Master's Thesis

in Computer Science

Towards an Explicit-State Model Checking Framework

M.A. Kattenbelt

August 2006

Committee

dr. ir. Theo Ruijs
dr. ir. Arend Rensink
prof. dr. ir. Joost-Pieter Katoen

Research Group

Formal Methods and Tools
Faculty of EEMCS
University of Twente

Formal Methods & Tools

University of Twente
Enschede - The Netherlands

Abstract

In the field of formal verification we use model checkers to verify models of systems against a specification. Often these model checkers are specialised for a a limited number of model specification languages, property specifications and verification algorithms. The purpose-built nature of these tools results in poor reusability, and in order to achieve optimisations the source code is generally not modular. As a result new tools are often developed from scratch and interoperability of tools is minimal.

This thesis presents a conceptual architecture for a framework to support these tools. This architecture is based on a layered design. A *generic layer* is to provide generic functionality for the simulation, testing and verification of models. An *abstract layer* provides a model implementation, such that it can use the functionality in the generic layer. Finally, a *tool layer* maps a particular model specification language to the abstract layer. The conceptual architecture has the objective to support reuse of code and to encourage a modular design in tools.

Besides the conceptual architecture, we also introduce a framework in the sense of a library. The design of this libary introduces an implementation of the conceptual architecture for explicit-state model checking. Firstly, a generic layer for explicit-state model checking with generic simulation and verification functionality is discussed. This generic layer uses type abstraction to abstract from the types of *states*, *labels* and *transitions* in the state spaces of the models.

Similarly, an abstract layer is introduced which uses a state representation based on graphs. We use the generic and abstract layer to model check **PROM**$^+$ models. **PROM**$^+$ is based on a minimalistic version of **PROMELA**, but adds the notion of shared pointer variables and dynamic object allocation.

Although the framework offers the flexibility and modularity for which is was made, the required verification time of the tool is significantly slower than **SPIN**. This can be ascribed to some design decisions that were made to make the framework more flexible. In particular this is the use of reference counting pointers and the design of the search module in the generic layer, and expensive operations in the abstract layer, such as linearisation and copying of the state graphs.

However, the statevector size achieved in our tool is generally smaller than those in **SPIN** using default settings and without any optimisations. This implies that potentially our tool is competitive with **SPIN** in terms of memory usage.

Finally, the graph-based state representation enables the exploitation of symmetry reductions in the model. Thread-symmetry is exploited by means of a small change in the state linearisation procedure. This significantly reduces the number of visited states and only poses little overhead.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Interfaces

List of Code Listings

Acknowledgements

First and foremost I would like to thank dr. ir. Theo Ruijs and dr. ir. Arend Rensink for their support over the course of this project. Without their constructive comments and the discussions during the weekly meetings I would not have been able to complete this project to the same standard. It is due to an interesting course by dr. ir. Theo Ruijs and prof. dr. ir. Joost-Pieter Katoen that I got interested in formal methods to begin with, for which I would like to thank them.

I would also like to thank my friends and family, and in particular Amy, for their interest in my work, as well as their supportiveness, encouragements and patience.

Finally, I would like to thank my fellow students of the formal methods and software engineering lab. Many interesting discussions have made this project a pleasant experience.

Chapter 1

Introduction

In the field of software engineering a large part of the development process is spent on testing [45]. Testing helps to identify faults in the implementation of a system, which in turn helps developers to make a system meet its requirements. Because tests typically do not cover all possible system runs, testing does not guarantee the absence of faults [63]. Faults can lead to failures, which are undesirable in any system and intolerable in safety-critical systems such as electronic flight control systems in aircraft or medical systems. For these systems it could be argued that testing alone is not sufficient.

Formal verification, and in particular *model checking*, is a field that complements testing. Rather than ensuring some selective runs of the system behave correctly, the intention of model checking is to formally prove all runs conform to the requirements. Typically model checking is not exercised on the system and requirements directly, but on a *model of the system* and *properties* representing the requirements. Formal verification can be included in the development process to further reduce the number of faults in a system.

Verifying a model enables the ability to abstract from certain aspects of the system that are irrelevant to the property under consideration. Models can usually be expressed as a *state space*, which is a transition system that defines the behaviour of the model. A problem that is frequently encountered is the *state space explosion* [18], which usually refers to the exponential growth of the state space that occurs when a model exists of multiple asynchronous components. The state space explosion makes model checking challenging. A significant amount of research is spent in improving the efficiency of the verification process, developing techniques to reduce the size of the state space and developing other means of enabling the verification of ever more complex models.

Verification tools, or *model checkers*, are the key to applying model checking in practice. This thesis discusses the introduction of a framework for model checkers. The rest of this chapter discusses the domain of model checking, and discusses the objectives of the new framework. Chapter 2 introduces a conceptual design which globally introduces the approach that is used in the framework. Furthermore, this chapter relates this design to existing frameworks. Chapter 3 through 5 introduce an implementation of the conceptual design for explicit-state verification. Finally, chapter 6 presents a case-study which uses the framework to verify **Promela**-like models.

1.1 Formal Methods

Model checking is part of the more general field of formal methods, which is concerned languages, techniques, and tools based on mathematical principles [20]. There are a few reoccurring terms in this field:

System In the context of formal methods a system is usually a software program or a hardware device. The objective of formal verification is to reduce the number of faults in a system, but the system itself is usually too complex to verify. Therefore, verification usually involves models of systems.

Model A model, often denoted by $\mathcal{M}$, formally describes the behaviour of a system. The model can be derived from an existing system, either automatically or manually, but can also be constructed before any system is created. Verification prior to implementation could be applied to ensure a certain design or protocol is correct before any implementation efforts have been completed. Models of systems tend to abstract from some irrelevant aspects of the system, for instance by means of abstract interpretation [23, 22]. The terms *model* and *system* are used pretty much interchangeably throughout this report, but one should bear in mind that in reality model checking typically is performed on models of systems rather than the systems themselves. Different means of formally specifying a model are described in section 1.2.2.

Specification The specification formally describes the requirements of the system. In this report the specification is presumed to be in the form of *properties*, which are described in section 1.2.1.

The field of formal methods consists not only of numerous formal languages and techniques, but also offers many tools to put the formal theory into practice. The functionality of tools in the field of formal methods can be categorised into the categories of *simulation*, *formal testing* and *formal verification*.

1.1.1 Simulation

A simulation is a step-by-step execution of a *model*. This is not only useful to see whether a model is behaving as expected, but is also useful as a means to provide a trace to an erroneous state. Although simulation is not really a field of research within formal methods, it is a useful function to have in tools that work with models. Simulation is useful for quickly finding simple faults in a model. Faults in the model can reflect faults in the system, or could have been introduced during the modelling phase, meaning the model does not accurately reflect the system.

1.1.2 Formal testing

In the context of this document 'testing' should be interpreted as formal conformance testing [63]. Testing assumes the existence of a formal specification and treats a system as a black-box. Selected runs of the system are analysed to see whether the implementation conforms to the specification. The runs are chosen in such a way that if all tests succeed then it is likely that the implementation is correct. Because testing is not exhaustive, testing can't formally guarantee that the implementation conforms to the specification.

1.1.3 Formal verification

A definition of verification from the ISO/IEC 12207 standard is 'confirmation by examination and provision of objective evidence that specified requirements have been fulfilled' [45]. The two main approaches for verification are *theorem proving* and *model checking* [20]. This thesis focuses on model checking.

Theorem proving 'Theorem proving is a technique by which both the system and its desired properties are expressed as formulas in some mathematical logic. [..] Theorem proving is the process of finding a proof of a property from the axioms of the system' [20].

Model checking Model checking is the *formal verification* of a *model* against a *specification*. Where theorem proving can be considered an elegant way of mathematically proving a model behaves correctly, model checking is a more direct approach where every possible run of the model is examined. Model checking uses a verification algorithm to confirm whether the model ($\mathcal{M}$) models the specification (p), which can be denoted as $\mathcal{M} \models p$. The verification algorithm is a systematic process that formally concludes whether $\mathcal{M} \models p$ or $\mathcal{M} \not\models p$. The verification algorithm differs significantly for different types of models and specifications.

Simulation, testing and verification are complementary. Simulation is a way of quickly finding obvious faults early in the design process. Testing and verification should go hand in hand to ensure the implementation conforms to the specification. Verification only proves that a model of a system conforms to the specification, whereas testing makes it probable that the implementation conforms to the model of the system.

Although our main focus in this document is on a framework to support model checking, the functionality required for simulation and formal testing are closely related to this. For instance, the model representation could be the used for simulation purposes as well as model checking.

1.2 Model Checking Domain

The framework presented in this thesis is intended to support model checking tools (e.g. *model checkers*). The intention of this section is to provide an overview of the model checking domain. As ideally a framework for model checking would support the model checking domain as a whole. We start with discussion different means of specifying properties in section 1.2.1 followed by a discussion of model specification languages in 1.2.2 and 1.2.3. Both properties and model specification languages will be related to tools in section 1.2.4.

1.2.1 Property Specification

In the context of model checking a model either conforms to a property or it does not conform to the property, there is no room for ambiguity. This means the specification of the system has to be presented in terms of some formal language rather than a natural language. There are many formalisms in which one could express properties over a system, the most popular of which are *invariants*, **LTL** *properties* and **CTL** *properties*. These properties all are based on *atomic propositions*.

Atomic propositions The most elementary part of properties are *atomic propositions*. For each state of the model these expressions are either true or false. Depending on the type of model such an atomic proposition could be of many different types. Potentially valid atomic propositions are 'the value of global variable y is equal to 1', 'process A is blocked' or 'value v is open'.

Invariants Propositional logic can combine atomic propositions and logical operators to form propositional formulae. For each state of the system these formulae hold or don't hold. These propositional formulae can be used as a means of expressing an *invariant* of a model. Invariants are safety properties that hold if for every state of the model the propositional formula holds. For instance, say there is an atomic proposition d that denotes a desirable state of the model, i.e. 'variable x equals 0'. A possible invariant could be denoted by the trivial propositional formula $\neg d$. This informally would be the property 'variable x never equals 0'. Invariants are considered a very important type of safety properties. A safety property is a property whose violation can be proven by means of a finite trace [1], in contrast to liveness properties. For expressing liveness properties or complex safety properties, temporal logics are required such as **LTL** and **CTL**.

Linear Temporal Logic Invariants are not powerful enough to formulate the more complex specifications one would wish to express over a model. This is mainly due to the fact that it cannot distinguish between states of the model. Temporal logics allow us to do just this. Linear temporal logic (**LTL**) expresses properties over paths. All possible paths of a model consist of a series of states. An **LTL** formula is either true or false for each path of the model, and the **LTL** formula holds for the entire model only if for each possible path of the model the **LTL** formula holds.

In order to express properties over paths, **LTL** has some additional operators compared to propositional logic. Most importantly it has the temporal operators next ($\mathcal{X}$) and until ($\mathcal{U}$). Consider a and b are **LTL** formulae and p is a path of the model under consideration. If a is an atomic proposition then it holds for p if it holds for the first state of p. The formula $\mathcal{X}a$ holds for path p if a holds for the path that omits the first state of p. In order to explain the until operator more terminology needs to be introduced. Consider path p^i to be path p with the first i states removed. The formula $a\,\mathcal{U}\,b$ holds for p if there is a $n \in \mathbb{N}$ such that a holds for $p^0, \ldots, p^{n-1}$ and b holds for p^n.

For convenience one could derive more operators from the given operators, such as eventually ($\mathcal{F}$), globally ($\mathcal{G}$), release ($\mathcal{R}$) and the weak until operator ($\mathcal{W}$). A more concise and more complete reference on the semantics and usage of **LTL** properties can be found in [40]. An example of a valid **LTL** property is $\mathcal{G}(d \rightarrow \mathcal{F}\neg d)$, which informally means 'for each state on the path, if variable x equals 0 then eventually it will not equal 0'. This type of property could be used to guarantee the model does not get stuck in a situation where x is continuously 0.

Computation Tree Logic As **LTL** properties are defined over paths, they cannot express properties that discriminate between paths. Computation tree logic (**CTL**) introduces an existential (E) and an universal (A) operator such that path formulae can be joined to form state formulae. For example, properties such as 'there does not exist a path such that variable x can equal 0' can now be formulated as $\neg E\mathcal{F}d$, which is equivalent to $A\mathcal{G}\neg d$. Because all path subformulea in **CTL** are wrapped with a quantifier, **CTL** properties express properties over states of a model rather than paths. A **CTL** formula holds for a model if it holds for all initial states of the model. A more elaborate discussion of **CTL** can be found in [40].

Invariants, **LTL** and **CTL** properties are arguably the most commonly used methods of expressing properties over systems. But besides these, there are numerous other ways in which one could express the desired behaviour of a system, most of which are adaptations of **LTL** and **CTL**. It is not the goal of this section to provide an exhaustive list of property specifications, but some important alternatives will be discussed briefly.

One aspect which cannot be expressed in **LTL** nor **CTL** is probability. An adaptation of **CTL** called probabilistic computation tree logic (**PCTL**) introduces a probability operator over path subformulae. An example would be $A\,\mathcal{P}_{>.99}(\mathcal{G}\neg p)$ which informally states 'for all paths the probability that $\neg p$ always holds is more than 99%'. Although **PCTL** formulae are syntactically very similar to **CTL**, the verification algorithms that are to used are significantly different. Also it should be taken into account that in order for a **PCTL** formula to be meaningful a model would need to have a notion of probabilities.

All previously defined property specifications cannot handle time requirements very well. The only means of addressing time in **LTL**, **CTL** or **PCTL** is by means of saying one event occurs after another. It is not quite difficult to express properties like 'd occurs within 5 time units'. Continuous stochastic logic (**CSL**) combines the probabilistic features of **PCTL** with an additional *timed-until* operator. An example of a property specified in **CSL** is $A\,\mathcal{P}_{>.99}(\top\mathcal{U}^{<5}d)$ which informally states 'for all paths the probability that d holds within 5 time units is at least 99%'.

A more extensive explanation of both **PCTL** and **CSL** can be found in [50]. Again, it should be noted that there exist many other variations of property specifications.

1.2.2 Model Specification

Similar to properties, system models can be specified in numerous different ways. In theory any means of formally specifying a model could be used for the purpose of model checking. In practice there are some model specifications that are used quite often. In this section some of these formalisms will be described.

Labelled Transition System A labelled transitions system (**LTS**) is arguably the most elementary way of describing a system. A system is defined by means of a set of states (S), a set of labels (L) and a transition relation ($R \subseteq S \times L \times S$). Although usually not included in the definition of **LTS**, one could argue that you also need an initial state ($I \in S$), and in order to check the properties described in section 1.2.1 one would need to be able to evaluate whether an atomic proposition holds in a state. Similar to **LTS** are finite-state automata such as **DFA** and **NFA**, but also Büchi automata and Kripke structures.

Timed Automata Timed automata (**TA**) introduce the notion of clocks. Each model consists of a set of clocks (C), states (S) and labels (L), an initial state ($I \in S$), and a transitions relation

($R \subseteq S \times L \times \mathcal{B}(C) \times \mathcal{P}(C) \times S$) where $\mathcal{B}(C)$ is a set of clock constraints that guard the transitions. Clock constraints are simple expressions on clocks such as 'clock c has a value smaller than 5 time units' ($c < 5$). Transitions of a **TA** are labelled with these constraints. $\mathcal{P}(C)$ is the powerset of clocks, such that each transition is labelled with a set of clocks that is to be reset. Additionally one can label states with invariants ($Y : S \longrightarrow \mathcal{B}(C)$). Clocks in timed automata have rational values which increase as time progresses. More detailed information on **TA** in the context of model checking can be found in [2, 68, 9].

Stochastic Timed Automata Stochastic timed automata (**STA**) are fairly similar to timed automata in the sense that they have clock variables. However, clocks in **STA** do not increase in value as time progresses, but they are triggers that count down. A **STA** consists of triggers (C), states (S), labels (L) and a transition function ($R \subseteq S \times L \times \mathcal{P}(C) \times S$), where $\mathcal{P}(C)$ is the trigger set (the set of triggers that need to be expired for the transition to be enabled). An **STA** also has a clock setting function ($Set : S \longrightarrow \mathcal{P}(C)$) and a distribution function ($F : C \longrightarrow (\mathbb{R} \longrightarrow [0, 1])$). When the **STA** is in state s, all triggers $c \in Set(s)$ are given a value according to their distribution function $F(c)$. This means that **STA** models are both timed and probabilisitic. More information on **STA** can be found in [25, 24].

Continuous-Time Markov Chains Continuous-time Markov chains (**CTMC**) consist of a set of states (S), a set of labels (L) and a transition function ($R \subseteq S \times S \rightarrow [0, \infty)$). The labels no longer are associated with transitions, but they are associated with the states by means of some labelling function ($L' : S \longrightarrow L$). Transisitions in R are paired with a parameter $\lambda \in [0, \infty)$ which is the parameter in a memoryless distribution function $F(x) = \lambda e^{-\lambda x}$. The chance that the transition $s \rightarrow s'$ occurs within t time units is equal to $\int_0^t F(x)dx = 1 - e^{-R(s,s')t}$. More information on **CTMC**s can be found in [3]. There also exists a discrete-time Markov chain variant, **DTMC**, which directly associates each transitions with a probability rather than a distribution function.

Petri nets In a Petri net the transitions are defined different to the models presented so far. A petri net has states (S), transitions (T) and a transition function ($R : (S \times T) \cup (T \times S) \rightarrow \mathbb{N}$). At any one time all states in S have a particular number of tokens associated with them. These tokens make up the marking of the Petri net. If a transition from T is fired, then R helps to redistribute these tokens. For instance if $R(s_1, t_1) = 1$ then if t_1 is fired, the number of tokens in s_1 is decreased by one. Similarly, if $R(t_1, s_2) = 2$ and t_1 is fired, the number of tokens in s_2 is increased by two. The initial marking of a Petri net is given by a function $I : S \longrightarrow \mathbb{N}$. A more detailed discussion on the semantics of Petri nets can be found in [53, 34].

Graph Transition Systems In graph transformation systems (**GTS**) states (S) are graphs. The transitions relation (R) consists of injective graph morphisms on graphs in S. Graph grammers can be used to specify the transitions of a **GTS** in terms of graphs as well. The interested reader is referred to [57] for more information.

The formalisms that have been presented so far are all mathematical models. This is an advantage in the sense that the semantics of these type of models are well-defined. However, as the systems that are modelled grow increasingly complex a more scalable approach can prove profitable. The formalisms that will be presented from here on will be based on textual representations rather than mathematical models, and are comparable to programming languages,. This offers the advantage that abstract concepts such as processes, locks, classes and functions are easily included in a model, but has the disadvantage that the exact semantics of these models is much harder to define and comprehend. It is not uncommon for the semantics of these textual representations to be interpretable as one of the previously defined mathematical formalisms. For instance one could say that the *semantic model* of **MoDeST**-models are stochastic timed automata [24].

