bank = DB
customers[1].branch = 0
customers[1].city = Heidelberg
customers[1].name = Elke
customers[2].advisor = 0
customers[2].bank = DB
customers[2].branch = 0
customers[2].city = Mannheim
customers[2].name = Matthias
```

Save in: var.out Clear Close

Figure 5.2: Test Case 1: Initial values on the left-side and results on the right-side

6. Summary and conclusion

Model checking is a concept to formally verify finite-state machines / automatas. While it was primarily developed and optimized for automatized verification of hardware systems current research is dealing with the application of model checking to (imperative) software systems. Facing the gap between current model checker languages for modeling finite-state machines on the one hand and object oriented software concepts on the other hand the goal of this thesis is to explore selected leading-edge model checkers with the goal to support the bridging of this gap.

The examined model checkers are SPIN (modeling language: Promela), STEPS (modeling language: SPL) and SMV (modeling language: SMV modeling language). This thesis compares these modeling languages

- with respect to a set of imperative programming language criteria
- with respect to language-specific concepts (not comparable according to the chosen criteria)
- with each other and
- with respect to concepts specific to the object oriented paradigm.

A set of design proposals and test implementations for the mapping of the individual modeling languages to object oriented concepts result from this exploration. Besides several test implementations a simplified object oriented reference example has been built to verify the design proposals where possible. The implementations have shown that it is partially possible to realize object oriented concepts using the individual input languages of the model checking tools when applying the mappings designed in this thesis. However, some of the object oriented concepts have proved to be difficult to implement.

A brief conclusion:
The Promela language allows, with some design detours, the realization of all considered object oriented concepts (with an exception of inheritance), but the SMV and SPL failed in central requirements for a reasonable mapping.

Further work

The results of this thesis can be refined with respect to an even more thorough study of SMV and SPL. There may exist further design solutions circumventing the problems experienced in sections 4.2, 4.3 and 5.1.3. The comparison should be extended to further object oriented concepts like inheritance and polymorphism. Several concepts have been neglected in this thesis focusing on the most important concepts to get a broad overview about the major problems.

In this project only three model checking tools are used. Although these are the leading-edge model checkers there exist several other tools that could be of interest, and some of these could prove to be easier to use in an object oriented system than the examined ones. Moreover, a study of the efficiency (with respect to memory, run time, code size and manageability by the programmer as well as with respect to the verification costs) would be useful as complementation to this work.

Further future work is the development of a higher-level modeling language. Common concepts of the different model checking languages meet at a higher abstraction level. In this abstraction level, a unified model is the goal, which can directly be translated to the different input languages of selected model checking tools. A mapping of such a language to the lower-level current model checking languages would be useful. However, such a language mapping is not straight-forward as there are many incompatibilities already between the examined different model checker languages (e.g. *Modules* in SMV versus *Processes* in Promela. Bandera 2.4 realizes a similar concept. It takes Java source code as input and produces a model, which can be translated for its use in SPIN or SMV (or even further model checkers if a mapping is implemented). For this purpose an intermediate representation is defined. Similar principles may be also found in other domains, e.g. CORBA IDL [CORB04]. As an alternative to providing a language mapping, a dedicated model checker for processing this new higher-level language could be developed.

A. Promela implementation part 1

Here is the source code for the first Promela implementation (see section 5.1.1). The code for the tests in section 5.2.1 are included in the code.

```
typedef Account {
int accountnumber;
int credit;
/* key values */
mtype customer;
mtype bank;
}

typedef Customer {
mtype name;
mtype city;
/* key values */
mtype bank;
mtype advisor;
mtype branch;
}

typedef Bank {
mtype name;
int banknumber;
}

typedef Branch {
mtype city;
/* key values */
mtype bank;
}

typedef Advisor {
mtype name;
mtype city;
/* key values */
mtype branch;
}
```

```
int numberOfAccounts = 4;
int numberOfBranches = 2;
int numberOfAdvisors = 3;
int numberOfCustomers = 3;

