Formal Verification of UML-RT Capsules using Model Checking

Master of Science Thesis in Secure and Dependable Computer Systems

MATS CARLSSON
LARS JOHANSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, June 2009

Formal Verification of UML-RT Capsules using Model Checking

Mats Carlsson
Lars Johansson

©Mats Carlsson, June 2009.
©Lars Johansson, June 2009.

Examiner: Wolfgang Ahrendt

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden, June 2009

Abstract

Formal verification methods have successfully been used to ensure correctness of both hardware and software systems. In contrast to testing methods, that can demonstrate the presence of faults in a system, formal methods can prove their absence.

A department of the telecommunications company Ericsson AB in Gothenburg, Sweden, uses the UML-RT language to model software used in WCDMA radio base stations. These concurrent and reactive systems can be modeled in the Eclipse-based RSARTE environment.

Previous work underlines a need of narrowing the gap between software development tools used in industry and formal verification tools. This thesis examines the feasibility of using model checking to verify properties of UML-RT capsules. We present a prototype tool for generating verification models in the Promela language for the model checker Spin. The tool is implemented as a model-to-text transformation using the JET tool and is integrated into RSARTE.

The result of the work establishes that it, for a subset of constructs in UML-RT, is possible to automate generation of verification models that can be used to demonstrate properties of the original UML-RT capsules. We demonstrate this with example models created in RSARTE.

Keywords formal verification, model checking, model-to-text, Promela, RSARTE, Spin, UML-RT.

Sammanfattning

Formella verifieringsmetoder har med framgång använts för att säkerställa korrekthet av både hårdvaru- och mjukvarusystem. Till skillnad från testmetoder, vilka enbart kan visa förekomst av fel i system, kan formella metoder bevisa frånvaron av dessa.

En avdelning på telekommunikationsföretaget Ericsson AB i Göteborg använder modelleringsspråket UML-RT för att modellera mjukvara för användning i radiobasstationer för WCDMA. Dessa parallella och reaktiva system kan modelleras i den Eclipse-baserade utvecklingsmiljön RSARTE.

Tidigare arbeten understryker ett behov av att minska avståndet mellan industriella mjukvaruutvecklingsverktyg och verktyg för formell verifiering. Detta arbete utforskar möjligheten att använda *model checking* för att verifiera egenskaper hos UML-RT-kapslar. Vi presenterar ett prototypverktyg för att generera verifieringsmodeller i språket Promela, som används av *model checking*-verktyget Spin. Prototypverktyget är implementerat i form av en *model-to-text*-transformation med hjälp av verktyget JET och är integrerat i RSARTE.

Resultatet av arbetet fastställer att det, för en delmängd av byggstenarna i UML-RT, är möjligt att automatisera framställning av verifieringsmodeller, som därefter kan användas för att påvisa egenskaper hos de ursprungliga UML-RT-kapslarna. Vi demonstrerar detta med hjälp av ett antal exempelmodeller som skapats med RSARTE.

Sökord formell verifiering, model checking, model-to-text, Promela, RSARTE, Spin, UML-RT.

Preface

This report is the result of a Master of Science thesis project in the program Secure and Dependable Computer Systems at Chalmers University of Technology in Gothenburg, Sweden. The work has been conducted between February and June of 2009 for Ericsson AB, at a department developing application software for WCDMA radio base stations.

The examiner for the thesis has been Dr. Wolfgang Ahrendt from the Department of Computer Science and Engineering at Chalmers University of Technology, whose advice and support we gratefully acknowledge. Our supervisors at Ericsson have been Anders Borghed, Peter Eriksson and Sebastian Holmgren, all of whom have provided much appreciated support, guidance and encouragement throughout our work.

Contents

List of Figures

Listings

Terminology

Acronyms

CTL	Computation Tree Logic
FSM	Finite State Machine
JET	Java Emitter Template
LTL	Linear Time Logic
OTD	ObjecTime Developer
OTI	Object Technology International
Promela	Process Meta Language
PSL	Property Specification Language
ROOM	Real-Time Object-Oriented Methodology
RoseRT	IBM Rational Rose RealTime
RSARTE	Rational Software Architect Real Time Edition
SMARRT	Static Model checking and Analysis for Rose RealTime
SMV	Symbolic Model Verifier
Spin	Simple Promela Interpreter
TABU	Tool for the Active Behaviour of UML
UML-RT	UML Real Time
UML	Unified Modeling Language
VIP	Visual Interface to Promela
WCDMA	Wideband Code Division Multiple Access

XMI	XML Metadata Interchange
XML	Extensible Markup Language
XPath	XML Path Language

CHAPTER 1

Introduction

The presence of errors in computer systems, both originating from incorrect design choices as well as from implementation mistakes, remains a challenge in the industry. Despite traditional validation and verification counter measures, such as simulation and testing, mistakes are costly. Figures released in 2002 estimated that the industry cost of software faults in the United States alone, approached 60 billion dollars (NIS 2002).

Faults may be present in the design or implementation of a system and certain conditions may activate a fault, causing an error to be produced. Unless such an error is properly handled, it may propagate and cause the system output to deviate from the specified output, resulting in a failure (Avizienis, Laprie, Randell & Landwehr 2004).

Software and hardware testing can expose faults that may exist in the system, provided that a test case has been constructed, which activates the fault and exposes the resulting error. The task of constructing suitable test cases is not trivial and attempting to ensure that all faults are found by testing the system exhaustively is often an impossible task (Clarke, Grumberg & Peled 1999, p. 2). As will be illustrated by the following two examples, gathering even overwhelming amounts of empirical evidence to support a claim does not *prove* its correctness.

The first example is the floating point unit of Pentium processors which was found to be flawed in 1994. The fault lay in a lookup table used for division operations and was due to entries missing from the table. This, in turn, caused certain instructions related to floating point division to produce results deviating from the correct output. The probability of activating the fault by applying one of the affected instructions to a randomly chosen value from the input space was 1 to 9 billion, according to analysis by Intel (Int 1994).

A more theoretical example of a problem which is difficult to analyze correctly using testing is the following: We can claim that there exists no positive integer n such that $991 \cdot n^2 + 1$ is a perfect square, apart from the trivial solution of $n = 0$. This claim happens to be incorrect, but the first positive integer solution does not occur until

$$n = 12,055,735,790,331,359,447,442,538,767$$

(Rotman 1998), suggesting that it is difficult to falsify the hypothesis using testing. The only feasible method of refuting the hypothesis is to use a method that relies on proofs rather than on an exhaustive search.

Formal verification methods rely on techniques from logic and mathematics, in practice usually supported by computer tools, to prove properties about systems. Such methods have been used successfully in the development of systems that require a very high degree of confidence that the product meets its specification.

There are many reasons to why a system may demand rigorous design validation. Safety requirements, such as in air traffic control systems or flight control systems; security requirements, such as in implementations of cryptographic protocols; or the cost of failure may all motivate the use of formal verification. Legal requirements, such as in digital signing of programs or documents to ensure authenticity and non-repudiation; certification requirements; or legal implications of failure in any of the above scenarios may also warrant the use of formal methods (Elamkulam, Glazberg, Rabinovitz, Kowlali, Gupta, Kohli, Dattathrani & Macia 2006).

1.1 Background

The telecommunications company Ericsson AB develops software for radio base stations for the Wideband Code Division Multiple Access (WCDMA) standard at a department at Lindholmen in Gothenburg (hereinafter referred to only as the "department"). The software is developed with visual tools that support modeling in a dialect of UML and C++ code is then generated directly from the models. The modeling language in use at the department is UML Real Time (UML-RT), currently supported by the tool IBM Rational Rose RealTime (RoseRT) but being superseded by the Rational Software Architect Real Time Edition (RSARTE) tool.

UML-RT, described in more detail in Chapter 4, is for example used at the department to model hierarchies of concurrently operating system components called *capsules*, whose behaviors are defined using state machines and that communicate with each other using message passing. The behavior of a single capsule can be very complex and the combined behavior of several capsules even more so.

Testing of both normal and failure cases is performed at the department. However, there is always a possibility that faults in capsules remain undiscovered and appear when least expected. In the words of Dijkstra (1970, p. 7), "Program testing can be used to show the presence of bugs, but never to show their absence!" For this reason, the department is interested in understanding and exploring the possibility of using formal verification methods as a complement to the testing procedures in current use.

1.2 Aim

The purpose of the thesis is to evaluate how formal verification can be integrated in the model based development process used for developing the application software for WCDMA base stations at the department. This thesis aims to provide answers to the following questions:

1. What methods and tools for formal verification of software models are available in the academic, industrial or open source communities? We decompose this question into the following sub-questions:
   (a) What approach could be used as a basis for verification of models developed at the department?
   (b) Are there existing tools for formal verification that can be directly used to verify properties of models in the development environment at the department? If this is not the case, are there existing tools that can serve as a foundation for implementing such a tool?
   (c) Can these tools be integrated into the development environment at the department?
2. What properties are of interest to the department to verify?
3. How should models be developed to allow such properties to be formally verified using the chosen method?
4. What can be gained from applying formal verification methods to the department's software models?

Furthermore, since it is of interest to the department that the viewpoints of the software designer are considered throughout the work, decisions regarding the verification approach, tool selection and design choices will be made with the potential future user of the tool in mind.

1.3 Limitations

Any prototypical tool implemented is intended to serve as a proof of concept, outlining a possible method for property verification. For this reason, it is not an aim to handle models of the same size or complexity as can be found in the department's products. In addition to varying greatly in complexity, the models developed at the department are produced with tools that support a large number of modeling constructs. The scope of this thesis is limited to a subset of the available constructs (see Section 6.2.2).

The verification is intended to target only properties of the model and not to make any claims about correctness of code embedded in the model. Therefore, no code analysis methods for extracting information from embedded code are considered. This in turn implies that any fully automated verification procedure is beyond the scope of this thesis.

Any model properties related to time (in the sense that can be measured numerically, not in the sense of ordering of events) will be either treated on an abstract level or be completely disregarded. This is a common practice when modeling concurrent or distributed systems, because as Ben-Ari (2008, p. 173) explains,

> "Algorithms for these systems are designed to be independent of the speed of execution of a process or the speed at which a message is delivered, so it is sufficient to know that there are no errors caused by interleaving statements and messages."

1.4 Disposition

Abbreviations, terminology and names used in this disposition are explained as they are later introduced. The remainder of this thesis is ordered as follows:

Chapter 2 presents an introduction to the field of formal verification, as well as a brief introduction to the verification tool Spin, which is used in the prototype tool.

Chapter 3 presents the method chosen for addressing the questions that this thesis aims to answer.

Chapter 4 presents the background of UML-RT and a description of the modeling tools used at the department. A presentation of modeling concepts and building blocks of UML-RT is also given, together with an explanation of the link between UML-RT and the two modeling tools RoseRT and RSARTE.

Chapter 5 presents a survey of previous work in the field of software model verification. An assessment of previous work and existing verification tools as they relate to the modeling language used at the department is also given.

Chapter 6 presents a description of a prototype tool implemented for integrating the model checker Spin with the modeling tool RSARTE.

Chapter 7 presents a categorization of system properties and a description of how such properties may be specified. The two properties that are of primary interest to the department are also presented.

Chapter 8 describes the models and presents the properties that are verified using the prototype tool.

Chapter 9 presents verification results for each model and property.

Chapter 10 presents answers to the questions that the thesis aims to answer. A discussion on the limitations of the chosen verification approach and prototype tool is presented, together with suggestions for possible future extensions and improvements of the work.

Appendix A presents the verification model for a capsule discussed in Section 8.4.

Appendix B contains two sample JET templates that are part of the prototype tool.

CHAPTER 2

Theory

This chapter provides an introduction to the use of model checking tools, to formally verify systems. We describe formal specification languages using the model checker Simple Promela Interpreter (Spin) and its input languages as the example.[1]

2.1 Formal Verification

The objective of formal verification is to extend beyond what is possible to achieve using testing procedures, by providing a formal proof that a system conforms with design specifications. The system in question can, for example, be a hardware circuit or communication protocol (Clarke et al. 1999), or a software design (Holzmann 2003). The two major methods used to formally verify systems are *deductive verification* and *model checking* (Clarke et al. 1999).

Deductive verification relies on using a system of axioms and application of inference rules, to prove a system correct. This can be, and was originally done, manually, but generally requires expertise in both the area of the target system and in mathematics or logic. Tools such as KeY[2] and HOL Light[3] have been developed to aid the user by automatically providing a complete proof, or by serving as a proof assistant that guides the user's interaction with a theorem prover. According to Ben-Ari (2008, p. 23), one advantage of deductive methods is that they may allow verification of systems where the size of the state space would otherwise be limiting.

[1]Spin is used as the example since it is used in later parts of this thesis. A motivation for this choice is given in Section 5.6.

[2]`http://www.key-project.org`

[3]`http://www.cl.cam.ac.uk/~jrh13/hol-light/`

2.2 Model Checking

The model checking field originated in the early 1980s as a method for addressing the problem of verifying concurrent programs (Clarke 2008). Such programs are hard to debug due to the frequent difficulty of reproducing errors (Clarke 2008). If the tasks executing concurrently in the program communicate or otherwise depend upon each other, it can very quickly become problematic to establish under what conditions deadlocks may occur or mutual exclusion constraints may be violated, due to the large number of possible ways in which multiple tasks may interleave.

Clarke et al. (1999) partition the process of model checking a design into three steps; modeling, specification and verification.

Modeling The design must be represented or encoded in some formal language. This can be done by directly writing code in the language of the model checker, or by transforming some other description, such as source code, into the language of the model checker. This transformation can be done manually, or by using some automated pre-processor (see for example Beyer, Henzinger, Jhala & Majumdar (2007) and Holzmann & Smith (1999)).

Specification The properties that the model should satisfy must also be stated in an unambiguous, formal notation. This requires firstly an understanding of what properties are of interest to verify, and secondly a language sufficiently expressive to capture those properties. Examples of such languages are Linear Time Logic (LTL)[4] and Computation Tree Logic (CTL), both members in the temporal logic family.

Verification The tool will accept the model and the specification and determine whether the design meets the requirements. The tool will determine if, for every possible path through an algorithm or for every possible state of a hardware circuit, the model satisfies the specification.

This task is not performed with brute-force methods since that would be infeasible for most target systems. For example, circuits with more than 10^{90} states have been verified using model checking (Emerson 2008) which is far beyond the reach of any brute-force algorithm.

The two dominating model checking methods are divided by Holzmann (2003) into those using symbolic verification methods and those using explicit verification methods. An example falling into the first category is the Symbolic Model Verifier (SMV) tool that is based on Ordered Binary Decision Diagram manipulation techniques (Clarke et al. 1999). The Spin tool falls into the second category and uses partial order reduction (see section 2.3.5) as part of its strategy to cope with the state explosion problem (Holzmann 2003).

2.2.1 Model checking workflow

The practical usage of a model checker will typically follow the workflow outlined in Figure 2.1. A design is transformed into a description suitable for the model

[4]LTL is described in more detail in Section 2.3.3.

checker, either manually or with the aid of tools, and a specification of a wanted or unwanted behavior is captured as a property. The result produced by the model checker on the basis of this information is either confirmation that the property has been verified to hold, or an error trace showing how the property can be invalidated (Clarke 2008, pp. 2–3). If given sufficient resources, the model checking tool will always terminate with an answer (Clarke 2008, pp. 3).

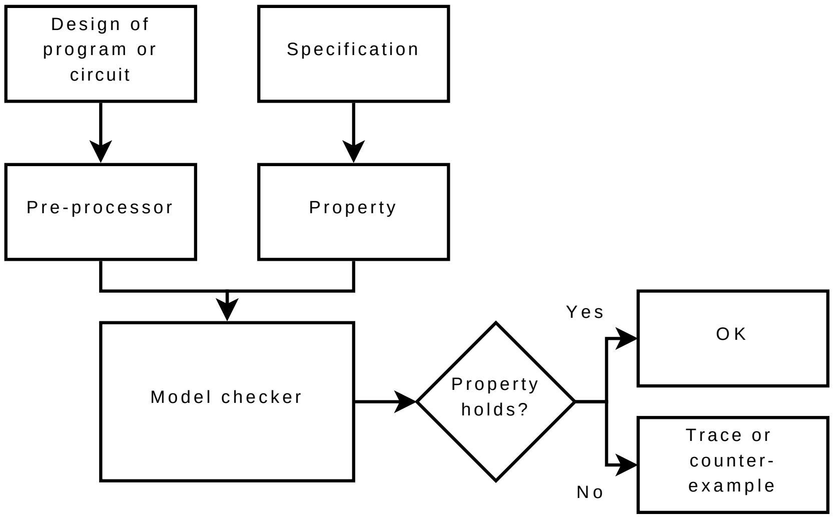


Figure 2.1: Outline of a general model checking workflow.

2.3 The Spin model checker

The Spin model checker was originally developed at Bell Labs in the beginning of the 1980s (Holzmann 1997) and has since been continuously developed. Spin has been available since 1991 and can be used freely for educational purposes (Holzmann 2003). Commercial use of Spin does not command a fee but requires a license agreement to be accepted (LIC 2001).

Spin is a system for proving properties of software systems by enabling a designer to create abstract models of the target system, specify properties that must hold for the model and verify if this is in fact the case or not.[5] Spin allows the designer to explore — through simulation or verification — system behavior which results from interaction between processes.

Spin uses verification models specified in Process Meta Language (Promela) (Holzmann 1997) and can be used in several ways including random, interactive or guided simulation mode, and verification mode. When used as random simulator, Spin will follow one randomly chosen execution path of the modeled system. In interactive simulation mode, the user is called upon to decide how to proceed when a choice must be made between possible execution paths. In guided simulation mode, Spin follows a trail file that describes a specific execution path through the system.

[5]For the underlying theory of how models and properties are handled by Spin to perform the verification, see Holzmann (1997).

Note that no amount of repeated random simulation guarantees that every execution path is eventually taken. The results from a simulation run is only applicable to the particular execution path through the system selected in that specific simulation run. A verification run is therefore necessary to verify claims about *all* of the possible simulation runs, i.e., about every possible execution path of the system. If Spin finds that a model does not satisfy a particular property, then a trail file is generated. The file details the exact steps that produces a violation of the property and can be used in guided simulation mode to replay the execution (Holzmann 2003, pp. 245–252).

2.3.1 The Promela specification language

Promela is a specification language, visually resemblant of C, for describing models of concurrent systems. The number of constructs supported by the language is intentionally small and the focus of those constructs is on describing behavior and interaction between system components, rather than computation. The reason for this is that, “Promela is not meant to be an implementation language but a systems description language.” Holzmann (2003, p. 8)

The Promela language provides features for describing concurrently executing processes and communication between such processes using message passing over buffered or unbuffered message channels. Promela also makes it easy to model indeterministic choice through the use of control statements similar to Dijkstra’s guarded commands (Holzmann 2003, p. 407).[6] However, there are no features for returning values from function calls, no support for floating point numbers and no notion of time beyond event ordering.(Holzmann 2003, p. 8)

2.3.2 A Promela example

It is beyond the scope of this thesis to describe details of Spin and Promela, but Listing 2.1 serves to give an impression of syntax and concepts of the Promela language. For a complete description of Spin, Promela and further examples see, e.g., Holzmann (2003), Ben-Ari (2008) and Holzmann (1997).

The example models a scenario with two bank tellers and two bank customers, communicating in pairs using message passing over a channel. The customers share one bank account with an initial balance of 60 currency units. The customers both carry out three transactions, choosing indeterministically between attempting to withdraw or to deposit 50 currency units. When a bank teller receives an order to withdraw money, the balance of the account is checked to ensure that it holds sufficient funds. If this is the case then the balance is decreased and the teller then verifies that the balance has not somehow fallen below zero (see lines 14–15 in Listing 2.1). When a bank teller receives an order to deposit money, the balance is increased.

A verification run with Spin can be used to find a process interleaving that can cause the account balance to fall below zero. The resulting trail file can in turn be used to playback the execution and locate the error. Listing 2.2 presents condensed output from the execution playback. It demonstrates that the assertion violation occurs when the first teller is interrupted in the withdrawal procedure *after* checking the account balance, but *before* recording the

[6]See for example lines 28–31 in Listing 2.1.

Listing 2.1: Promela model vulnerable to a data race situation.

```
short balance = 60;

mtype = {withdraw_50, deposit_50};

chan channel_a = [0] of { mtype };
chan channel_b = [0] of { mtype };

proctype teller(chan in) {
end:
   do
   :: in ? withdraw_50 ->
      if
      :: balance >= 50 ->
         balance = balance - 50;
         assert(balance >= 0);
      :: else -> skip;
      fi
   :: in ? deposit_50 ->
      balance = balance + 50;
   od
}


proctype customer(chan out) {
   byte transactions = 3;
   do
   :: transactions > 0 ->
      if
      :: out ! withdraw_50;
      :: out ! deposit_50;
      fi;
      transactions = transactions - 1;
   :: else -> break;
   od
}


init {
   atomic {
      run teller(channel_a);
      run customer(channel_a);
      run teller(channel_b);
      run customer(channel_b);
   }
}
```

Listing 2.2: Condensed Spin output demonstrating a process interleaving resulting in an assertion violation.

```
proc  2 (customer 1) [out!withdraw_50]
proc  1 (teller 1)   [in?withdraw_50]
proc  1 (teller 1)   [((balance>=50))]

proc  4 (customer 2) [out!withdraw_50]
proc  3 (teller 2)   [in?withdraw_50]
proc  3 (teller 2)   [((balance>=50))]

proc  3 (teller 2)   [balance = (balance-50)]
proc  3 (teller 2)   [assert((balance>=0))]

proc  1 (teller 1)   [balance = (balance-50)]
spin: Error: assertion violated
spin: text of failed assertion: assert((balance>=0))
proc  1 (teller 1)   [assert((balance>=0))]
```

new balance. The consequence is that the tellers are able to make a withdrawal each, based on the same account balance, incorrectly causing a final negative balance.

2.3.3 Property specification in Spin using LTL

The properties that we wish to prove or refute for a given Promela model must be specified in some formal notation. The example in Section 2.3.2 showed a correctness claim specified using an assertion statement, but Spin also supports verification of correctness claims specified in LTL (Holzmann 1997). This temporal logic allows specifications that refer to the future (Huth & Ryan 2004, p. 175) by extending propositional logic formulas with temporal connectives (Ben-Ari 2008, pp. 71–72).