In this section only a very general description of these model specification languages will be given, but a comparison will follow in the next section (1.2.3).

NuSMV **NuSMV** is the input language of the **NuSMV** model checker [15]. The language is derived from **SMV** and is basically a definition of a transition relation. It is possible to compose models in a hierarchical fashion by using modules. **SMV** is most suited for modelling synchronous systems, making it a suitable language for specifying models based on hardware.

Murphi The specification language **Murphi** is based on guarded rules. At any one time a rule may execute if its guard is enabled. There are no such things as processes, functions, or other high-level features. **Murphi** is mostly used for protocol verification. An interesting feature of **Murphi** is the support of scalar sets, which enables symmetry reductions during verification (see also [61, 62]).

Promela Process meta language **Promela** is the input language of model checker **SPIN** [36]. Due to the popularity of this tool, **Promela** is a prominent language in the world of software and protocol verification. As the name suggests a **Promela**-model is defined as a set of asynchronous processes. The exact semantics of **Promela** are dictated by the implementation of **SPIN**, although formal semantics have been derived in [67].

MoDeST The semantic model of **MoDeST**-models is based on stochastic timed automata, therefore **MoDeST**-models are probabilistic and timed. The main objective of **MoDeST** is to be able to express many types of submodels [24], such as timed automata and **CTMC**s. Due to the complex nature of this language and its semantic model, building a verification tool for **MoDeST** could prove difficult.

BIR Bandera intermediate language (**BIR**) is the input language of the model checker **Bogor**. **BIR** has a lot of high-level constructs in order to model object-oriented software. In particular it is well suited for modelling **Java** programs. The language **BIR** itself is extensible, its syntax can be extended in **BIR** itself and the semantics can be implemented in **Java**. The formal semantics of **BIR** have not been described in literature, and are presumably fairly complex. See also [27, 60, 59].

Java Obviously **Java** is a programming language, but in the current context it is suggested that **Java**-programs are also models. The **JPF**-tool is capable of verifying and simulating **Java** models, and the java virtual machine (**JVM**) could also be seen as a tool for simulating (e.g. executing) **Java**-models. Not all **Java**-programs are closed systems, as they might require some interaction with the user. In order to exhaustively verify a **Java**-program it needs to be closed.

1.2.3 Comparison of High-Level Model Specification Languages

The high-level model specification languages described in section 1.2.2 have only been discussed briefly. To get more insight into the class of models that can be expressed in these languages, a selection of model features is presented. Which features are present in which model specification language is depicted in table 1.3.

Control-Flow A first category of model features concerns different ways of defining the control-flow in sequential systems. Some basic control-flow constructs such as conditional statements and loops are not considered features, as they are very common. A less common feature is is the ability to *jump* to arbitrary locations in the control-flow, similar to the old-fashion `goto` statement in **BASIC**. Although jumps are not considered very modular from a programming perspective, in modelling languages they can simplify modelling systems with complicated control-flows. *Functions* are a way to easily model a complicated or repetitive control-flow by composing it of smaller control-flow units. An execution unit, called a thread or process, has a stack of functions associated with it in order to keep track of the control-flow. *Exception handling* is a way of abruptly changing the flow of control in case of exceptional circumstances. If such an exceptional circumstance occurs, an exception is thrown down the function stack until there is a function that knows how to handle such a exception. This function will *catch* the exception. Finally, it is considered a feature if a model specification language explicitly allows the composition of a model from multiple *parallel threads*. Note that such threads could execute either synchronically or asynchronically.

Communication As most models allow compositions from parallel components[1], this category

[1]These components are mostly threads or processes, but in the case of **Murphi** they are *guarded rules*, and in **NuSMV** they are referred to as *modules*.

of model-features concerns the communications and synchronisation between those components. An *atomic* construct enforces that a single component is the only component to be active while its control-flow is within such an atomic construct. Atomic statements are useful when modelling systems where mutual exclusion is required, and can significantly reduce de size of the state space to be verified. *Locks* in effect accomplish the same, but in a different way. Where atomic constructs are explicitly part of a specification language, locks are objects which grant access to certain resources. Only one component at a time can gain the lock, through which mutual exclusion is achieved. A method of communications between components is via *channels*, where messages are interchanged amongst components. If the case of a non-buffered rendez-vous channel the components need to synchronise in order to exchange messages.

Data Abstraction The following category of features concerns data abstraction. It is presumed that all major model specification languages support primitives data types and variables, such as integers and booleans. More interesting are *composite types* such as records. These composite types can be composed from a number of other types. Although this is not necessarily very useful by itself, it does make it possible to bundle information that intuitively belongs to the same entity. *Classes* are similar to composite types, but also include functions. A model specification language will be shown as capable of *inheritance* if it is possible to define inheritance relations between data types. A subtype inherits all the fields of all its parent-type. Inheritance can also be defined over classes. Functions of a super-class can be overloaded by sub-classes. A language can model *method overriding* if a model can dynamically resolve the function to execute during run-time, depending on which sub-class the function call was made. Another feature to consider is the ability to define *custom data types*. With this term we do not mean the composite types or classes mentioned previously, but new data types that can be included as if they were supported by the modelling language natively. Finally, a special mention is given to *symmetric data types*. These are sets for which the order of the elements is not interesting, as only symmetry preserving operations are allowed. Scalar sets can be a useful way of defining symmetries in the data of models [43, 42].

Furthermore, it is considered a feature if the model specification language supports the dynamic creation and deletion of *objects* or *threads*. If threads cannot be created dynamically, then every single state of the model consists of the same number of threads, which might not be flexible enough to model some systems. Dynamic object creation is useful for modelling heap-like memory models of programs.

Other features are the inclusion of *assertions* in the model specification. These are in fact part of the property specification rather than the model specification. An assertion is basically an expression that should hold in a certain set of states of the model, dictated by the location at which it is defined in the model. Also, some specification languages have means of modelling *clocks*, or *probabilities*. Table 1.1 shows which features can be found in the modelling languages presented in the previous section.

1.2.4 Overview of Tools

So far the tools belonging to the model checking domain have been mentioned only sporadically. In table 1.2 an overview of tools is given in relation to the property specification and model specification languages presented in previous sections. In this section these tools will be addressed only briefly, whereas in the next section a comparison of these tools will be presented. Again, the selection of the tools that are presented here is not exhaustive and is slightly biased towards explicit-state verification, but it should give the reader some idea of the capabilities of the tools in this field. The scope of this document does not allow a very extensive discussion on the features and techniques in the field of model checking. Besides the brief discussion of the features some relevant literature will be provided. Furthermore, chapter 5 explains how these features see fit in the framework, which might provide some additional insight into the features discussed in this section.

[2]It might be possible to implement symmetric data types by means of a custom type in **BIR**.

Table 1.1 – A comparison of model specification languages based on selected language features.

		Control Flow				Comm.			Data Abstraction						Miscellaneous				
	Semantic Model	Jumps	Function Stacks	Exception Handling	Parallel Threads	Atomic	Locks	Channels	Composite Types	Classes	Inheritance of Data	Method Overriding	Custom Data Types	Symmetric Data Types	Dynamic Data Creation	Dynamic Thread Creation	Assertions	Clocks	Probabilities
NuSMV	**LTS**				✓			✓								✓			
Murphi	**LTS**		✓						✓					✓			✓		
Promela	**LTS**	✓			✓	✓		✓	✓							✓	✓		
MoDeST	**STA**		✓	✓	✓	✓			✓							✓		✓	✓
BIR	**LTS**	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓	2	✓	✓	✓		
Java	**LTS**		✓	✓	✓	✓	✓		✓	✓	✓	✓			✓	✓	✓		

Table 1.2 – A selection of tools put into the context of the previously presented property specifications and model specification languages.

	Simulation	Testing	Verification			
			Invariants	LTL	CTL	PCTL, CSL
Labelled Transition System	**TORX**	**TORX**				
Timed Automata	**UPPAAL**		**UPPAAL**		**UPPAAL**	
Stochastic Timed Automata						
Continuous-Time Markov Chains			**MRMC**			**MRMC**
Petri nets						
Graph Transition Systems	**GROOVE**		**GROOVE**		**GROOVE**	
NuSMV	**NUSMV**		**NUSMV**	**NUSMV**	**NUSMV**	
Murphi	**MURPHI**		**MURPHI**			
Promela	**SPIN**		**SPIN**	**SPIN**		
MoDeST	**MOTOR**					
BIR	**BOGOR**		**BOGOR**	**BOGOR**		
Java	**JPF**		**JPF**			

TorX The **TorX** tool is a tool for simulating and testing labelled transition systems. The nature of a test tool is to check a chosen set of runs of a system to see whether they conform the the specifcation. **TorX** allows models to be specified in **LOTOS**, **SDL** and **Promela** [64].

UPPAAL **UPPAAL** is a tool for the simulation and verification of timed automata (**TA**). Timed automata are directly drawn in a graphical user interface. **UPPAAL** is capable of verifing **CTL** properties. The verification algorithm first translates a **TA** into a zone-automaton (i.e. an **LTS**-like representation in which each state is a zone is which the clock values are similar; see [9]). An interesting feature of **UPPAAL** is the possibility of verifying models in a distributed fashion [7].

MRMC The model checker **MRMC** is a tool for the verification of discrete-time and continuous-time Markov chains as well as Markov reward models. Properties can be specified in a subset of **PCTL** as well as **CSL**. Verification algorithms used in **MRMC** are significantly different to those used for non-probabilistic models [3].

GROOVE **GROOVE** is a tool for the exploration of **GTS** models as defined by graph grammars [58, 57]. Graph grammars allow a representation of states and transition rules in the form of graphs. This arguably is a more natural way of representing systems. Recently, **GROOVE** was extended with the capability of **CTL** model checking [47].

NuSMV The model checker **NuSMV** is different to other tools presented here in the sense that it is oriented toward verifying models based on hardware. It uses a symbolic verification algorithm for the verification of mainly **CTL** properties. It is possible to specify some properties in **LTL**, which are translated to equivalent **CTL** properties by the tool.

Murphi **Murphi** is a tool for the verification of models specified in the **Murphi** language. As this language is based on guarded rules, it is not very suitable for modelling object-oriented systems, but it is well suited for the verification of protocols. As mentioned in section 1.2.2 an important feature of **Murphi** is the support of symmetry reductions by means of scalar sets [43, 42]. The tool supports distributed verification as well as use of a magnetic disk (to increase the size of models that can be checked) [62, 61].

SPIN The **SPIN** model checker is arguably the most popular tool in the field of formal verification. It a tool for verifying **Promela** models against **LTL** properties, and is efficient in doing so. It applies specialistic techniques such as partial-order reductions, bit-state hashing and hash-compaction in order to improve performance, and therefore the size of the models that can be verified with **SPIN** [36]. Also, it is capable of enforcing weak fairness and checking progress requirements. As can be seen in table 1.3 **SPIN** supports a lot of optimisation features. The source-code of **SPIN** is optimised for performance. The downside of this is that the tool is not written in a modular fashion. Reusing and extending **SPIN** is therefore difficult.

MoToR The tool **MoToR** is currently only a simulator for **MoDeST**-models. As part of the **HaaST** project it is the first step towards verification of these complex models.

Bogor **Bogor** is a model checker for models specified in **BIR**. It is an example of a tool that is written in a modular and extensible way. Modules can be written to be used in **Bogor**, but also the **BIR** language can be extended. The correlation between **BIR** and **Java** is very high, making **Bogor** an interesting choice for the verification of object-oriented programs. There exist modules for **Bogor** that perform symmetry reductions as well as partial-order reductions.

JPF **Java Pathfinder** (**JPF**) is a verification tool for **Java** programs. It is useful for finding deadlocks in **Java** programs. It is also possible to define custom properties in **Java** itself. The first version of **JPF** translated **Java** programs into **Promela** models, so that they could be verified by **SPIN**. The current version is built like a virtual machine which explores every possible path of execution. It applies partial-order reduction, state collapsing and hash compaction in order to improve efficiency [36].

1.2.5 Comparison of Verification Tools

Comparing all tools mentioned in section is rather difficult, as they were all built for different purposes. Therefore this section will limit the comparison to the comparison of model checkers. The intention of this section is to provide a list of features that can occur in model checking tools, and should therefore be compatible with the framework. Chapter 5 will discuss how these features can be included in the framework. The comparison is based on the features presented below:

Verification Algorithms There are a few categories of verification algorithms that can be distinguished. The *explicit-state* category applies algorithms in which states are individually represented. Usually the verification procedure is some type of exhaustive search over all states in the state space. Examples of such explicit-state verification algorithms are given in [39, 38].

In *symbolic* verification algorithms states are not individually represented. The algorithm is based on operations on *sets* of states. These sets can be symbolically represented by means of **BDD**s. This approach is useful for model checking **CTL** properties [16]. Finally, there is a distinguishable category called *bounded* model checking. This technique first transforms the state space into a formula in propositional logic. This formula is constructed in such a way that it is only satisfiable if some states of interest are reachable, after which a satisfiability solver determines whether there exists a solution [17]. It is called bounded model checking because only paths upto a particular number of transitions are considered.

Henceforth, our comparison criteria will be biased towards explicit-state verification algorithms.

Search Strategies If a tool uses an *explicit-state* verification algorithm, there is usually a choice as to the order in which states in the state space are visited. The most common choice is to use a *depth-first* traversal strategy. If a violation is found during depth-first traversal, the path to the current state is given by the search stack, which can be used as a path to the erroneous state (e.g. counter-example).

Another possibility is a *breadth-first* traversal strategy. The advantage of a breadth-first exploration is that when a property violation is found, it is guaranteed that the counter-example of this violation is the shortest counter-example possible. However, unlike a depth-first traversal, you do not get this counter-example for free, as there is no longer a search stack to extract the counter-example from. Another disadvantage is that a breadth-first approach is usually more memory intensive.

Finally, the category of *directed* strategies is distinguished. The search algorithms in this category use some heuristic to determine the order of traversal, in the hope of finding property violations sooner. Popular directed search methods are *best-first* and A^* [29, 28, 52].

Most other features in explicit-state verification are a direct result of the *state space explosion* [18]. Tools strife to verify continuously large models. The ways in which they realise this can be categorised into *performance improvements*, *reduction methods* and *approximative methods*.

Performance improvements The capabilities of tools, in both time and memory, is continuously improved upon, which enables the verification of models with increasingly large state spaces. Performance can be improved by means of more efficient algorithms and more efficient storage techniques, but also by finding alternative resources.

State collapsing uses the fact that models are made of of components [35]. Rather than storing each individual state separately, states could share information about the components. If in two global states have a process component in common, which is coincidentally also in the same state, it is sufficient to store this information only once.

State compression attempts to reduce the size of the bitvectors that represent states [35]. A Huffman encoding could be used for this purpose. Some training runs might be necessary to analyse the frequency of bytes in order to optimise the compression.

Rather than storing every single state that was once visited, one could employ a *caching* strategy [33]. The larger the cache, the less likely it is that states are revisited. The cache could store as many states as the memory allows. The risk of caching is that states could be visited more than necessary because they have been deleted from the cache.

If states are represented by vectors of bits, it is possible to use a *minimised* automaton to serve the function of a state store [37, 36]. This automaton is labelled with bit values, and accepts only the bitvectors that represent the states that have been visited.

A technique that is used in the context of symbolic verification algorithms is that of binary decision trees **BDD** [13]. Such a decision tree is capable of storing binary function in a memory-efficient way. As each state is distinguishable by the set of atomic propositions it satisfies, we could interpret the set of atomic propositions as a set of boolean variables. The **BDD** can very efficiently represent a set of states.

Some tools offer the possibility of *distributed verification*. The processing and memory load is shared over multiple processing units. Although this feature can greatly increase the size of models that can be verified, it is hard to devise algorithms that allow the verification to be distributed [51, 61, 6].

Besides using memory for storing the visited states, it is also possible to use *external storage* such as a hard drive. Although the capacity of hard drives is greater than memory, access to the drive is relatively slow. This means using the magnetic disk for verification comes at a performance hit. Special algorithms can be devised to use the disk without too many drawbacks [62].

Reduction methods An active field of research is developing methods to significantly reduce the size of a model's state space, whilst ensuring the verification result remains formally correct.

First of all *partial-order reduction* is a commonly applied technique to reduce the size of the state space. This reduction is based on the commutativeness of transitions. For instance if there are two threads that are about to perform a transition, and those two transitions are do not affect each other, there might be a situation where a reduction can be applied. Rather than verifying all possible orders in which the transitions could execute, it might be sufficient to explore only some orders of execution. This technique is quite complex and dependent on the type of model at hand, therefore the interested reader is referred to chapter 10 of [19] or [32, 55] for more specific information. The inclusion of partial-order reduction in the framework is discussed in section 5.4.1.

Another reduction method is that of *symmetry reduction* [52, 10]. Symmetries within a model are used to reduce the state space of models considerably. If two states are symmetrical, and the property does not distinguish between those states, it is sufficient to explore only one of those states. There are numerous of possible symmetries that can occur within models, depending on the type of model at hand. However, in the context of software-based models one could distinguish *thread-symmetry*, *heap-symmetry* and *symmetric data types*.

First we consider *thread-symmetry* [60]. Consider a dining philosophers system where each philosopher is modelled by a process. One could argue that it doesn't matter which philosopher takes the first step, because if any other philosopher would have made this step the result would be is symmetrical. Two states are equal except for a permutation of processes instances.

Another symmetry that could occur is *heap-symmetry*, which focusses on permutations of heap objects rather than threads [60]. The idea of heap symmetry is that it is usually irrelevant what the identity of an object is, or where it is located in memory, as long as the heap structure is the same.

Finally one could also introduce special data types to help to identify symmetry, such as scalar sets [42]. For instance one could have a scalar set of process identifiers. On such a set one can only perform operations that preserve symmetry. It is useful if one would like to iterate over processes and the order of iteration is not relevant. We will refer to these as *symmetric data types*.

Approximative methods Some techniques abandon the principle of formal correctness for an approximate result. This allows the verification of model checking much larger models. Bit-state hashing and hash-compaction are commonly applied approximative methods [48].

Bit-state hashing or supertrace is such an approximative method. A very large bitvector of is created, and with a hash function each state is mapped to a bit in this vector. A search algorithm uses this vector to see whether it already visited a state instead of a store. This means that there is a small chance that an erroneous state will not be found, because two states can map to the same bit in the bit vector. This means it is possible for the search to wrongly conclude it has visited a state, thereby omitting a portion of the state space in which erroneous states could still reside [48, 36].