Account accounts[4];
Branch branches[2];
Advisor advisors[3];
Customer customers[3];
Bank bank;



mtype = {Mads, Elke, Miriam, Ole, Per, Tore, Matthias, DB, Karlsruhe,
Heidelberg, Mannheim}


init {
chan token = [1] of {int};

token!0;
run setAllValues(token);
(len(token) == 0);


token!0;
run assignAdvisor(Mads, token);
(len(token) == 0);

token!0;
run assignAdvisor(Elke, token);
(len(token) == 0);

token!0;
run assignAdvisor(Matthias, token);
(len(token) == 0);



token!0;
run openAccount(Mads, token);
(len(token) == 0);

token!0;
run openAccount(Elke, token);
(len(token) == 0);

token!0;
run openAccount(Matthias, token);
(len(token) == 0);


/* Depositing money in Mads' account */
int i=0;
do
:: accounts[i].customer == Mads -> break;
:: accounts[i].customer != Mads ->
  if
  :: i ≥ 3 -> break
```

```
  :: i < 3 -> i=i+1
  fi;
od;

token!0;
run deposit(50, i, token);
(len(token) == 0);

token!0;
run withdraw(30, i, token);
(len(token) == 0);

token!0;
run withdraw(30, i, token);
(len(token) == 0);

/* Transfering money from in Mads' account to Elke's account */
int j=0;
do
:: accounts[j].customer == Elke -> break;
:: accounts[j].customer != Elke ->
  if
  :: j ≥ 3 -> break
  :: j < 3 -> j=j+1
  fi;
od;

token!0;
run transfer(20, i, j, token);
(len(token) == 0);




}


proctype deposit(int amount; int account; chan intoken) {
accounts[account].credit = accounts[account].credit + amount;
intoken?0
}

proctype withdraw(int amount; int account; chan intoken) {
if
:: (accounts[account].credit − amount) ≥ 0 ->
accounts[account].credit = accounts[account].credit − amount
:: (accounts[account].credit − amount) < 0
fi;
intoken?0
}

proctype transfer(int amount; int fromaccount; int toaccount;
chan intoken) {
if
:: (accounts[fromaccount].credit − amount) ≥ 0 ->

accounts[fromaccount].credit = accounts[fromaccount].credit − amount;
   accounts[toaccount].credit = accounts[toaccount].credit + amount
:: (accounts[fromaccount].credit − amount) < 0 -> skip
```

```
fi;
intoken?0
}

proctype assignAdvisor(mtype incustomer; chan intoken) {
int i=0;
do
:: customers[i].name == incustomer -> break;
:: customers[i].name != incustomer -> i=i+1;
od;

int j=0;
int k=0;

do
:: if
  :: (j < numberOfBranches) -> if
    :: (customers[i].city == branches[j].city) -> do
      :: (branches[j].city == advisors[k].city) ->
customers[i].advisor = advisors[k].name; break
      :: (branches[j].city != advisors[k].city) -> k = k+1
      od; break
    :: (customers[i].city != branches[j].city) -> j=j+1
    fi;
  :: j == numberOfBranches -> break;
  fi;
od;

intoken?0;
}

proctype openAccount(mtype incustomer; chan intoken) {
int i=0;
do
:: customers[i].name == incustomer -> break
:: customers[i].name != incustomer -> i=i+1
od;

int j=0;
if
:: (customers[i].advisor == 0)
:: (customers[i].advisor != 0) ->
  do
  :: accounts[j].customer == 0 -> accounts[j].customer = incustomer;
break
  :: accounts[j].customer != 0 -> j=j+1
  od;
fi;
intoken?0;
}


proctype setAllValues(chan intoken) {
bank.name = DB;
bank.banknumber = 123;

/* Initializing accountnumbers and bank */
int i = 0;
do
```

```
:: (i < 4) -> accounts[i].accountnumber = i+1000; accounts[i].bank = DB;
i = i+1
:: (i ≥ 4) -> break;
od;