The operators inherited from propositional calculus are: *negation*, *conjunction*, *disjunction*, *implication* and *equivalence*. In addition to this, LTL provides the temporal operators *always*, *eventually*, *(strong) until*, the dual of *until* (commonly referred to as *release*) and *next*. These allow us to state claims about the behavior of the Promela model and to use Spin to assert or refute those claims. See Holzmann (2003, pp. 135–136) for a complete description of the semantics of the operators and the syntax used in Spin.

Always ($\Box$) captures properties that are related to invariance, e.g., the formula $\Box\, p$ specifies that condition p always holds true. The operator is written `[]` in Promela.

Eventually ($\Diamond$) captures properties related guaranteed behavior, e.g. the formula $\Diamond\, p$ specifies that the condition p holds in the current state or will hold in some future state. The operator is written `<>` in Spin.

Until ($\mathcal{U}$) captures properties of relative behavior, e.g. $p\ \mathcal{U}\ q$ specifies that the condition p must hold until q becomes true (now or in the future).

The definition of *weak* until does not require that q ever becomes true, while the definition of *strong* until does require that q at some point holds true. Spin uses the strong definition of until (Ben-Ari 2008, p. 91). The operator is written `U` in Spin.

Release ($\mathcal{R}$) is the dual of the strong until operator, e.g., $p \mathcal{R} q$ specifies that q holds true until p becomes true, which *releases* q. If p never becomes true, then q must hold forever. The operator is written `V` in Spin.

Next ($\mathcal{X}$) captures properties that relate a state to its successor, e.g., $\mathcal{X}\ p$ holds in the current state iff p holds in the *next* state. The operator is written `X` in Spin.

The *next* operator requires caution because of the restrictions on its use that are imposed by Spin. The default behavior of Spin is to disallow the use of the *next* operator, due to the possibility of conflicts with the state space reduction method used by Spin (see Section 2.3.5).

2.3.4 LTL property verification

Spin can be used both to prove desired behaviors (i.e., properties that should always hold) or error behaviors (i.e., properties that should never hold). For reasons of verification efficiency, Spin does not attempt to prove that a behavior is guaranteed; instead Spin attempts to show how a behavior claimed to be impossible can in fact be achieved (Holzmann 1997, p. 97).

This means that to prove a desired behavior, an LTL formula that captures that behavior is specified and then *negated*. Spin then attempts to show a run of the system in which the negated formula holds. If Spin succeeds, then the desired behavior can be violated; but if Spin determines that the negated claim cannot be refuted, then the model must exhibit the desired behavior. We can conclude that Spin is used "to check for *violations* of requirements." (Holzmann 2003, p. 149)

The challenge for any designer using model checking is to construct a small yet sufficiently detailed verification model that ideally only captures the features of the design that "must be considered to establish correctness" (Clarke et al. 1999, p. 13). Unnecessary details that make the model more complicated without affecting the "correctness of the checked properties" (Clarke et al. 1999, p. 13) should be omitted.

2.3.5 Problem space reduction

Holzmann (2003, p. 191) writes that Spin makes use of two types of strategies for addressing the state space explosion problem, the aims of which are either "to reduce the number of reachable states that must be searched to verify properties, or to reduce the amount of memory that is needed to store each state."

One method used by Spin is called partial order reduction. According to Clarke et al. (1999) partial order reduction relies on selecting and examining only a subset of all possible execution paths. One example of how this is achieved is by detecting interleaving of processes such that the relative ordering of the processes' execution steps do not affect the final outcome of the execution, with regards to the property being verified. This reduces the problem size because, as Clarke et al. (1999) writes,

> "When a specification cannot distinguish between two interleaving sequences that differ only by the order in which concurrently executed events are taken, it is sufficient to analyze only one of them."

Another means of reduction is to exploit stutter equivalence, in that,

> "a pair of sequences are considered to be equivalent if they differ in at most the number of times a state may adjacently repeat." (Peled, Wilke & Wolper 1995)

Spin's partial order reduction strategy assumes that stutter equivalence can be used and is therefore only guaranteed to be valid for stutter invariant properties. Although it is not impossible to write stutter invariant properties that make use of the next operator (Holzmann 2003), an LTL formula which does not contain the next operator is guaranteed to be stutter invariant (Peled et al. 1995). It is therefore also guaranteed not to invalidate the results of the partial order reduction algorithm (Holzmann 2003). Nonetheless, Ben-Ari (2008) comments that the abstract treatment of time explains why it would be of limited benefit to allow the next operator,

> "For example, in a client-server system, we want to specify that a client process *eventually* receives a service from a server process but it doesn't really matter if that occurs in the next state or ten states later."

Lamport (1983, p. 661) also argues against the inclusion of a next operator in a temporal logic with the motivation that this allows requirements to be specified that distinguish between models on the basis of properties that are irrelevant in an abstract specification.

CHAPTER 3

Method

In order to address the questions posed in Section 1.2, this project is divided into two phases:

Phase 1: An initial study of previous work describing methods and tools for formal verification of state machines and UML-RT models is carried out. The purpose of this is to provide an understanding of what verification approach can be suitable in the modeling environment at the department. We assess the applicability of existing work in the context of this environment, based on two factors:

1. The ability of the tool to verify properties that are of current interest, but also of possible future interest to the department.
2. The potential for integrating the proposed approach (and possibly already existing tool) into the department's tool environment.

The findings of the initial study determine if

1. a tool considered suitable is available, or if
2. a prototype tool based on principles suggested by previous work has to be implemented.

The results of the first phase determined that the second phase was directed towards the latter possibility, i.e., towards implementing a prototype tool.

Phase 2: We then demonstrate the chosen verification approach by conducting a case study, exploring the use of model checking in the setting of an RSARTE environment. In the case study, we

1. model selected problems with the modeling tools used at the department, and

2. apply the selected verification approach and tool to demonstrate how properties of those models can be verified.

The problems studied and modeled are chosen mainly from training material for the modeling tools used at the department. Starting with models from training material is suitable since such models introduce fundamental building blocks and important constructs used in the modeling environment. Avoiding complex models is also suitable for a prototype demonstration, since the intended focus is on the feasibility of verification rather than on tool performance.

In addition to problems from training material, the problem of the dining philosophers — which is well-known within verification and concurrent programming — is also modeled.

3.1 Configuration of test system

The experiments with verifying the properties of modeled problems are conducted on a test system running a 64-bit GNU/Linux operating system, using version 5.1.7 of the Spin model checker. Each experiment is restricted to using a maximum of 3200 MB of memory, and has a maximum time limit set to two hours.

CHAPTER 4

Description of modeling environment

The department develops software using the modeling language UML-RT, supported by modeling tools such as RoseRT and RSARTE. RoseRT is the tool in current use at the department but it is in the process of being replaced by RSARTE. This chapter gives a historic perspective on UML-RT and provides an introduction to modeling constructs that are important in UML-RT and that should be supported by a verification tool.

4.1 Historic context

To give a better understanding of UML-RT, and how it relates to the tools evaluated in Chapter 5, we present a brief historical overview of some of the standards, concepts and tools that have influenced UML-RT and that have been used to model reactive systems[1] over the past two decades.

4.1.1 Modeling reactive systems

Statecharts were introduced by Harel in the 1980s as a visual formalism for specifying complex reactive systems, such as, "telephones, automobiles, communication networks, computer operating systems, missile and avionics systems, and the man-machine interface of many kinds of ordinary software." (Harel 1987)

Statecharts form an extension to *state diagrams* that can be used to represent Finite State Machines (FSMs), that were in turn already being used to describe reactive components. Harel's work allowed specification of systems that were larger compared to those that could be conveniently described using FSMs. This was achieved by the introduction of, e.g., *hierarchy* and *concurrency* or

[1]A component which performs a fresh computation for each invocation is called *transformational*. A component which may rely on prior computations, in addition to new values, to perform a practically continuous computation is called *reactive* (Drusinsky 2006). For such components there is some notion of memory and they may therefore be called stateful, whereas the transformational components are stateless.

orthogonality (Drusinsky 2006). Classic FSMs are flat and sequential and for these reasons they do not scale well to larger systems. This limitation was reduced by the introduction of Harel's extensions (Drusinsky 2006).

The next modeling formalism was Real-Time Object-Oriented Methodology (ROOM) and its ROOM charts, a modified variant[2] of Harel's statecharts. ROOM was supported by the ObjecTime Developer (OTD) tool, developed by the Canadian company ObjecTime Limited.[3]

The ROOM language introduced the *actor* as a primary element (Selic 1996). The actor concept has propagated through the evolution of languages and remains in UML-RT, where it is referred to as a *capsule*. An actor is a concurrent object that communicates with its environment through interfaces known as ports. The ports are instances of protocol classes that define the message passing communication between actors. Figure 4.1 shows a ROOM example with two Client actors connected via ports to a FileSystem actor.

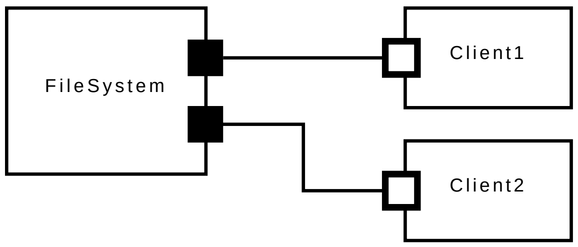


Figure 4.1: Example of actors in ROOM. Figure adapted from Selic (1996, p. 215).

The behavior of an actor is completely defined by a ROOM chart and the hierarchical modeling made possible by ROOM charts permits a gradual refinement of complex behavior (Selic 1996). Actors, or capsules, ports and protocols are described in more detail in Section 4.2.

4.2 Modeling constructs in UML-RT

This section gives an overview of some of the constructs that have been inherited by UML-RT from its ancestors. The components covered are *capsules*, *ports*, *protocols*, and *state machines*.

A *capsule* object corresponds to a logical execution thread and is defined by its *structure* and its *behavior*. The structure of the capsule describes its relation to other objects in the system and the behavior describes how the capsule reacts to its environment. A capsule has precisely one *state machine* that defines its behavior, but may contain any number of sub-capsules, referred to as *capsule roles*, and any number of connections to other capsules in its environment. The behavior and internal structure are completely contained

[2]For example, ROOM charts do not support concurrent states, as a result of a trade-off decision between modeling power and code generation efficiency (Selic 1996).

[3]In the year 2000, the Rational Software Corporation acquired ObjecTime, after which their products Rational Rose and OTD, respectively, where merged into the tool RoseRT. The Rational Software Corporation was in turn purchased by IBM in 2003 and RoseRT became part of IBM's product portfolio.

within a capsule, allowing other objects in the system to view the capsule as a black box.

Capsules communicate exclusively through message passing. Messages that are delivered to the capsule (by a run-time service library) will be received by its structure and are then processed according to its behavior. The *run-to-completion semantics* of capsules ensure that no more than one message at a time is delivered from the capsule's structure to its behavior. "When the capsule receives a message, a transition chain is triggered. The entire transition chain must be executed before the run-time service library delivers the next message." (Rat 2003)

The messages sent and received by capsules are defined by sets of signals grouped into *protocols*. A signal has a name, a direction (in or out) and can optionally be associated with a payload that is delivered in the message along with the signal.

The interface for communication with a capsule is called a *port*. A *public port* is an interface between a capsule and its environment, and a *private port* is an interface between a capsule and its capsule roles. Ports are associated with protocols, ensuring that only specific signals can be sent and received by the capsule using that particular port. The association also restricts how capsules can be connected to each other, e.g., by requiring that one of the connected ports is *conjugated* so that the out signals sent by one capsule correspond to the in signals received by the other, and vice versa.

A state machine defines the behavior of a capsule by describing how the capsule responds to stimuli, i.e., signals sent from other capsules. Signal reception may trigger a transition from one state to another, which in turn causes a sequence of *actions* to be executed. Each state may define entry and exit actions and each transition may define a transition action. In a situation where a signal received in state a triggers a transition t to state b, the exit action of a will be executed, followed by the transition action of t, followed by the entry action of b.

4.3 UML-RT tools at the department

RoseRT is the currently used tool at the department, but migration to RSARTE is in progress. The RSARTE tool is built on top of the Eclipse platform, which originated as a development platform at IBM's subsidiary Object Technology International (OTI) in 1998 (Cernosek 2005). Eclipse has since been released as an open source project. In 2004, IBM announced that several of its products such as Rational Software Modeler and Rational Software Architect would be built on top of the Eclipse platform, under the name IBM Rational Software Development Platform (Cernosek 2005).

RoseRT and RSARTE are both used for modeling in UML-RT. Even though both tools are based on Unified Modeling Language (UML), there are differences between the two. UML-RT models created in RoseRT are based on the 1.4 standard of UML, with custom extensions to provide the constructs that have been inherited from ROOM, e.g., capsules. The current version of RSARTE uses UML version 2.1 with a UML profile called *UMLRealTime*, which provides the necessary constructs.

Differences between model representations means that models are not in practice trivially interchangeable between different tools. These compatibility issues also become clear in Chapter 5, wherein existing tools for software model verification are evaluated in relation to the UML-RT tools used at the department.

CHAPTER 5

Previous work and tools for software model verification

Previous work exists in the area of property verification of UML models and this chapter presents a selection[1] of such research projects and tools. Each tool is presented with a short description of important features and an assessment of the tool's potential for use at the department. The assessment is based on compatibility with the modeling tools and modeling language at the department, and in some cases the availability of the tool.

5.1 vUML

The vUML tool, for automated property verification of state machines in UML models, is presented in (Lilius & Porres Paltor 1999a) and (Lilius & Porres Paltor 1999b). The tool translates a given UML model into a Promela model for use in the Spin model checker. Feedback to the user is given in the form of sequence diagrams generated from the error trails produced by Spin. The process of model transformation, verification and interpretation of error trails is fully automated, which frees the user from having to know Promela or interact with Spin directly.

vUML provides support for automatically verifying a set of pre-defined properties. These properties are specified by assigning special meaning to certain states of the state machine, by marking those states with labels. An example of such a label is the *invalid* label, which signifies that the marked state should never be reached. The labels are recognized in the transformation to Promela, automatically creating a verification model that includes these properties.

vUML lacks support for user specified properties, which may be limiting if properties beyond the default set are of interest. Moreover, the input language to vUML is non-standard (Lilius & Porres Paltor 1999b, p. 12) and vUML no longer appears to be distributed.

[1]In addition to the tools discussed in more detail in this chapter, previous work has also been presented by, e.g., Mikk, Lakhnech, Siegel & Holzmann (1998), Shen, Compton & Huggins (2002) and Jussila, Dubrovin, Junttila, Latvala & Porres (2006).

5.2 Hugo

The Hugo[2] tool (Schäfer, Knapp & Merz 2001, Knapp & Wuttke 2007) allows transformation of UML models diagrams, such as state machines and collaboration diagrams, into the modeling languages of several different model checkers, including Spin. The main purpose of Hugo is to "to verify whether certain specified collaborations are indeed feasible for a set of UML state machines."(Schäfer et al. 2001, p. 9) This is achieved by producing a Promela model from the state machines and by making the claim to Spin that the behavior described by the collaboration diagram is *impossible*. Spin then attempts to refute this claim and will, if successful, produce an "error" trail demonstrating that the collaboration is in fact possible.

As in the case of vUML, Hugo relieves the user from direct interaction with the model checker and from using LTL for specifying properties. Unfortunately, Hugo does not appear to provide necessary support for UML-RT specific constructs, such as capsules.

5.3 VIP and v-Promela

The Visual Interface to Promela (VIP) (Kamel & Leue 2000) is a tool for creating models visually and verifying properties using the Spin model checker as a back-end. The models created in VIP are based on v-Promela (Leue & Holzmann 1999), which is a modeling language that resembles UML-RT in several aspects, e.g., by supporting constructs such as capsules, ports and protocols, very similar to those that are found UML-RT.

The model transformation into Promela is automatic, but the verification process requires direct interaction with Spin and no information from the verification results is fed back into VIP. The exclusive use of v-Promela models in VIP also means that manipulation or verification of models created at the department is not immediately accessible. Nonetheless, the Promela code produced by VIP is interesting from a perspective of demonstrating principles for how modeling of state machines, protocols and interaction between capsules can be achieved in Promela.

5.4 TABU

The work of Beato, Barrio-Solórzano, Cuesta & de la Fuente (2005) presents the Tool for the Active Behaviour of UML (TABU), an automatic verification tool for UML. The tool accepts UML models stored in the XML Metadata Interchange (XMI) format and performs an automated transformation into the modeling language of the SMV model checker. An interesting feature of TABU is that the tool provides a property writing assistant based the work of Dwyer, Avrunin & Corbett (1999), that aids the user in specifying model properties in temporal logic.

TABU does not appear to provide support for necessary UML-RT constructs.

[2]Hugo has been released in several versions and is now called Hugo/RT. It is available at `http://www.pst.ifi.lmu.de/projekte/hugo/`.

5.5 SMARRT

The Static Model checking and Analysis for Rose RealTime (SMARRT) tool, presented in (Elamkulam et al. 2006), permits verification of properties for UML-RT models by integrating the modeling tool RoseRT with IBM's model checker RuleBase.[3] SMARRT is capable of automatically translating a UML-RT model into Property Specification Language (PSL), a very expressive language understood by RuleBase. It also provides functionality that aids the user in specifying model properties by using a variant of UML sequence diagrams. Verification results are presented in the form of sequence diagrams within RoseRT, making error interpretation simpler for the user.

5.6 Summary and conclusions of review

The SMARRT tool is closely integrated with the development tool already in use at the department but further evaluation of SMARRT and its potential for use in the future RSARTE environment is abandoned in this work, since the tool is not publicly available. The VIP tool supports several constructs similar to the ones available in UML-RT and is closely integrated with the Spin model checker. However, the tool is too restricted in its support of input models. TABU provides an interesting property assistant but does not support capsules. Neither does Hugo, although it demonstrates an interesting method for proving the feasibility of certain state machine behaviors. The vUML tool, finally, is not used for further evaluation since it appears to have become unavailable. Nonetheless, it does feature an interesting method for verifying a certain set of properties by extending an input model with special state labels.

The majority of the examined tools target some subset of UML or UML-RT models. No available tool has been found that provides sufficient support for important modeling constructs such as capsules, combined with being compatible with either RoseRT or RSARTE. The review suggests that a general approach for verifying properties of the types of models that are of interest, is to extract a verification model from the original model and to use a model checking tool to perform the verification. This corresponds to the general model checking workflow, as outlined in Figure 2.1.

Although the conclusion is that none of the reviewed tools can be integrated and used directly at the department, they highlight important concepts and interesting features that should be considered in an implementation of a prototype tool for property verification of UML-RT models. Chapter 6 describes such a prototype tool, based on the findings of this review. Spin is the model checker which dominates in reviewed previous work and it is also widely used in software model checking. It is therefore used as the back-end of the prototype tool.

[3]http://www.haifa.ibm.com/projects/verification/RB_Homepage/

CHAPTER 6

Prototype system integrating RSARTE with Spin

This chapter presents design decisions for the prototype tool and outlines its implementation. The prototype tool is intended to integrate Spin with the modeling environment RSARTE, used at the department.

6.1 Verification model extraction options

The *pre-processor* step in Figure 2.1 corresponds to extraction of a verification model from the original model. Since the proof of concept prototype tool of this thesis is implemented using Spin as the model checking back-end, the verification model must be specified in Promela based on the original model from RSARTE. The extraction can be performed in several ways, each with advantages and disadvantages:

Manual translation A verification model can be extracted by manual implementation in Promela. Choosing this option eliminates the need to implement a transformation tool, but would be very impractical for targeting anything beyond a small number of simple capsules.

There is also a risk of losing or overlooking information when the verification model must be manually extracted and kept synchronized with a possibly changing original model. As a consequence, this option does not sufficiently consider the viewpoints of a designer and is therefore discarded.

Re-modeling The VIP tool supports visual modeling constructs similar to those of RSARTE. Using VIP, it is possible to model an equivalent or more abstract version of the original model, and then rely on VIP's ability to export the new model to Promela.

Compared with the previous option, the likelihood of losing information can be considered slightly lower, but the impracticalities of manually maintaining two versions of the same model disqualifies also this option.

Parsing generated C++ code The department uses code generation to produce compilable C++ code from models and Holzmann & Smith (1999) describes work done to allow automatic extraction of verification models from source code. Generating source code from a model results in a loss of abstraction, which is both unnecessary and undesirable when producing a verification model. Furthermore, verification of code is beyond the scope of this thesis and for these reasons the option is not considered further.

Parsing an exported model RSARTE allows models to be exported in XMI format. An external tool capable of parsing Extensible Markup Language (XML) could then be used to traverse the XMI file structure and generate a verification model based on the structure of the original mode. This option could spare the designer much unnecessary work required for maintaining two model versions but does not provide full integration with RSARTE.

Code generation RSARTE is built on the Eclipse framework and can as such be extended with tools and plug-ins for Eclipse, e.g., tools for model transformation and code generation. This option provides close integration with RSARTE and a high degree of automation, which is desirable from the viewpoint of a designer.

Code generation is selected for extracting a verification model in the prototype tool, since it automates much of the necessary work. The Eclipse platform's *Model to Text*[1] project provides support for transforming models into textual artifacts. The Java Emitter Template (JET)[2] tool is part of this project and can be used to generate source code in any language and is therefore used for code generation in the prototype tool. The resulting tool chain for model verification is illustrated in Figure 6.1.

The extraction of a verification model is performed within RSARTE using JET and the resulting Promela model is handed over to the Spin model checking workflow. The verifier, generated from the verification model and optional LTL property, produces verification statistics and an error trail if a property or assertion violation is detected. Holzmann (2003, pp. 245–246) provides more detailed information about Spin's verification process.

6.1.1 Code generation in RSARTE using JET

JET uses a *template* system similar to Java ServerPages to describe the transformation from model to text files. The template system allows textual artifacts to be produced by mixing plain text with dynamic content extracted from the model using control tags. For example, the tags allow iteration over model elements, conditional branching and retrieval of model information. A collection of templates are combined to form a JET transformation project and can be applied to a model, producing text files where the dynamic content of the templates is expanded with information from the model (Ackerman, Elder, Busch, Lopez-Mancisidor, Kimura & Balaji 2008, p. 474).