Hash-compaction extends the principle of bit-state hashing. Rather than storing the bitvector, a smaller hashtable is used. One hash function is used to find a location in the hashtable, and the result of another hash function is the value that is stored in the hashtable. This is an approximate technique as multiple states could map to the same hash value, and if a hashtable without a linked list is used, some hash values might be discarded [48].

Table 1.3 – A comparison of verification tools. Note that most features of comparison are chosen to compare explicit-state verification, rendering the inclusion of **NuSMV** and **MRMC** rather useless. Features that aren't applicable for a tool are marked by a '- '.

	Algorithm			Search			Performance Improvements							Reduction Methods				Approx.	
	Explicit-State	Symbolic	Bounded	Depth-First	Breadth-First	Directed	State Collapsing	State Compression	Binary Decision Diagrams	Minimised Automata	State Caching	Distributed Verification	Use of External Storage	Parial-Order Reduction	Thread Symmetry	Heap Symmetry	Symmetric Data Types	Bit-State Hashing	Hash-Compaction
UPPAAL	✓			✓	✓			✓	✓										
MRMC				-	-	-	-	-	-	-	-			-	-	-	-	-	-
GROOVE	✓			✓	✓		✓								✓	✓			
NuSMV		✓	✓	-	-	-	-	-	✓	-	-			-	-	-	-	-	-
Murphi	✓			✓	✓						✓	✓	✓				✓	✓	✓
SPIN	✓			✓	✓		✓	✓		✓				✓				✓	✓
Bogor	✓			✓	✓	✓	✓							✓	✓	✓			
JPF	✓			✓		✓						✓		✓				✓	✓

1.3 Problem Statement

As is illustrated by table 1.2 most tools are very specialised, meaning they have been developed with the goal of supporting only a single model specification language and only a small number of different property specifications.

This specialisation enables the code-base of a tool to be *optimised* for a particular purpose and does not encourage a *modular* and *generic* design. As a result, it is usually quite fruitless to attempt to reuse the source code of these tools, and new tools are often written from scratch. Any opportunity to share the functionality of tools is not exploited. Because tools do not share a common code-base the interoperability of tools is quite poor. Interaction between tools can only be achieved with considerable effort.
To address these issues this thesis introduces a *framework* for model checkers, where the meaning of framework is two-fold:

Conceptual architecture A conceptual design of model checking tools which enables the reuse of code and encourages a modular design. This conceptual design is applicable in a wider perspective than just explicit-state model checking. It provides an architecture in which algorithms can be reused for different kinds of models. The conceptual architecture is discussed in chapter 2.

The library A library which has been designed using the principles of the conceptual architecture. On a high level of abstraction this library defines means of simulating and verifying explicit-state models (chapter 3). On a low level of abstraction it provides an implementation of PROMELA-like models (chapters 4, 5 and 6).

Both will be referred to as 'framework' in the remainder of this thesis. The introduction of a framework should help towards improving the previously described scenario in the model checking domain. More specificly, this thesis works towards the following objectives:

- Provide and encourage the creation of generic functionality that can be used across several verification tools. For instance, the framework could provide simulation or verification functionality which can be used for a number of different models. By providing reusable functionality and algorithms in a framework, new tools would no longer have to be build from scratch. The development of new tools would be easier and the resulting code more modular.

- Improvements in the interoperability of tools. If tools would be based on a common state space and model representation, it would not require an extensive effort to make tools cooperate.

Chapter 2

Conceptual Architecture

The previous sections are intended to justify the development of a new framework in the model checking field. In particular section 1.2.4 shows that most tools are very specialised and integration of tools in this field is difficult. Rather than introducing a new black-box framework, which takes input in the form of some complex and expressive intermediate language, this framework should be more open, as is explained in the next few sections.

2.1 Unification of the Model Checking Domain

The main challenge when designing a model checking framework is to find a way to combine all the modelling languages, property specifications and verification algorithms in one single framework in a way such that reuse of code and modular design are possible.

Unfortunately, it is not necessarily the case that all system models share a common representation at a high level of abstraction. For instance probabilistic models have a different state space representation to non-probabilistic models. However, it is possible to identify models types that share an identical representation on a high-level of abstraction, such as those depicted in figure 2.1. When we abstract from the internal structure of states we can describe models of this type as a labelled transition system.

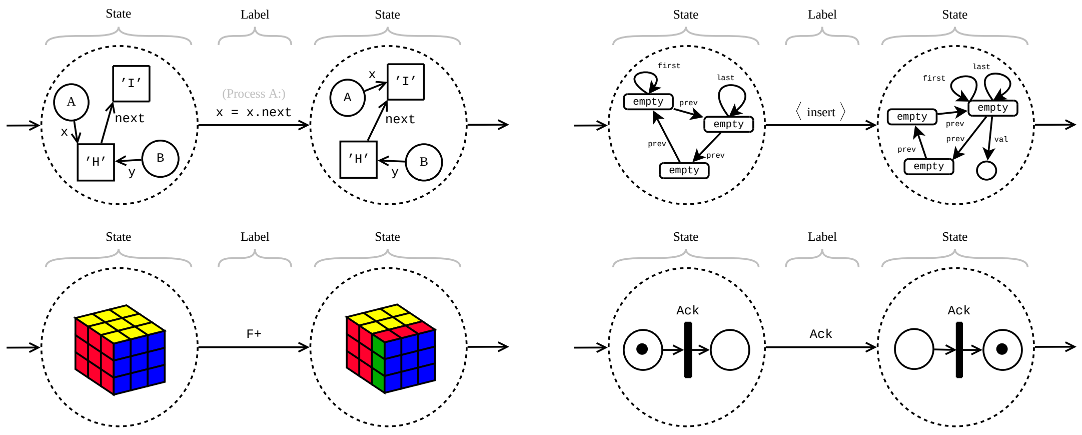


Figure 2.1 – These model types are identical on a high level of abstraction. However, on a lower level the internal structure of states, transitions and labels are totally different. These figures are shown in more detail in appendix C.

All models in figure 2.1 consist of *states*, *labels* and *transitions*. Although the internal structure and interpretation of the models are different, the algorithms that could be used for the simulation, testing and verification of these models are identical. This is where we argue there is an opportunity to provide reusable functionality, in terms of *generic algorithms*, which can be used for multiple model types.

We identify two main approaches that can be used to create a new framework, namely the *black-box approach* and the *layered approach*.

2.1.1 The Black-Box Approach

In general model checkers have one formalism in which the system model has to be defined (e.g. **Bogor** requires models to be specified in **BIR**). One could continue this philosophy when designing a framework. Preferably, one would choose a semantically rich modelling language, such that a number of other model types can be expressed in this language. For example, if one chooses **MoDeST** as the input language for a framework, the framework would also support **CTMC**-models and **TA**-models, as these can be expressed in **MoDeST**.

Whether there exists one particular modelling language in which all other models can be expressed is questionable. Therefore a framework using this approach would probably only support a limited part of the model checking domain. There are a few reasons why one could argue defining a common input formalism is not a suitable approach for a model checking framework:

- By choosing one particular input formalism the framework is limited to a certain class of system models that can be expressed in this formalism. Additionally, even if it is possible to express a model in this formalism this does not necessarily mean this is the optimal way to verify such a model.
- By limiting the framework to a particular input formalism one would also limit the framework to work on a particular level of abstraction. Although it might be possible to express models of a different level of abstraction in the given formalism, this conversion might not be optimal.
- Consider the chosen formalism is quite complex, which is a valid assumption given the fact that the intention is able to express many different types of models in this formalism. Arguably, devising a verification algorithm for such a complex language is difficult.

However, besides these disadvantages, this approach also has some some advantages:

- The required functionality to provide a framework would be straight-forward. By providing simulation, testing and verification functionality for the chosen formalism, the framework would be complete.
- Because the framework would only work with a single formalism the verification procedures can be optimised for this single formalism.

In figure 2.2 the black-box principle is illustrated by means of the model checkers **SPIN**, **Murphi** and **Bogor**. The figure shows that different model checkers require different input formalisms. Note that we omitted the input of property specifications. Converting one formalism to the other is at the least inefficient and undesirable, at the worst this is impossible. Model checkers are designed in this way is because they don't need to be reusable or generic, and optimisation of the tool is easier on a single well-defined input formalism. In principle one could call any model checkers a model checking framework in terms of the black-box approach, which is **Bogor**'s approach [59]. However, **Bogor** does allow a custom implementation of some modules and alterations of its input language (see section 2.2.1).

2.1.2 The Layered Approach

Rather than employing a black-box approach, a framework could employ a more open approach. For instance, the framework could support **Promela**-like models, but at the same time allow other types of models to reuse lower-level parts of the framework where appropriate. The intention of a layered design is that a framework would exploit the similarity of models and provide generic

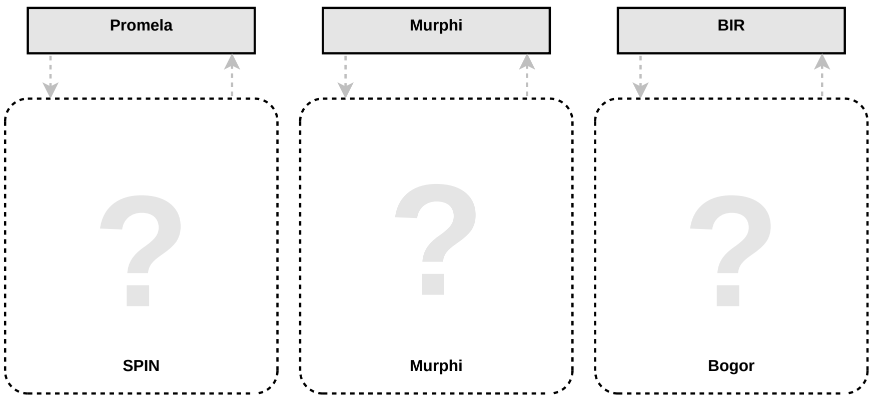


Figure 2.2 – In general model checkers can be considered black-box systems with models as input, specified in some particular formalism, and a verification result as output. In this figure the black-box nature of **SPIN**, **MURPHI** and **BOGOR** is illustrated.

algorithms where possible. Note that the models provided in figure 2.1 is not the only group of models for which one could denote similarities.

Figure 2.3 introduces a layered architecture. The bottom layer represents the model on a high level of abstraction, whereas the top layers are more concrete. Reuse can be accomplished by sharing layers with a high abstraction. We can distinguish three layers, the *generic layer*, the *abstract layer* and the *tool layer*. As is intuitively clear, the generic layer and abstract layer are provided by the framework, and the tool layer is meant for tools that use the framework.

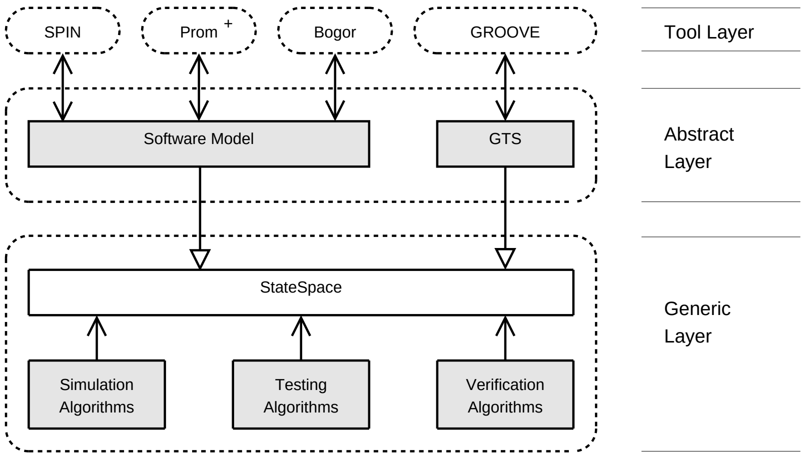


Figure 2.3 – The proposed framework can be divided into three main levels of abstraction: the *generic layer*, the *abstract layer* and the *tool layer*.

Generic layer The generic layer provides algorithms for certain types of models. For instance, one could provide simulation, testing and verification algorithms for the class of models depicted in figure 2.1.

It is not feasible to define just one generic layer for all models, this is due to the diversity of models in the model checking domain. For instance, the fields of *explicit-state*, *symbolic*, *bounded* and *probabilistic* model checking are too different to be encapsulated within the same generic layer, and should probably be defined in separate generic layers.

The most important requirement of such a generic layer is that the algorithms in this layer are oblivious to the abstract layer. Also, the generic layer should provide a means in which model can be defined such that they can use the generic functionality. In figure 2.3 this is the abstract base class `StateSpace`.

Abstract layer The abstract layer is the layer on top of the generic layer. In particular it gives an internal structure to the generic layer. For example, in figure 2.1 where the generic layer was only concerned with models on the level of states, labels and transition, the abstract layer provides the internal structure to these concepts, such as Rubik's cubes or Petri nets for states.

The idea is that it is possible to have multiple abstract layers on the same generic layer such that the algorithms of generic layers are reused.

Tool layer The tool layer is not provided by the framework, it is included in the figure to show how tools could use the framework. In chapter 6 there is small case study of a tool for **PROMELA**-like models, in which the intention of the tool layer should become clear. The nature of the tool layer is that it is not reusable, however, the idea is that a well-defined abstract layer could be used by multiple tools.

The layered approach has a few advantages compared to the black-box approach:

- The framework would not be limited to one input formalism, nor would it be necessary to convert between input formalism.
- Usage of the framework is not limited to a single level of abstraction. As the architecture consists of layers, it is possible to use the functionality provided in the framework at several levels of abstraction.

Besides these advantages, the layered approach also has some disadvantages:

- The required functionality of the framework is not straight-forward. It would be difficult to know which generic and abstract layers are required in order to truly call the framework a model checking framework.

We argue the *layered approach* suits our goals much better than the *black-box* approach. It enables the reuse of algorithms, and it also enforces a modular design due to the modularisation in layers. Note that this chapter only presents a *conceptual architecture* of the model checking framework. The rest of this thesis concerns a possible implementation of this conceptual architecture.

To be more specific, the *generic layer* presented in chapter 3 is based on explicit-state model checking. It provides functionality to support the simulation as well as verification of 'explicit-state models', no testing algorithms are taken into consideration. The *abstract layer* uses a graph-based representation of states to model software-based models in chapter 4. Finally, the *tool layer* uses the generic and abstract layer to verify a **PROMELA**-based specification language called **PROM**$^+$ in chapter 6.

2.2 Related Work

Although chapter 1 explained why a framework for model checking is desirable, there has not been a discussion of other frameworks in this field of research. In this section these frameworks will be introduced. A brief overview of the features of these frameworks will be given, alongside with an explanation how their approach relates to our conceptual design.

2.2.1 Bogor

Besides being a model checker for **BIR**, **BOGOR** is often portrayed as a model-checking framework [59, 27]. This framework chooses **BIR** as the language in which all models are to be specified. Although **BOGOR** could be described as a black-box framework (see section 2.1.1), it does have a few features that provide some additional flexibility:

- **BIR** is an extensible language. It is possible to introduce new language features to extend **BIR**s syntax. More specifically, one can introduce new native types, and define operations on such types.

- The model checker **Bogor** is written in a modular fashion. It defines several interfaces for which custom modules can be implemented. Examples of such interfaces are the `IStateManager` and the `ISchedulingStrategist`.

This approach has several disadvantages, which convince us this architecture is not ideal for a model checking framework. Firstly, the black-box nature of **Bogor** is our main concern. As all input is given in terms of **BIR**, we argue that **Bogor** is only a suitable framework for models that are semantically similar to **BIR**. As the **BIR** specification is fairly similar to object-oriented programming languages such as **Java**, it could be argued that **Bogor** is only suitable for this type of models.

The second issue concerns the flexibility provided by the interfaces. These are custom modules that can be used to implement features. A problem arises if multiple features require the custom implementation of the same module. For instance, in order to implement both symmetry reductions and state collapsing, the state store module, or `IStoreManager`, requires a custom implementation. With two independent features there are 4 possible configurations in which the verification can be executed. It is undesirable to implement a custom implementation of the module for each configuration.

2.2.2 Concurrency Workbench

In article [21] a framework for verifying concurrent systems is described. It is a combination of a toolkit called the concurrency workbench (**CWB-NC**) and a process algebra compiler (**PAC-NC**). In this section this framework will simply be referred to as 'the concurreny workbench'. This framework assumes a language has well-defined semantics in the form of structured operational semantics (**SOS**). These rules could be compared to inference rules which describe a transitions of the system [21]. The process algebra compiler is used to generate a parser from a set of **SOS**-rules. This parser can parse a model specification and generate a **LTS** that describes the semantics of the model. The concurrency workbench toolkit is used to verify the **LTS**.

In effect this framework is language-independent, which could be considered a feature as well as a shortcoming. The advantage is that if the semantics of some model specification language are available in **SOS**, then by giving the framework these rules we should in theory be able to verify this type of models. This would involve minimal effort, and provides a great deal of flexibility.

The disadvantage is that not all languages have formally defined semantics. For a specification language like **Promela**, it is very difficult to construct **SOS**-rules [67]. We argue it is fair to say that providing these rules is not always trivial.

2.2.3 Model Checking Kit

The Model-Checking Kit of the University of Stuttgart uses a layered approach [54]. This is similar to the approach described in section 2.1.2, in the sense that specification languages are mapped to a generic layer by means of abstraction. The verification algorithms work on this generic layer, which is based on 1-safe Petri-nets. These Petri nets could be seen as a very basic semantic model to model concurrent systems, similar to **LTS** [34]. Properties in the Model Checking Kit are also described in terms of Petri nets.

The main feature of the Model Checking Kit is its layered design. Algorithms work on a low-level of abstraction, making them reusable across different model specification languages. The Model Checking Kit is totally based on 1-safe Petri-nets. Rather than using generics in such a way that algorithms are oblivious to the internal structure of the models, all models are expressed using the same Petri-net representation. By using type abstraction to abstract from the internal structure of state objects the internal structure could still be preserved and used in specialised parts of the framework. For instance, if we have a complex specification language that is verified by this framework, than by mapping the model to the Petri-net representation would mean losing internal information. This information can no longer be used when writing counter-examples or storing states in a store.

2.2.4 IF Toolkit

The **IF** toolkit is another framework for the formal verification of models [12]. The principle of the **IF** Toolset is based around a three layers of abstraction, which is similar to our philosophy. The lowest layer is based on **LTS**, which is similar to our generic layer in the sense that simulation and verification are performed on this level. The second layer is one based the intermediate description language (**IF**). **IF** is a rich modelling language that includes notions of time, processes, channels and other high-level features. The idea is that this layer provides a broad range of functionality to support the **IF** language, such as slicing algorithms, static analysis tools, and functionality to export to languages such as **Promela**. The third layer is similar to our tool layer, in the sense that custom languages are supported by translation to the **IF** representation of the second layer.