/* Initializing customers */
customers[0].name = Mads;
customers[0].city = Karlsruhe;
customers[0].bank = DB;

customers[1].name = Elke;
customers[1].city = Heidelberg;
customers[1].bank = DB;

customers[2].name = Matthias;
customers[2].city = Mannheim;
customers[2].bank = DB;

/* Initializing Advisors */
advisors[0].name = Ole;
advisors[0].city = Karlsruhe;
advisors[0].branch = Karlsruhe;

advisors[1].name = Tore;
advisors[1].city = Heidelberg;
advisors[1].branch = Heidelberg;

advisors[2].name = Miriam;
advisors[2].city = Mannheim;
advisors[2].branch = Heidelberg;

/* Initializing Branches */
branches[0].city = Karlsruhe;
branches[0].bank = DB;

branches[1].city = Heidelberg;
branches[1].bank = DB;

intoken?0;
}
```

B. Promela implementation part 2

Here is the source code for the second Promela implementation (see section 5.1.1). The code for the tests in section 5.2.1 are included in the code.

```
mtype = {Mads,Elke}

/* our main method */
init {
/* Creating an account1 */
chan token1 = [1] of {int};
chan account1 = [3] of {chan};
token1!0;
run Account(account1, token1);
(len(token1) == 0);

/* Creating an account2 */
chan token2 = [1] of {int};
chan account2 = [3] of {chan};
token2!0;
run Account(account2, token2);
(len(token2) == 0);


/* Initializing an integer parameter */
chan param = [1] of {int};

/* Setting the account1 number */
param!1;
account1!param;
token1!0;
run Account_setAccountNumber(account1);
(len(token1) == 0);

/* Setting the account2 number */
param!2;
account2!param;
token2!0;
run Account_setAccountNumber(account2);
(len(token2) == 0);
```

```
/* Getting the account1 number */
account1!param;
token1!0;
run Account_getAccountNumber(account1);
(len(token1) == 0);

int returnValue;
account1?param;
param?returnValue;

/* Checking if account1's number is correct */
assert(returnValue == 1);

/* Adding param to a's tail */
account1!param;
/* Executing account1's deposit method */
param!50;
token1!0;
run Account_deposit(account1);
(len(token1) == 0);

/* Executing account1's withdraw method */
param!30;
account1!param;
token1!0;
run Account_withdraw(account1);
(len(token1) == 0);

/* Getting the account1's value */
token1!0;
account1!param;
run Account_getValue(account1);
(len(token1) == 0);
account1?param;
int value;
param?value;

/* Checking if the value is correct */
assert(value == 20);


/* Executing account2's withdraw method */
param!30;
account2!param;
token2!0;
run Account_withdraw(account2);
(len(token2) == 0);

/* Getting account2's value */
token2!0;
account2!param;
run Account_getValue(account2);
(len(token2) == 0);
account2?param;
param?value;

/* Checking if the value is correct */
assert(value == 0);
```

```
/* Creating a customer */
chan custToken = [1] of {int}
chan customer = [3] of {chan}
custToken!0;
run Customer(customer, custToken);
(len(custToken) == 0);

/* Setting Customer's name */
mtype name = Mads;
chan nameParam = [1] of {mtype}
nameParam!name;
customer!nameParam;
custToken!0;
run Customer_setName(customer, custToken);
(len(custToken) == 0);

/* Getting Customer's name */
mtype returName;
chan nameParam2 = [1] of {mtype}
customer!param;
custToken!0;
run Customer_getName(customer, custToken);
(len(custToken) == 0);
customer?nameParam2;
nameParam2?returName;

/* Checking if the value is correct */
assert(returName == Mads);

/* Adding an account to customer */
chan accountParam = [1] of {chan, chan}
accountParam!account1,token1;
customer!accountParam;
custToken!0;
run Customer_addAccount(customer);
(len(custToken) == 0);

/* Adding another account to customer */
accountParam!account2,token2;
customer!accountParam;
custToken!0;
run Customer_addAccount(customer);
(len(custToken) == 0);