JET uses model loaders that allow different kinds of models to be manipulated and navigated in a transformation project. The two model loaders bundled with JET do not provide sufficient support for loading models created in

[1]`http://www.eclipse.org/modeling/m2t/`
[2]`http://www.eclipse.org/modeling/m2t/?project=jet#jet`

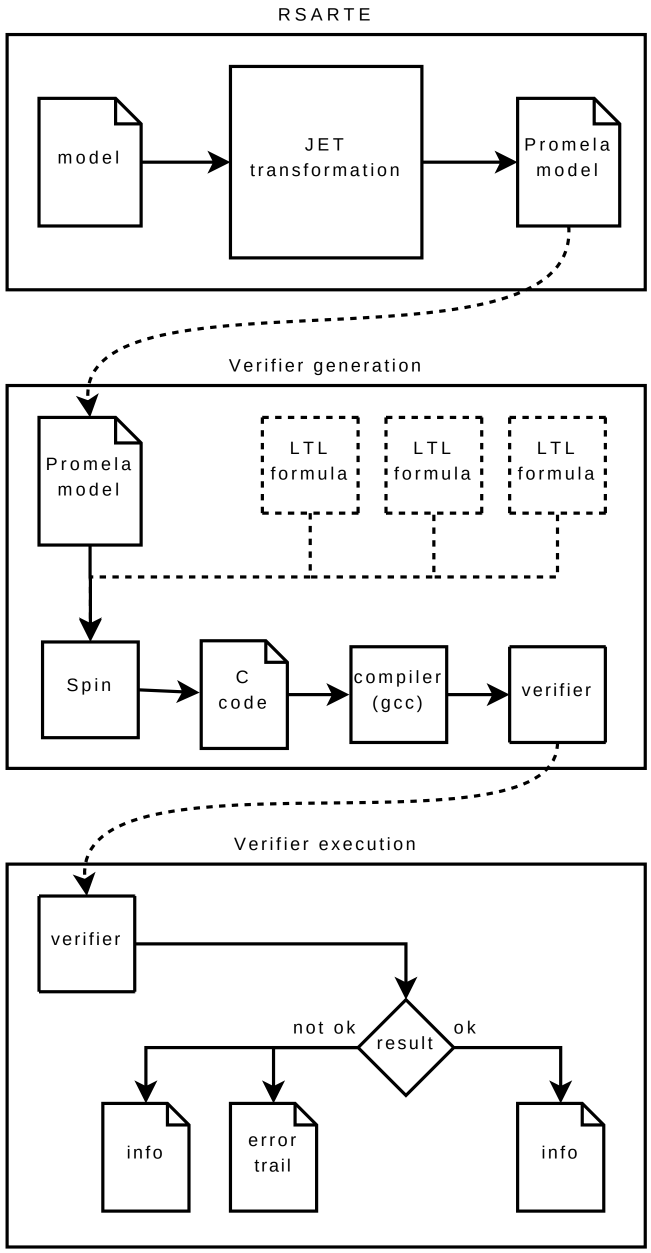


Figure 6.1: Illustration of the verification procedure in the prototype system.

RSARTE, and the prototype tool therefore uses a customized model loader that is not part of JET by default.[3]

Once a model is loaded, JET allows navigation of the model structure using XML Path Language (XPath)[4] expressions. Information from the model, such as the names of elements, is also retrieved and inserted into the text artifacts using XPath expressions, as the template is expanded (Ackerman et al. 2008, p. 490).

6.2 Verification model overview

The prototype tool transforms elements from the model into a verification model in Promela. This section outlines choices made for closing the generated verification model and for transforming concepts of UML-RT to Promela.

6.2.1 Modeling capsule interaction with the environment

A capsule can use public ports to interact with its environment. Spin requires that a verification model "must always contain *all* the information that could possibly be required to verify its properties." (Holzmann 2003, p. 68) Therefore, if the top-level capsule in the model has public ports and is expected to interact with its environment, then the environment of the capsule must also be included in the verification model.

A capsule's only interface for interaction is its ports. The internals of a capsule is hidden from the environment's point of view and, conversely, the capsule's view of its environment is restricted to the reception and transmission of signals. Modeling a capsule's environment from the capsule's point of view can therefore be done by sending signals to the capsule's ports, and by receiving signals sent from the capsule's ports. An illustration is found in Figure 6.2, where components modeling the environment are connected to a capsule with three ports.

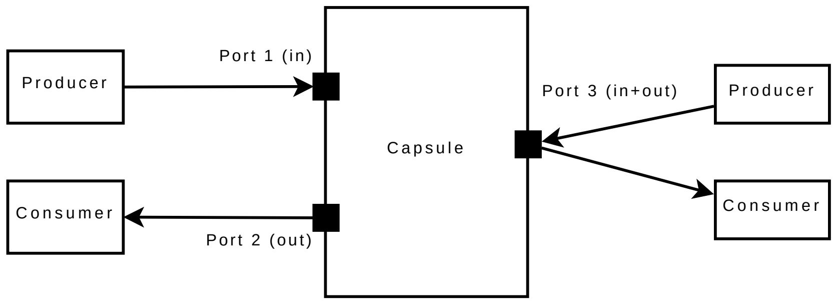


Figure 6.2: Illustration of a capsule that has public ports. Producer and consumer components are attached to the capsule in the verification model to model the behavior of the environment and close the verification model.

It is possible that the behavior of the environment is somehow restricted. However, if no restrictions are imposed on the signal sequences that can be sent

[3]The customized model loader has been provided by Paul Elder, IBM.
[4]`http://www.w3.org/TR/xpath`

to the system (by ensuring that the choice of signals is non-deterministic) then *any* behavior of the environment is in effect modeled, including what would be controlled and intentional signal sequences. One benefit of modeling the environment in this way is that erratic and unexpected behavior of the environment can be precisely that which causes failure in a real system (Holzmann 2003, p. 5).

6.2.2 Mapping concepts in UML-RT to Promela

The structure of the Promela code is based to a large extent on the structure of verification models generated using the VIP tool. This section explains the more important points of the mapping between RSARTE models and the verification models. Section 8.4 contains a description of a UML-RT model created in RSARTE. The state machine of the only capsule in the model is found in Figure 8.13(a). The Promela code generated to verify that model is available in its entirety in Appendix A and is referred to for comparison throughout the remainder of this section.

The prototype system does not provide a complete mapping of all available UML-RT constructs. The recognized subset excludes, for example, hierarchical states, payloads carried by signals, and multiplicity values for model elements exceeding one. Moreover, no guard conditions in state machines are recognized, and neither are pseudostates (such as choice points) except the initial state.

Capsules

- Each capsule is modeled as a Promela process. The example verification model contains one such process called *DemonstrationCapsule* (line 63).
- Each process uses the `run` operator to start the processes that correspond to its capsule roles. The *DemonstrationCapsule* does not contain capsule roles, so it runs no other processes (lines 71–76).
- The top-most capsule is started by the `init` process. The *DemonstrationCapsule* is the top-most capsule and is started at line 257.
- There are no assumptions made on the scheduling of processes or priorities between them.

Protocols and signals

While v-Promela and UML-RT distinguish between *in* and *out* signals in a protocol, the VIP tool does not support this distinction (Kamel & Leue 2000, p. 474). The prototype system does, and generates a Promela model in which this *in* and *out* signals are treated separately.

- Protocols are modeled using a user-defined structured data type (using the `typedef` declarator). The original model in the example contains one protocol called *DemonstrationCommunication* (line 20).
- All protocol signals are modeled with `mtype` declarations. The *DemonstrationCommunication* protocol has three in signals (lines 24–28).
- The *in* and *out* signals of a protocol are separated into two `typedef`s.

Ports and connectors

- A connector joins two capsules and is modeled with Promela channels; one channel for *in* signals and one channel for *out* signals.
- A capsule communicates over one or more ports that are connected to other capsule ports with a connector. The channels used by a process are passed as arguments to the process when it is started. The name, used in the process to refer to the channel, corresponds to the port name to which the connector is connected (line 64).
- Each process creates the channels needed for communication both between its capsule roles, and between the capsule itself and its capsule roles.
- Channels required for communication between the top-most capsule and the environment are created by the `init` process (line 253).

State machines

- Each state in a capsule's state machine is modeled as a labeled, atomically executed code block (e.g., lines 99, 153 and 207).
- Transitions are modeled using `goto` jumps to the label of the destination state.
- A process modeling a capsule is blocked in execution until a signal message is sent to it on one of its channels. For *StateA* in *DemonstrationCapsule*, this corresponds to the `if` block spanning lines 101–150.
- A received signal (line 103) is compared to the signals handled by the state as defined in the UML-RT model and if a match is found, the corresponding transition is taken (lines 106, 117 and 128 for signals *goA*, *goB* and *goC* respectively in *StateA*).
- When a signal is received that triggers a transition to a new state, the exit code for the first state is executed, followed by the transition code and the entry code for the second state. Finally, the jump to the new label is taken (lines 105–114 for the *StateA* having received *goA*). These action chains are obtained in the transformation and corresponding Promela code blocks are generated for each trigger in a state.

6.2.3 Signal producers and consumers

Producer and consumer processes are generated for each protocol, based on the signals that are part of that protocol. A producer process continuously sends signals, nondeterministically chosen from the available signals in the protocol, while a consumer process consumes all protocol signals received. The producer process for the in signals of the *DemonstrationCommunication* protocol is defined on lines 30–47.

Producer and consumer processes can be automatically connected to a top-most capsule that has public ports. Lines 253–258 show how a channel for communication between the *DemonstrationCapsule* and the signal producing process is established.

6.2.4 Embedded Promela code

In the UML-RT models at the department, communication between capsules is performed by sending signals using action code written in C++. The action code can be embedded as an entry or exit action for a state, or as an action in a transition between states. Since the action code in the models is written in the C++ language it cannot be directly transferred to the Promela verification model.

This problem is addressed by separating C++ code from Promela code with pre-processor directives. This allows the C++ code to coexist with the equivalent Promela code directly in the model created in RSARTE. An example of this separation is found in Listing 8.1, which contains action code from the traffic light system model.

6.2.5 Property verification

The LTL formulas used for verifying properties of the Promela models consist of operators (see Section 2.3.3) and propositional symbols. A propositional symbol used in Promela is a "boolean expression that can be evaluated in a single state independently of a computation" (Ben-Ari 2008, p. 72).

As described in Section 6.2.2, a state in the state machine of a capsule in the UML-RT model corresponds to an atomic code block in the Promela verification model. The block is uniquely labeled with the name of the state. To be able to specify properties about a state machine in the original model, it must be possible to specify properties that depend on the equivalent construct in the verification model.

This can be accomplished in Promela using *remote label references*. It is possible to define a propositional symbol by a boolean expression that evaluates to *true* if and only if the location counter of a certain process is currently at precisely such a specific labeled atomic block. Listing A.1 (line 10) shows how a propositional symbol *stateA* is defined using a remote label reference to a label in the process *DemonstrationCapsule*.

The propositional symbols are not defined automatically by the prototype tool and must therefore be added manually to the verification model.

6.3 Verification model options

In addition to the various options available to customize simulation and verification in Spin[5], the verification model generated by the prototype tool can be parameterized by the following macros:

`-DPROMELA` Selects the Promela code which is separated from C++ code in the modeling tool.

`-DXU` In and out signals are sent on different channels which means that a Promela process in practice exclusively reads from certain designated channels and exclusively writes to other channels. The `-DXU` option specifies that this separation is recognized in the verification model, using

[5]See for example Holzmann (2003) or `http://spinroot.com/spin/Man/index.html`

Promela's channel assertions, which improves the performance of the verification (Holzmann 2003, pp. 69-70). Using the option requires that the standard Spin `-DXUSAFE` directive is used when compiling the verifier.

`-DVERBOSE` Specifies that each process should print the name of each state it enters in simulation runs, and that this information should be included in generated message sequence diagrams (visual representations of error trails).

`-DCHANLEN=n` Alters the channel length, used for all communication channels, from the default value of 1 to n.

6.4 JET transformation structure

The JET transformation project is hierarchically organized and the transformation of the main model concepts, such as capsules and protocols, are sectioned into different templates. This allows for a clear project structure and also facilities the process of introducing modifications or extensions to the existing project.

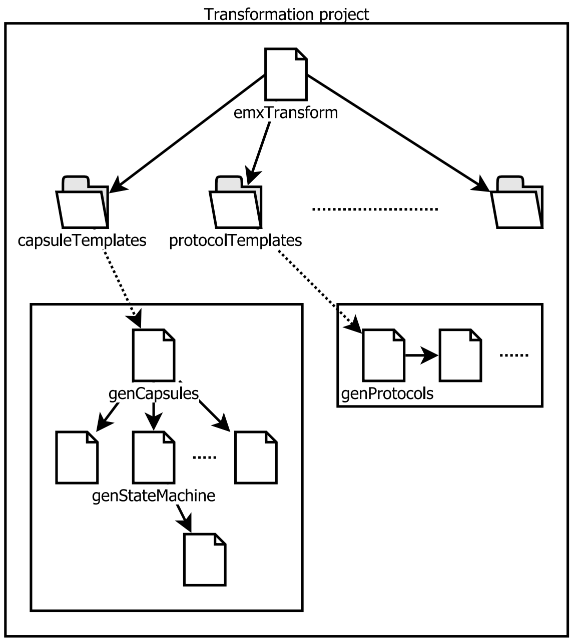


Figure 6.3: Structure of the JET transformation project.

Figure 6.3 illustrates the general structure of the transformation project. The *emxTransform* template is the main project template and includes other template files, such as the *genProtocols* template which transforms model protocols into Promela equivalents.

Appendix B shows an excerpt of the transformation project and contains two template files: the main project template *emxTransform* (see Listing B.1), and the *genStatemachine* template (see Listing B.2).

CHAPTER 7

Property specification

The Promela model produced by the transformation is uninteresting, unless it can be shown that the verification model does or does not satisfy properties that pertain to the original UML-RT model. Correctness properties can be specified directly in the Promela model using, e.g., assertions or labels, and they can also be generated from LTL formulas (Holzmann 2003, p. 75). In previous work (see Chapter 5), several approaches are presented for achieving verification of model properties.

Temporal properties can be classified in several ways; such as Lamport's (1977) separation between *safety* and *liveness* properties,[1] or Manna & Pnueli's (1992) classification, which is based on the structure of LTL formulas. In this chapter we describe and categorize properties on the basis of how specification and verification can be achieved in the prototype system.

7.1 Internally specified properties

The properties in the first category share the feature that they are not specified outside[2] the generated verification model, but are instead expressed as part of the structure of the Promela code. Since such properties are part of the Promela structure itself, they will be automatically included in any verification of the model.

The objective of the prototype tool is to transform UML-RT capsules into a Promela representation, and this means that certain properties can be identified that should hold for any UML-RT capsule, regardless of other constraints on its behavior. An example of such a property — one that can be considered an error in any UML-RT capsule — is the presence of *deadlocks*.

A deadlock occurs in a situation "in which control is in the middle of the program, yet no part of the program is able to proceed." (Manna & Pnueli 1992,

[1]Safety properties are those that specify that something *will not* happen whereas a liveness properties specify that something *must* happen. (Lamport 1977)

[2]For example by using LTL formulas.

p. 309) The absence of deadlock is checked by default by Spin (Holzmann 2003, p, 75), which means that a standard verification run of a verification model of communicating UML-RT capsules will automatically detect if a deadlock situation between the capsules is possible. If, for example, a UML-RT model contains only two communicating capsules and they are both permitted to enter a state where they wait for a signal from the other party, then the verification model will be able to reach a point where no process can continue, causing a deadlock.

Other properties that are not quite as general as deadlock absence can be automatically reflected in the verification model if particular states in the original model are given a role that can be recognized by Spin. An example of this is the option, available both in vUML (Lilius & Porres Paltor 1999b, pp. 6–7) and VIP (Kamel & Leue 2000, p. 483) but not in the prototype tool[3], to mark certain states with a *progress*[4] label. The use of the label in the UML-RT model can be interpreted thus: entering the marked state signifies that some goal has been accomplished or that useful work is performed by the state machine. This is a liveness property and if the model can exhibit cyclic behavior, wherein no state machine performs useful work, then Spin is able to detect this possibility.

A third example of a correctness properties encoded automatically in the Promela model is discussed in Section 7.3.1.

7.2 Externally specified properties

The second category contains those properties that are specified outside the generated model (for example by writing LTL formulas) in contrast to the correctness properties from the previous category. There are several possible ways of constructing properties that can be used to verify models. We highlight three possible options discussed in previous work.

First, LTL formulas can be specified manually. LTL allows a wide variety of properties to be captured in a notation with a very simple syntax (see Section 2.3.3). Still, formalizing even quite basic requirements in LTL can become challenging (Dwyer et al. 1999).

The second option is therefore to use a tool for guiding a user in specifying properties and aiding the user in writing correct syntax, but more importantly in writing properties that capture the intended meaning of the requirement. This approach is used by the TABU tool (see Section 5.4).

The third option is to avoid direct use of LTL altogether and specify properties in some visual notation. This approach is used by the Hugo tool (see Section 5.2). The property can then be translated from visual notation to an LTL formula.

The prototype tool does not provide an option for external property specification beyond manually written LTL formulas.

[3]The possibility of extending the prototype tool with such an option is discussed in Section 10.3.1.

[4]Both tools utilize the *progress* label (see Holzmann (2003, p. 459)) that already exists in the Promela language.

7.3 Properties of primary interest

The example models in Chapter 8 will be used to demonstrate verification of several properties from the two categories. The two properties that are of *primary* interest to the department are related to the behavior of a single capsule. Informally stated, they are that,

1. a capsule's state machine must, regardless of state, handle every signal that it can receive; and
2. it must be detectable if leaving some state becomes impossible, once it has been entered.

The LTL formula and mechanisms of the Promela model that enable verification of these two properties are described in the following sections.

7.3.1 Signal handling guarantee

Every state of a capsule's state machine is associated with a set of transitions leading to other states and a transition is taken when it is triggered. This triggering occurs when a signal, received at a port of the capsule, is delivered to the behavior of the capsule. If a particular signal can arrive at a particular port and be delivered to the state machine when it is in a certain state which has no outgoing transition triggered by that signal, then this must be detected in the verification model.

Verification of this property is achieved by taking advantage of Promela's assertion statement in the verification model. Section 6.2.2 outlines how signals and transition triggering is modeled in Promela. If no match for a received signal is found then an assertion statement is intentionally violated.

The property becomes part of the verification model itself and is therefore considered to be one of the properties that are internally specified.

7.3.2 Trap detection

It is possible that entering a particular state makes it impossible to leave that state. This can for example be caused by a lack of outgoing transitions. If complete signal handling has been guaranteed by the previous property, then a trap can still exist if the outgoing transitions from the state lead back to the same state.

We let the propositional symbol s_0 denote that the Promela process is at a point in the code corresponding to the capsule's state machine being in the state *State 0*. The property that *State 0* is trapping is then captured by the LTL formula

$$\Box(s_0 \rightarrow \Box s_0) \tag{7.1}$$

Manna & Pnueli (1992, p. 192) refers to such a property (for the symbol p) as "once p, always p."

To show that *State 0* is trapping in all executions, the LTL formula supplied to Spin is the negation of Property 7.1. This negated LTL formula formalizes violations of the original property. In Spin syntax it is written `! [](s0 -> []s0)`.

If Spin is able to show a violation of the original property, then an error trail will be produced that demonstrates how to leave *State 0* once entered. If Spin is unable to show a violation of the original property then *State 0* must in fact be trapping.

7.4 Limitations in property specification

Although LTL is expressive, there are limitations on the kinds of properties that can be stated in the language. One such limitation of LTL is that it does not allow quantification over paths, which means that it cannot be used to express properties that are related to the possible existence of paths. Huth & Ryan (2004, p. 184) gives examples of properties that consequently are impossible to express in LTL, such as:

> "From any state *it is possible* to get to a `restart` state (i.e., there is a path from all states to a state satisfying `restart`)."

Such properties may instead be expressed in other logics that allow quantification, such as CTL. CTL*, which is a superset of both CTL and LTL, allows any formula in the two, and additionally permits formulas that can be expressed in neither CTL nor LTL alone. See Huth & Ryan (2004, pp. 217-221) for examples and a comparison of the expressive powers of these logics.

CHAPTER 8

Model examples

This chapter presents a number of models created with RSARTE together with the properties that have been used in verification. The models have been selected from tool training material (Rat 2003) in addition to Dijkstra's well-known problem of the *dining philosophers*. A model illustrating the problem of verification model complexity is presented, as well as a model intended to illustrate verification of the two properties found to be of primary interest (see Section 7.3).

8.1 A model of a traffic light system

The traffic light system model illustrates concepts of capsule communication using channels, as well as hierarchically constructed capsules (capsules containing capsule roles). The model is implemented in RSARTE using capsules, state machines and a communication protocol. We present three versions of the traffic light system to illustrate how properties can be proven and refuted, leading to modifications of the design. The initial model is based on an exercise from training material for RoseRT (Rat 2003) but differs in certain behavioral characteristics, as discussed later. The traffic light system example also illustrates how design mistakes can be introduced in a system when the design is based solely on sequence diagrams.[1]

8.1.1 Model description

Two roads meet at an intersection and traffic through the intersection is guarded by traffic lights facing in the four directions of the compass. A central traffic light controller sends directives to the traffic lights, signaling which states they should enter. The states of the controller correspond to its view of how the lights facing in the intersecting directions should behave.

[1] A sequence diagram is a graphical notation describing the behavior of a particular system scenario.

The behavior of the controller is cyclic and the transitions between its control states are triggered by reception of timing events.

8.1.2 Properties

For an implementation of a traffic light system to be correct, the following list of properties should, at minimum, be fulfilled:

1. The signal handling guarantee property (see Section 7.3.1) must hold for all states in the model.

2. In an infinite run of the system, all traffic lights should display green infinitely often.

   This liveness property specifies that the system should be starvation free, which is of interest since it proves that a car approaching the intersection will eventually be allowed to pass through it, regardless of when the car arrives or from which direction.

   We let the propositional symbol *northGreen* denote that the *north* traffic light is in state *Green*. The property for the *north* traffic light is expressed in LTL as:

$$\Box\Diamond northGreen \tag{8.1}$$

3. The system may never simultaneously signal green in such a way that traffic can collide.

   This safety property is an example of using mutual exclusion to control access to a shared resource (the part of the intersection where traffic meets) and where a violation of the mutual exclusion property could have severe consequences.