However, there are some shortcomings to be mentioned about this approach. Most importantly these concern the choice of a single formalism representation of models in the second layer. It is curious how the second layer requires models to be represented in **IF**. Although the architecture of the **IF** toolkit is designed according to the layered approach, the choice of a common representation in the form of a single formalism is more in line with the black-box approach, and seems to infer some limitations. The principle behind the **IF** toolkit is that new specification languages are supported by means of a tranformation to the second layer, and thus an **IF** specification. This transformation could be difficult if the original source language is not closely related to **IF**. Also it should be noted that the **IF** toolkit does not use type abstraction to abstract from the internal structure of states.

Chapter 3

The Generic Layer

This chapter is devoted to the generic layer. Firstly, we introduce a method of specifying a model to be used by this layer, then we provide functionality that uses this model. As this framework will focus on explicit-state verification, we will provide a state space interface for the class of models associated with this type of verification, as well as simulation and verification algorithms. Note that explicit-state verification methods are typically simple exhaustive searches over the state space.

3.1 State Space Interface

To start with, the generic layer needs information about the model under investigation. The models we are targeting in this layer are the models that can be verified with explicit-state verification techniques. In these models, of which a few examples were presented in figure 2.1, we can distinguish the notion of *states*, *labels* and *transitions*. The combination of these items is called a *state space*. It is presumed that models that use the functionality of this layer have a state space. The formal definition of a state space is as follows:

Definition 3.1. *A statespace is a tuple $\langle L, S, I, R\rangle$, where L is a set of labels, S is a set of states, $I \in S$ is the initial state and R is a* collection *of transitions. Each transition in R consists of at least a source and target state from S, and is labelled with a label from L.*

Note that R is deliberately not limited to a subset of $S \times L \times S$. For instance, a model have two transitions with the same source state, the same target state and the same label, but have some additional information like clock guards or probabilities that distinguish the transitions. Although this information might not need to be taken into account, a simple cartesian product might be more limiting than necessary. Note that the state space definition is very similar to that of a labelled transition system (**LTS**) and is therefore especially suitable for abstract layers based on model specification languages that have this as their semantic model (see also table 1.3). There are a number of ways in which one could provide the information encapsulated in the formal definition of a state space. The first choice is between an 'explicit representation' and a 'symbolic representation':

Explicit representation A representation of the state space which contains information about individual states, labels and transitions is called an explicit representation. This approach is most suitable for explicit-state model checking algorithms, which are mostly exhaustive searches of the state space.

Symbolic representation A symbolic representation of the state space is a representation where the state space interface provides information in terms of sets of states and transitions rather than individual states. These sets can be represented symbolically by for instance **BDD**s.

Because the generic layer focusses on explicit-state model checking, the explicit representation is an obvious choice. Given this choice, one could present information about the state space in a static fashion, or on-the-fly:

Static approach A static approach implies the programmatic interface is a direct translation of the state space formalism. The complete sets of states, labels and transitions can be retrieved as a whole. This approach is static because if verification is to be performed on such a model then first *all* states, labels and transitions will have to be computed.

On-the-fly approach The interface provides the state space information piece-by-piece. Rather than providing all states simultaneously, one can determine the successors of a given state. The reachable state space can be explored in an iterative fashion. Application of the on-the-fly approach often means that states are generated dynamically only when they are required.

It is usually preferable to use the on-the-fly approach, in most cases the static approach provides more information than necessary. For instance when simulating or verifying a state space, there is not always a need to visit all of the state space. An on-the-fly approach might provide some performance benefits in this case. In case an algorithm needs a static representation this can always still be reconstructed from the on-the-fly interface, by simply exhaustively exploring the state space.

3.1.1 Formal Interface of a State Space

There is need for an interface that uses a explicit-state representation and presents information about the state space in an on-the-fly manner. Interface 3.1 provides the required functionality:

$$\begin{aligned}
InitialState &: S \\
FirstTransition &: S \longrightarrow R \cup \{\epsilon\} \\
NextTransition &: R \longrightarrow R \cup \{\epsilon\} \\
Source &: R \longrightarrow S \\
Label &: R \longrightarrow L \\
Target &: R \longrightarrow S
\end{aligned}$$

Interface 3.1 – StateSpace

The StateSpace abstracts from the fact that states (S), labels (L) and transitions (R) have a different internal structure for different types of models. This allows us to write algorithms in readable pseudo-code for these interfaces. The fact that we do not know the internal structure of states, labels and transitions does come at a price; we cannot assume anything about these objects. For instance we cannot assume transitions in R have probabilities associated with them. If we do need probabilities for some algorithm we would need to provide these through some other interface with a function ($Probability : R \longrightarrow [0,1]$).

There might be a few functions in the StateSpace interface that leave room for discussion:

- By means of $InitialState$ a state space has exactly one initial state associated with it. It could be argued that some models contain multiple initial states. However, by adding a new artificial initial state with transitions to the original initial states one could solve this problem.
- The $FirstTransition$ and $NextTransition$ function could be perceived as an overly complicated method of retrieving the set of outgoing transitions from some state in the state space. Given a state $s \in S$ the function $FirstTransition$ gives the first outgoing transitions of state s, or ϵ if s has no outgoing transitions. Given a transition $r \in R$, $NextTransition$ retrieves the next outgoing transition with the same source, or ϵ if there aren't any. Note that this method implies the presence of a total order over outgoing transitions of each state in the state space. This means there could be models we exclude due to these methods.

  One could replace these methods by a single method $Transitions : S \longrightarrow \mathcal{P}(R)$. However, in accordance with the on-the-fly principle it makes more sense to provide as little information as possible. This could dramatically improve performance; for instance a depth-first search does not continuously need all outgoing transitions of a state space, the use of the $FirstTransition$ and $NextTransition$ is much more efficient.

- Finally, it could be argued that $Source$, $Label$ and $Target$ should be part of transitions themselves. However, it is preferable to not assume anything about the internal structure of transitions, and thus we use this method instead. This also means that the state space of the model can be retrieved by means of one single entity, namely the STATESPACE interface.

As interfaces in this document are just a convenient method of denoting some programmatic concept, the next section will show how the STATESPACE interface is represented in the framework.

3.1.2 Programmatic Interface of a State Space

An interface such as STATESPACE denotes nothing more than an abstract base class. Each 'function' in the interface maps to an abstract function in the class. By implementing these functions in a subclass, one can provide an implementation of a state space. This is exactly what the abstract layer does, as was shown in figure 2.3.

However, there is one important aspect that needs to be addressed. In the previous section we abstracted over the internal structure of states, labels and transitions. This principle needs to be mapped to the programmatic interface of the state space. There are two methods with which one could accomplish this:

Inheritance One could introduce base classes for states, labels and transitions, and define the functions in the state space interface on these types. Algorithms would be generic in the sense that they work with a particular class of states, labels and transitions. Models that implement the state space interface would define subclasses of these states, labels and transition to provide internal structure to these concepts. The algorithms could be used on models only if their states, label and transition classes are derived from those of the algorithm.

Note that this is much more flexible than is required, as during the lifetime of a tool all states used within that tool would most likely be of the same type. Also, the memory footprint of using inheritance for elementary objects such as states and labels, and the runtime overhead of continuous casting is unnecessary.

Type abstraction Another option is to abstract from the types of states, labels and transitions by means of generics. Implementations of the state space interface would have to instantiate the interface for particular type of states, labels and transitions. The advantage is that there is no additional overhead of run-time polymorphism, whereas the disadvantage is that 'generic algorithms' also have to abstract from types of states, labels and transitions by means of generics. Once a generic algorithm is instantiated for a particular model at compile-time, this algorithm should not sacrifice any performance compared to a specialised algorithm for a particular type of model.

Because model checking is a field where performance is essential, the decision is to implement the state space interface by means of type abstraction. In figure 3.1 the abstract base class representing the STATESPACE interface is presented.

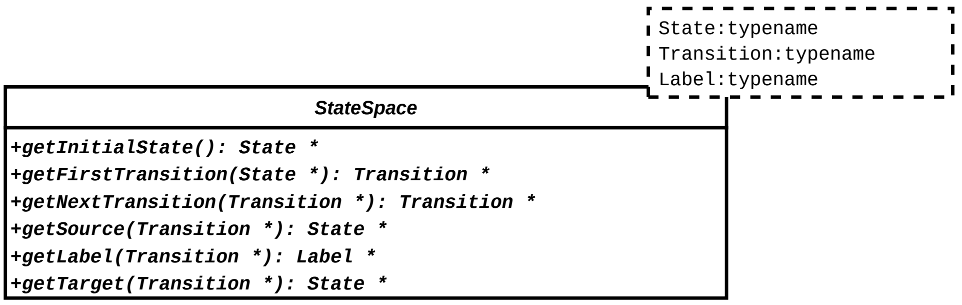

State:typename
Transition:typename
Label:typename
StateSpace
+getInitialState(): State *
+getFirstTransition(State *): Transition *
+getNextTransition(Transition *): Transition *
+getSource(Transition *): State *
+getLabel(Transition *): Label *
+getTarget(Transition *): State *

Figure 3.1 – A class representing the STATESPACE interface, presented in UML. Each function in the interface maps to an abstract function in the abstract base class.

Finally, it should be noted that this class is implemented in a slightly different way than presented. Instead of defining the functions in terms of normal pointers, in reality the implementation uses a pointer with reference counting. In programming languages without garbage collection, some entity should be appointed as responsible for managing the different state, label and transition instances. To provide a more flexible approach we use reference counting pointers.

3.1.3 Providing Generic Functionality

The STATESPACE interface is the means by which abstract layers can provide information about a model. The generic layer uses this information to provide generic functionality such as simulation and verification algorithms. The notation 'STATESPACE' denotes the definition of an interface, whereas '$[\![StateSpace]\!]$' denotes an implementation of this interface. An example of an algorithm that uses an implementation of STATESPACE is algorithm 3.1. Algorithms are also just a way of formally denoting a programmatic concept, in this case an algorithm somewhere in the framework.

Algorithm 3.1: $Transitions(s')$

```
Require: $[\![StateSpace]\!]$
1: $R' \leftarrow \emptyset$
2: $r' \leftarrow [\![StateSpace]\!].FirstTransition(s')$
3:
4: while $(r' \neq \epsilon)$ do
5:     $R' \leftarrow R' \cup \{r'\}$
6:     $r' \leftarrow [\![StateSpace]\!].NextTransition(r')$
7: end while
8:
9: return $R'$
```

Algorithm 3.1 is a function that retrieves all outgoing transitions of a state, and is added as a function in the `StateSpace` class. It could be seen as an additional function in the STATESPACE interface ($Transitions : S \longrightarrow \mathcal{P}(R)$). Arguably, the notation of interfaces and algorithms is quite limited, however, it does provide a useful abstraction for denoting algorithms compared to simply providing source-code. The context of the interfaces and algorithms will be provided in the text.

3.2 Generic Simulation

A simulation of a model can be seen as the exploration of it's state space. In order to achieve a modular design of the simulation module, we introduce two new interfaces, SIMULATIONSTRATEGY and SIMULATIONOBSERVER.

3.2.1 Simulation Strategy

In general we distinguish *interactive*, *random* and *guided* simulations. During an interactive simulation the user decides which execution-path to follow. If at any time during the execution of the model there is more than one way to proceed then the user is prompted to state his preference. A random simulation follows a random execution-path of the model. Random simulation is useful to confirm if the model behaves as expected. A guided simulation follows a predefined execution-path. This path could be a trace from the initial state to an erroneous state, which could be the counter-example of a failed verification attempt.

We consider these to be strategies that guide a simulation through the state space. This is something that is essentially independent to other parts of the simulation module and is therefore it is a good candidate to be implemented by means of a strategy pattern. The interface of SIMULATIONSTRATEGY consists of a single function $ChooseTransition$.

Given a set of transitions, the $ChooseTransition$ function determines which transition to take. Obviously the set of transitions cannot be $\emptyset$. As is intuitively clear, the set of transitions all share a

$$ChooseTransition : \mathcal{P}(R) \longrightarrow R \cup \{\epsilon\}$$

Interface 3.2 – SIMULATIONSTRATEGY.

common source state. It is possible that ϵ is returned which indicates the simulation should finish. The SIMULATIONSTRATEGY is consulted throughout the simulation and determines the path the simulation will take through the state space.

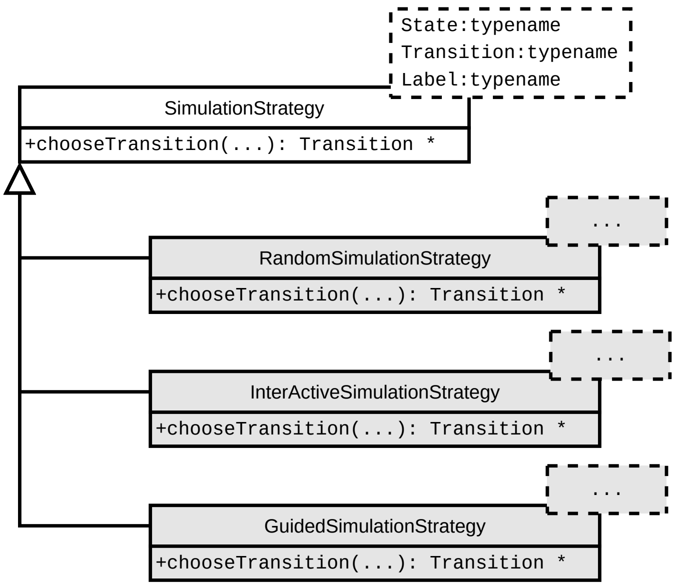

State:typename
Transition:typename
Label:typename
SimulationStrategy
+chooseTransition(...): Transition *
...
RandomSimulationStrategy
+chooseTransition(...): Transition *
...
InterActiveSimulationStrategy
+chooseTransition(...): Transition *
...
GuidedSimulationStrategy
+chooseTransition(...): Transition *

Figure 3.2 – The interface SIMULATIONSTRATEGY with three implementations, depicted as they see fit in the framework.

As is shown in figure 3.2, each type of strategy coincides with an implementation of SIMULATIONSTRATEGY. The `InterActiveSimulationStrategy` and the `GuidedSimulationStrategy` will need require additional information to realise their functionality, whereas the `RandomSimulationStrategy` does not.

3.2.2 Simulation Observer

To encourage a modular design, the simulation module also provides an interface for observation. This information could be used to print information about the progress of the simulation to the console, or to update a graphical user interface. The SIMULATIONOBSERVER interface consists of a set of procedures, rather than functions, and therefore has no influence on the simulation algorithm itself.

$$\begin{aligned} SimulationStarted &: () \\ TransitionTaken &: (R) \\ SimulationEnded &: () \end{aligned}$$

Interface 3.3 – SIMULATIONOBSERVER.

An observer of a simulation is notified when the simulations starts and when it ends, as well as each time a transition is taken. One could argue that in order to do anything useful as an observer, you need to know what type of state, label and transition is being used. Fortunately, as interfaces are simply generic abstract base classes it is possible to specialise an interface to work on just a particular type of model, as is shown in figure 3.3.

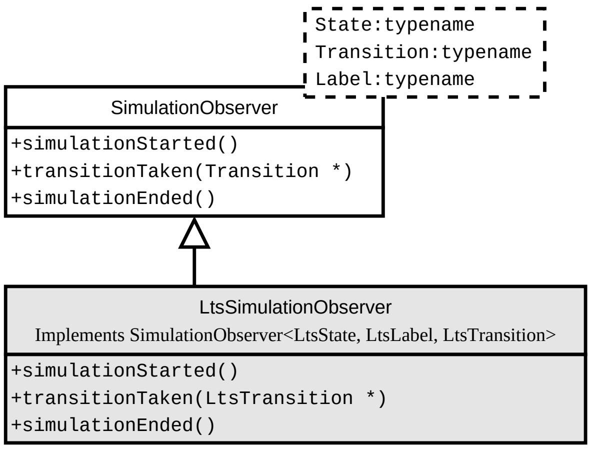


Figure 3.3 – The SIMULATIONOBSERVER interface with a specialised implementation for **LTS** models.

3.2.3 Simulation Algorithm

The simulation algorithm is given in algorithm 3.2. Note that this algorithm requires a $[\![StateSpace]\!]$, $[\![SimulationStrategy]\!]$ and $[\![SimulationObserver]\!]$, all instantiated on the same type of states, labels and transitions. One might argue that simulation algorithm 3.2 is quite basic, because it does not take into account probabilities or priorities or other advanced features. However, one could easily define a specialised $[\![SimulationStrategy]\!]$ that does take into account this additional information.

Algorithm 3.2: Simulation algorithm

```
Require: $[\![StateSpace]\!]$, $[\![SimulationStrategy]\!]$, $[\![SimulationObserver]\!]$.
 1: $[\![SimulationObserver]\!].SimulationStarted()$
 2:
 3: $s' \leftarrow [\![StateSpace]\!].InitialState()$
 4: $R' \leftarrow [\![StateSpace]\!].Transitions(s')$
 5:
 6: while $(R' \neq \emptyset)$ do
 7:    $r' \leftarrow [\![SimulationStrategy]\!].ChooseTransition(R')$
 8:
 9:    if $(r' \neq \epsilon)$ then
10:       $[\![SimulationObserver]\!].TransitionTaken(r')$
11:       $s' \leftarrow [\![StateSpace]\!].Target(r')$
12:       $R' \leftarrow Transitions(s')$
13:    else
14:       $R' \leftarrow \emptyset$
15:    end if
16: end while
17:
18: $[\![SimulationObserver]\!].SimulationEnded()$
```

Algorithm 3.2 is implemented in a class `Simulation`, which uses an implementation of $[\![StateSpace]\!]$, $[\![SimulationStrategy]\!]$ and $[\![SimulationObserver]\!]$. This could be seen as an application of the strategy pattern, as can be seen in figure 3.4.