/* Getting an account from a customer */
param!2;
customer!param;
custToken!0;
run Customer_getAccount(customer);
(len(custToken) == 0);
customer?accountParam;
chan accout = [3] of {chan};
chan tokout = [1] of {chan};
accountParam?accout,tokout;

/* Getting the accounts balance */
tokout!0;
```

```
accout!param;
run Account_getValue(accout);
(len(tokout) == 0);
accout?param;
param?value;

/* Param checking account2's balance is correct */
assert(value == 0);

/* Getting a non-existing account */
param!0;
customer!param;
custToken!0;
run Customer_getAccount(customer);
(len(custToken) == 0);

/* Checking if no accounts are returned */
assert(len(customer) == 1);


/* terminating accounts and customer */
chan b =[1] of {chan}
account1?b;
account2?b;
customer?b;
}

proctype Customer(chan a; chan token) {
/* Class variables */
mtype name;
mtype city;
chan accounts[4] = [1] of {chan};
chan accountsToken[4] = [1] of {chan};

int accCounter = 0;
/* Method and class identifiers */
int getName = 10;
int setName = 11;
int getCity = 20;
int setCity = 21;
int addAccount = 30;
int getAccount = 31;

int classId = 12223;

/* inserting classid into object reference */
chan z = [1] of {int}
z!classId;
a!z;

/* Creating channels for receiving method calls
and their parameters */
chan methodcall = [1] of {int};
int methodvalue;
chan inMtypeParameter = [1] of {mtype};
chan addAccountParameter = [1] of {chan, chan};
chan inIntParameter = [1] of {int};
chan tempIntParameter = [1] of {int}
mtype inMtype;
```

```
chan inAccount = [1] of {chan};
chan inToken = [1] of {chan};
int inAccountNumber;
int tempInt;

token?0; /* Object construction complete */

/* Waiting for method calls */
do
:: (len(a) == 3) -> a?methodcall; methodcall?methodvalue;
   if
   :: (methodvalue == addAccount) & (accCounter < 4) ->
a?addAccountParameter;
addAccountParameter?accounts[accCounter],accountsToken[accCounter];
accCounter = accCounter + 1;
   :: (methodvalue == getAccount) -> a?inIntParameter;
inIntParameter?inAccountNumber; int i=0; do
     :: (i<accCounter) -> accounts[i]!inIntParameter;
accountsToken[i]!0;
run Account_getAccountNumber(accounts[i]);
(len(accountsToken[i]) == 0);
accounts[i]?tempIntParameter; tempIntParameter?tempInt; if
       :: (tempInt == inAccountNumber) ->
addAccountParameter!accounts[i],accountsToken[i]; a?z;
a!addAccountParameter; a!z; break
       :: (tempInt != inAccountNumber) -> i=i+1
       fi;
     :: (i≥accCounter) -> break
     od;
   :: (methodvalue == setName) -> a?inMtypeParameter;
inMtypeParameter?name
   :: (methodvalue == getName) -> a?inIntParameter;
inMtypeParameter!name; a?z; a!inMtypeParameter; a!z
   fi; token?0

:: (len(a) == 0) -> break /* Object termination */
od;

}

proctype Customer_addAccount(chan a) {
int classId = 12223;
int inClassId;
chan methodId = [1] of {int}
methodId!30;

/* Getting classid out of object reference */
chan inClass = [1] of {int};
a?inClass;
inClass?inClassId;
inClass!inClassId;

chan parameter = [1] of {int};

if
:: (inClassId == classId) -> a!methodId; a?parameter; a!parameter;
a!inClass
:: (inClassId != classId) -> skip
fi;
```

```
}

proctype Customer_getAccount(chan a) {
int classId = 12223;
int inClassId;
chan methodId = [1] of {int}
methodId!31;

/* Getting classid out of object reference */
chan inClass = [1] of {int};
a?inClass;
inClass?inClassId;
inClass!inClassId;

chan parameter = [1] of {int};

if
:: (inClassId == classId) -> a!methodId; a?parameter; a!parameter;
a!inClass
:: (inClassId != classId) -> skip
fi;