   We let the propositional symbol *northGreen* denote that the *north* traffic light is in state *Green* and analogously for the remaining three traffic lights. The mutual exclusion property for the traffic light system is expressed in LTL as:

$$\Box \neg(\ (northGreen \lor southGreen) \land (eastGreen \lor westGreen)\ ) \tag{8.2}$$

8.1.3 First version

The first implementation is based directly on diagrams from the training material (Rat 2003). The state machines describing the behavior of both the traffic lights and the controller (see Figure 8.1(a) and Figure 8.2(a), respectively) are transferred directly from this material.

Model elements

The model consists of the three capsules *TrafficLight*, *Controller* and *Intersection*, and the protocol *LightControl*.

Listing 8.1: Initial transition effect code for the state machine of the Controller capsule in the first version of the traffic light system.

```
#ifdef PROMELA
    nControl_out ! green;
    sControl_out ! green;
    eControl_out ! red;
    wControl_out ! red;
#else
    myTiming.informIn(RTTimespec(greenTime));
    nControl.green().send();
    sControl.green().send();
    eControl.red().send();
    wControl.red().send();
#endif
```

The behavior of the traffic light capsule is defined by its state machine (see Figure 8.1(a)). The capsule also has a port *control* (see Figure 8.1(b)) through which it receives signals from the controller. Each transition is triggered by a specific signal from the controller capsule, e.g., the transition *goGreen* is triggered by the signal *green*. A traffic light always starts in its *Red* state.

The purpose of the *Controller* capsule is to cycle the traffic lights through their different states according to a given pattern (see Figure 8.2(a)). The signals sent to the traffic lights, using action code in the transitions of the *Controller* capsule state machine, are derived from the message sequence diagram from the training material (see Figure 8.3). Although the sequence diagram only describes the signals sent to the *north* and *west* traffic lights, the action code is extended to include also the *south* traffic light (being sent the same signal as *north*) and the *east* traffic light (being sent the same signal as *west*). The action code of the *Controller*'s transition from its initial state to the state *nsGreen* is found Listing 8.1.

The *Intersection* capsule contains all other capsules in the model and establishes how the *Controller* capsule is connected to each *TrafficLight* capsule. The structure of the *Intersection* is visible in Figure 8.4.

The *LightControl* protocol defines the control signals *green*, *yellow* and *red* that can be sent to the traffic lights.

8.1.4 Second version

The difference between the first model and the second lies in the state machine of the *Controller* capsule. Considering that a traffic light always enters the *Red* state when it is started, it is unnecessary for the *Controller* to send the *red* signal to the two traffic lights *east* and *west*, as Figure 8.3 suggests.

Modifications to model elements

The *Controller* capsule state machine is altered so that it does not send the *red* signal to the *east* and *west* traffic lights in the initial transition. The consequence is that lines 4, 5, 10 and 11 are removed from the transition effect code (see Listing 8.1) in second version.

8.1.5 Third version

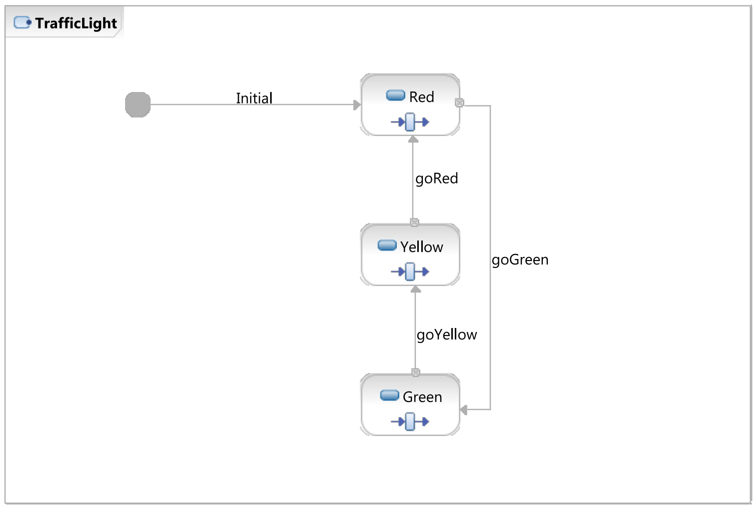
The third version continues to build on the second version and introduces an acknowledgement scheme so that the *Controller* capsule also requires a response for each control signal sent to the traffic lights.

Modifications to model elements

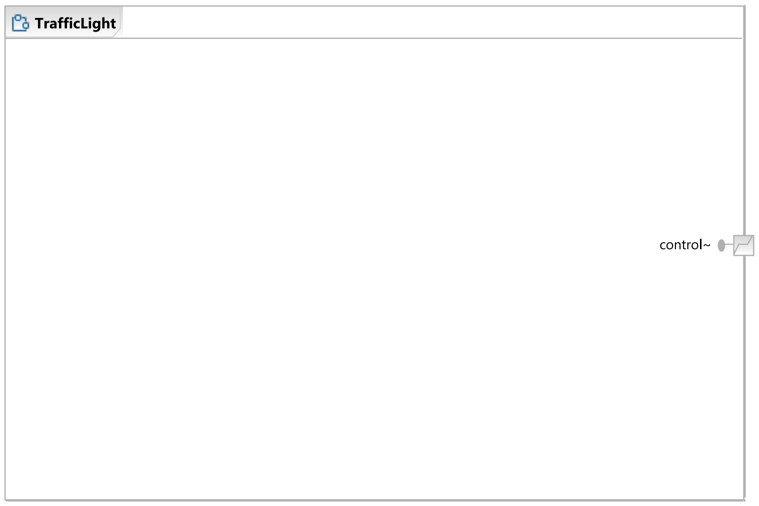
An acknowledgment signal, *ack*, is added to the existing communication protocol *LightControl* and a message with this signal is sent by a *TrafficLight* in each of the transitions *goGreen*, *goYellow* and *goRed*.

The state machine of the *Controller* capsule is modified to require *ack* signals to be received from the traffic lights for each control signal they are sent. This is accomplished by adding intermediary states whose outgoing transitions are triggered on reception of the *ack* signal. The *Controller* is no longer allowed to enter its new control state until it has received *ack* messages from the correct traffic lights.

The prototype tool does not support hierarchical states or transitions triggered by multiple signals, which explains why the new state machine of the *Controller* capsule (see Figure 8.5) becomes quite large.

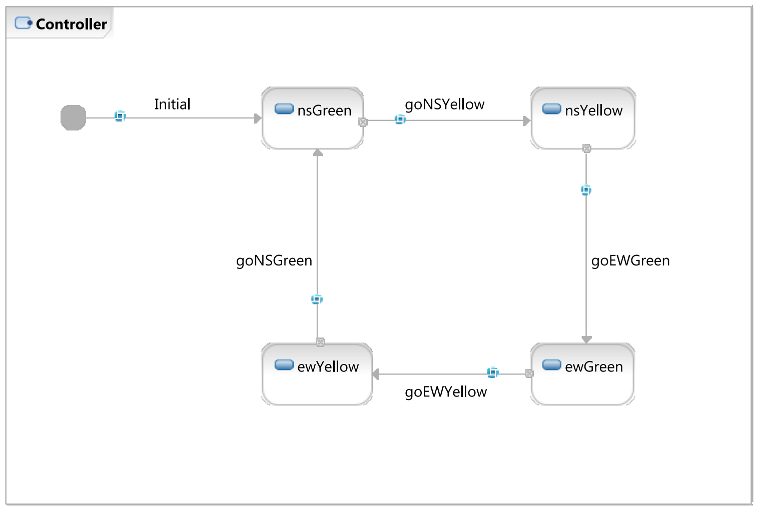


(a) TrafficLight state machine

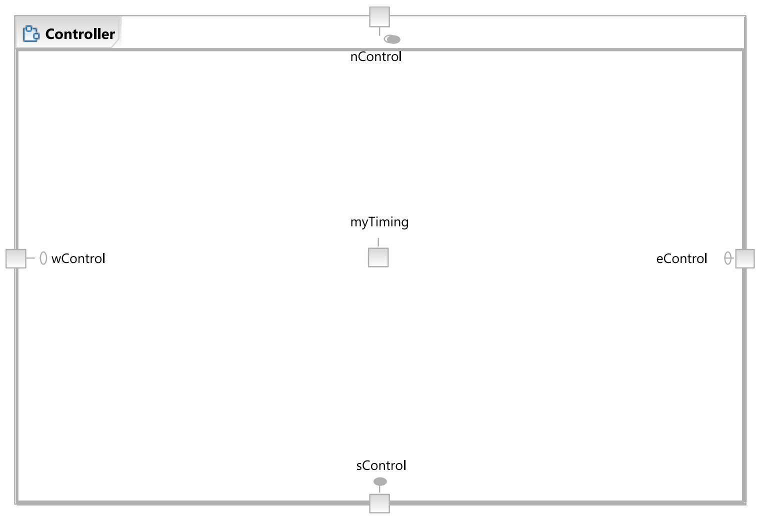


(b) TrafficLight structure diagram

Figure 8.1: State machine diagram and structure diagram of the traffic light capsule.



(a) Controller state machine



(b) Controller structure diagram

Figure 8.2: State machine diagram and structure diagram of the original controller capsule.

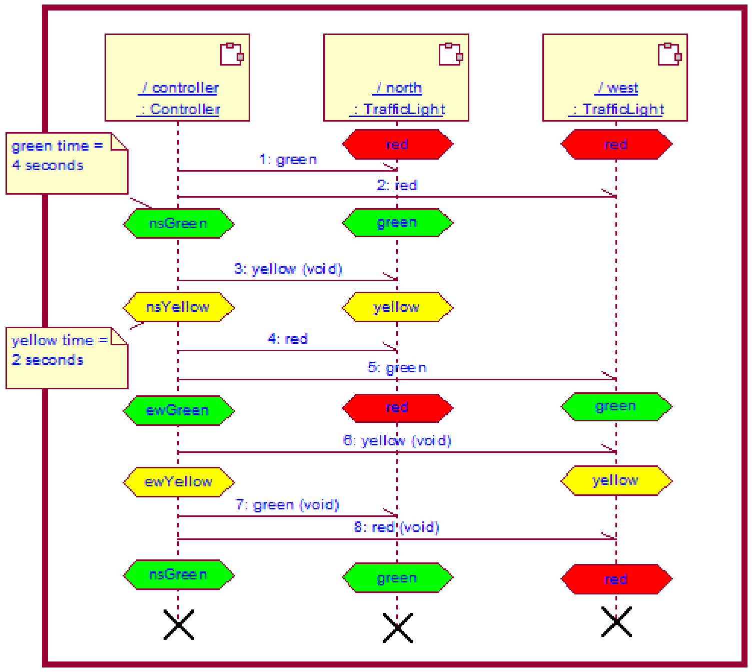


Figure 8.3: Sequence diagram of the traffic light system model illustrating interactions between the controller and two traffic lights. Figure from Rat (2003).

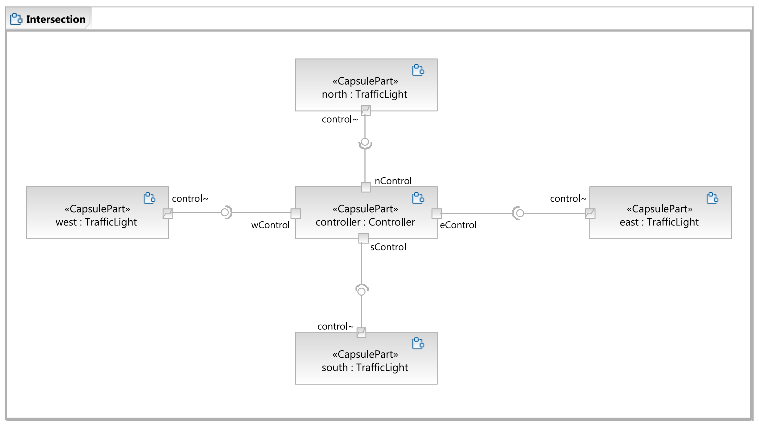

Intersection
«CapsulePart» north : TrafficLight
control~
nControl
«CapsulePart» west : TrafficLight
control~
wControl
«CapsulePart» controller : Controller
eControl
control~
«CapsulePart» east : TrafficLight
sControl
control~
«CapsulePart» south : TrafficLight

Figure 8.4: Structure diagram of the intersection capsule.

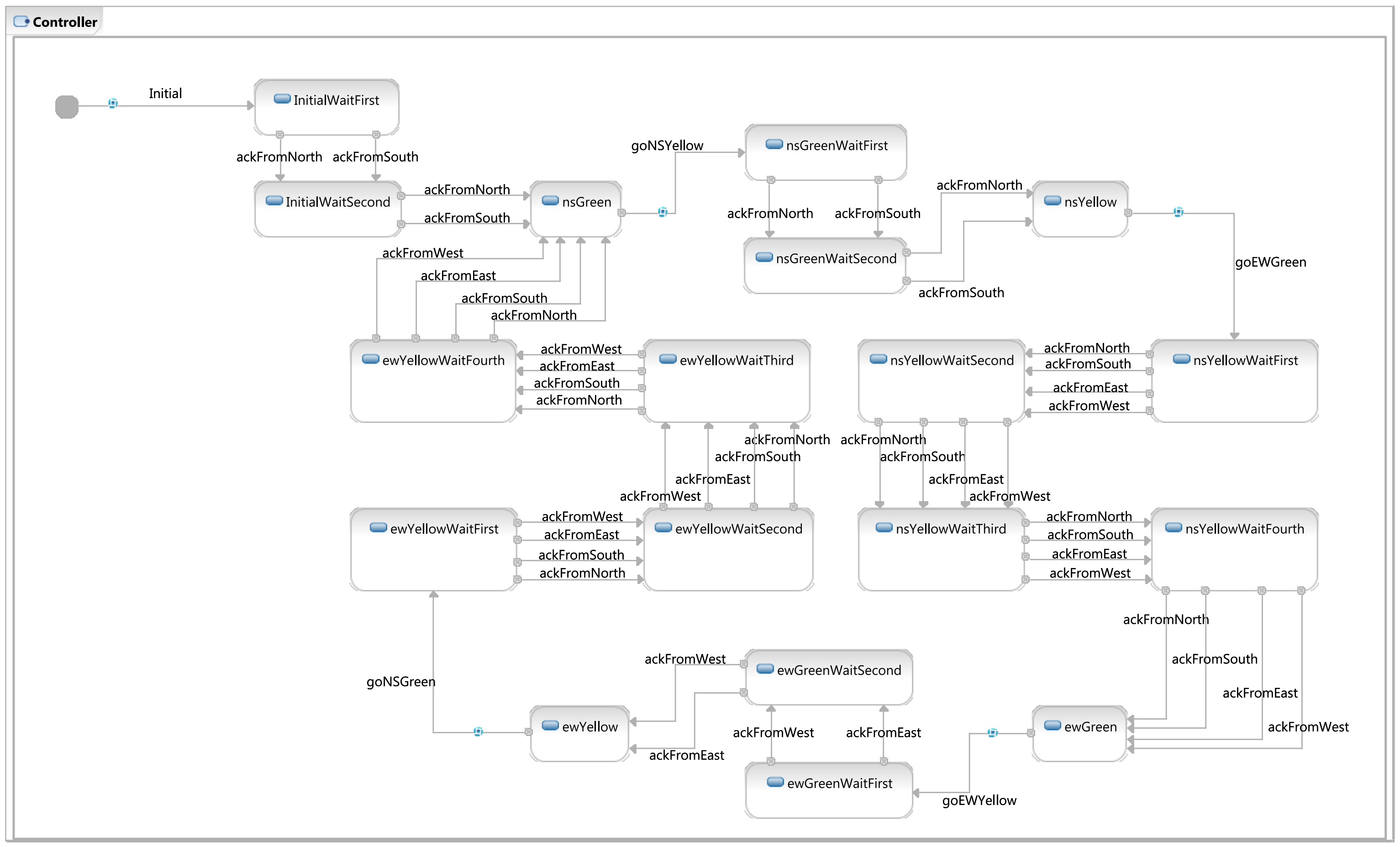


Figure 8.5: Modified state machine of the controller capsule.

Listing 8.2: Promela code for the stimulus producer process of the electronic lock model.

```
/* Producer process for the LockCommunication signals */
proctype LockCommunication_prod (chan com) {
#ifdef XU
xs com;
#endif
send:
   do
      :: true -> send_one: com ! one ;
      :: true -> send_two: com ! two ;
      :: true -> send_question: com ! question ;
      :: true -> send_other: com ! other ;
      :: true -> send_lock: com ! lock ;
   od
}
```

8.2 A model of an electronic lock

Training material for RoseRT (Rat 2003) describes a passive class[2] that models the behavior of a simple electronic combination lock. The model has a state machine that changes state based on input in the form of characters. Entering the correct sequence of characters (`1`, `2`) unlocks the lock, the character `L` locks the lock and the character `?` causes a help message to be printed.

8.2.1 Model description

The prototype tool does not support signals that carry a payload (such as a character), so we implement the model as a capsule that reacts to a set of input signals grouped into a protocol. This is a form of *data type abstraction* (Holzmann 2003, p. 236) where the entire input space (all characters) are reduced to a set of signals. Such a change "can be justified if the correctness properties of a model do not depend on detailed values, but only on the chosen value ranges."(Holzmann 2003, p. 236) The protocol signals are named *one*, *two*, *question*, *lock* and *other*.

Closing the model

The *Lock* capsule has a public port (see Figure 8.6(b)), which means that the verification model must be closed using a stimulus process to simulate nondeterministic behavior of its environment. The generated Promela code modeling the signal producer is available in Listing 8.2.

[2]Unlike an active object, such as a capsule, a passive class does not have its own logical thread of execution. While capsules are restricted to message passing for communication, passive classes communicate by procedure or function invocations.

8.2.2 Properties

A correct implementation of the lock model should satisfy the following properties:

1. The signal guarantee property (see Section 7.3.1) must hold for all states in the model.

2. The trap detection property (see Section 7.3.2) must be refuted for all states in the model.

3. Ordering the lock to lock must always result in the lock eventually being locked, regardless of its current state.

   We define propositional symbols as follows: *snd_lock* denotes that the environment sends the signal *lock*, and *locked* denotes that the lock is currently in the state *Locked*. The property specifying the desired behavior is then expressed in LTL as:

   $$\Box(snd\_lock \rightarrow \Diamond locked) \tag{8.3}$$

   The property used by Spin to describe a violation of this behavior is the negation of Property 8.3.

4. It should be possible to unlock the electronic lock when it is locked.

   We define propositional symbols as follows: *locked* denotes that the lock currently is in state *Locked*, and *unlocked* denotes that that the lock is in state *Unlocked*. We demonstrate that this behavior is feasible by intentionally claiming that it is impossible, and using Spin to produce the counterexample (Schäfer et al. 2001). We make the intentionally incorrect claim that: it is always the case that when the lock is locked, it will *not* eventually become unlocked.

   This property is expressed in LTL as:

   $$\Box(locked \rightarrow \neg(\Diamond unlocked)) \tag{8.4}$$

   If Spin can refute this claim, then a trail will be produced demonstrating how to unlock the locked lock.

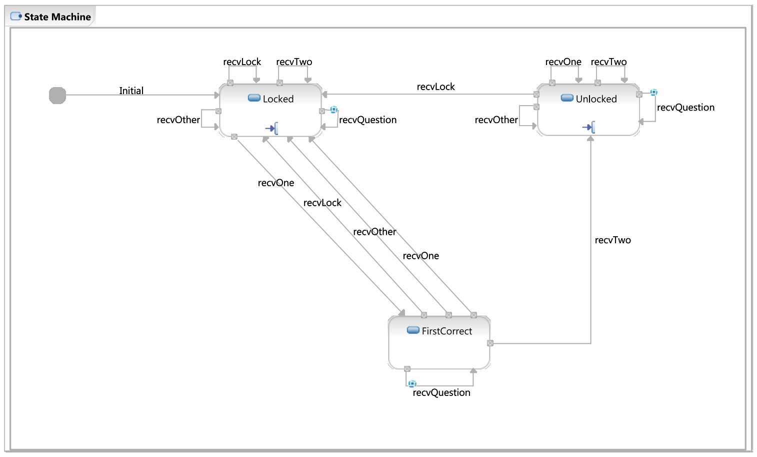
8.2.3 First version

Model elements

The model consists of the capsule *Lock* and the protocol *LockCommunication*. During the transformation process into Promela the verification model is extended with the stimulus process whose task is to provide the lock with input signals.

The *Lock* capsule has a state machine consisting of three states (see Figure 8.6(a)). The states signify the lock being either locked, in the state when the first correct signal of the unlocking sequence has been received, or unlocked. The *Lock* capsule receives input through the port *lockCom* (see Figure 8.6(b)) and is always initialized in the *Locked* state.

The *LockCommunication* protocol defines the set of abstract signals that can be sent to the electronic lock.



(a) Lock state machine

Lock
lockCom

(b) Lock structure diagram

Figure 8.6: State machine diagram and structure diagram of the lock capsule.

8.2.4 Second version

The faulty model of the electronic lock is identical to the first model version except for an intentional error introduced in the state machine of the *Lock* capsule. The purpose of the second model is to demonstrate how such an error in the model can be detected in the verification model.

Modifications to model elements

The *Lock* capsule's state machine is modified (see Figure 8.7) by removing the transition triggered by the signal *other* in the *Unlocked* state.

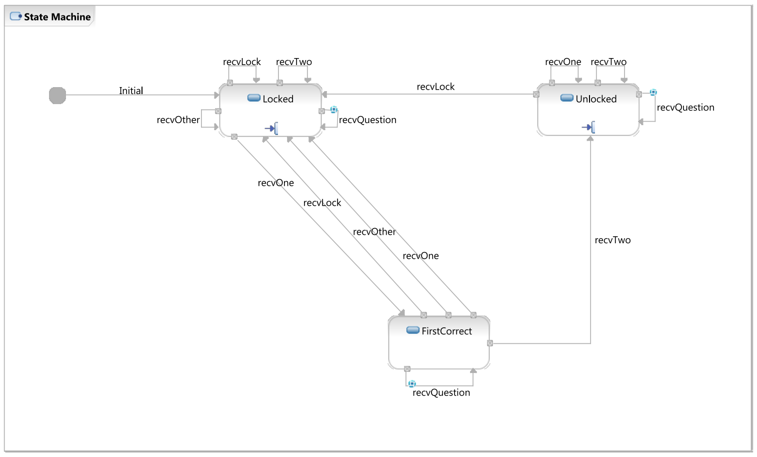


Figure 8.7: Modified state machine diagram of the lock capsule.