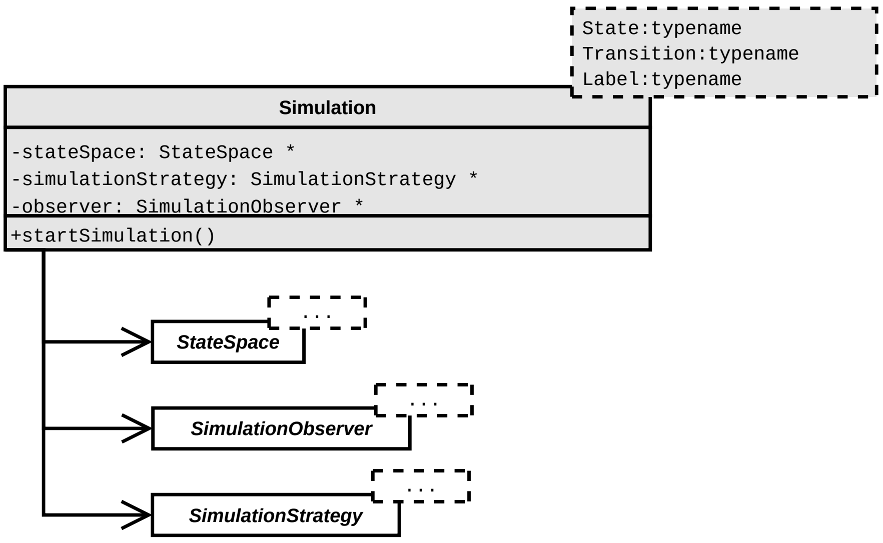


Figure 3.4 – An overview of the generic simulation module, which combines the functionality of $[\![StateSpace]\!]$, $[\![SimulationStrategy]\!]$ and $[\![SimulationObserver]\!]$.

3.3 Generic Search

Verification of a model that can be expressed as a state space is an exhaustive search of the state space. One could argue that even **LTL** model checking is basically searching for acceptance cycles in the state space of the synchronous product of the the Büchi automata representing the model and the negation of the property. There are a number of ways in which a search module could be provided. In particular there is always a trade-off between the flexibility of the functionality provided by module and the performance of the module. The search functionality should provide at least the following possibilities:

- Providing different search strategies, such as depth-first, breadth-first and directed search. It is essential for an explicit-state model checking framework to be capable of providing multiple strategies of state space traversal.
- Providing the ability to perform *nested* searches, such as nested depth-first search [38, 39]. Nested searches are particularly useful for searching for acceptance cycles.
- Providing the ability to search for a number of different goals, such as accepting states, deadlock states, back-edges [4].

Which ever solution is used, there are a few requirements that are to be taken into consideration. Ideally, the solution should be modular, flexible and not compromise on performance.

There are two approaches we will take into consideration. The first is is a search module consisting of multiple solutions, each of which is dedicated to provide a particular part of the required functionality. The second is a more general design which provides all functionality in one solution.

Dedicated solutions For each desired type of search (e.g. depth-first search, breadth-first search, directed-search, nested depth-first search, . . .) we could introduce a dedicated solution. The dedicated design for a simple depth-first is described in appendix B, which is similar to the depth-first algorithms presented in literature [38, 39, 10, 19]. Although a dedicated solution for each search type could allow for an optimisation for the particular algorithm, this design is not very flexible. The search module would consist of several optimised solutions for different search types, and code reuse would be minimal, even though the functionality of those solutions might be very similar.

Modular solution Alternatively, we could design a search module that is flexible enough to take into account all search types, at the cost of some performance overhead. A design which is capable of providing all required functionality is presented in this section.

The choice between the different alternatives depends on the sacrifices in performance one is willing to make to benefit a modular design. As the focus of this thesis is on modularity we choose the 'modular solution'.

To illustrate the need for a modular design, the search method in [4] is discussed. This method attempts to find acceptance cycles (cycles with at least one accepting state), as is common in the context of **LTL** model checking [38, 39]. The algorithm uses a breadth-first traversal method for finding 'back-edges' in the state space, which are transitions of which the target state is of a lower level than the source state. The level of a state is the iteration in which a breadth-first traversal first visits the state. The general idea is that each cycle, and thus each acceptance cycle contains a back-edge. For each back-edge a depth-first search is started to see whether it is part of an acceptance cycle. Note that to realise this functionality, one would need to be able to target transitions (e.g. back-edges) as well as states (accepting states), and it has to be possible to start a nested search of a different strategy.

Similar issues arise when we try to find acceptance cycles with a nested depth-first search as it is described in [38, 36, 19]. The intention of a nested depth-first search is also to find acceptance cycles. The initial search searches for accepting states, and a nested depth-first search is started from accepting states after all its successors were visited. This nested search which will look for a cycle. This requirement implies that there are different kinds of events within search strategies that trigger certain events.

The next section will introduce a simple design which provides a high degree of flexibility.

3.3.1 Search Feedback

To encourage a more modular design, we separate the function of search strategies. Strategies dictate the order of traversal of the state space, but abstract from things like goal states and state stores. Obviously, the need to identify goal states and storing the visited state space is still there, this is provided by the SEARCHFEEDBACK interface.

During some events in the search strategy algorithm it will require some feedback. For instance when its just explored a state, it would be useful to know whether this is a goal state, or whether its been visited before. The set of all events that require 'feedback' is denoted by E. Obviously, the type of events that occur during a search depends on the search strategy used. Also the feedback that it returns depends on the event. The set of all feedback types is called F.

$$ProvideFeedback : (E) \longrightarrow F$$

Interface 3.4 – SEARCHFEEDBACK.

To provide useful feedback the information present in the event from E is not sufficient. It is likely that the implementation of SEARCHFEEDBACK will need to query the search strategy in order to extract more information.

The general idea is that a search strategy consults its feedback at every important event that occurs. In table 3.1 a small overview of the interaction between the search strategy and SEARCH-FEEDBACK is given. How an implementation of SEARCHFEEDBACK realises the required functionality will be explained in section 3.3.3, for now it is sufficient to assume the `SearchFeedback` as an abstract base class.

3.3.2 Search Strategies

In figure 3.5 it is shown how search strategies use the SEARCHFEEDBACK interface to perform a search. This section will focus on the `DepthFirstStrategy`, but a similar discussion could discuss a breadth-first or heuristic strategy.

Table 3.1 – Interaction between SEARCHFEEDBACK and `SearchStrategy`. A `SearchStrategy` triggers certain events and expects SEARCHFEEDBACK to provide appropriate feedback.

Strategy	Event (E)	Expected Feedback (F)
Depth-First Strategy	**EXPLORE**	**NEW**, **SKIP**, **GOAL**
	BACKTRACK	**CONTINUE**, **GOAL**
Breadth-First Strategy	**EXPLORE**	**NEW**, **SKIP**, **GOAL**
Best-First Strategy	**EXPLORE**	**NEW**, **SKIP**, **GOAL**

As shown in table 3.1, a depth-first strategy has two events, namely **EXPLORE** and **BACKTRACK**. An **EXPLORE** event occurs when a new step is explored in the state space. This new step is a state of which the strategy is unsure whether it is a goal state (**GOAL**), whether it is a state that was previously visited (**SKIP**), or a new state (**NEW**). The SEARCHFEEDBACK interface is responsible for providing this information, and to help the SEARCHFEEDBACK interface, the `DepthFirstStrategy` has a few functions to retrieve the current state and transition. Note that the **EXPLORE** event is likely to be the same for all possible strategies.

The **BACKTRACK** event occurs just after the depth-first algorithm pops a transition from the stack to start back-tracking. This event is useful for starting a nested search. Possible feedback is that the algorithm can continue normally (**CONTINUE**), or that the goal state was found (**GOAL**).

Similar to the iterative depth-first search algorithm B.3, a depth-first strategy is defined in algorithm 3.5. This strategy has two 'helper' functions for exploring (algorithm 3.3) and backtracking (algorithm 3.4). The `getCurrentState` and `getCurrentTransition` in figure 3.5 are simply getter functions for the $currentState$ and $currentTransitions$ variables in algorithm 3.5. The most important difference to the iterative depth-first search is that detection of goal states and detecting previously visited states is done simultaniously, using the functionality of $[\![SearchFeedback]\!]$. The search strategy is shown in algorithm 3.5 on page 30. This algorithm uses a $[\![StateSpace]\!]$, $[\![SearchFeedback]\!]$ and a $[\![Stack]\!]$, which is simply a stack of transitions with two functions $Push : (R)$ and $Pop : R \cup \{\epsilon\}$ (see also appendix B).

Algorithm 3.3: $TryExploreStep(s \in S, r \in R \cup \{\epsilon\})$

```
$currentState \leftarrow s$
$currentTransition \leftarrow r$

$feedback \leftarrow [\![SearchFeedback]\!].ProvideFeedback($EXPLORE$)$

{* Note that SKIP is ignored. *}

if ($feedback = $ GOAL) then
    if ($r \neq \epsilon$) then
        $[\![Stack]\!].Push(r)$
    end if
    $foundTarget \leftarrow$ TRUE
else if ($feedback = $ NEW) then
    if ($r \neq \epsilon$) then
        $[\![Stack]\!].Push(r)$
    end if
    $foundNewState \leftarrow$ TRUE
end if
```

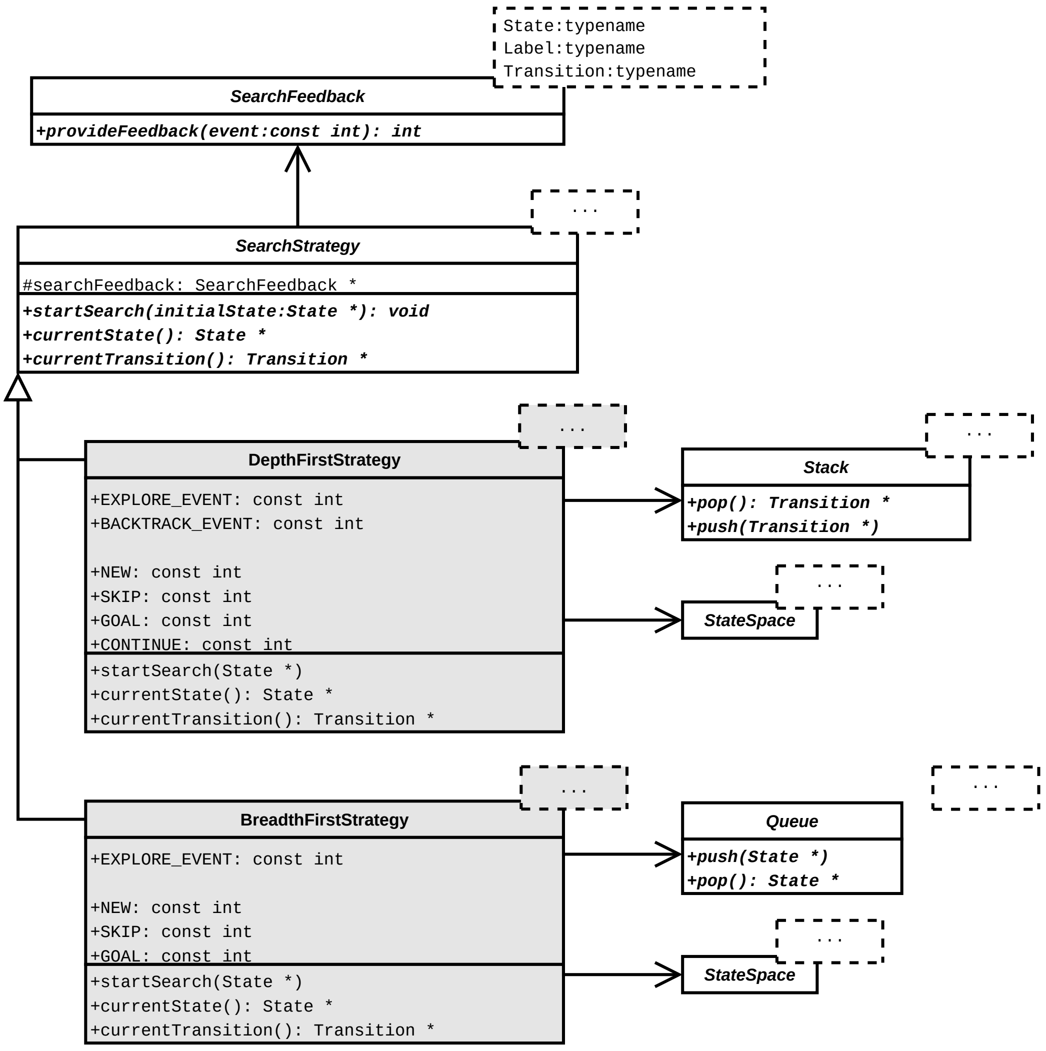


Figure 3.5 – Design of `SearchStrategy` and SEARCHFEEDBACK. The strategies now no longer have a $[\![Store]\!]$ or $[\![Goal]\!]$ but a $[\![SearchFeedback]\!]$ takes over this functionality.

Algorithm 3.4: $TryBacktrackStep(s \in S, r \in R \cup \{\epsilon\})$

```
1:  $currentState \leftarrow s$
2:  $currentTransition \leftarrow r$
3:
4:  $feedback \leftarrow [\![SearchFeedback]\!].ProvideFeedback(\text{BACKTRACK})$
5:
6:  {* Note that CONTINUE is ignored. *}
7:
8:  if ($feedback = \text{GOAL}$) then
9:    if ($r \neq \epsilon$) then
10:     $[\![Stack]\!].Push(r)$
11:   end if
12:   $foundTarget \leftarrow \text{TRUE}$
13: end if
```

Algorithm 3.5: Depth-first search strategy

```
Require: $[\![StateSpace]\!]$, $[\![SearchFeedback]\!]$, $[\![Stack]\!]$.
$foundTarget \leftarrow$ FALSE
$foundNewState \leftarrow$ FALSE

$currentState \leftarrow [\![StateSpace]\!].InitialState()$
$currentTransition \leftarrow \epsilon$

$TryExploreStep(currentState, currentTransition)$

while $(\neg foundTarget \wedge currentState \neq \epsilon)$ do
   {* Invariant: State $currentState$ is a new state. *}
   $foundNewState \leftarrow$ FALSE

   $r' \leftarrow [\![StateSpace]\!].FirstTransition(s')$
   while $(\neg foundTarget \wedge \neg foundNewState \wedge r' \neq \epsilon)$ do
      {* Iterate over outgoing transitions of $s'$. *}
      $s' \leftarrow [\![StateSpace]\!].Target(r')$
      $TryExploreStep(s', r')$
      if $(\neg foundTarget \wedge \neg foundNewState)$ then
         $r' \leftarrow [\![StateSpace]\!].NextTransition(r')$
      end if
   end while

   if $(\neg foundTarget \wedge \neg foundNewState)$ then
      $r' \leftarrow [\![Stack]\!].Pop()$
      while $(\neg foundTarget \wedge \neg foundNewState \wedge r' \neq \epsilon)$ do
         $s' \leftarrow [\![StateSpace]\!].Target(r')$ {* Iterate over stack. *}
         $TryBacktrackStep(s', r')$
         $alt \leftarrow [\![StateSpace]\!].NextTransition(r')$
         while $(\neg foundTarget \wedge \neg foundNewState \wedge alt \neq \epsilon)$ do
            {* Iterate over alternative outgoing transitions of $r'$. *}
            $s' \leftarrow [\![StateSpace]\!].Target(alt)$
            $TryExploreStep(s', alt)$
            if $(\neg foundTarget \wedge \neg foundNewState)$ then
               $alt \leftarrow [\![StateSpace]\!].NextTransition(alt)$
            end if
         end while
         if $(\neg foundTarget \wedge \neg foundNewState)$ then
            $r' \leftarrow [\![Stack]\!].Pop()$
         end if
      end while
   end if

   if $(\neg foundNewState)$ then
      $currentState \leftarrow \epsilon$
      $currentTransition \leftarrow \epsilon$
   end if
end while

if $(\neg foundTarget \wedge \neg foundNewState)$ then
   $currentState \leftarrow [\![StateSpace]\!].InitialState()$
   $TryBacktrackStep(currentState, \epsilon)$
end if
```

3.3.3 Search Adapter

So far there has been the assumption that there exists a functional implementation of SEARCHFEEDBACK, but implementing such an interface is not trivial. This section describes an implementation that is fairly general, and can be used for a number of commonly used search algorithms in model checking. In figure 3.6 it is shown that `SearchAdapter` implements SEARCHFEEDBACK and uses a SEARCHSTRATEGY. What is more important is how this class implements its feedback requirements. By means of the method `addHandler` we can associate an event with a default feedback value, as well as a list of CONDITIONs and ACTIONs. The idea is that ACTIONs provide feedback, and they are taken into account if the CONDITION they are associated with holds. To explain this in more detail we should first describe these interfaces.

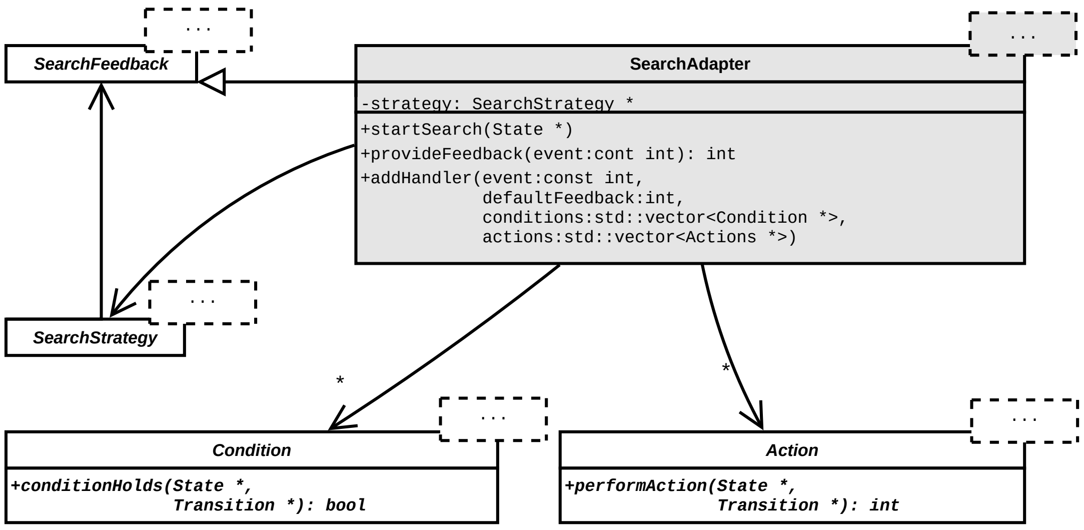


Figure 3.6 – A `SearchAdapter` implements SEARCHFEEDBACK and uses a SEARCHSTRATEGY. It can generate feedback from events by means of CONDITIONs and ACTIONs.

A condition is similar to the GOAL interface (see B.2 on page 86) of the dedicated solution, in the sense that it looks at the current state or transition and decides whether the condition holds or not. Example conditions could be accepting conditions, deadlock condition, back-edge condition or tautologic condition. As for other interfaces the implementation can be generic, or specialised for a particular type of model. A condition uses the current state, and the transition leading to the current state (or ϵ in case of the initial state). The CONDITION interface is defined in interface 3.5.

$$ConditionHolds : (S \times (R \cup \{\epsilon\})) \longrightarrow \{\textbf{TRUE}, \textbf{FALSE}\}$$

Interface 3.5 – CONDITION.