}

proctype Customer_getName(chan a) {
int classId = 12223;
int inClassId;
chan methodId = [1] of {int}
methodId!10;

/* Getting classid out of object reference */
chan inClass = [1] of {int};
a?inClass;
inClass?inClassId;
inClass!inClassId;

chan parameter = [1] of {int};

if
:: (inClassId == classId) -> a!methodId; a?parameter; a!parameter;
a!inClass
:: (inClassId != classId) -> skip
fi;

}

proctype Customer_setName(chan a) {
int classId = 12223;
int inClassId;
chan methodId = [1] of {int}
methodId!11;

/* Getting classid out of object reference */
chan inClass = [1] of {int};
a?inClass;
inClass?inClassId;
inClass!inClassId;
```

```
chan parameter = [1] of {int};

if
:: (inClassId == classId) -> a!methodId; a?parameter; a!parameter;
a!inClass
:: (inClassId != classId) -> skip
fi;

}


proctype Account(chan a; chan token) {
/* Class variables */
int value;
int accountNumber;
/* Method and class identifiers */
int getValue = 3221;
int withdraw = 1234;
int deposit = 4242;
int getAccountNumber = 10;
int setAccountNumber = 11;

int classId = 45521;
/* inserting classid into object reference */
chan z = [1] of {int}
z!classId;
a!z;


/* Creating channels for receiving method calls
and their parameters */
chan methodcall = [1] of {int};
int methodvalue;
chan inparameter = [1] of {int}; /* only one necessary */
chan outparameter = [1] of {int};
int invalue;


token?0; /* Object construction complete */

/* Waiting for method calls */
do
:: (len(a) == 3) -> a?methodcall; methodcall?methodvalue;
  if
  :: (methodvalue == getValue) -> a?inparameter; a?z;
outparameter!value; a!outparameter; a!z; token?0
  :: (methodvalue == withdraw) -> a?inparameter; inparameter?invalue;
    if
    :: (value ≥ invalue) -> value = value − invalue
    :: (value < invalue) ->
    fi; token?0
  :: (methodvalue == deposit) -> a?inparameter; inparameter?invalue;
value = value + invalue; token?0
  :: (methodvalue == getAccountNumber) -> a?inparameter; a?z;
outparameter!accountNumber; a!outparameter; a!z; token?0
  :: (methodvalue == setAccountNumber) -> a?inparameter;
inparameter?invalue; accountNumber = invalue; token?0
  fi;
:: (len(a) == 0) -> break /* Object termination */
```

```
od;
printf("Killing object")
}

proctype Account_getValue(chan a) {
int classId = 45521;
int inClassId;
chan methodId = [1] of {int}
methodId!3221;

/* Getting classid out of object reference */
chan inClass = [1] of {int};
a?inClass;
inClass?inClassId;
inClass!inClassId;

chan parameter = [1] of {int};

if
:: (inClassId == classId) -> a!methodId; a?parameter; a!parameter;
a!inClass
:: (inClassId != classId) -> skip
fi;

}


proctype Account_withdraw(chan a) {
int classId = 45521;
int inClassId;
chan methodId = [1] of {int}
methodId!1234;

/* Getting classid out of object reference */
chan inClass = [1] of {int};
a?inClass;
inClass?inClassId;
inClass!inClassId;

chan parameter = [1] of {int};

if
:: (inClassId == classId) -> a!methodId; a?parameter; a!parameter;
a!inClass
:: (inClassId != classId) -> skip
fi;

}

proctype Account_deposit(chan a) {
int classId = 45521;
int inClassId;
chan methodId = [1] of {int}
methodId!4242;

/* Getting classid out of object reference */
chan inClass = [1] of {int};
a?inClass;
inClass?inClassId;
```

```
inClass!inClassId;

chan parameter = [1] of {int};

if
:: (inClassId == classId) -> a!methodId; a?parameter; a!parameter;
a!inClass
:: (inClassId != classId) -> skip
fi;

}

proctype Account_getAccountNumber(chan a) {
int classId = 45521;
int inClassId;
chan methodId = [1] of {int}
methodId!10;