8.3 The dining philosophers

The problem of the dining philosophers (Dijkstra 1977) is a commonly used illustration of how deadlocks can occur in the context of concurrently operating computer systems. We model the problem using capsules, state machines and communication channels in RSARTE, and transform it into a verification model in Promela.

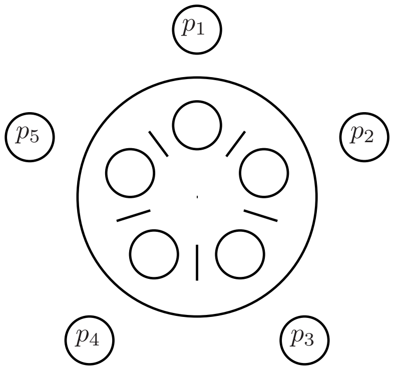


Figure 8.8: Illustration of the problem of the dining philosophers, with five philosophers (labeled $p_1, \ldots, p_5$) that sit around a table, sharing five chopsticks.

8.3.1 Problem scenario

Five philosophers each sit with a bowl of food around a circular table and spend their lives eating and thinking, without communicating with each other. Between each philosopher pair lies one chopstick (see Figure 8.8) and to be able to eat, a philosopher must use the two chopsticks that are shared with the philosopher on either side.

One possible scheme for each philosopher to follow is described by Algorithm 1. However, the consequence of using this scheme is that an error situation can occur if the philosophers time their actions such that they all pick up their left chopstick and then all turn their attention to the right chopstick, which is then forever unavailable. This can happen since the scheme introduces a situation that fulfills all of the four necessary conditions (Coffman, Elphick & Shoshani 1971) for a potential deadlock:

Mutual exclusion condition Every chopstick is shared by two philosophers.

Wait for condition After obtaining one chopstick the philosopher must obtain yet another one.

No preemption condition The philosophers do not communicate and cannot forcibly take chopsticks from each other.

Circular wait condition The philosophers are arranged around a table so that each philosopher can hold a chopstick that is required by a neighbor.

Algorithm 1: Naïve procedure for acquiring and releasing resources in the problem of the dining philosophers.

```
while true do
    begin Thinking
        Wait for some time
    end
    begin Hungry
        Request(left chopstick)
        Request(right chopstick)
        begin Eating
            Wait for some time
        end
        Release(left chopstick)
        Release(right chopstick)
    end
```

Consequently, the problem of the dining philosophers is to establishing a scheme or protocol for the philosophers to follow that is both deadlock free and starvation free (ensuring that all philosopher eventually are permitted to eat). There are several possible solutions to the problem.

One solution is to introduce a butler at the table that regulates how many philosophers can attempt to acquire chopsticks at the same time. The deadlock situation can then be avoided by having the philosophers stand up (they cannot obtain chopsticks or eat standing up) and requiring the philosophers to be granted permission from the butler before being seated at the table. After finishing a meal, a philosopher will stand up again and notify the butler. The butler will not allow all of the philosophers to be seated at once; one chair is always kept empty, which breaks the circular wait condition.

8.3.2 Model description

We present an RSARTE model of the problem based on the butler solution. The model has only four philosophers instead of the usual five, since a model with five philosophers produces a verification model that exhausts the resources of the test machine (see Section 3.1). The reduced number of philosophers does not affect the principle behind the solution.

8.3.3 Properties

The following properties should hold in a correct solution to the problem of the dining philosophers:

1. The signal handling guarantee property (see Section 7.3.1) must hold for all states in the model.
2. If a philosopher is eating, the philosopher's two neighbors cannot also be eating.

   We let the propositional symbol *p1Eating* denote that philosopher one is currently eating, and analogously for philosophers two and three. This *safety* property is expressed in LTL as:

   $$\Box\,(\mathit{p2Eating} \rightarrow \neg(\mathit{p1Eating} \;\vee\; \mathit{p3Eating})) \tag{8.5}$$

   The property used to describe a violation of the property to Spin is the negation of Property 8.5.
3. If a philosopher is eating, then the chopsticks lying immediately to the left and right must be in use.

   We define propositional symbols as follows: *c2TakenRight* denotes that chopstick two is acquired by the philosopher to its right and *c2TakenRight-ReqLeft* denotes that chopstick two is both acquired by the philosopher to its right and requested by the philosopher to its left (see Figure 8.9 and Figure 8.12(a)).

   The property is expressed in LTL as:

   $$\begin{aligned}\Box\,(\mathit{p2Eating} \rightarrow ((&\mathit{c1TakenLeft} \;\vee\; \mathit{c1TakenLeftReqRight}) \\ &\wedge (\mathit{c2TakenRight} \;\vee\; \mathit{c2TakenRightReqLeft})))\end{aligned} \tag{8.6}$$

   The property used to describe a violation of the property to Spin is the negation of Property 8.6.
4. In an infinite run of the model, a philosopher must be allowed to eat infinitely often. This property specifies that model is starvation free and must hold for all philosophers.[3]

   The property is expressed in LTL as:

   $$\Box\,\Diamond\,\mathit{p2Eating} \tag{8.7}$$

   The property used to describe a violation of the property to Spin is the negation of Property 8.7.

[3]The property does not hold for the model, as will be demonstrated and explained in Section 9.3.

8.3.4 Model implementation

Model elements

The model consists of the four capsules *Table*, *Butler*, *Philosopher*, and *Chopstick*; and the protocol *ResourceAllocation*.

The *Table* capsule has no purpose other than to contain capsule roles and to define the relationships and communication channels between the capsule roles (see Figure 8.9).

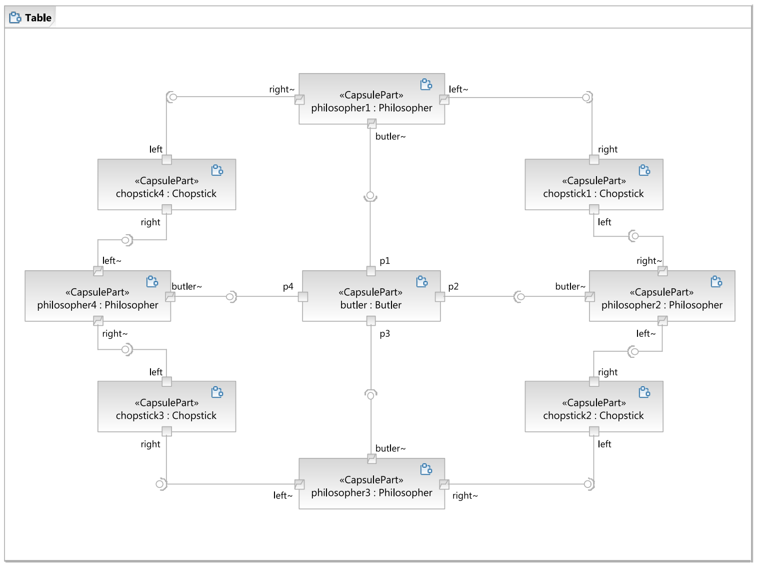


Table
«CapsulePart» philosopher1 : Philosopher
«CapsulePart» philosopher2 : Philosopher
«CapsulePart» philosopher3 : Philosopher
«CapsulePart» philosopher4 : Philosopher
«CapsulePart» chopstick1 : Chopstick
«CapsulePart» chopstick2 : Chopstick
«CapsulePart» chopstick3 : Chopstick
«CapsulePart» chopstick4 : Chopstick
«CapsulePart» butler : Butler
right~
left~
butler~
left
right
p1
p2
p3
p4

Figure 8.9: Structure diagram of the table capsule.

The behavior of a philosopher is defined by the state machine (see Figure 8.10(a)) of the *Philosopher* capsule. The primary tasks for a philosopher are to think and eat, captured by the states *Thinking* or *Eating*, respectively. Before a philosopher can eat it must first be allowed a place at the table and then acquire two chopsticks. The *Philosopher* capsule has three communication ports: *butler*, through which requests for a seat at the table is made and granted; and *left* and *right*, through which the philosopher requests and is granted access to the chopsticks to the left and right (see Figure 8.10(b)).

The butler determines when a philosopher is allowed a seat. This behavior is defined in the *Butler* capsule's state machine, see Figure 8.11(a). Communication with the philosophers is conducted through four[4] ports, one for each philosopher (see Figure 8.11(b)).

A chopstick can be acquired or returned by a philosopher sitting either to the left or to the right of the chopstick (see Figure 8.12(a)). The signal to grant the philosopher on the *left* access to a chopstick is sent in the transition *leftReq* (because the chopstick was available and then requested by *left*) and the transition *rightRet* (because the chopstick was taken by *right*, requested by *left*

[4]The prototype tool does not support replicated ports, which are possible to use in UML-RT.

and then returned by *right*), and vice versa for the philosopher on the *right*.[5]

The *ResourceAllocation* protocol defines the different signals of the model, used for either requesting or returning a resource and notifying if a request is granted. The signals are *req*, *ret* and *grant*, respectively. The protocol is both used for communication between a philosopher and a chopstick, and for communication between a philosopher and the butler.

8.4 A model with intentional errors

The model described in this section is indented to demonstrate the results of verifying only the two properties that are of primary interest to the department (see Section 7.3). The model is therefore very simple and contains two deliberate errors of the types that must be found during model verification. The Promela model produced for this model is available in Appendix A.

8.4.1 Model description and properties

The model is verified with respect to the following properties:

1. The signal handling guarantee property (see Section 7.3.1) must hold for all states in the model.

2. The trap detection property (see Section 7.3.2) must be refuted for all states in the model.

Model elements

The model consists of a single capsule *DemonstrationCapsule* (see Figure 8.13) and the protocol *DemonstrationCommunication*. In the transformation process into Promela, the model is extended with a stimulus process whose task it is to close the verification model by simulating an indeterministic environment of the *DemonstrationCapsule*.

The demonstration capsule's behavior is defined by a state machine containing three states: *StateA*, *StateB* and *StateC*. The *DemonstrationCommunication* protocol defines the signals that can be sent to the demonstration capsule. Each signal is intended to trigger a transition being taken to the state whose name corresponds to the name of the signal. The signals of the protocol are: *goA*, *goB* and *goC*.

StateA and *StateC* both handle all signals correctly, triggering transitions to the other states upon reception of the signals, whereas *StateB* contains two errors. The transition named *goToA*, triggered by the signal *goA*, is a self-transition leading back to *StateB*. The transition named *goToC* is missing completely from *StateB*.

The consequence is that *StateB* is both a trapping state, in violation of property 2, and fails to handle all signals, in violation of property 1.

[5]It is also possible to sent the signals directly in the entry action of the two states *TakenLeft* and *TakenRight*.

8.5 A complexity experiment

While it is beyond the scope of this thesis to cope with models of the size or complexity as of those used at the department, it is nonetheless of interest to give an idea of how fast a verification model can grow. The complexity of the verification model depends on the complexity and modeling constructs used in the original model. It is necessary to control the complexity of the verification model to avoid exhausting memory resources or exceeding the time available for verification. Holzmann (2003, p. 119) writes "the worst-case computational expense of verifying any type of correctness property with a model checker increases with the number of reachable system states R of a model."

A UML-RT capsule typically has many public ports sending and receiving many possible signals. This communication diversity introduces complexity in the verification model attributable to three factors:

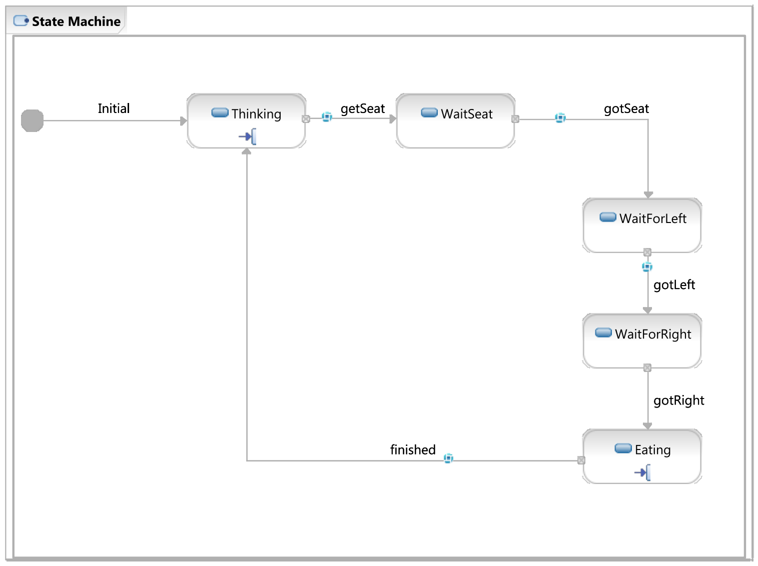
1. The number of Promela channels in the verification model, which corresponds to the number of ports in the original model.
2. The number of different message types in the verification model, which corresponds the number of different signals of the protocols in the original model.
3. The length of the Promela channels. This parameter must be fixed in order to close the verification model.

The complexity introduced by a set of message channels in a verification model can be understood by the number of states that such a set of objects can be in. Holzmann (2003, pp. 119-120) gives the number of states R_Q in terms of the number of channels q, the number of different messages m and the length of the channel s:

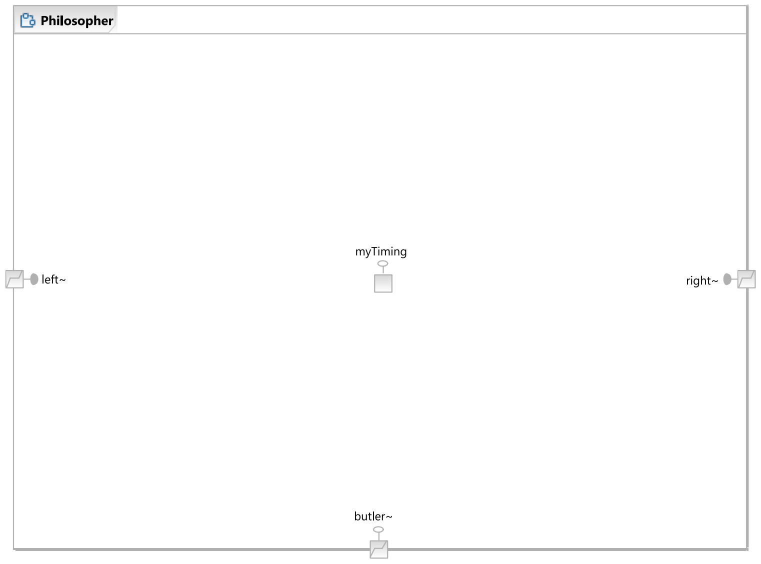
$$R_Q = \left(\sum_{i=0}^{s} m^i \right)^q \tag{8.8}$$

We illustrate the complexity introduced by communication channels by modeling a UML-RT capsule with one state and one port that receives messages of a protocol with one signal. All signals from all ports are handled by triggering a transition back to the same single state.

The effects in practice of increasing the number of possible signal are compared with the effect of increasing the number of ports receiving those signals. Those are the parameters in Equation 8.8 that are derived directly from the original UML-RT model. The length of a channel is a restriction in Promela which does originate from the UML-RT model, and is therefore fixed in all models. The comparison is made by performing a verification of the signal handling guarantee property (see Section 7.3.1) for each model version and obtaining the number of global system states stored in the state space for each verification run.

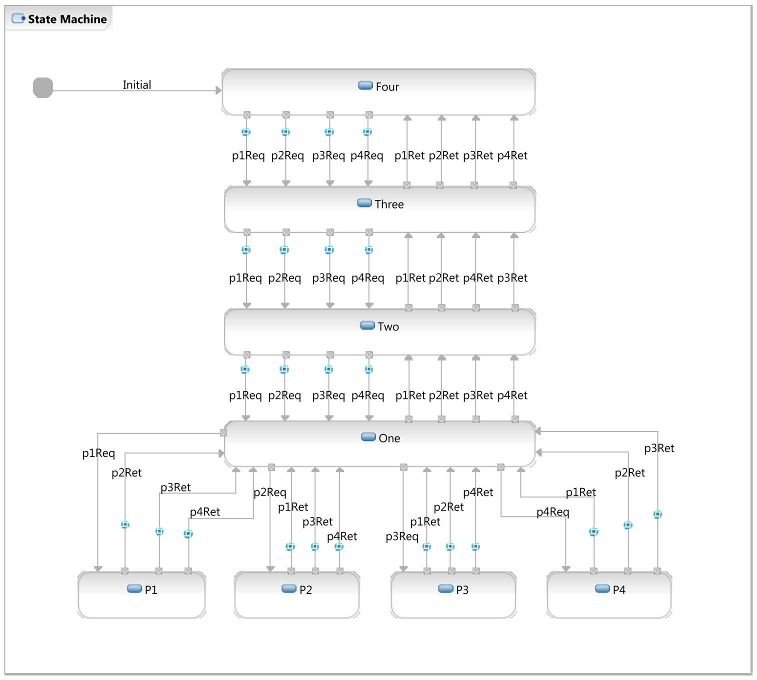


(a) Philosopher state machine

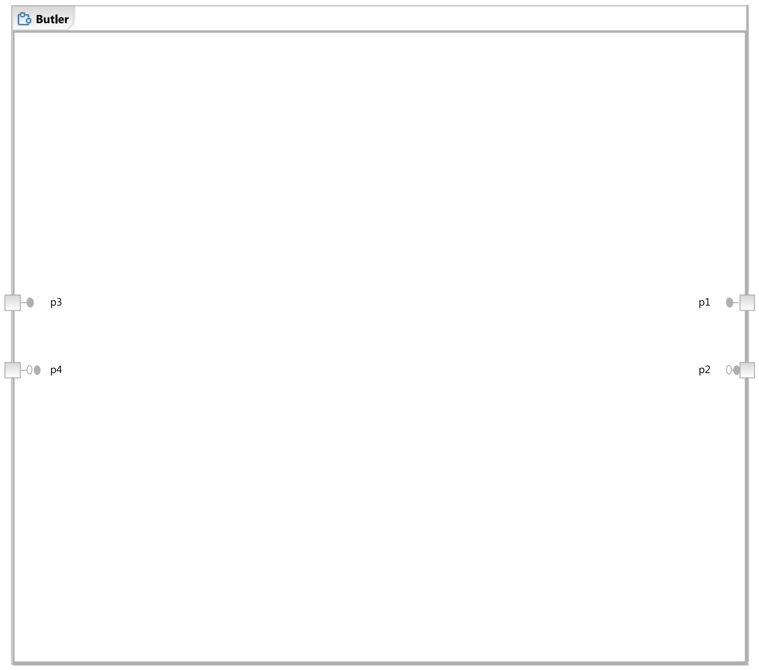


(b) Philosopher structure diagram

Figure 8.10: State machine diagram and structure diagram of the philosopher capsule.

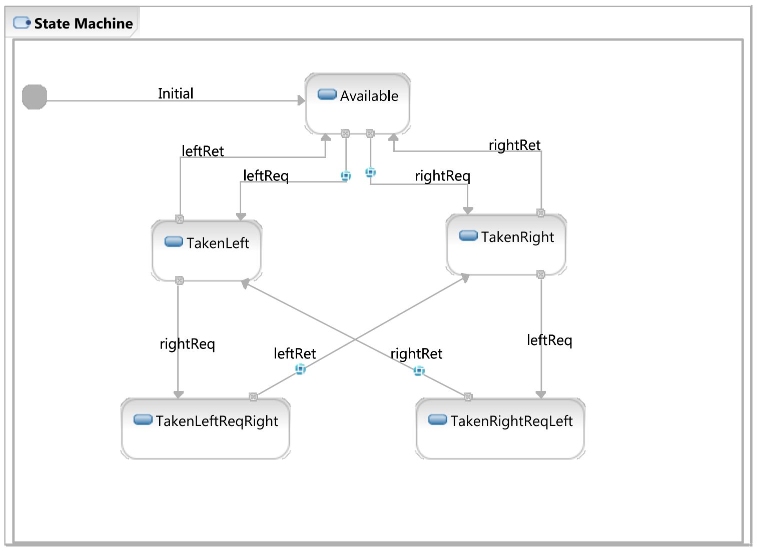

State Machine
Initial
Four
p1Req p2Req p3Req p4Req p1Ret p2Ret p3Ret p4Ret
Three
p1Req p2Req p3Req p4Req p1Ret p2Ret p4Ret p3Ret
Two
p1Req p2Req p3Req p4Req p1Ret p2Ret p3Ret p4Ret
One
p1Req
p2Ret
p3Ret
p4Ret
p2Req
p1Ret
p3Ret
p4Ret
p3Req
p1Ret
p2Ret
p4Ret
p1Ret
p4Req
p2Ret
p3Ret
P1
P2
P3
P4

(a) Butler state machine

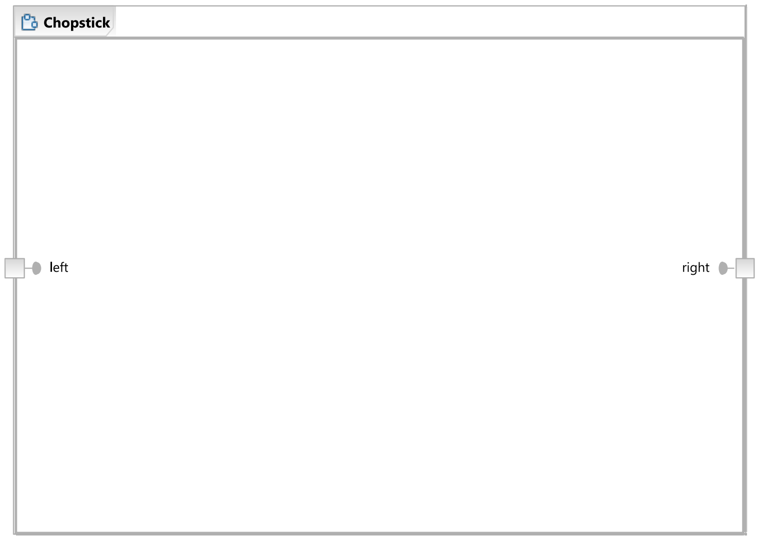


(b) Butler structure diagram

Figure 8.11: State machine diagram and structure diagram of the butler capsule.