An action object performs some action, such as the storage of a state in a store, starting a nested search, or printing out the visited states. These actions are associated with a condition, and are only executed if the condition holds. The execution of an action provides feedback which is taken into account in the `SearchAdapter` implementation. The ACTION interface is defined in interface 3.6.

$$PerformAction : (S \times (R \cup \{\epsilon\})) \longrightarrow F$$

Interface 3.6 – ACTION.

The current state and the transition leading to the current state are arguments of both the $ConditionHolds$ function of CONDITION and the $PerformAction$ function in ACTION. From figure

3.6 it is clear that one can interpret search strategies as an interface SEARCHSTRATEGY, which provides the required information.

$$CurrentState : S$$
$$CurrentTransition : R \cup \{\epsilon\}$$

Interface 3.7 – SEARCHSTRATEGY.

Furthermore, a `SearchAdapter` potentially receives feedback from multiple actions, and will have to make a choice as to which to return to the search strategy. For this a function $Priority : (F) \longrightarrow \mathbb{N}$ is required, which implies the importance of certain feedback. As is clear from figure 3.5 feedback is represented as integer members anyway, so there is no need to explicitly define such a function if the integer values are chosen correctly. The only influence it has is that if the feedback of multiple ACTIONs is taken into account, the $Priority$ function is used to select the most important one. Naturally, in the case of our depth-first strategy, it is the assumption that $Priority(\textbf{GOAL}) > Priority(\textbf{SKIP}) > Priority(\textbf{NEW})$ and $Priority(\textbf{GOAL}) > Priority(\textbf{CONTINUE})$.

As previously said the `addHandler` function of `SearchAdapter` enables events to be associated with a default feedback value and a sequence of pairs of conditions and actions. Once feedback is requested by the search strategy this information is taken into account. Informally for each condition-action pair in the sequence the condition is evaluated. If the condition holds, the action is executed and its feedback is taken into account. The most prioritised feedback is dominant and is returned. A more formal description is given in algorithm 3.6 on page 32.

Algorithm 3.6: $ProvideFeedback(e \in E) \longrightarrow F$

```
Require: $[\![SearchStrategy]\!]$
Require: $defaultFeedbackValue$ is the default value of feedback for $e$.
Require: $\langle C_0, A_0\rangle, \ldots, \langle C_n, A_n\rangle$ is the sequence of pairs of $[\![Condition]\!]$ and $[\![Action]\!]$
    associated with $e$.
1: $feedback \leftarrow defaultFeedbackValue$
2: $i \leftarrow 0$
3: while $(i \leq n)$ do
4:    $s \leftarrow [\![SearchStrategy]\!].CurrentState()$
5:    $r \leftarrow [\![SearchStrategy]\!].CurrentTransition()$
6:    if $(C_i.ConditionHolds(s, r))$ then
7:       $actionFeedback \leftarrow A_i.PerformAction(s, r)$
8:       if $(Priority(actionFeedback) > Priority(feedback))$ then
9:          $feedback \leftarrow actionFeedback$
10:      end if
11:   end if
12:   $i \leftarrow i + 1$
13: end while
14: return $feedback$
```

To illustrate the flexibility of the design described in this section, a few examples will be provided.

- Consider a scenario where a tool needs to find accepting state in a model. To start with one would need to decide upon a search strategy. Assume one would choose the breadth-first approach, which has one event (**EXPLORE**) which behaves identically to its depth-first counterpart. Assume the user implemented an `AcceptCondition` specialised for a particular model. This condition holds if the current state is accepting. Also the user has implemented a `StoreAction`, which returns **SKIP** when a state is already in the store, and **NEW** if not. Furthermore, assume there are generic implementations of `TautologicCondition` and `GoalAction`, which always hold and always return **GOAL**, respectively.

One could set up a `SearchAdapter` with a `BreadthFirstStrategy` and associate the EXPLORE event with default feedback value NEW, and the condition-action sequence ⟨`AcceptCondition`,`GoalAction`⟩, ⟨`TautologicCondition`,`StoreAction`⟩.

- Now consider the scenario that the same user wishes to find acceptance cycles using a nested depth-first search (see [38]). Assume two store actions are implemented, `FirstStoreAction` and `SecondStoreAction`, one for the original search and one for the nested search. Both store actions use the same state store, as is described in [38]. Also we have a `IsSeedCondition` which evaluates to TRUE only if the current state matches some custom seed that can be set. Furthermore, we have a `SetSeedAction` action which is responsible for setting the custom state of the `IsSeedCondition` to the current state (and always returns CONTINUE). A `NestedSearchAction` starts a second search and returns GOAL if the search was successful, or CONTINUE otherwise.

  The initial search would be a `SearchAdapter` with a `DepthFirstStrategy`, with the EXPLORE event associated with default feedback value NEW, and the condition-action sequence ⟨`TautologicCondition`,`FirstStoreAction`⟩. Also associate the BACKTRACK event with default value CONTINUE and sequence ⟨`AcceptCondition`,`SetSeedAction`⟩, ⟨`AcceptCondition`,`NestedSearchAction`⟩.

  The nested search (defined in `NestedSearchAction`) has a `SearchAdapter` with a `DepthFirstStrategy`. The EXPLORE event is associated with with default feedback value NEW, and the condition-action sequence ⟨`TautologicCondition`, `SecondStoreAction`⟩, ⟨`IsSeedCondition`,`GoalAction`⟩.

Perhaps the examples seem more complex than expected. For instance, the store functionality in the second example is fairly complex to implement. However, this type of functionality is required to implement a nested depth-first search as it is described in [38]. The modular design does not simplify the problem, but provides a more natural way of solving it. As explained before, the generic design of this layer does enable the possibility of implementing such functionality in a generic fashion, i.e. for a whole class of models, rather than a specific type of model.

3.4 Transformations of State Spaces

In the search module, functionality was introduced that could search for, say, accepting states, as well as accepting cycles. Although both search types are defined over state spaces, these state spaces are typically not the same, or even of the same type. When we search for acceptance cycles this is often because we have performed a synchronous product of two Büchi automata, one for the system, and for for a negation of a property [66, 19, 36]. However, it is not desirable to make two implementations of a model type, one for **LTL** model checking and one for model checking simple properties. Therefore it is necessary to be able to make transformations in state spaces.

See figure 3.7 for an example of a transformation. A `FooModel` of a state space exists, and we can use generic functionality over this type of model. However, we also want a transformation of this type of model to `BarModels`. The typical way to implement this is to make an association from `BarModel` to `FooModel`. Both model types implement the STATESPACE interface with different types of states, labels and transitions.

3.4.1 Examples of State Space Transformations

Although the possibilities of transformations are very broad, there are a few typical examples that can be discussed.

- Consider we have an implementation of a model in which each state, label and transition consists of a lot of objects. As memory requirements are strict these need to be converted to bitvectors before verification, which is a common approach model checking. This can be done with a transformation, as can be seen in figure 3.8. A `MemoryIntensiveModel` is tranformed into a `BitvectorModel`. This principle is applied in our abstract layer in section 4.2.

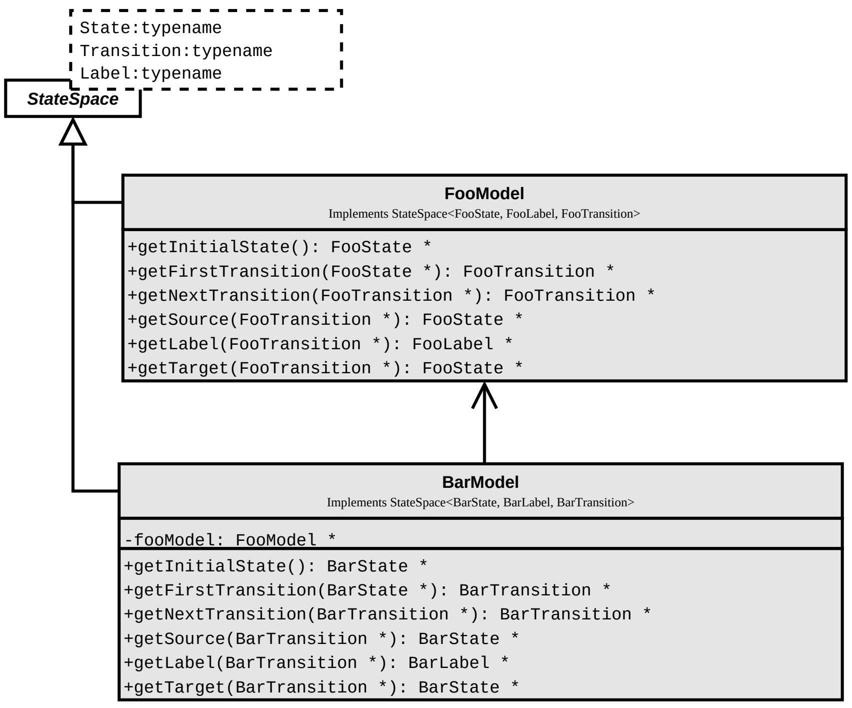

State:typename
Transition:typename
Label:typename
StateSpace
FooModel
Implements StateSpace<FooState, FooLabel, FooTransition>
+getInitialState(): FooState *
+getFirstTransition(FooState *): FooTransition *
+getNextTransition(FooTransition *): FooTransition *
+getSource(FooTransition *): FooState *
+getLabel(FooTransition *): FooLabel *
+getTarget(FooTransition *): FooState *
BarModel
Implements StateSpace<BarState, BarLabel, BarTransition>
-fooModel: FooModel *
+getInitialState(): BarState *
+getFirstTransition(BarState *): BarTransition *
+getNextTransition(BarTransition *): BarTransition *
+getSource(BarTransition *): BarState *
+getLabel(BarTransition *): BarLabel *
+getTarget(BarTransition *): BarState *

Figure 3.7 – An example tranformation from a `FooModel` state space to a `BarModel` state space.

- Consider a type of model whose 'global' state space is composed purely of a product of state spaces of the components in the model. This too can be a transformation, although perhaps this illustrates that the naming convention of 'transformation' does not cover the intended usage, as this transformation transforms multiple state spaces into a single one (see figure 3.9).

- Finally, we can call the synchronous product of the Büchi automaton that represents the negation of an **LTL** property and the Büchi automaton of the system model as a transformation. Each automaton implements a state space and the `LtlProductModel` is a generic class for making this synchronous product for any kind of model. Although this might be tricky to implement, once this is functionality is available it can be used to model check **LTL** model checking for any type of state space. Note that figure 3.10 abstracts from the fact that the `LtlProperty` will probably need a set of atomic propositions of a type that is consistent with the model type under investigation. Note that as the states in the result of the synchronous product are basically pairs of `LtlState` and the type of states over which the system model is defined, the `LtlProductState` need to be generic as well.

3.4.2 Enabling Reuse for Transformations

The transformations presented in this section present us with a new design problem. Consider the transformation from `MemoryIntensiveModel` to `BitvectorModel` from figure 3.8. The `MemoryIntensiveModel` uses state, label and transitions types `MemoryIntensiveState`, `MemoryIntensiveLabel` and `MemoryIntensiveTransition`.

Deadlock conditions, strategy observers etcetera, are all defined on these types of states, labels and transitions, rather than their `BitvectorModel` equivalents. In the current setup, if one wishes to search `BitvectorModels` one would also need to implement the interfaces such as conditions and observers for `BitvectorState`, `BitvectorLabel` `BitvectorTransition`. However, implementing the same condition for both seems redundant, and it is preferable to use classes defined over the memory-intensive representation for the bitvector representation too.

However, as figure 3.9 and 3.10 show, this reuse is not always intuitive, as there is not always

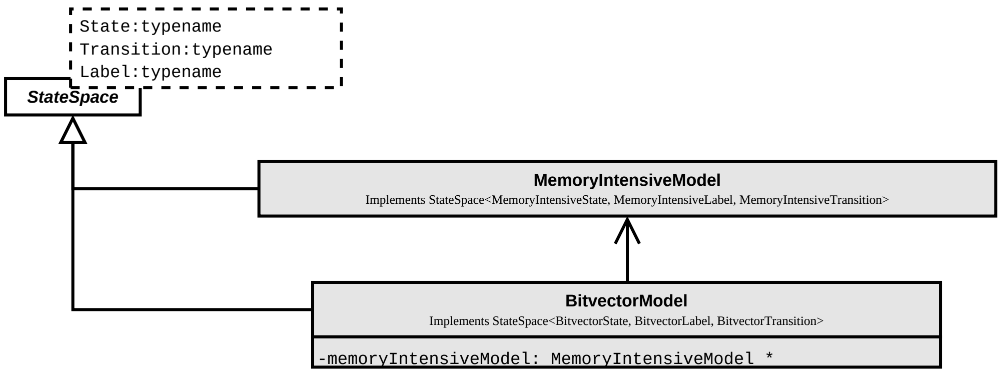

State:typename
Transition:typename
Label:typename
StateSpace
MemoryIntensiveModel
Implements StateSpace<MemoryIntensiveState, MemoryIntensiveLabel, MemoryIntensiveTransition>
BitvectorModel
Implements StateSpace<BitvectorState, BitvectorLabel, BitvectorTransition>
-memoryIntensiveModel: MemoryIntensiveModel *

Figure 3.8 – A transformation to a more memory-efficient representation of the state space.

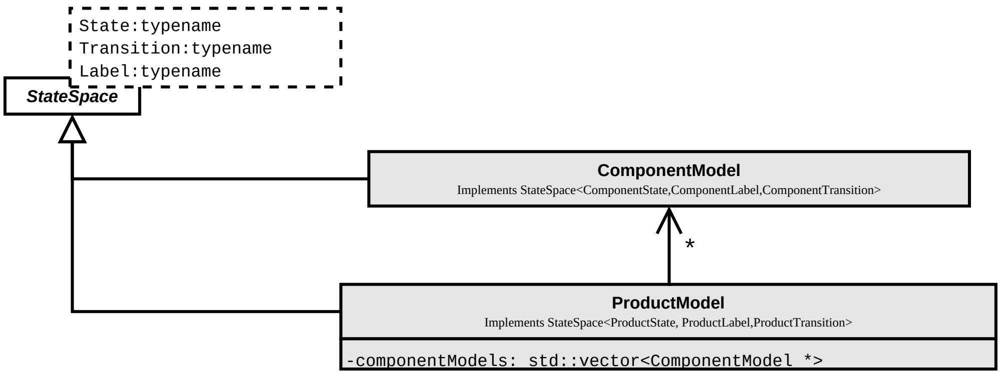

State:typename
Transition:typename
Label:typename
StateSpace
ComponentModel
Implements StateSpace<ComponentState,ComponentLabel,ComponentTransition>
*
ProductModel
Implements StateSpace<ProductState, ProductLabel,ProductTransition>
-componentModels: std::vector<ComponentModel *>

Figure 3.9 – The parallel composition of the state spaces of components into a product state space.

an obvious mapping from the transformed state space to one of the original state spaces. For instance, for figure 3.10 one could have conditions implemented for the `LtlProperty` and for the system model. It should be possible to use both types of conditions for `LtlProductModels` as there is a clear mapping from this product to the property and system state spaces.

The solution is to provide one implementation of each generic interface that can provides the required implementation by means of an implementation of the same interface for the original state space and a mapping from the transformed state space to the original state space. For instance, for the case of `MemoryIntensiveModel` and `BitvectorModel` one would need a mapping from `BitvectorModel` to `MemoryIntensiveModel` and an implementation of a SIMULATIONOBSERVER for `MemoryIntensiveModels`, then it is possible to realise a SIMULATIONOBSERVER for `BitvectorModels`. This is shown in figure 3.11. In the worst case this design would require each interface to have one generic implementation that provides reuse for transformed models.

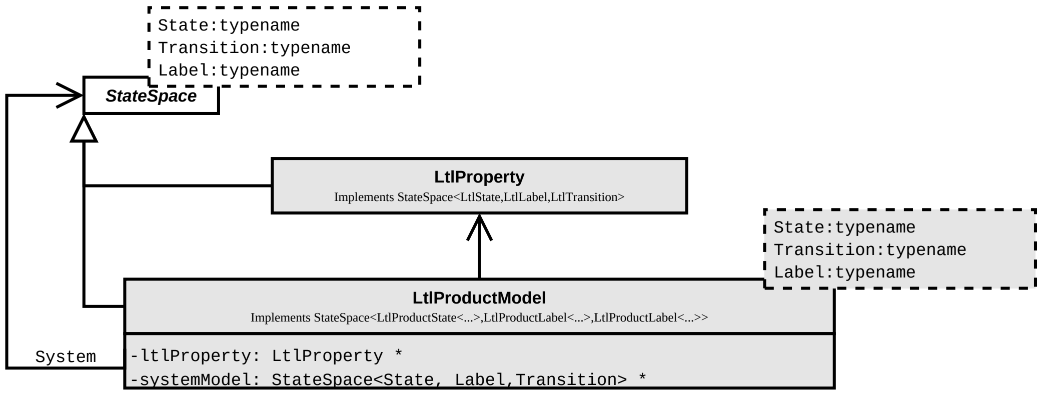

State:typename
Transition:typename
Label:typename
StateSpace
LtlProperty
Implements StateSpace<LtlState,LtlLabel,LtlTransition>
State:typename
Transition:typename
Label:typename
LtlProductModel
Implements StateSpace<LtlProductState<...>,LtlProductLabel<...>,LtlProductLabel<...>>
System
-ltlProperty: LtlProperty *
-systemModel: StateSpace<State, Label,Transition> *

Figure 3.10 – Synchronous product of an **LTL** property and the system model as a transformation.