/* Getting classid out of object reference */
chan inClass = [1] of {int};
a?inClass;
inClass?inClassId;
inClass!inClassId;

chan parameter = [1] of {int};

if
:: (inClassId == classId) -> a!methodId; a?parameter; a!parameter;
a!inClass
:: (inClassId != classId) -> skip
fi;

}

proctype Account_setAccountNumber(chan a) {
int classId = 45521;
int inClassId;
chan methodId = [1] of {int}
methodId!11;

/* Getting classid out of object reference */
chan inClass = [1] of {int};
a?inClass;
inClass?inClassId;
inClass!inClassId;

chan parameter = [1] of {int};

if
:: (inClassId == classId) -> a!methodId; a?parameter; a!parameter;
a!inClass
:: (inClassId != classId) -> skip
fi;
}
```

C. SMV implementation

Here is the source code for the unfinished SMV implementation.

```
MODULE main
VAR
bank : Bank;
customers : array 0 .. 2 of Customer(1);
advisors : array 0 .. 3 of Advisor(1);
nullcustomer : Customer(0);
nullcustomeraction : Customeraction(0);
customeractions0 : array 0 .. 3 of Customeraction(1);
customeractions1 : array 0 .. 3 of Customeraction(1);
customeractions2 : array 0 .. 3 of Customeraction(1);
currentaction : Customeraction(0);

ASSIGN
bank.branches[0].city := Karlsruhe;
bank.branches[1].city := Mannheim;
customers[0].name := Mads;
customers[0].city := Karlsruhe;
customers[1].name := Matthias;
customers[1].city := Muehlacker;
customers[2].name := Elke;
customers[2].city := NA;

nullcustomeraction.type := idle;

customeractions0[0].type := openaccount;

customeractions0[1].type := deposit;
customeractions0[1].val2 := 3;
customeractions0[1].val1 := 1;
customeractions0[1].toaccount := 0;

customeractions0[2].type := idle;
customeractions0[3].type := withdraw;
customeractions0[3].val2 := 0;
customeractions0[3].val1 := 3;
```

```
init(currentaction) := nullcustomeraction;
next(currentaction) := case
currentaction = nullcustomeraction : customeractions0[0];
currentaction = customeractions0[0] : customeractions0[1];
currentaction = customeractions0[1] : customeractions0[2];
1 : customeractions0[3];
esac;

advisors[0].name := Per;
advisors[1].name := Tore;
advisors[2].name := Harald;
advisors[3].name := Ole;

bank.branches[0].advisors[0] := advisors[0];
bank.branches[0].advisors[1] := advisors[1];
bank.branches[1].advisors[2] := advisors[2];
bank.branches[1].advisors[3] := advisors[3];

init(bank.customers[0].name) := NA;

init(bank.newcustomer) := customers[0];

next(bank.newcustomer) := case
bank.newcustomer = customers[0] : customers[1];
bank.newcustomer = customers[1] : customers[2];
1 : nullcustomer;
esac;

init(bank.accountaction) := idle;
next(bank.accountaction) := currentaction.type;

next(bank.val2) := case
currentaction.type in {deposit, withdraw, transfer1} : currentaction.val2;
1: bank.val2;
esac;
next(bank.val1) := case
currentaction.type in {deposit, withdraw, transfer1} : currentaction.val1;
1: bank.val1;
esac;


DEFINE

bank.accounts[0].accountnumber := 1;
bank.accounts[1].accountnumber := 2;
bank.accounts[2].accountnumber := 3;
bank.accounts[3].accountnumber := 4;


SPEC
AG(bank.accounts[0].accountnumber = 1) &
AG(bank.branches[0].city = Karlsruhe) &
AG(advisors[1].name = Tore) &
AG(customers[0].name = Mads) &
AG(bank.customers[0].nullpointer ->
(bank.customers[0].name = Mads)) &
AG(bank.customers[2].nullpointer ->
(bank.customers[2].name = Matthias) ) &
AG(!bank.customers[3].nullpointer) &
```