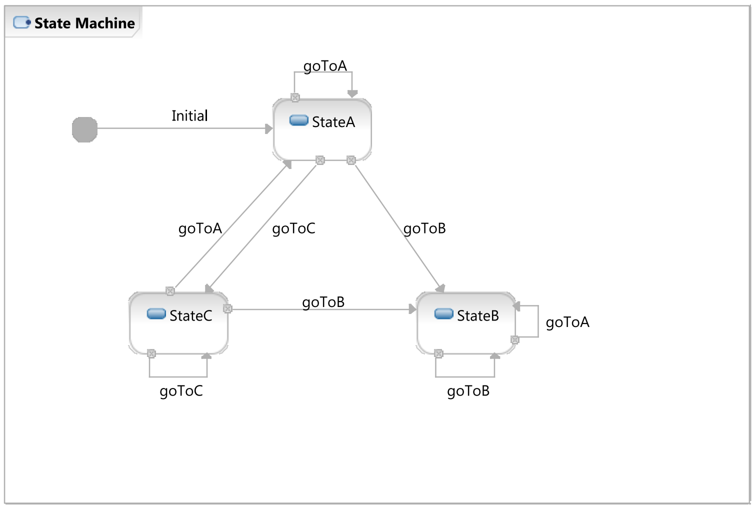


(a) Chopstick state machine

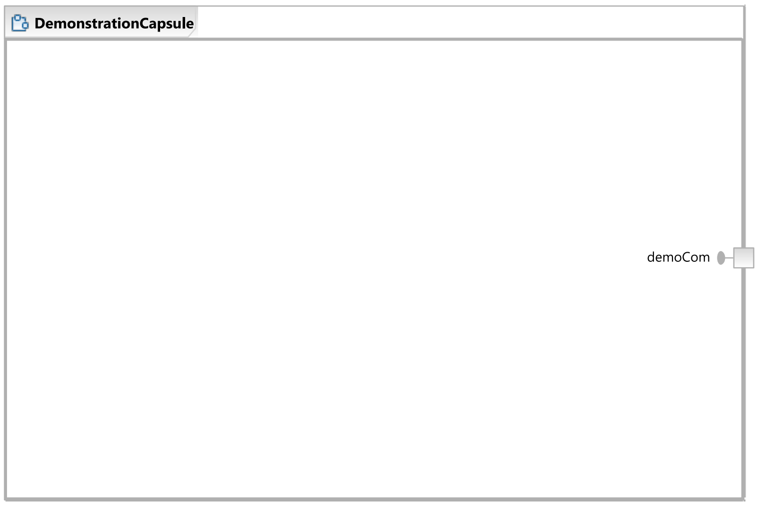


(b) Chopstick structure diagram

Figure 8.12: State machine diagram and structure diagram of the chopstick capsule.



(a) Demonstration capsule state machine



(b) Demonstration capsule structure diagram

Figure 8.13: State machine diagram and structure diagram of the demonstration capsule.

CHAPTER 9

Results

This chapter presents verification results for the sample models described in Chapter 8.

9.1 Traffic light system

The properties listed in Section 8.1.2 are verified one by one for each version of the model. In those cases where a property is proven not to hold, any remaining properties for that model version are excluded from verification. In some cases the number of errors for a particular property and model version exceeds one. This is due to an iterative search used to find the shortest path to the error.

9.1.1 First version

Property 1 (Signal handling guarantee)

The verification results in errors (see Listing 9.1), demonstrating that there is at least one state in the model which does not handle every possible input signal. The generated error trail demonstrates how an assertion due to unhandled signals can be violated and can be illustrated visually using a message sequence diagram (see Figure 9.1).

We can see that as the *controller* capsule makes its initial transition (lines 27–35) it sends signals to all four traffic lights (lines 31–34). The *east* traffic light receives the *red* signal (line 39) even though it is already in the *Red* state (line 16). The *red* signal is not handled in the *Red* state (see Section 8.1.3) which causes an assertion violation (line 41).

Listing 9.1: Verification results for property 1 (signal handling guarantee) for the first version of the traffic light system model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          - (none specified)
   assertion violations +
   cycle checks         - (disabled by -DSAFETY)
   invalid end states   +

State-vector 76 byte, depth reached 32, errors: 2
       15 states, stored
        8 states, matched
       23 transitions (= stored+matched)
       26 atomic steps
hash conflicts:         0 (resolved)

    2.501   memory usage (Mbyte)
```

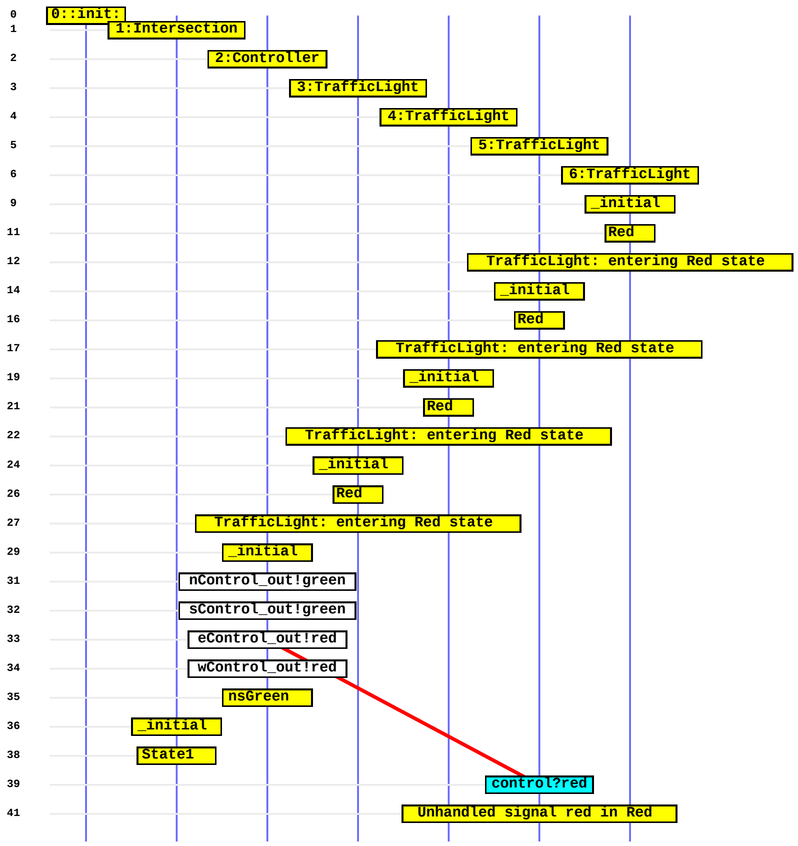


Figure 9.1: Error scenario for property 1 (signal handling guarantee) for the first version of the traffic light system model.

Listing 9.2: Verification results for property 1 (signal handling guarantee) for the second version of the traffic light system model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          - (none specified)
   assertion violations +
   cycle checks         - (disabled by -DSAFETY)
   invalid end states   +

State-vector 76 byte, depth reached 202, errors: 0
      325 states, stored
      493 states, matched
      818 transitions (= stored+matched)
      945 atomic steps
hash conflicts:         0 (resolved)

    2.501   memory usage (Mbyte)
```

9.1.2 Second version

Property 1 (signal handling guarantee)

The verification produces no errors (see Listing 9.2), proving that assertion violations due to unhandled signals cannot occur.

Property 2 (recurrence)

The property is verified for all four traffic lights in the model and does not produce any errors, proving that all traffic lights exhibit a cyclic behavior and enter the *Green* state infinitely often in an infinite run of the model. The property also holds for the *Yellow* state and *Red* state for each traffic light.

Only the result from the verification of the *north* traffic light's *green* state is presented (see Listing 9.3). The remaining eleven out of twelve verification runs (four traffic lights and three states) produce near-identical results and are omitted.

Property 3 (mutual exclusion)

The verification produces errors (see Listing 9.4), proving that the mutual exclusion property does not hold for the four traffic lights guarding the intersection. It is therefore possible that two traffic lights guarding intersecting roads can simultaneously be in their respective *Green* states, which could lead to collisions in a real system.

The visual representation (see Figure 9.2) of the error trail illustrates a message sequence by which the property claim can be refuted. The root cause of the error is that the *controller* capsule is allowed to switch between control states without first establishing that its previously sent messages have been received by the traffic lights.

Listing 9.3: Verification results for property 2 (recurrence) for the second version of the traffic light system model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          +
   assertion violations + (if within scope of claim)
   acceptance   cycles  + (fairness disabled)
   invalid end states   - (disabled by never claim)

State-vector 80 byte, depth reached 288, errors: 0
      615 states, stored (905 visited)
     2054 states, matched
     2959 transitions (= visited+matched)
     3157 atomic steps
hash conflicts:        12 (resolved)

    2.501   memory usage (Mbyte)
```

Listing 9.4: Verification results for property 3 (mutual exclusion) for the second version of the traffic light system model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          +
   assertion violations + (if within scope of claim)
   acceptance   cycles  + (fairness disabled)
   invalid end states   - (disabled by never claim)

State-vector 80 byte, depth reached 157, errors: 18
      318 states, stored (324 visited)
      671 states, matched
      995 transitions (= visited+matched)
     1191 atomic steps
hash conflicts:         0 (resolved)

    2.501   memory usage (Mbyte)
```

Listing 9.5: Verification results for property 1 (signal handling guarantee) for the third version of the traffic light system model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          - (none specified)
   assertion violations +
   cycle checks         - (disabled by -DSAFETY)
   invalid end states   +

State-vector 104 byte, depth reached 155, errors: 0
      283 states, stored
      421 states, matched
      704 transitions (= stored+matched)
     1063 atomic steps
hash conflicts:         0 (resolved)

    2.501   memory usage (Mbyte)
```

Figure 9.2 shows a situation in which the *north*, *east* and *west* traffic lights simultaneously are in the *Green* state. The chain of events leading to this situation is the following:

1. The *north* traffic light is in state *Red* (line 77), the *east* traffic light is in state *green* (line 72) and the *west* traffic light is in state *Green* (line 82).
2. The *controller* transitions from the state *ewGreen* to the state *ewYellow* (lines 68–88) and sends the signal *yellow* to both the *east* and *west* traffic lights (lines 86–87).
3. Those signals have not yet been received when the *controller* transitions to its next state and sends the signal *Green* to the *north* traffic light (line 91).
4. The *north* traffic light receives the *green* signal (line 92) and enters its *Green* state (line 95).

This causes green lights in intersecting directions, which is a violation of the mutual exclusion property.

9.1.3 Third version

Property 1 (signal handling guarantee)

The verification produces no errors (see Listing 9.5), proving that assertion violations due to unhandled signals cannot occur. This is the same result as for the first and second versions of the traffic light system.

Property 2 (recurrence)

The property is verified for all four traffic lights in the model and does not produce any errors. This is the same result as for the second version of the

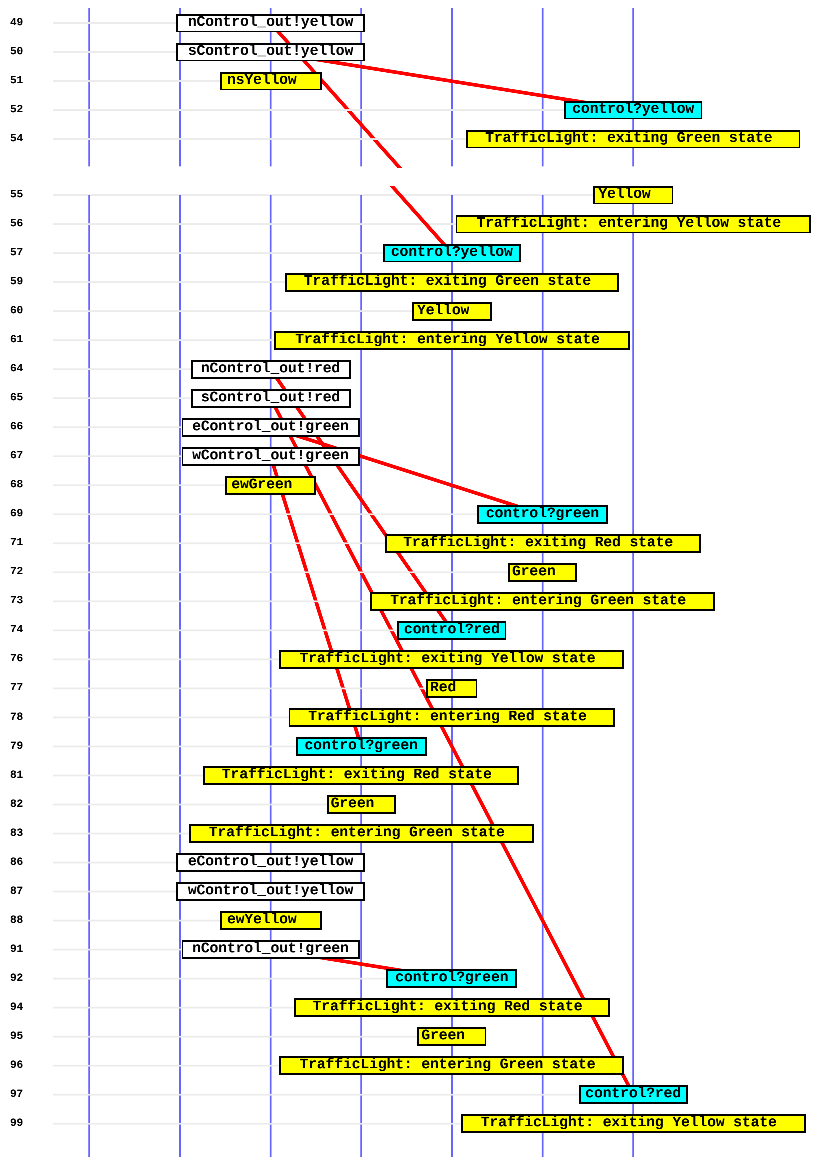

49
nControl_out!yellow
50
sControl_out!yellow
51
nsYellow
52
control?yellow
54
TrafficLight: exiting Green state
55
Yellow
56
TrafficLight: entering Yellow state
57
control?yellow
59
TrafficLight: exiting Green state
60
Yellow
61
TrafficLight: entering Yellow state
64
nControl_out!red
65
sControl_out!red
66
eControl_out!green
67
wControl_out!green
68
ewGreen
69
control?green
71
TrafficLight: exiting Red state
72
Green
73
TrafficLight: entering Green state
74
control?red
76
TrafficLight: exiting Yellow state
77
Red
78
TrafficLight: entering Red state
79
control?green
81
TrafficLight: exiting Red state
82
Green
83
TrafficLight: entering Green state
86
eControl_out!yellow
87
wControl_out!yellow
88
ewYellow
91
nControl_out!green
92
control?green
94
TrafficLight: exiting Red state
95
Green
96
TrafficLight: entering Green state
97
control?red
99
TrafficLight: exiting Yellow state

Figure 9.2: Error scenario for property 3 (mutual exclusion) for the second version of the traffic light system model.

Listing 9.6: Verification results for property 2 (recurrence) for the third version of the traffic light system model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          +
   assertion violations + (if within scope of claim)
   acceptance   cycles  + (fairness disabled)
   invalid end states   - (disabled by never claim)

State-vector 108 byte, depth reached 214, errors: 0
      527 states, stored (771 visited)
     1784 states, matched
     2555 transitions (= visited+matched)
     3574 atomic steps
hash conflicts:         0 (resolved)

    2.501   memory usage (Mbyte)
```

traffic light system. Listing 9.6 presents the result from the verification run of the *north* traffic light's *green* state.

Property 3 (mutual exclusion)

The verification produces no error (see Listing 9.7), proving that the mutual exclusion property holds for the traffic system. The traffic lights for the intersecting roads do not enter their *Green* states in such a way that traffic can collide.

9.2 Electronic lock

The properties listed in Section 8.2.2 are verified one by one for each version of the model. In those cases where a property is proven not to hold, any remaining properties for that model version are excluded from verification. In some cases the number of errors for a particular property and model version exceeds one. This is due to an iterative search used to find the shortest path to the error.

9.2.1 First version

Property 1 (signal handling guarantee)

The verification produces no error (see Listing 9.8), proving that assertion violations due to unhandled signals cannot occur.

Property 2 (trap detection)

The property is verified for all states in the model and produces counterexamples in every case. This demonstrates that no state in the model is a trapping state.

Listing 9.7: Verification results for property 3 (mutual exclusion) for the third version of the traffic light system model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          +
   assertion violations + (if within scope of claim)
   acceptance   cycles  + (fairness disabled)
   invalid end states   - (disabled by never claim)

State-vector 108 byte, depth reached 214, errors: 0
      283 states, stored
      421 states, matched
      704 transitions (= stored+matched)
     1063 atomic steps
hash conflicts:         0 (resolved)

    2.501   memory usage (Mbyte)
```

Listing 9.8: Verification results for property 1 (signal handling guarantee) for the first version of the electronic lock model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          - (none specified)
   assertion violations +
   cycle checks         - (disabled by -DSAFETY)
   invalid end states   +

State-vector 28 byte, depth reached 169, errors: 0
      424 states, stored
      245 states, matched
      669 transitions (= stored+matched)
      302 atomic steps
hash conflicts:         0 (resolved)

    2.501   memory usage (Mbyte)
```

Listing 9.9: Verification results for property 2 (trap detection) for the first version of the electronic lock model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          +
   assertion violations + (if within scope of claim)
   acceptance   cycles  + (fairness disabled)
   invalid end states   - (disabled by never claim)

State-vector 32 byte, depth reached 20, errors: 2
      125 states, stored
      216 states, matched
      341 transitions (= stored+matched)
       78 atomic steps
hash conflicts:         0 (resolved)

    2.501   memory usage (Mbyte)
```

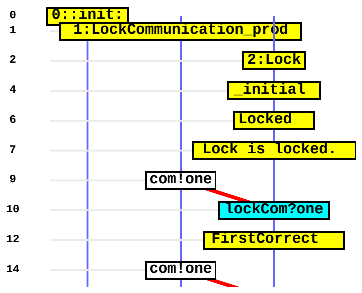


Figure 9.3: Error scenario for property 2 (trap detection) for the first version of the electronic lock model.

We present only the result for state *Locked* (see Listing 9.9) since the results for the other states are near-identical.

The visual representation of the error trail (see Figure 9.3) produced by Spin shows a counterexample to the claim that state *Locked* is trapping. In the counterexample, we can see that the lock is in state *Locked* (line 6). This state can be exited by following the transition triggered on reception of the signal *one* (line 10), leading into state *FirstCorrect*.

Property 3 (locking the lock)

The verification produces no error (see Listing 9.10). This proves that if the environment sends the signal *lock*, the lock will eventually become locked.

Listing 9.10: Verification results for property 3 (the lock can be locked) for the first version of the electronic lock model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          +
   assertion violations + (if within scope of claim)
   acceptance   cycles  + (fairness disabled)
   invalid end states   - (disabled by never claim)

State-vector 32 byte, depth reached 296, errors: 0
      484 states, stored (531 visited)
      472 states, matched
     1003 transitions (= visited+matched)
      464 atomic steps
hash conflicts:         0 (resolved)

    2.501   memory usage (Mbyte)
```

Property 4 (unlocking the lock)

The verification produces errors (see Listing 9.11) and a trail, refuting the claim that a locked lock is impossible to unlock.

The message sequence diagram (see Figure 9.4) illustrating the error trail demonstrates a counterexample to the claim. As expected, the lock can indeed be unlocked (line 24) if the environment provides the correct sequence of input signals, i.e., by sending signal *one* (line 9), followed by signal *two* (line 14).

9.2.2 Second version

Property 1 (signal handling guarantee)

The verification produces errors (see Listing 9.12) demonstrating that there is at least one state in the model that does not handle all types of input signals.

The message sequence diagram (see Figure 9.5) illustrating the error trail demonstrates a signal sequence that causes an assertion violation, due to an unhandled signal. The violation occurs in state *Unlocked* (line 23) on reception of the signal *other*, sent by the environment (line 20).

9.3 Dining philosophers

The properties listed in Section 8.3.3 are verified one by one for the model of the dining philosophers.

Property 1 (signal handling guarantee)

The verification produces no errors (see Listing 9.13), proving that no assertion violations due to unhandled signals can occur.

Listing 9.11: Verification results for property 4 (the lock cannot be unlocked) for the first version of the electronic lock model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          +
   assertion violations + (if within scope of claim)
   acceptance   cycles  + (fairness disabled)
   invalid end states   - (disabled by never claim)

State-vector 32 byte, depth reached 41, errors: 4
      485 states, stored
      818 states, matched
     1303 transitions (= stored+matched)
      574 atomic steps
hash conflicts:         0 (resolved)

    2.501   memory usage (Mbyte)
```

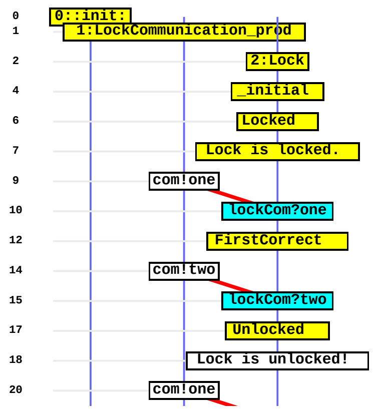


Figure 9.4: Error scenario for property 4 (the lock cannot be unlocked) for the first version of the electronic lock model.

Listing 9.12: Verification results for property 1 (signal handling guarantee) for the second version of the electronic lock model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          - (none specified)
   assertion violations +
   cycle checks         - (disabled by -DSAFETY)
   invalid end states   +

State-vector 28 byte, depth reached 58, errors: 11
      389 states, stored
      805 states, matched
     1194 transitions (= stored+matched)
     1013 atomic steps
hash conflicts:         0 (resolved)

    2.501   memory usage (Mbyte)
```

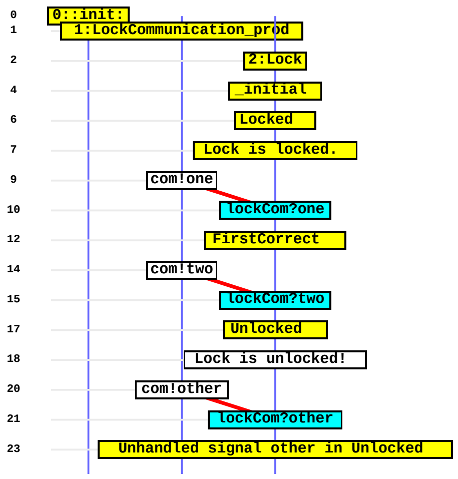


Figure 9.5: Error scenario for property 1 (signal handling guarantee) for the second version of the electronic lock model.