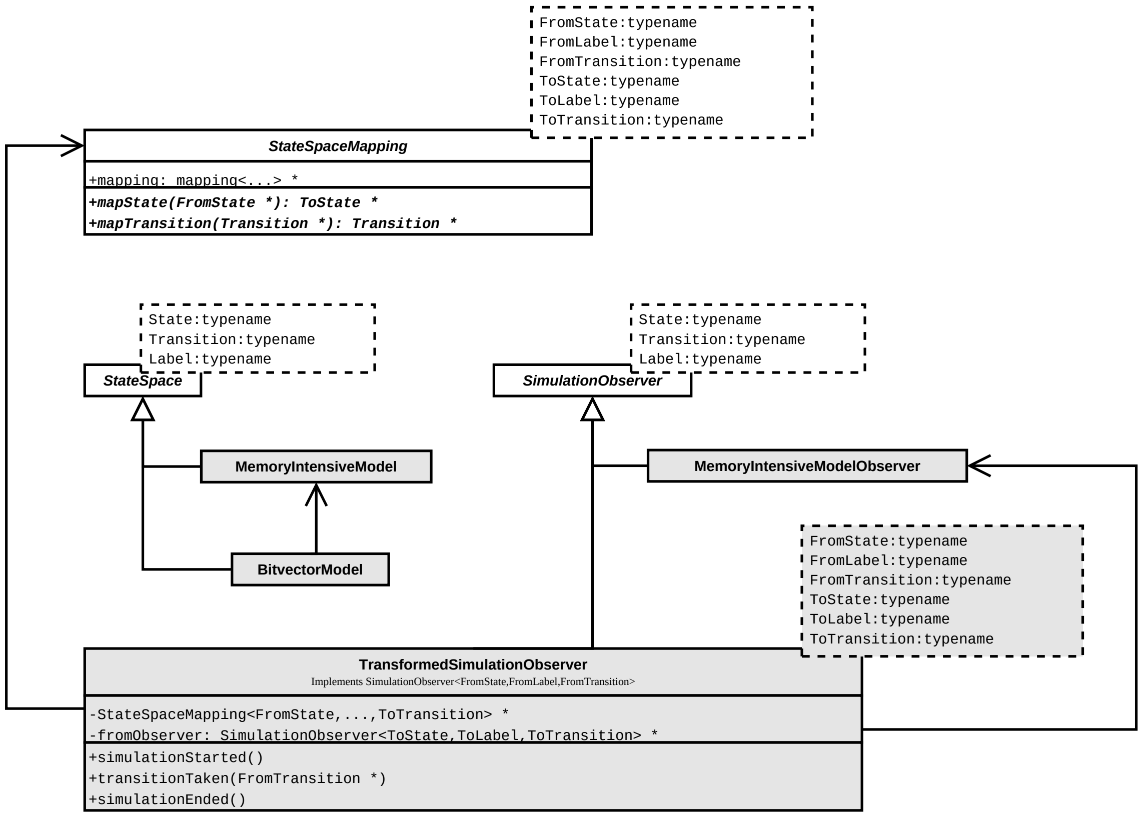


Figure 3.11 – Reuse of a `MemoryIntensiveModelObserver` by means of a generic class `TransformedSimulationObserver`, given a mapping from `BitvectorModel` to `MemoryIntensiveModel`.

Chapter 4

The Abstract Layer

So far the focus has been on the generic layer of the framework, which was concerned with models on a high level of abstraction. This layer requires information about models on the level of a state space, and did not assume anything about the internal structure of states, labels and transitions. The *abstract layer* provides an internal structure to these states, labels and transition. Although a specification on the level of a state space might be appropriate for some models, it is generally desirable to specify models using a more abstract specification language. The naming convention might introduce some confusion; the 'abstract layer' refers to the abstraction made from individual states and transitions in the state space. However, when we say this layer has a low level of abstraction we refer to the fact that the states, labels and transitions in this layer are more conrete than in the generic layer, as they have an internal structure.

There are many possible ways in which one could provide an abstract layer, but this chapter focuses on only one possibility. Some models that could be implemented as an abstract layer in figure 2.3 in section 2.1.2, different implementations of the abstract layer would all share the funcitonality in the generic layer.

As the initial drive behind this project was to develop a more modular version of **SPIN**, the abstract layer in this chapter is mostly tailored to be able to model **PROMELA**-like models. For this we introduce high-level features such as data, variables, processes, functions, control-flow and statements. The models defined by the current implementation of the abstract layer will be referred to as 'software models'.

It should be noted that the state representation in this abstract layer is based on [60] (e.g. **BOGOR**). The graph-like nature of the states presented in this article seem to be a natural way of expression states of *software models*, and open up opportunities for symmetry reductions.

4.1 Design of Software Models

In chapter 2 and 3 it was explained that in order to use the generic layer, one would need to provide an implementation of the STATESPACE interface, specialised with a particular type of state, label and transition. The big picture of the abstract layer should thus include classes like `SoftwareModel`, `SoftwareState` and `SoftwareTransition`. However, as transitions in software-based models are mostly associated with statements of code, we choose to use the naming convention `Statement` for labels rather than `SoftwareLabel`.

In figure 4.1 it is shown how these classes implement a state space interface. However, it shows little detail of the funcitonality involved to transform the high-level features into a state space. To illustrate this consider a `SoftwareState`. A `SoftwareState` remembers the state of the `SoftwareModel`. At any one time, a model could have multiple parallel processes running, each with their own function stack and local variables. Also there could be global variables, and a heap of objects that is shared amongst processes. However, remembering the state is not enough, we also must have enough information to be able to correctly implement the semantics of the model in terms of transitions and changes of states. For this we require the notion of statement and control-flow.

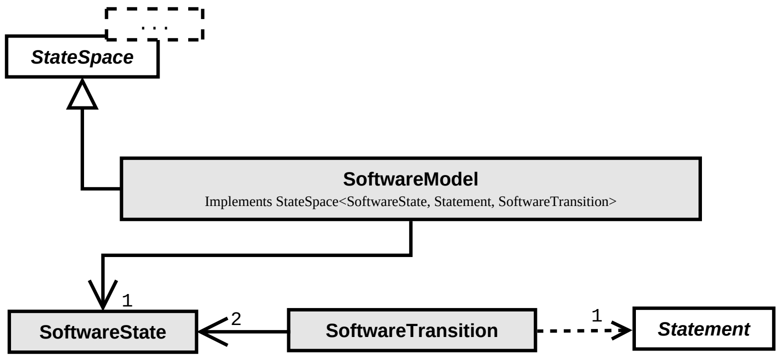


Figure 4.1 – Simplified view of the abstract layer, consisting of `SoftwareModels`. Note that `Statements` are not directly related to `SoftwareTransitions`, but rather `SoftwareTransitions` link to a `ControlFlowTransition` of the control-flow of a process or function type, which is associated with a `Statement`

The information required is two-fold, not only do we need information about instances (e.g. data values, process states) but also about types (e.g. data types, process types). To show why this is useful, consider the control-flow of processes. This control-flow is the same for all process instances of the same type. It makes sense to store this information per process type rather than per instance. Throughout this chapter, the distinction between information about *types* and information about *instances* is very explicit. Note that the inclusion of typing information does make the design of the abstract layer more complex.

Firstly, the focus will be on the representation of states. `SoftwareStates` are considered to be graphs, similar to [60], and as we also have typing information, we first consider the type graph of states. This type graph is model-wide, shared by each state in the state space. See definition 4.1 and 4.2 for the formal definition of both a graph and a type graph.

Definition 4.1. *A directed graph is a triplet $\langle S,\ L,\ R\rangle$, where S is a set of nodes, L is a set of labels and $R \subseteq S \times L \times S$ is set of directed labelled edges.*

Definition 4.2. *Consider graphs $G = \langle S_G,\ L_G,\ R_G\rangle$, $H = \langle S_H,\ L_H,\ R_H\rangle$ and a typing function $\tau : S_G \longrightarrow S_H$ such that $(s, l, s') \in R_G \Longrightarrow (\tau(s), l, \tau(s')) \in R_H$. H is called a type graph of G.*

The reason why this chapter formally treats states as graphs and introduces type graphs as well is because this opens up new possibilities to formally reason about them. For instance, the algorithm used to linearise states to a bitvector makes strong use of fact that it is possible to interpret states as graphs (see section 4.2.2). Also, treating states as graphs will proof useful when trying to realise symmetry reductions in the state space (see also section 5.4.2).

4.1.1 Formal Description of the Model-Wide Type Graph

Consider `SoftwareModel` $\mathcal{M}$ which is to be expressed in terms of a state space $\langle S_\mathcal{M}, L_\mathcal{M}, I_\mathcal{M}, R_\mathcal{M}\rangle$. The idea is that each `SoftwareState` in $S_\mathcal{M}$ will be interpreted as a graph. Also, we assume there exists a type graph $\Gamma_\mathcal{M}$ which is a type graph for each state in $S_\mathcal{M}$. Each type node in $\Gamma_\mathcal{M}$ has additional information associated with it. For example a process type would have a control-flow. To build this type graph it is necessary to find out more about $\mathcal{M}$. It is presumed that the information needed to build a type graph can be made available by means of static analysis over the model specification.

$$\begin{aligned} \mathcal{M}_{Proc} &: \text{The set of all process types in } \mathcal{M}. \\ \mathcal{M}_{Func} &: \text{The set of all function types in } \mathcal{M}. \\ \mathcal{M}_{Data} &: \text{The set of all data types in } \mathcal{M}. \\ \mathcal{M}_{Var} &: \text{The set of all variables used in } \mathcal{M}. \end{aligned}$$

The intuition is that process types, function types and data types are nodes in our type graph, whereas variables are labels in the type graph. This information by itself is not sufficient to construct the type graph, therefore we introduce some auxiliary functions.

$$\begin{aligned}\mathcal{M}_{Scope} &: \mathcal{M}_{Var} \longrightarrow (\{\iota\} \cup \mathcal{M}_{Proc} \cup \mathcal{M}_{Func} \cup \mathcal{M}_{Data})\\ \mathcal{M}_{VarType} &: \mathcal{M}_{Var} \longrightarrow \mathcal{M}_{Data}\\ \mathcal{M}_{Call} &: (\mathcal{M}_{Proc} \cup \mathcal{M}_{Func}) \longrightarrow \mathcal{P}(\mathcal{M}_{Func})\end{aligned}$$

The function $\mathcal{M}_{Scope}$ maps variables to the scope in which they are defined, whereas $\mathcal{M}_{VarType}$ maps variables to the data type these variables resemble. All not only all types in the model are nodes of the type graph, we also define an additional ι-type, which denotes the global type. Global variables can use this type as their scope. Note that the definition of $\mathcal{M}_{VarType}$ implies we cannot have references to process instances, or function instances. Finally, the relation $\mathcal{M}_{Call}$ specifies the functions a process and function type might call. This information is sufficient to reconstruct a call graph of a model. Now, the type graph $\Gamma_{\mathcal{M}}$ consists of the triplet $\langle S_{\Gamma_{\mathcal{M}}}, L_{\Gamma_{\mathcal{M}}}, R_{\Gamma_{\mathcal{M}}}\rangle$ where:

$$\begin{aligned}S_{\Gamma_{\mathcal{M}}} =& \{\iota\} \cup \mathcal{M}_{Proc} \cup \mathcal{M}_{Func} \cup \mathcal{M}_{Data}\\ L_{\Gamma_{\mathcal{M}}} =& \{*\} \cup \mathcal{M}_{Var}\\ R_{\Gamma_{\mathcal{M}}} =& \{(\mathcal{M}_{Scope}(var), var, \mathcal{M}_{VarType}(var)) | var \in \mathcal{M}_{Var}\} \cup\\ & \{(s, *, s') | s \in (\mathcal{M}_{Proc} \cup \mathcal{M}_{Func}) \cdot s' \in \mathcal{M}_{Call}(s)\}\end{aligned}$$

The edge label '$*$' is used to model function calls. As one process or function *type* may call multiple functions during its lifetime, it is clear that this label introduces non-determinism in the type graph. But at any one time, a process or function *instance* can only call one function. This means that this is not a cause for non-determinism in the state graphs.

Consider the model specification in listing 4.1. Although without defining formal semantics it is not precisely clear how we could extract the required information, the intention is that the syntax is intuitive enough to convince most readers. As before, $\mathcal{M}$ will denote the model under consideration. Given $\mathcal{M}_{Proc}$, $\mathcal{M}_{Func}$, $\mathcal{M}_{Data}$, $\mathcal{M}_{Var}$, $\mathcal{M}_{Scope}$, $\mathcal{M}_{VarType}$ and $\mathcal{M}_{Call}$, which ideally should be extractable from listing 4.1 by means of static analysis, the type graph $\Gamma_{\mathcal{M}}$ will be constructed.

Using the information that is provided, the type graph $\Gamma_{\mathcal{M}}$ can be constructed. In figure 4.2 this graph is shown using a graphical representation that is presumed to be intuitive, and is supposed to naturally extend the graphical notation for state graphs given in [60].

$$\begin{aligned}S_{\Gamma_{\mathcal{M}}} &= \{\iota, init, Philosopher, takeFork, releaseFork, Fork, Boolean\}\\ L_{\Gamma_{\mathcal{M}}} &= \{*, f1, f2, f3, left, right, take, release, value\}\\ R_{\Gamma_{\mathcal{M}}} &= \{(Philosopher, *, takeFork), (Philosopher, *, releaseFork),\\ &\quad (init, f1, Fork), (init, f2, Fork), (init, f3, Fork),\\ &\quad (Philosopher, left, Fork), (Philosopher, right, Fork),\\ &\quad (takeFork, fork, Fork),\ (releaseFork, fork, Fork),\\ &\quad (Fork, value, Boolean)\}\end{aligned}$$

4.1.2 Design to Support Typing Information

Perhaps it is hard to relate last section to the previously presented design diagrams of certain parts of the framework. This section presents how the typing information can be put into practice. Figure 4.3 shows a **UML** diagram that relates the `SoftwareModel` to the typing information.

Types that are present in our model are nodes in the type graph ($S_{\Gamma_{\mathcal{M}}}$), and map to the `Type` class in the design. As can be seen in the definition of $S_{\Gamma_{\mathcal{M}}}$ it consists of four 'types', ι, $\mathcal{M}_{Proc}$, $\mathcal{M}_{Func}$ and $\mathcal{M}_{Data}$, which map to `GlobalType`, `ProcessType`, `FunctionType` and `DataType`, respectively.

Edges in the type graph consist of both variables and function calls. As function calls have no edge-specific information (i.e. they always have the same label $*$), they can be incorporated by a simple list in each function and process to the functions they can call ($\mathcal{M}_{Call}$). Variable edges

Listing 4.1 – A dining philosophers model in DSPIN-style.

```
typedef Fork {
    bool isTaken;
};

init {
    Fork & f1 = new Fork;
    Fork & f2 = new Fork;
    Fork & f3 = new Fork;

    f1 = false;
    f2 = false;
    f3 = false;

    run Philosopher(f1, f2);
    run Philosopher(f2, f3);
    run Philosopher(f3, f1);
};

proctype Philosopher(Fork & left, Fork & right) {
    do ::
        takeFork(left);
        takeFork(right);
        releaseFork(right);
        releaseFork(left);
    od;
};

function takeFork(Fork & take) : void {
    atomic
    {
        !fork.isTaken -> fork.isTaken = true;
    };
};

function releaseFork(Fork & release) : void {
    fork.isTaken = false;
};
```

however do have edge-specific information, as the edges are labelled with the variable itself. This means the introduction of a `Variable` class can be justified. The definition of $\mathcal{M}_{Scope}$ is present by means of association from the `Type` class to a number of `variables`. Each `Variable` has an association with a `DataType` to represent $\mathcal{M}_{VarType}$.

Although this section does not provide a thorough understanding of the functionality of the classes presented in figure 4.3, hopefully it does provide the link between the presented formal description of the type graph and the design of the abstract layer presented so far.

4.1.3 Formal Description of the State Graphs

Now that we have constructed the type graph for all state graphs of $\mathcal{M}$, we can describe what the states of $\mathcal{M}$ will look like. Consider we would like to build the state graph of a state $\sigma \in S_{\mathcal{M}}$ defined by $\langle S_\sigma, L_\sigma, R_\sigma \rangle$ as well as a typing function τ_σ that maps nodes from the state graph to nodes in the type graph ($\tau_\sigma : S_\sigma \longrightarrow S_{\Gamma_{\mathcal{M}}}$). Nodes in the state graph also have 'internal data', for example the internal data of a boolean data instance would be a boolean value, and the internal state of a process instance would consist of a process identifier and a control-flow state. This is described in more defail in the section about state graph linearisation (section 4.2). Again, we

$$
\begin{aligned}
\mathcal{M}_{Proc} &= \{init, Philosopher\}\\
\mathcal{M}_{Func} &= \{takeFork, releaseFork\}\\
\mathcal{M}_{Data} &= \{Fork, Boolean\}\\
\mathcal{M}_{Var} &= \{f1, f2, f3, left, right, take, release, value\}\\
\mathcal{M}_{Scope}(f1) &= init\\
\mathcal{M}_{VarType}(f1) &= Fork\\
\mathcal{M}_{Scope}(f2) &= init\\
\mathcal{M}_{VarType}(f2) &= Fork\\
\mathcal{M}_{Scope}(f3) &= init\\
\mathcal{M}_{VarType}(f3) &= Fork\\
\mathcal{M}_{Scope}(left) &= Philosopher\\
\mathcal{M}_{VarType}(left) &= Fork\\
\mathcal{M}_{Scope}(right) &= Philosopher\\
\mathcal{M}_{VarType}(right) &= Fork\\
\mathcal{M}_{Scope}(fork) &= takeFork\\
\mathcal{M}_{VarType}(fork) &= Fork\\
\mathcal{M}_{Scope}(fork) &= Philosopher\\
\mathcal{M}_{VarType}(fork) &= Fork\\
\mathcal{M}_{Scope}(value) &= Fork\\
\mathcal{M}_{VarType}(value) &= Boolean\\
\mathcal{M}_{Call}(init) &= \emptyset\\
\mathcal{M}_{Call}(Philosopher) &= \{takeFork, releaseFork\}\\
\mathcal{M}_{Call}(takeFork) &= \emptyset\\
\mathcal{M}_{Call}(releaseFork) &= \emptyset
\end{aligned}
$$

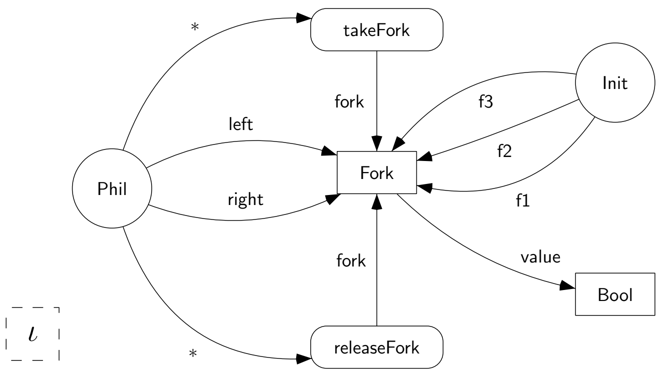


Figure 4.2 – Type graph of the states of the dining philosophers model given in listing 4.1. The ι-node has no outgoing edges as there are no global variables.

need to know more about the internal structure of σ:

$$
\begin{aligned}
\sigma_{\underline{Proc}} &: \text{The set of all process } \textit{instances} \text{ in } \sigma.\\
\sigma_{\underline{Func}} &: \text{The set of all function } \textit{instances} \text{ in } \sigma.\\
\sigma_{\underline{Data}} &: \text{The set of all data } \textit{instances} \text{ in } \sigma.
\end{aligned}
$$

These set of instances should be disjoint. Additionally, it is required to have all instances map to their type; these functions will be used to define τ_σ later.