```
AG(bank.customers[2].nullpointer ->
(bank.customers[2].name = Elke)) &
AG({Karlsruhe} in {bank.branches[1].city, Heia, Karlsruhe}) &
EF(bank.accounts[0].val2*10+bank.accounts[0].val1 = 31)



MODULE Bank
VAR
accounts : array 0 .. 3 of Account(1);
branches : array 0 .. 1 of Branch;
customers : array 0 .. 3 of Customer(0);
newcustomer : Customer(0);
currentcustomer : Customer(0);
val2 : { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
val1 : { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
accountaction : {deposit, withdraw, transfer1, transfer2, idle};
currentaccount : { -1, 0, 1, 2, 3 };
successwithdraw : boolean;

ASSIGN
init(accounts[0].val2) := 0;
init(accounts[0].val1) := 0;
init(accounts[1].val2) := 0;
init(accounts[1].val1) := 0;
init(accounts[2].val2) := 0;
init(accounts[2].val1) := 0;
init(accounts[3].val2) := 0;
init(accounts[3].val1) := 0;


next(accounts[currentaccount].val2) := case
accountaction = deposit & ((accounts[currentaccount].val2+val2)*10 +
accounts[currentaccount].val1+val1) < 100 :
accounts[currentaccount].val2+val2 +
!( ((accounts[currentaccount].val1+val1) mod 10) =
((accounts[currentaccount].val1+val1) mod 10));
1 : accounts[currentaccount].val2;
esac;



next(customers[0]) := case newcustomer.nullpointer &
!customer[0].nullpointer &
{newcustomer.city} in {branches[0].city, branches[1].city} :
newcustomer;
1 : customers[0];
esac;

next(customers[1]) := case newcustomer.nullpointer &
!customer[1].nullpointer &
{newcustomer.city} in {branches[0].city, branches[1].city} :
newcustomer;
1 : customers[1];
esac;

next(customers[2]) := case newcustomer.nullpointer &
!customer[2].nullpointer &
{newcustomer.city} in {branches[0].city, branches[1].city} :
```

```
newcustomer;
1 : customers[2];
esac;

next(customers[3]) := case newcustomer.nullpointer &
!customer[3].nullpointer &
{newcustomer.city} in {branches[0].city, branches[1].city} :
newcustomer;
1 : customers[3];
esac;



DEFINE
hade := accounts[0].output;



MODULE Account(null)
VAR
Type : {Savings, house};
val2 : { 0, 1, 2, 3, 4, 5, 6, 7, 8 ,9 };
val1 : { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
DEFINE
nullpointer := null;



MODULE Branch
VAR
hei : boolean;
city : {Karlsruhe, Mannheim, Muehlacker};
advisors : array 0 .. 3 of Advisor(1);



MODULE Advisor(null)
VAR
name : {Per, Tore, Harald, Ole};
DEFINE
nullpointer := null;



MODULE Customer(null)
VAR
name : {Mads, Matthias, Elke};
city : {Karlsruhe, Muehlacker, NA};
accounts : array 0 .. 4 of Account(0);
advisor : Advisor(0);
DEFINE
nullpointer := null;

MODULE Customeraction(null)
VAR
type : {idle, openaccount, transfer, deposit, withdraw, getadvisor};
val2 : { 0, 1, 2, 3, 4, 5, 6, 7, 8 ,9 };
val1 : { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
fromaccount : { 1, 2, 3, 4 };
```