Listing 9.13: Verification results for property 1 (signal handling guarantee) for the dining philosophers model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          - (none specified)
   assertion violations +
   cycle checks         - (disabled by -DSAFETY)
   invalid end states   +

State-vector 248 byte, depth reached 480492, errors: 0
  1493090 states, stored
  6997300 states, matched
  8490390 transitions (= stored+matched)
 11225119 atomic steps
hash conflicts:   5920646 (resolved)

  394.554   memory usage (Mbyte)
```

Property 2 (simultaneous eating)

The property is verified for all four philosophers in the model and does not produce errors in any case. This proves, for all philosophers, that if a philosopher is eating then the two neighboring philosophers are not eating (since they share chopsticks). We present only the verification result for philosopher one (see Listing 9.14) since the results for the other philosophers are near-identical.

Property 3 (chopstick usage)

The property is verified for all four philosophers in the model and does not produce errors in any case. This proves, for all philosophers, that if a philosopher is eating, it uses the chopsticks on both sides. We present only the verification result for philosopher one (see Listing 9.15) since the results for the other philosophers are near-identical.

Property 4 (recurrence)

Verification of the property produces a counterexample for each philosopher in the model. Listing 9.16 shows the verification result for philosopher one and contains an error, which proves that there exists an infinite execution of the model in which it is not the case that philosopher one is allowed to eat infinitely many times. The reason for this is that the *Butler* process has a separate communication channel for each *Philosopher* and there is no facility in the verification model to force the *Butler* to choose fairly between *Philosophers.*

Listing 9.14: Verification results for property 2 (simultaneous eating) for the dining philosophers model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          +
   assertion violations + (if within scope of claim)
   acceptance   cycles  + (fairness disabled)
   invalid end states   - (disabled by never claim)

State-vector 252 byte, depth reached 657968, errors: 0
  1493090 states, stored
  6997300 states, matched
  8490390 transitions (= stored+matched)
 11225119 atomic steps
hash conflicts:   5929190 (resolved)

Stats on memory usage (in Megabytes):
  381.611   equivalent memory usage for states
      (stored*(State-vector + overhead))
  325.017   actual memory usage for states (compression:
      85.17%)
            state-vector as stored = 212 byte + 16 byte
                overhead
    2.000   memory used for hash table (-w19)
   21.362   memory used for DFS stack (-m700000)
  348.265   total actual memory usage
```

Listing 9.15: Verification results for property 3 (chopstick usage) for the dining philosophers model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          +
   assertion violations + (if within scope of claim)
   acceptance   cycles  + (fairness disabled)
   invalid end states   - (disabled by never claim)

State-vector 252 byte, depth reached 657968, errors: 0
  1493090 states, stored
  6997300 states, matched
  8490390 transitions (= stored+matched)
 11225119 atomic steps
hash conflicts:   5929190 (resolved)

Stats on memory usage (in Megabytes):
  381.611   equivalent memory usage for states
      (stored*(State-vector + overhead))
  325.016   actual memory usage for states (compression:
      85.17%)
            state-vector as stored = 212 byte + 16 byte
                overhead
    2.000   memory used for hash table (-w19)
   21.362   memory used for DFS stack (-m700000)
  348.265   total actual memory usage
```

Listing 9.16: Verification results for property 4 (recurrence) for the dining philosophers model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          +
   assertion violations + (if within scope of claim)
   acceptance   cycles  + (fairness disabled)
   invalid end states   - (disabled by never claim)

State-vector 252 byte, depth reached 129, errors: 1
    26596 states, stored (39893 visited)
   199034 states, matched
   238927 transitions (= visited+matched)
   359188 atomic steps
hash conflicts:      2200 (resolved)

   29.319   memory usage (Mbyte)
```

Listing 9.17: Verification results for property 1 (signal handling guarantee) for the demonstration model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          - (none specified)
   assertion violations +
   cycle checks         - (disabled by -DSAFETY)
   invalid end states   +

State-vector 28 byte, depth reached 38, errors: 7
       79 states, stored
       89 states, matched
      168 transitions (= stored+matched)
      138 atomic steps
hash conflicts:         0 (resolved)

    2.501   memory usage (Mbyte)
```

9.4 Model with intentional errors

The properties listed in Section 8.4.1 are verified for the model. In some cases the number of errors for a particular property exceeds one. This is due to an iterative search used to find the shortest path to the error.

Property 1 (signal handling guarantee)

The verification produces errors (see Listing 9.17) which proves that there is at least one state in the model that does not handle all input signals.

The message sequence chart (see Figure 9.6) generated from the error trail demonstrates how an assertion can be violated as a result of a signal not being handled. The figure illustrates that when the capsule is in *StateB* (line 11) and the environment sends the signal *goC* (line 13) which is received while still in *StateB* (line 14), the result is an assertion violation (line 16).

Property 2 (trap detection)

The verification is performed for every state in the model and produces a counterexample in all cases except for *StateB*, proving that *StateB* is the only trapping state in the model. The verification result in Listing 9.18 shows that Spin cannot refute the claim that specifies that *stateB* is a trapping state.

9.5 Complexity experiment

The results of the complexity experiment for comparing the state space effects of increased number of ports and signals are presented in Figure 9.7. We see that the behavior predicted by Equation 8.8 is exhibited by the verified models.

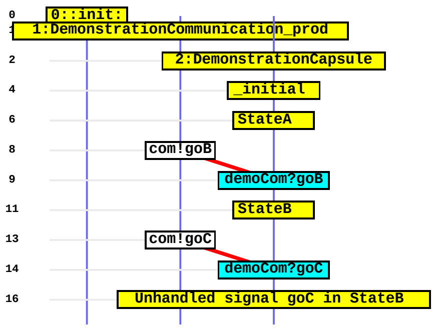


Figure 9.6: Error scenario of property 1 (signal handling guarantee) for the demonstration model.

Listing 9.18: Verification results for property 2 (trap detection) for the demonstration model.

```
(Spin Version 5.1.7 -- 23 December 2008)
   + Partial Order Reduction

Full statespace search for:
   never claim          +
   assertion violations - (disabled by -A flag)
   acceptance   cycles  + (fairness disabled)
   invalid end states   - (disabled by never claim)

State-vector 32 byte, depth reached 89, errors: 0
      139 states, stored
      121 states, matched
      260 transitions (= stored+matched)
      137 atomic steps
hash conflicts:         0 (resolved)

    2.501   memory usage (Mbyte)
```

The number of stored states increases exponentially when the number of ports is increased, but more slowly when the number of signals is increased.

We can also observe a plateau effect in Figure 9.7 which is due to insufficient memory resources. For larger values for the number of states or ports, the memory required to perform the verification exceeds the available memory of the test machine (see Section 3.1).

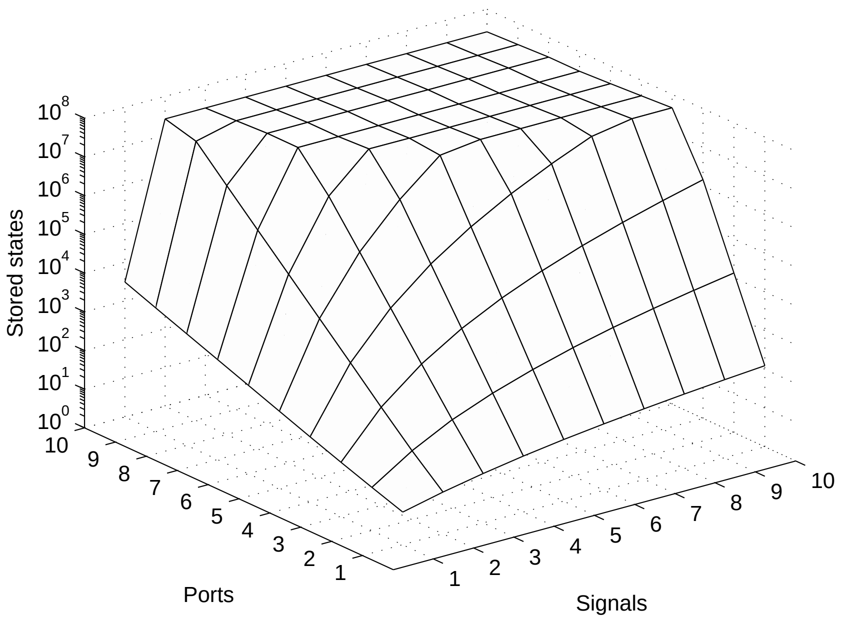


Figure 9.7: Number of stored states used for verifying the signal handling guarantee property (see Section 7.3.1) depending on the number of ports and signals used. Note that the plateau effect is due to insufficient memory resources which results in incomplete verification.

CHAPTER 10

Conclusions

This chapter presents answers the questions posed in Section 1.2 and a discussion on the verification approach, the prototype tool and their limitations.

10.1 Answers to questions

Question 1

What methods and tools for formal verification of software models are available in the academic, industrial or open source communities?

This question is addressed by decomposition into sub-questions.

Sub-question 1a

What approach could be used as a basis for verification of models developed at the department?

Previous work for verifying properties of UML state machines and a visual formalism similar to UML-RT has been found in the literature. The reviewed tools all use a transformational approach, in the sense that UML models, often consisting of a subset of the available model constructs, are transformed to the input language of a general model checker. Properties of the transformed model are then verified with the model checker and the results of the verification are interpreted in the context of the original model.

The model checkers used as back-ends for the verification include Spin, SMV and RuleBase. Spin is a dominating tool in the domain of software model checking. A majority of the previous work transforms the original model into a verification model specified in Promela, the input language of Spin, and this is also the case in our work.

Sub-question 1b

Are there existing tools for formal verification that can be directly used to verify properties of models in the development environment at the department? If this is not the case, are there existing tools that can serve as a foundation for implementing such a tool?

No commercial or industrial strength tool has been found, tailored to the problem of verifying properties of UML state machines in general, and of UML-RT or RSARTE models in particular. The majority of previous work is at a research stage and targets earlier versions of UML that do not incorporate the capsule concept. Not all tools have been available for evaluation or experimentation; conclusions about the applicability of those tools have had to be drawn from presented tool descriptions only.

For these reasons, a prototype tool has been implemented which draws heavily on ideas from tools such as vUML, Hugo and VIP, e.g., the Promela output of the prototype tool is similar to that which is produced by the VIP tool.

Sub-question 1c

Can these tools be integrated into the development environment at the department?

The prototype tool for transforming RSARTE models into Promela models, demonstrates that it to a large degree is possible to automate the transformation procedure. The implementation of the prototype tool using the JET tool, demonstrates that it is possible to integrate a verification tool in the RSARTE environment.[1]

Question 2

What properties are of interest to the department to verify?

Two properties have been identified in discussions with Ericsson engineers as being of primary interest to the department (see Section 7.3). They can be stated informally as:

1. guaranteed handling of all possible signals sent to a capsule, and
2. detection of trapping states that are impossible to exit.

The second of these two primary properties is specified in LTL, a formalism which allows great flexibility in the kinds of properties that can be expressed. Other properties that are of potential interest to verify can therefore also be specified, such as the mutual exclusion property verified for the third version of the traffic system model.

[1]Previous work integrating a model checker in a development environment exists. See e.g. Beyer, Henzinger, Jhala & Majumdar (2004) where an Eclipse plug-in for model checking is described.

Question 3

How should models be developed to allow such properties to be formally verified using the chosen method?

State machines and states

The properties that are specified by LTL formulas specify behaviors of a capsule's state machine, or the combined behaviors of several capsules' state machines. To be able to specify meaningful properties, the states of the state machine must be selected so that the behavior of the capsule is visible in the structure of the state machine itself.

This means that although it is entirely possible to *camouflage* statefulness of a UML-RT model through constructs below the visible surface of the state machine, this possibility should be avoided. Values of capsule attributes can, for example, be modified in action code and then be used to determine which transition to take when a signal is received. This complicates automatic transformation to a verification model, since the transformation must then depend not only on the structure of the state machine, but also on analysis of the embedded action code.

Using only a few or even only one state also makes it difficult to specify properties of interest, since the LTL properties describe paths through the states of the state machine. Similarly, adding unnecessary states should be avoided since it increases the complexity of the generated verification model without increasing the potential for specifying meaningful properties.

The current version of the transformation tool recognizes a limited subset of the model constructs in UML-RT (see Section 6.2.2). Until more constructs are recognized, the UML-RT model must be restricted to this limited subset. Support for more constructs can be added by creating additional templates that capture the transformation rules for those features.

Embedded code

The prototype tool is built to automatically generate verification models in Promela code, capturing state machines of capsules and simple structural relationships between capsules. As stated in the limitations of this thesis, behavior described by the action code embedded in the state machines is not subject to automatic transformation.

Nonetheless, since signal sending is necessary to enable communication between state machines, a facility to combine C++ code with Promela code is discussed in Section 6.2.4. It allows a Promela model of the action code to be stored within the UML-RT model without affecting the generated C++ code, and consequently allows the verification model to capture signal sending. The separation is achieved using directives recognized by both the C pre-processor used by Spin and the pre-processor used before compiling the generated C++ code at the department.

Signal reception is handled automatically by the transformation tool. This is possible since signals trigger transitions in a way which is captured by the UML-RT model structure.

Question 4

What can be gained from applying formal verification methods to the department's software models?

The chosen formal verification method is model checking. Applying model checking to the department's software models has the potential of providing several benefits in the development process.

Firstly, the department uses test cases that target both normal and failure cases. Constructing suitable test cases is difficult and always carries with it the risk of overlooking hazardous scenarios. Holzmann (2003, p. 5) writes, "For every one failure scenario that is considered, there are a million others that may have been overlooked." For this reason, it can be beneficial to the department to use methods that can provide confidence that no failure scenario has been neglected.

Whalen, Cofer, Miller, Krogh & Storm (2007, pp. 72–73) observe an important difference in the verification process between test-based verification and verification by model checking and write, "Instead of focusing on the creation of test vectors, the focus is on the creation of *properties* and *environmental assumptions*." Rather than attempting to construct ways in which to demonstrate a behavior, whether required or prohibited, the objective in model checking is to specify *exactly* the property that is of interest and leave the task of demonstration to the tool.

Secondly, a capsule may be constructed with preconceived ideas on how it will be used. This means that faults may remain latent in the capsule, due to assumptions on its future environment. If the environment changes or if the preconceptions were incorrect, signals may be sent to the capsule in patterns that were not originally considered and latent faults may be activated and trigger errors. Since the model checker makes no such assumptions, unless explicitly instructed, faults that could otherwise have been missed can be discovered. Holzmann (2003, p. 5) writes, "Automated tools have no trouble constructing the bizarre scenarios that no sane human could imagine — just the type of scenario that causes real-life systems to crash." Ensuring correct capsule behavior without assumptions on the environment can therefore increase the robustness of the department's product.

Thirdly, the test methods in current use at the department generally target an already implemented version of a software model, which means that they are used at a late stage in development. Detecting errors early in the lifecycle can have significant cost benefits, but can be difficult unless behavior of early models can be tested. This is the domain of model checking; allowing abstract verification models to be verified and errors to be detected even at an early stage (Holzmann 2003, p. 6).

Finally, model checking has the potential of discovering errors that occur as a result of emergent behavior of interacting system components, such as communicating capsules. The two primary properties (see Section 7.3) do not focus directly on such behavior. The detection of the mutual exclusion violation in the second version of the traffic light system, followed by the correct verification of this property in the third version of the traffic light system, demonstrates that the method proposed in this thesis can be used also to verify properties of such emergent behavior.

10.2 Discussion

This section presents a discussion of some benefits and limitations of the implemented prototype tool. We also present cautionary advice on how results of the verification can be interpreted in the context of the original UML-RT model.

10.2.1 Benefits and drawbacks

We consider the main benefit and novelty of the prototype tool to be that it narrows the gap between the visual modeling tool RSARTE and the model checker Spin. Although the VIP tool is a predecessor and supports a wider range of constructs, e.g., hierarchical states, VIP is a separate environment, whereas the prototype tool is directly integrated in RSARTE.

The use of a template system to carry out the transformation is convenient since it permits further refinement and extension of the prototype tool. Support for more UML-RT constructs can be added with relative ease.

The main drawback, apart from the limited set of supported UML-RT model constructs, is the rapid growth of the verification models. The Promela models that are created have large state spaces, even for UML-RT models that are seemingly relatively small. One cause is the channel communication between processes, which follows from the communication between capsules. Channel communication, a known source of complexity (Holzmann 2003, p. 125), is used extensively, not only between processes resulting from capsule transformations, but also between such processes and generated stimulus processes. It may be possible to reduce complexity when verifying capsules with public ports, by moving the generated stimulus processes into the target process, thereby removing what Holzmann (2003, p. 106) calls *sink* and *source* processes.

10.2.2 Primary application areas

This work suggests and exemplifies two applications of model checking in the development of UML-RT models at the department. Correctness properties that depend directly on action code are considered outside the structure of the model and are therefore also beyond the scope of this thesis. Interesting correctness properties can very well still be studied.

The first application concerns the capability to specify properties of a capsule that interacts with a non-deterministic environment. This can be used to verify properties that have been identified to be of interest to the department, as has previously been discussed. This is also demonstrated, e.g., in the example model of the electronic lock. The robustness of a capsule can be improved by verifying properties without assumptions on its environment.

The second application is to use the model checker to verify properties of a UML-RT model at an early design stage. The prototype tool supports the basic UML-RT construct for specifying concurrency and the basic constructs that relate to interaction. We propose that the prototype tool could be used to verify the interaction between several capsules, before the action code that implements the work performed by the capsule is added.

The prototype tool can then provide a means to ensure that communication between capsules is correctly implemented, and consistency in the combined behavior of several capsules. Spin stems from early tools designed for protocol analysis (Holzmann 1997, p. 6), and the prototype tool brings this capability into RSARTE.

The model of the traffic light system (see Section 8.1) can be used as an argument supporting this. It is, for example, not necessary to implement the details of how the traffic light interacts with hardware to actually turn on the correct light, in order to verify interesting properties of the system. The possible violation of the mutual exclusion property is a problem with the model that should be addressed, regardless of how hardware interaction is later implemented within a specific traffic light.

10.2.3 Important verification issues

Fixed channel lengths

The necessity of using fixed-length buffers for communication between processes in Promela is a restriction that is not imposed in UML-RT. A Promela process will be blocked if it attempts to send messages on a full channel, or on a rendezvous channel[2] when the receiving process is not ready to receive. This can cause an unintentional breach of atomicity in the verification model. If this happens, verification results can be produced that are difficult to interpret, since the process in the verification model will be blocked in an intermediate state that does not have a direct counterpart in the UML-RT model.

Using oversized[3] channels to avoid blocking has the obvious negative effect of dramatically increasing the state space of the verification model. A fixed value is necessary to obtain a closed verification model, but it is very easy to create a model that is far too large to verify by setting this value too high.

Environment dependent results

It is necessary to consider how to interpret results from a verification run of a Promela model in the context of the original UML-RT model. If a verification model is generated from a capsule that contains capsule roles, e.g., the traffic light system model, then the results obtained by verification reflect only the properties of that particular configuration.

For example, the change between the first and the second version of the traffic light system — the change that allowed the signal guarantee property to be verified — was not implemented by ensuring that all traffic lights handled all signals in all states. Instead, a restriction was made on the behavior of the *Controller* to ensure that it took into consideration the incomplete implementation of the *TrafficLight* capsule.

Consequently, the results of the verification should not be interpreted to show that a *TrafficLight* capsule always handles all signals in all states. The correct interpretation is that in the context of the *Intersection* capsule (which is the capsule that was verified), all capsule roles handled all signals that they could possibly receive. The *Intersection* capsule is not an indeterministic environment, which explains why the property held when verifying the *Intersection* capsule.

[2]A rendezvous channel in Promela has a buffer capacity of zero messages, which means that it forces synchronous message exchange.

[3]Leue, Mayr & Wei (2004) present work in the area of determining upper bounds on the number of messages in communication channels.

If we wish to verify the signal guarantee property for the *TrafficLight* capsule as a standalone unit, it must be verified outside the *Intersection* capsule. We would then find that the *TrafficLight* capsule's state machine contains states that do not handle all signals that can be received in an indeterministic environment.

10.3 Future work

This thesis shows that model checking can be used to verify properties of UML-RT models created in RSARTE and more work can be done to further explore the possibilities and limitations of the approach.

10.3.1 Properties

Two important property related areas that can be further developed are: how to specify properties and how to link properties with models.

In addition to the properties addressed in this thesis, we believe that there would be no shortage of interesting properties to verify for the department's models. The LTL formalism supported by Spin and consequently also by the prototype tool is very expressive and has a simple syntax, but can still be challenging to use correctly.

Dwyer et al. (1999) have presented work to address this by showing that over 90 % of properties targeted in finite-state verification conform to certain patterns.[4] An analysis of properties that can be extracted from specification documents at the department could be used for developing pattern repositories in the style of Dwyer et al.. It could also prove useful to integrate a tool in RSARTE for repository access, and for assisting in LTL property specification.

The practical aspects of linking properties with models should also be explored. UML can be extended using *profiles* and *stereotypes* (Booch, Rumbaugh & Jacobson 2005), and it could therefore be of interest to develop notation to supplement a capsule with LTL properties.

10.3.2 Model transformation

The prototype implementation should be extended to support a larger subset of common modeling constructs, such as hierarchical states. This will facilitate creation of more realistic models, which would be useful for future scalability tests. More modeling constructs will not necessarily lead to more complicated verification models; on the contrary, more compact model constructs, e.g., transitions that trigger on more than one signal, can reduce the number of required states in the original model as well as in the verification model.

The problem of having to create two versions of the action code to enable signal sending in the verification model should also be addressed. We speculate that specifying the action code in a higher-level action language could be useful in this respect. This type of specification could then be used both to generate

[4]The most common pattern found in the study was the *response* pattern, defined in the authors' classification by, "A state/event P must always be followed by a state/event Q within a scope."(Dwyer et al. 1999, p. 415) The property can be specified in LTL, and we verify a property that follows this pattern in the Electronic lock example model (see Section 8.2.2, property 3).

C++ code and to generate Promela code for use in the verification model, ensuring consistency between the two.

10.3.3 Scalability

Based on our experience with actual UML-RT models at the department, realistic models widely surpass the example models in terms of complexity. The feasibility of verifying properties of such models must be examined, and we find it likely that the problem of targeting more complex models will need to be addressed from several angles.

It could be of interest to evaluate scalability potential by increasing the memory and computational capacity of the machine running the verification. An approach that could be interesting in this context is that of parallelizing the verification on multiple cores or even multiple machines. The *swarm* tool[5] divides a large verification task into smaller tasks that can be run in parallel (Holzmann, Joshi & Groce 2008).

Since increased resources alone will most likely not be sufficient, more support for abstraction methods should be considered and perhaps also Spin's lossy state compression methods that can provide significant memory reduction at the expense of no longer guaranteeing complete coverage.

[5] `http://www.spinroot.com/swarm`

Bibliography

Ackerman, L., Elder, P., Busch, C., Lopez-Mancisidor, A., Kimura, J. & Balaji, R. S. (2008). *Strategic reuse with asset-based development*, IBM/Redbooks, IBM Corp., Riverton, NJ, USA.

Avizienis, A., Laprie, J. C., Randell, B. & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing, *Dependable and Secure Computing, IEEE Transactions on* **1**(1): 11–33.

Beato, M. E., Barrio-Solórzano, M., Cuesta, C. E. & de la Fuente, P. (2005). UML automatic verification tool with formal methods, *Electronic Notes in Theoretical Computer Science* **127**(4): 3–16.

Ben-Ari, M. (2008). *Principles of the Spin Model Checker*, Springer.

Beyer, D., Henzinger, T. A., Jhala, R. & Majumdar, R. (2004). An Eclipse plug-in for model checking, *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC'04)*, pp. 251–255.

Beyer, D., Henzinger, T. A., Jhala, R. & Majumdar, R. (2007). The software model checker Blast, *International Journal on Software Tools for Technology Transfer (STTT)* **9**(5–6): 505–525.

Booch, G., Rumbaugh, J. & Jacobson, I. (2005). *The Unified Modeling Language User Guide*, 2 edn, Addison-Wesley.

Cernosek, G. (2005). A brief history of Eclipse, IBM Software Group, IBM. `http://www.ibm.com/developerworks/rational/library/nov05/cernosek/`, retrieved on March 30, 2009.

Clarke, E. M. (2008). The birth of model checking, *in* Grumberg & Veith (2008), pp. 1–26.

Clarke, E. M., Grumberg, O. & Peled, D. A. (1999). *Model Checking*, MIT Press.

Coffman, E. G., Elphick, M. & Shoshani, A. (1971). System deadlocks, *ACM Computing Surveys* **3**(2): 67–78.

Dijkstra, E. W. (1970). Notes on Structured Programming. `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF`, retrieved on 20 May 2009.

Dijkstra, E. W. (1977). Two starvation-free solutions of a general exclusion problem. `http://www.cs.utexas.edu/users/EWD/ewd06xx/EWD625.PDF`, retrieved on 30 April 2009.

Drusinsky, D. (2006). *Modeling and Verification Using UML Statecharts: a working guide to reactive system design, runtime monitoring, and execution-based model checking*, Newnes.

Dwyer, M. B., Avrunin, G. S. & Corbett, J. C. (1999). Patterns in property specifications for finite-state verification, *ICSE*, pp. 411–420.

Elamkulam, J., Glazberg, Z., Rabinovitz, I., Kowlali, G., Gupta, S. C., Kohli, S., Dattathrani, S. & Macia, C. P. (2006). Detecting design flaws in UML state charts for embedded software, *in* E. Bin, A. Ziv & S. Ur (eds), *Haifa Verification Conference*, Vol. 4383 of *Lecture Notes in Computer Science*, Springer, pp. 109–121.

Emerson, E. A. (2008). The beginning of model checking: A personal perspective, *in* Grumberg & Veith (2008), pp. 27–45.

Grumberg, O. & Veith, H. (eds) (2008). *25 Years of Model Checking — History, Achievements, Perspectives*, Vol. 5000 of *Lecture Notes in Computer Science*, Springer.

Harel, D. (1987). Statecharts: A visual formalism for complex systems, *Science of Computer Programming* **8**(3): 231–274.

Holzmann, G. J. (1997). The model checker SPIN, *IEEE Transactions on Software Engineering* **23**: 279–295.

Holzmann, G. J. (2003). *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley Professional.

Holzmann, G. J., Joshi, R. & Groce, A. (2008). Swarm verification, *ASE*, IEEE, pp. 1–6.

Holzmann, G. J. & Smith, M. H. (1999). Software model checking, *in* J. Wu, S. T. Chanson & Q. Gao (eds), *FORTE*, Vol. 156 of *IFIP Conference Proceedings*, Kluwer, pp. 481–497.

Huth, M. & Ryan, M. (2004). *Logic in Computer Science: Modelling and Reasoning about Systems*, 2nd edn, Cambridge University Press.

Int (1994). Statistical analysis of floating point flaw, White paper, Intel Corporation. `http://support.intel.com/support/processors/pentium/fdiv/wp/`, retrieved on February 12, 2009.

Jussila, T., Dubrovin, J., Junttila, T., Latvala, T. & Porres, I. (2006). Model checking dynamic and hierarchical UML state machines, *MoDeV^{2}a: Model Development, Validation and Verification; 3rd International Workshop*, Genova, Italy, pp. 94–110.

Kamel, M. & Leue, S. (2000). VIP: A visual editor and compiler for v-Promela, *in* S. Graf & M. I. Schwartzbach (eds), *TACAS*, Vol. 1785 of *Lecture Notes in Computer Science*, Springer, pp. 471–486.

Knapp, A. & Wuttke, J. (2007). Model checking of UML 2.0 interactions, *Models in Software Engineering*, Vol. 4364 of *Lecture Notes in Computer Science*, Springer, pp. 42–51.

Lamport, L. (1977). Proving the correctness of multiprocess programs, *IEEE Transactions on Software Engineering* **3**(2): 125–143.

Lamport, L. (1983). What good is temporal logic?, *in* R. E. A. Mason (ed.), *Information Processing 83*, Elsevier Science Publishers B.V. (North-Holland), pp. 657–668.

Leue, S. & Holzmann, G. J. (1999). v-Promela: A visual, object-oriented language for SPIN, *ISORC*, IEEE Computer Society, pp. 14–23.

Leue, S., Mayr, R. & Wei, W. (2004). A scalable incomplete test for the boundedness of UML RT models, *in* K. Jensen & A. Podelski (eds), *TACAS*, Vol. 2988 of *Lecture Notes in Computer Science*, Springer, pp. 327–341.

LIC (2001). Spin commercial license. `http://www.spinroot.com/spin/spin_license.html`, retrieved on 12 May 2009,.

Lilius, J. & Porres Paltor, I. (1999a). vUML: a tool for verifying UML models, *Automated Software Engineering, 1999. 14th IEEE International Conference on.* pp. 255–258.

Lilius, J. & Porres Paltor, I. (1999b). vUML: a tool for verifying UML models, *TUCS Technical Report 272*, Turku Centre for Computer Science.

Manna, Z. & Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems*, Vol. 1, Springer-Verlag.

Mikk, E., Lakhnech, Y., Siegel, M. & Holzmann, G. (1998). Implementing statecharts in PROMELA/SPIN, *Industrial Strength Formal Specification Techniques. Proceedings. 2nd IEEE Workshop on*, pp. 90–101.

NIS (2002). The economic impacts of inadequate infrastructure for software testing, *Planning Report 02-3*, National Institute of Standards and Technology, Program Office Strategic Planning and Economic Analysis Group.

Peled, D., Wilke, T. & Wolper, P. (1995). An algorithmic approach for checking closure properties of ω-regular languages, *Proceedings of CONCUR '96: 7th International Conference on Concurrency Theory*, Springer-Verlag, pp. 596–610.

Rat (2003). *DEV470 Mastering Rational Rose RealTime using C++, Student Manual.* Part # 800-026285-000.

Rotman, J. J. (1998). *Journey into mathematics: an introduction to proofs*, Prentice Hall, Upper Saddle River, N.J.

Schäfer, T., Knapp, A. & Merz, S. (2001). Model checking UML state machines and collaborations, *Electronic Notes in Theoretical Computer Science* **55**(3): 357–369. Workshop on Software Model Checking (in connection with CAV '01).

Selic, B. (1996). Real-time object-oriented modeling (ROOM), *Real-Time and Embedded Technology and Applications Symposium, IEEE* **0**: 214–217. ObjecTime Limited.

Shen, W., Compton, K. & Huggins, J. (2002). A toolset for supporting UML static and dynamic model checking, *Computer Software and Applications Conference, 2002. COMPSAC 2002. 26th Annual International.*, pp. 147–152.

Whalen, M. W., Cofer, D. D., Miller, S. P., Krogh, B. H. & Storm, W. (2007). Integration of formal analysis into a model-based software development process, *in* S. Leue & P. Merino (eds), *FMICS*, Vol. 4916 of *Lecture Notes in Computer Science*, Springer, pp. 68–84.

APPENDIX A

Promela model example

This appendix presents the complete Promela code for a verification model. The original model (see Section 8.4) is the UML-RT model for demonstrating the two properties of primary interest to the department, identified in Section 7.3.

Listing A.1: Promela code for a demonstration capsule model.

```
/*                                                                  */
/* This code was automatically generated by the JET transformation */
/* emxtransform.pml.jet                                             */
/*                                                                  */
/* Model name: unknown                                              */
/* Package name: PropertiesDemonstration                            */
/*                                                                  */

/* Definitions */
#define stateA (
    DemonstrationCapsule@PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateA)
#define stateB (
    DemonstrationCapsule@PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateB)
#define stateC (
    DemonstrationCapsule@PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateC)

/* Options */
#ifndef CHANLEN
#define CHANLEN 1
#endif

/* Protocols */
/* Protocol: DemonstrationCommunication */
typedef DemonstrationCommunication {
  mtype _id
};
mtype = {
  goA
   ,    goB
   ,    goC
```

```
};
#ifdef STIMULUS
/* Producer process for the DemonstrationCommunication signals  */
proctype DemonstrationCommunication_prod (chan com) {
#ifdef XU
  xs com;
#endif
send:
  do
    :: true ->
send_goA:
        com ! goA ;
    :: true ->
send_goB:
        com ! goB ;
    :: true ->
send_goC:
        com ! goC ;
  od
}
/* Consumer process for the DemonstrationCommunication signals */
proctype DemonstrationCommunication_cons (chan com) {
#ifdef XU
  xr com;
#endif
recv:
  do
    :: com ? [goA] -> recv_goA: com ? _ ;
    :: com ? [goB] -> recv_goB: com ? _ ;
    :: com ? [goC] -> recv_goC: com ? _ ;
  od
```

```
}
#endif

/* Capsules */
proctype DemonstrationCapsule (
  chan demoCom
) {
#ifdef XU
  xr demoCom;
#endif
  /* Incoming message variables */
  DemonstrationCommunication demoCom_msg;
  /* Inter-connecting channels for contained capsules */
  /* Local ports */
  /* Initialization of all contained capsules. */
  atomic {
    skip
  };

PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1_initial:
  atomic {
#ifdef VERBOSE
    printf("MSC: _initial\n");
#endif
    if
      /* Take transition Initial (short name) */
      :: true ->
        /* Transition code for transition Initial */
        /* Entry code for state StateA (short name) */
#ifdef VERBOSE
        printf("MSC: StateA \n");
```

```
#endif
        /* Jump to next state StateA (short name) */
        goto PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateA

      /* Dummy guard for states without outgoing transitions */
      :: false
    fi
  }

PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateA:
  atomic {
    if
      /* Acquire message on port demoCom */
      :: demoCom?demoCom_msg ->
        if
          /* Take transition goToA (short name) */
          :: demoCom_msg._id == goA ->
            /* Exit code for state StateA (short name) */
            /* Transition code for transition goToA */
            /* Entry code for state StateA (short name) */
#ifdef VERBOSE
            printf("MSC: StateA \n");
#endif
            /* Jump to next state StateA (short name) */
            goto PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateA

          /* Take transition goToB (short name) */
          :: demoCom_msg._id == goB ->
            /* Exit code for state StateA (short name) */
            /* Transition code for transition goToB */
            /* Entry code for state StateB (short name) */
```

```
#ifdef VERBOSE
                printf("MSC: StateB \n");
#endif
                /* Jump to next state StateB (short name) */
                goto PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateB

             /* Take transition goToC (short name) */
             :: demoCom_msg._id == goC ->
                /* Exit code for state StateA (short name) */
                /* Transition code for transition goToC */
                /* Entry code for state StateC (short name) */
#ifdef VERBOSE
                printf("MSC: StateC \n");
#endif
                /* Jump to next state StateC (short name) */
                goto PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateC

             :: else ->
#ifdef VERBOSE
                printf("MSC: Unhandled signal %e in StateA\n", demoCom_msg._id);
#endif
                assert(false);
                /* Run the pan verifier with '-A' to suppress assertion violations */
                goto PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateA
          fi

      /* Transitions below lack trigger and are therefore disabled by default */
      /* Dummy guard for states without outgoing transitions */
      :: false
    fi
  }
```

```

PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateC:
  atomic {
    if
      /* Acquire message on port demoCom */
      :: demoCom?demoCom_msg ->
         if
            /* Take transition goToC (short name) */
            :: demoCom_msg._id == goC ->
               /* Exit code for state StateC (short name) */
               /* Transition code for transition goToC */
               /* Entry code for state StateC (short name) */
#ifdef VERBOSE
               printf("MSC: StateC \n");
#endif
               /* Jump to next state StateC (short name) */
               goto PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateC

            /* Take transition goToA (short name) */
            :: demoCom_msg._id == goA ->
               /* Exit code for state StateC (short name) */
               /* Transition code for transition goToA */
               /* Entry code for state StateA (short name) */
#ifdef VERBOSE
               printf("MSC: StateA \n");
#endif
               /* Jump to next state StateA (short name) */
               goto PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateA

            /* Take transition goToB (short name) */
            :: demoCom_msg._id == goB ->
            /* Exit code for state StateC (short name) */
            /* Transition code for transition goToB */
            /* Entry code for state StateB (short name) */
#ifdef VERBOSE
            printf("MSC: StateB \n");
#endif
            /* Jump to next state StateB (short name) */
            goto PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateB

          :: else ->
#ifdef VERBOSE
            printf("MSC: Unhandled signal %e in StateC\n", demoCom_msg._id);
#endif
            assert(false);
            /* Run the pan verifier with '-A' to suppress assertion violations */
            goto PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateC
        fi

      /* Transitions below lack trigger and are therefore disabled by default */
      /* Dummy guard for states without outgoing transitions */
      :: false
    fi
  }

PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateB:
  atomic {
    if
      /* Acquire message on port demoCom */
      :: demoCom?demoCom_msg ->
        if
          /* Take transition goToB (short name) */
```

```
                :: demoCom_msg._id == goB ->
                   /* Exit code for state StateB (short name) */
                   /* Transition code for transition goToB */
                   /* Entry code for state StateB (short name) */
#ifdef VERBOSE
                   printf("MSC: StateB \n");
#endif
                   /* Jump to next state StateB (short name) */
                   goto PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateB

                /* Take transition goToA (short name) */
                :: demoCom_msg._id == goA ->
                   /* Exit code for state StateB (short name) */
                   /* Transition code for transition goToA */
                   /* Entry code for state StateB (short name) */
#ifdef VERBOSE
                   printf("MSC: StateB \n");
#endif
                   /* Jump to next state StateB (short name) */
                   goto PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateB

                :: else ->
#ifdef VERBOSE
                   printf("MSC: Unhandled signal %e in StateB\n", demoCom_msg._id);
#endif
                   assert(false);
                   /* Run the pan verifier with '-A' to suppress assertion violations */
                   goto PropertiesDemonstration__DemonstrationCapsule__State_Machine__Region1__StateB
            fi

        /* Transitions below lack trigger and are therefore disabled by default */
```

```
      /* Dummy guard for states without outgoing transitions */
      :: false
    fi
  }
}

init {
#ifdef STIMULUS
  chan DemonstrationCapsule_demoCom = [CHANLEN] of { DemonstrationCommunication } ;
  /* Run top process and required stimulus processes to close the model */
  atomic {
    run DemonstrationCommunication_prod ( DemonstrationCapsule_demoCom ) ;
    run DemonstrationCapsule ( DemonstrationCapsule_demoCom )
  }
#else
  run DemonstrationCapsule ()
#endif
}
```

APPENDIX B

JET transformation templates

This appendix presents two of the templates that are part of the JET transformation project. Listing B.1 contains the top-most template in the transformation project and Listing B.2 contains the template that produces Promela code for representing a state machine.

Listing B.1: Main template for model transformation.

```
<%--
   Main template for model transformation.
   Includes all templates necessary for generating a Promela model.

   Variable dependencies: -
--%>

/*                                                                    */
/* This code was automatically generated by the JET transformation */
/* emxtransform.pml.jet                                               */
/*                                                                    */
/* Model name: <c:get select="/Model/@name" default="unknown"/> */
/* Package name: <c:get select="/Package/@name" default="unknown"/> */
/*                                                                    */

<%-- Generate definitions for Promela model. --%>
<c:include template="templates/definitionTemplates/genDefinitions.pml.jet" />

<%-- Generate extra options for Promela model. --%>
<c:include template="templates/optionTemplates/genOptions.pml.jet" />

<%-- Generate communication protocols for Promela model. --%>
<c:include template="templates/protocolTemplates/genProtocols.pml.jet" />

<%-- Generate all capsule processes for Promela model. --%>
<c:include template="templates/capsuleTemplates/genCapsules.pml.jet" />

<%-- Generate initial process for Promela model. --%>
<c:include template="templates/initTemplates/genInit.pml.jet" />
```

Listing B.2: Template for state machine generation.

```
<%--
  Template for generation of a capsule's state machine.
  Separates the generation of the state machine's pseudostates
  and ordinary states.

  Variable dependencies: capsule
--%>

<c:setVariable select="$capsule/StateMachine" var="statemachine" />

<%-- Pseudostates generation. --%>
<c:iterate select="$statemachine/region/Pseudostate[@kind = 'initial']" var="pseudostate">
  <f:replaceAll value="[^\\w]" replacement="_" regex="true"><c:get select="$pseudostate/owner/
   @qualifiedName" /></f:replaceAll>_initial:
  atomic {
#ifdef VERBOSE
    printf("MSC: _initial\n");
#endif
    if
      /* Take transition <c:get select="$pseudostate/outgoing/@name" /> (short name) */
      :: true ->
         /* Transition code for transition <c:get select="$pseudostate/outgoing/@name" /> */
        <c:get select="$pseudostate/outgoing/effect/@body" default=""/>

        /* Entry code for state <c:get select="$pseudostate/outgoing/target/@name" /> (short name) */
#ifdef VERBOSE
        printf("MSC: <c:get select="$pseudostate/outgoing/target/@name" /> \n");
#endif
        <c:get select="$pseudostate/outgoing/target/entry/@body" default=""/>

```

```
            /* Jump to next state <c:get select=”$pseudostate/outgoing/target/@name” /> (short name) */
            goto <f:replaceAll value=”[^\\w]” replacement=”_” regex=”true”><c:get select=”$pseudostate/
   outgoing/target/@qualifiedName” /></f:replaceAll>
         /* Dummy guard for states without outgoing transitions */
         :: false
      fi
   }
</c:iterate>

<%-- Ordinary states generation. --%>
<c:iterate select=”$statemachine/region/State” var=”state”>
   <f:replaceAll value=”[^\\w]” replacement=”_” regex=”true”><c:get select=”$state/@qualifiedName” /></f:
   replaceAll>:
   atomic {
      if
      <c:iterate select=”$capsule/ownedPort[
         (   stereotype(., 'UMLRealTime::RTPort')/@isConjugate = 'false'
            and count( ./type/owner/CallEvent[stereotype(., 'UMLRealTime::InEvent')])  \> 0
         )
         or
         (   stereotype(., 'UMLRealTime::RTPort')/@isConjugate = 'true'
            and count( ./type/owner/CallEvent[stereotype(., 'UMLRealTime::OutEvent')])  \> 0
         )
         ]” var=”port”>
      /* Acquire message on port <c:get select=”$port/@name” /> */
      :: <c:get select=”$port/@name” />?<c:get select=”$port/@name” />_msg ->
         if
         <c:iterate select=”$state/connectionPoint[./outgoing/trigger/port/@name = $port/@name]” var=”cp”>
            /* Take transition <c:get select=”$cp/outgoing/@name” /> (short name) */
            :: <c:get select=”$port/@name” />_msg._id == <c:get select=”$cp/outgoing/trigger/event/
   operation/@name” /> ->
```

```

          <%-- Generate state/transition action block. --%>
          <c:include template="templates/capsuleTemplates/genActionBlock.pml.jet" passVariables="state, cp" />

      </c:iterate>
        :: else ->
#ifdef VERBOSE
          printf("MSC: Unhandled signal %e in <c:get select="$state/@name" />\n", <c:get select="$port/@name" />_msg._id);
#endif
          assert(false);
          /* For RSARTE behavior simulation:                           */
          /* Run the pan verifier with '-A' to suppress assertion violations */
          /* ignoring the unhandled signal and continuing the execution      */
          goto <f:replaceAll value="[^\\w]" replacement="_" regex="true"><c:get select="$state/@qualifiedName" /></f:replaceAll>
      fi
    </c:iterate>

    <%-- Separate handling of transitions without a trigger. --%>
    /* Transitions below lack trigger and are therefore disabled by default */
    <c:iterate select="$state/connectionPoint[count(./outgoing/trigger) = 0]" var="cp">

      /* Take transition <c:get select="$cp/outgoing/@name" /> (short name) */
      :: false ->

        <%-- Generate state/transition action block. --%>
        <c:include template="templates/capsuleTemplates/genActionBlock.pml.jet" passVariables="state, cp" />
    </c:iterate>
```

```

        /* Dummy guard for states without outgoing transitions */
        :: false

      fi
  }
</c:iterate>

```