```
toaccount : { 1, 2, 3, 4 };
DEFINE
nullpointer := null;
```

Bibliography

[BBCF+98] Nikolaj Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner, Zohar Manna, Marc Pichora, Henny B. Sipma and Tomás E. Uribe. The Stanford Temporal Prover Educational Release, USER'S MANUAL. http://www-step.stanford.edu/manual/manual.ps.gz, Jul 1998.

[Bott03] Richard J. Botting. Glossary of Computer Language Terms. http://www.csci.csusb.edu/dick/samples/glossary.html, 2003.

[Chec97] Marsha Chechik. Lecture Notes, SMV. http://www.cs.toronto.edu/ chechik/courses97/csc2108/lectures/smvuse.ps, 1997.

[CORB04] CORBA Language Mapping Specifications. http://www.omg.org/technology/documents/formal/corba_language_mapping_s%pecs, 2004.

[Easy04] Easy Comp - homepage. www.easycomp.org, 2004.

[FoSc00] Martin Fowler and Kendall Scott. *UML konzentriert*. Addison-Wesley. 2000.

[Gert97] Rob Gerth. Concise Promela Reference. http://spinroot.com/spin/Man/Quick.html, Aug 1997.

[Get 04] Glossary of Object-Oriented Terms. http://www.getobjects.com/Examples/glossary.html, 2004.

[HaDw01] John Hatcliff and Matthew Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. *Lecture Notes in Computer Science* Band 2154, 2001, S. 39–??

[Holz97] Gerard J. Holzmann. The Model Checker SPIN. *Software Engineering* 23(5), 1997, S. 279–295.

[HoRu03] Gerard Holzmann and Theo C. Ruys. Advanced SPIN Tutorial. http://spinroot.com/spin/Doc/fm2003-spin-tutorial.pdf, Sep 2003.

[HuRy00] Michael R A Huth and Mark D Ryan. *Modelling and reasoning about systems: Logic in Computer Science*. Cambridge University Press. 2000.

[Lamp94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems* 16(3), May 1994, S. 872–923.

[MaPn95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc. 1995.

[McKi00] K. L. McKillan. The SMV system. http://www-2.cs.cmu.edu/~modelcheck/smv/smvmanual.ps, Nov 2000.

[NPAR03] Glossary of Verification and Validation Terms. http://www.grc.nasa.gov/WWW/wind/valid/tutorial/glossary.html, Aug 2003.

[Pars03] GOLD Parser. Parser concepts, Grammars and Backus Naur form. http://www.devincook.com/goldparser/concepts/bnf.htm, 2003.

[Pulv02] Elke Pulvermüller. Composition and Correctness. In *SC 2002: Workshop on Software Composition*, Band 65 der *Electronic Notes in Theoretical Computer Science (ENTCS)*, `http://www.elsevier.com/gej-ng/31/29/23/show/Products/notes/index.htt`, April 2002. Elsevier Science Publishers.

[RaRH03] Anthony Ralston, Edwin D Reilly and David Hemmendinger (Hrsg.). *Encyclopedia of Computer Science*. Wiley, John and Sons, Incorporated. 2003.

[Ruys02] Theo C. Ruys. SPIN Beginners Tutorial. http://spinroot.com/spin/Doc/SpinTutorial.pdf, Apr 2002.

[Seth96] Ravi Sethi. *Programming Languages*. Addison-Wesley. 1996.

[SMV04] The SMV System. http://www-2.cs.cmu.edu/~modelcheck/smv.html, feb 2004.

[SPIN96] Basic Spin Manual. http://spinroot.com/spin/Man/Manual.html, Aug 1996.

[SPIN97] Promela grammar. http://spinroot.com/spin/Man/grammar.html, Aug 1997.

[SPIN03] On-the-fly, LTL model checking with SPIN. http://spinroot.com/spin/whatispin.html, Dec 2003.

[SPJF02] Andreas Speck, Elke Pulvermüller, Michael Jerger and Bogdan Franczyk. Component Composition Validation. *International Journal of Applied Mathematics and Computer Science.* 12(4), December 2002, S. 581 – 589.

[SPL04] Context-free grammar for SPL. http://www.cs.yorku.ca/stateclock/STeP/-WWW/syntax-complete.ps, 2004.

[STeP03] The Stanford Temporal Prover. http://www-step.stanford.edu/, 2003.

[vHen03] F. W. von Henke. SMV - Überblick. http://www.informatik.uni-ulm.de/ki/Edu/Vorlesungen/Modellierung.und.Verifikation/WS0304/folien3.pdf, 2003.