$$
\begin{aligned}
\tau_{\sigma_{\underline{Proc}}} &: \sigma_{\underline{Proc}} \longrightarrow \mathcal{M}_{Proc}\\
\tau_{\sigma_{\underline{Func}}} &: \sigma_{\underline{Func}} \longrightarrow \mathcal{M}_{Func}\\
\tau_{\sigma_{\underline{Data}}} &: \sigma_{\underline{Data}} \longrightarrow \mathcal{M}_{Data}
\end{aligned}
$$

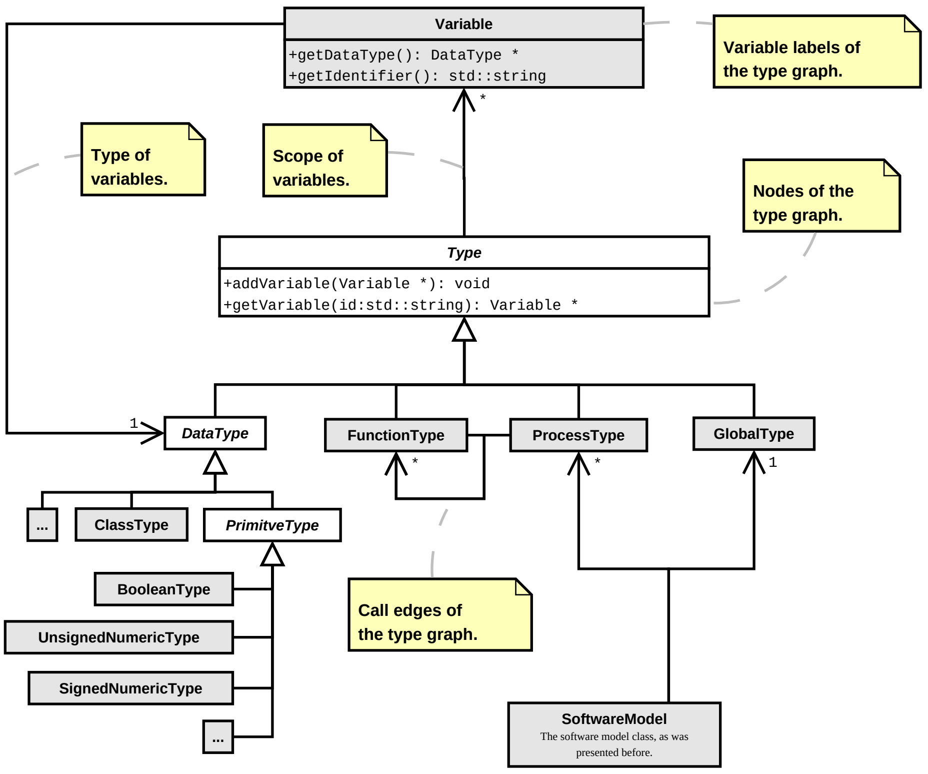

Variable
+getDataType(): DataType *
+getIdentifier(): std::string
Variable labels of the type graph.
Type of variables.
Scope of variables.
Nodes of the type graph.
Type
+addVariable(Variable *): void
+getVariable(id:std::string): Variable *
DataType
FunctionType
ProcessType
GlobalType
...
ClassType
PrimitveType
BooleanType
UnsignedNumericType
SignedNumericType
...
Call edges of the type graph.
SoftwareModel
The software model class, as was presented before.
1
*
*
*
1

Figure 4.3 – Design of typing information in the abstract layer. The notes map the diagram elements to their formal definition in the previous section.

Now we will define τ_σ by means of the functions we have just defined, and by adding the trivial case for the global scope (ι denotes the global instance as well as the global type). From τ_σ it should be clear that $S_\sigma = \{\iota\} \cup \sigma_{\underline{Proc}} \cup \sigma_{\underline{Func}} \cup \sigma_{\underline{Data}}$.

$$\tau_\sigma(s) = \begin{cases} \tau_{\sigma_{\underline{Proc}}}(s) & \text{, if } (s \in \sigma_{\underline{Proc}}) \\ \tau_{\sigma_{\underline{Func}}}(s) & \text{, if } (s \in \sigma_{\underline{Func}}) \\ \tau_{\sigma_{\underline{Data}}}(s) & \text{, if } (s \in \sigma_{\underline{Data}}) \\ \iota & \text{, if } (s = \iota) \end{cases}$$

Furthermore, for the states to map to the type graph, we need edges that reflect the variable edges in the type graph. This could be seen as the value of a variable and is defined by function $\sigma_{\underline{Value}}$. We use ϵ to denote a variable without value (e.g. a null pointer).

$$\sigma_{\underline{Value}} : (\mathcal{M}_{Var} \times S_\sigma) \rightarrow (\sigma_{\underline{Data}}) \cup \{\epsilon\}$$

A requirement is that the value function is consistent with the type graph:

$$val = \sigma_{\underline{Value}}((var, s)) \wedge val \neq \epsilon \Rightarrow \tau_\sigma(s) = \mathcal{M}_{Scope}(var) \wedge \tau_\sigma(val) = \mathcal{M}_{VarType}(var)$$

The final piece of information concerns call stacks. For each process and function instance it is required to know exactly which function is next on the call stack stack in the current state. This information is given in $\sigma_{\underline{Call}}$.

$$\sigma_{\underline{Call}} \subseteq (\sigma_{\underline{Proc}} \cup \sigma_{\underline{Func}}) \times \sigma_{\underline{Func}}$$

The $\sigma_{\underline{Call}}$ set also has to be consistent with the $\mathcal{M}_{Call}$ function in the typing information:

$$(p, f) = \sigma_{\underline{Call}} \Rightarrow \tau_\sigma(f) \in \mathcal{M}_{Call}(\tau_\sigma(p))$$

The $\sigma_{\underline{Call}}$ set comes with an additional constraint that each function instance is exactly once on a function stack. From all given information the graph representation of state σ can finally be defined:

$$\begin{aligned}
S_\sigma =& \{\iota\} \cup \sigma_{\underline{Proc}} \cup \sigma_{\underline{Func}} \cup \sigma_{\underline{Data}} \\
L_\sigma =& L_{\Gamma_{\mathcal{M}}} \\
R_\sigma =& \{(s, v, \sigma_{\underline{Value}}(v, s)) | v \in \mathcal{M}_{Var}, s \in S_\sigma \cdot \sigma_{\underline{Value}}(v, s) \neq \epsilon\} \\
& \cup \{(p, *, f) | (p, f) \in \sigma_{\underline{Call}}\}
\end{aligned}$$

According to the definition of a type graph (definition 4.2 on page 39) it should hold that $(s, l, s') \in R_\sigma \implies (\tau(s), l, \tau(s')) \in R_{\Gamma_{\mathcal{M}}}$. According to the definition of R_σ, there are two main types of edges in the state graph, edges representing variable values and edges representing function calls.

- To show the condition holds for call edges, consider that for each call edge $(s, l, s') \in R_\sigma$ implies $l = *$ and $(s, s') \in \sigma_{\underline{Call}}$. The requirement of $\sigma_{\underline{Call}}$ was that it is consistent with the typing information, e.g. $(s, s') \in \sigma_{\underline{Call}}$ implies $\tau_\sigma(s') \in \mathcal{M}_{Call}(\tau_\sigma(s))$. The definition of $R_{\Gamma_{\mathcal{M}}}$ shows that this means that $(\tau(s), l, \tau(s')) \in R_{\Gamma_{\mathcal{M}}}$.
- Similarly for variable edges, each variable edge $(s, l, s') \in R_\sigma$ implies $s \in S_\sigma$, $l \in \mathcal{M}_{Var}$ and $s' = \sigma_{\underline{Value}}(l, s)$. The requirement on $\sigma_{\underline{Value}}$ implies that $\tau_\sigma(s) = \mathcal{M}_{Scope}(l)$ and $\tau_\sigma(s') = \mathcal{M}_{VarType}(l)$. As $R_{\Gamma_{\mathcal{M}}}$ includes $(\mathcal{M}_{Scope}(var), var, \mathcal{M}_{VarType}(var))$ for each variable in $\mathcal{M}_{Var}$, it must hold that $(\tau(s), l, \tau(s')) \in R_{\Gamma_{\mathcal{M}}}$.

An example of a state graph is given in figure 4.4, which is again extracted from the model specification given in listing 4.1. The type graph of this figure is shown in figure 4.2.

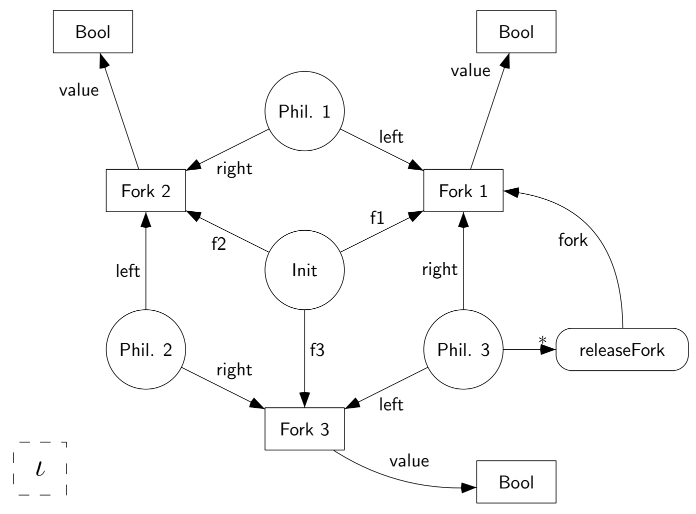


Figure 4.4 – A graph representation of a state of the dining philosophers model given in listing 4.1. The types of the nodes have been explicitly noted as text on the nodes. The example elaborates on the example in [60].

4.1.4 Design of State Representation

Similar as to how we related the type graph to the design of typing information in the framework, this section will associate the formal definition of states to the design of the framework that supports the representation of states. Figure 4.5 presents this design, which is similar to how figure 4.3 designed.

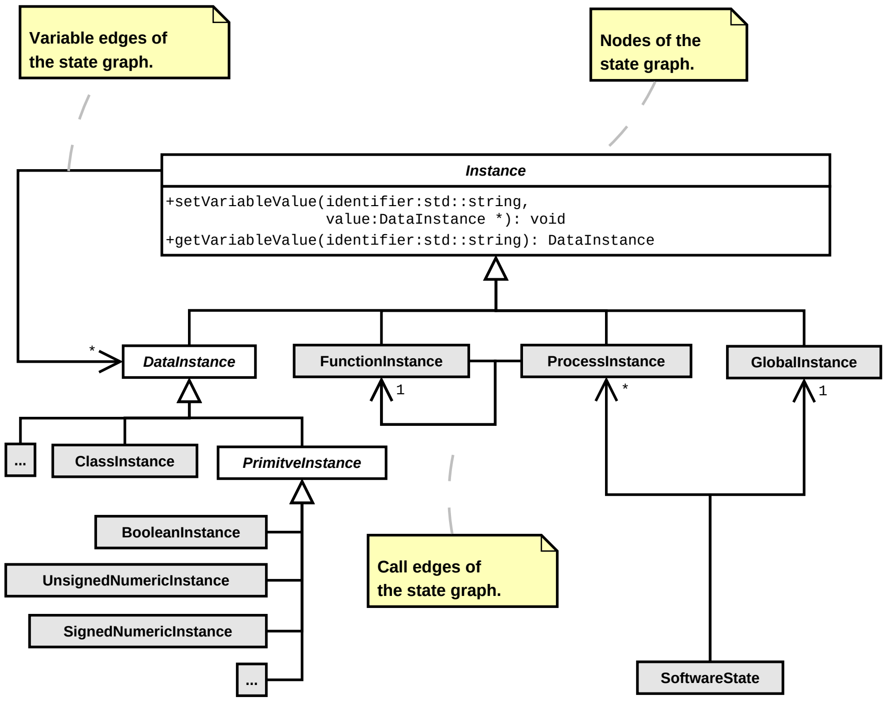

Variable edges of the state graph.
Nodes of the state graph.
Instance
+setVariableValue(identifier:std::string, value:DataInstance *): void
+getVariableValue(identifier:std::string): DataInstance
*
DataInstance
FunctionInstance
ProcessInstance
GlobalInstance
1
*
1
...
ClassInstance
PrimitveInstance
BooleanInstance
UnsignedNumericInstance
SignedNumericInstance
...
Call edges of the state graph.
SoftwareState

Figure 4.5 – Design of state representation in the abstract layer. The notes map the diagram elements to their formal definition in the previous section.

However, it is important to note that typing information is model-wide, and there only exists one 'type graph' per software model. In contrast there exist as many 'state graphs' as there exists states in the model.

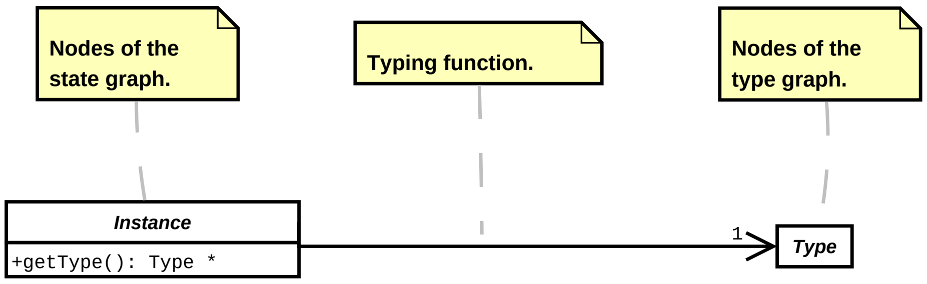


Figure 4.6 – Representation of typing function τ_σ by means of a function `getType` in each `Instance`.

The typing function τ is the link between state graphs and the type graph. An association from `Instance` to a `Type` represents the formal definition of this typing function. Obviously, this does imply the implicit requirement that `GlobalInstances` map to `GlobalTypes`, `ProcessInstances` map to `ProcessTypes` etc.

4.1.5 Formal Description of Transitions and Labels

The type graph and state graphs represent the type of state with which `SoftwareModels` implement the StateSpace interface. One would expect a similar discussion for the types of labels and transitions. As states are represented by state graphs, it makes sense to see transitions as graph morphisms. One could use formal specification methods like graph grammars [58, 57, 46] or other similar techniques to formally define the transitions and semantics of our abstract layer. There are a number of reasons why this abstract layer does not introduce formal means to describe transitions.

Firstly, state graphs are not purely graphs. The nodes in the state graph have internal values associated with them. For instance, a data instance would have a data value associated with it. These values could also change, meaning that a morphism on the state graphs alone is not expressive enough to model transitions. Secondly, we argue that it is not always trivial to describe the semantics of a specification language formally, especially if these semantics are to be expressed in graph grammars.

Rather than using a formal representation of transitions, we introduce a more practical design that closely follows the intuition.

4.1.6 Design of Transition and Label Representation

As previously mentioned, a `Statement` class represents the label type of the state space. Typically, a such a statement is responsible for a number of transitions in the state space of a `SoftwareModel`.

Each process type and function type has a control-flow. This control-flow is a separate implementation of the STATESPACE interface, and is labelled with `Statements`. The control-flow of process types and function types is not directly part of the type graph, but should be seen as information associated with the process type nodes and function type nodes in the type graph. Process and function instances contain a single control-flow node which models their *program counter*.

Note that modelling control-flow as an implementation of a STATESPACE opens up the possibility to simulate a control-flows, or check for the reachability of the final control-flow state.

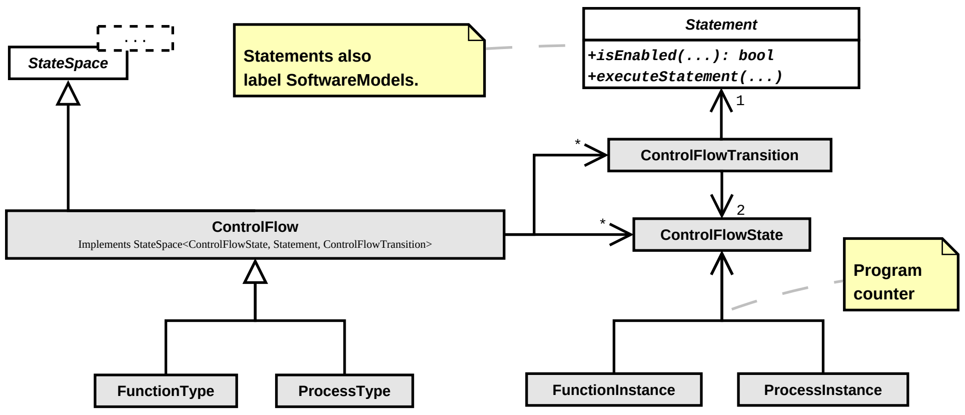


Figure 4.7 – The diagram that shows how control-flows are included in the model. Each process and function type are associated with some control-flow. Process and function instances merely point to their current state in the control flow of their type.

Figure 4.7 shows how the notion of control-flow is included in the design of the abstract layer. Statements are no longer directly linked to the `SoftwareModel` but rather are associated with transitions in the control flow of process and function types.

In order to implement the STATESPACE interface for `SoftwareModels`, we would need to be able to retrieve all outgoing transitions of a `SoftwareState`, and these outgoing transitions should be presented in a total order (due to the $FirstTransition$, $NextTransition$ functions in STATESPACE). We will address this procedure informally.

A `SoftwareState` is based on a state graph. From this state graph we can extract all the process instances ($\sigma_{\underline{Proc}}$). We look for outgoing transitions in a process instance in the order dictated by their process identifiers. Given a process instance, one can follow the '$*$' edges to find the top most frame on the call stack. This can be either the process instance or a function instance. All we currently know about this instance is its current control-flow state, as this is considered to be part of the internal information within function instances and process instances. With the τ_σ function, the type of this instance can be resolved. Within this type the control-flow associated with the instance can be found.

The $FirstTransition$ and $NextTransition$ of the control-flow can be used to find the next possible transitions in the control-flow. The `Statements` that label the control-flow state space, have an additional function that can determine whether the statement is executable for the current SoftwareState. This facilitates the need for conditional statements. Control-flow transitions that are not executable are not taken into account in the $FirstTransition$, $NextTransition$ of SoftwareModels.

Consider a transition that is obtained from the SoftwareModel, and this transition is based on a transition in the control-flow of a process or function type. The $getTarget$ function in SoftwareModel requires that based on this transition we can create a target SoftwareState. To realise this the original SoftwareState is first copied. The Statement that labels the control-flow transition is responsible for changing the new state to suit the semantics of the transition. Obviously, the control-flow state (or *program counter*) of the executing function or process instance is changed such that it is the target of the control-flow transition.

To facilitate the requirement that Statements themselves are responsible for changing the state, we introduces classes for Expressions and VariableReferences. Expressions evaluate to a DataInstance when given a SoftwareState. VariableReferences are a special kind of Expression which evalute to the value of a variable.

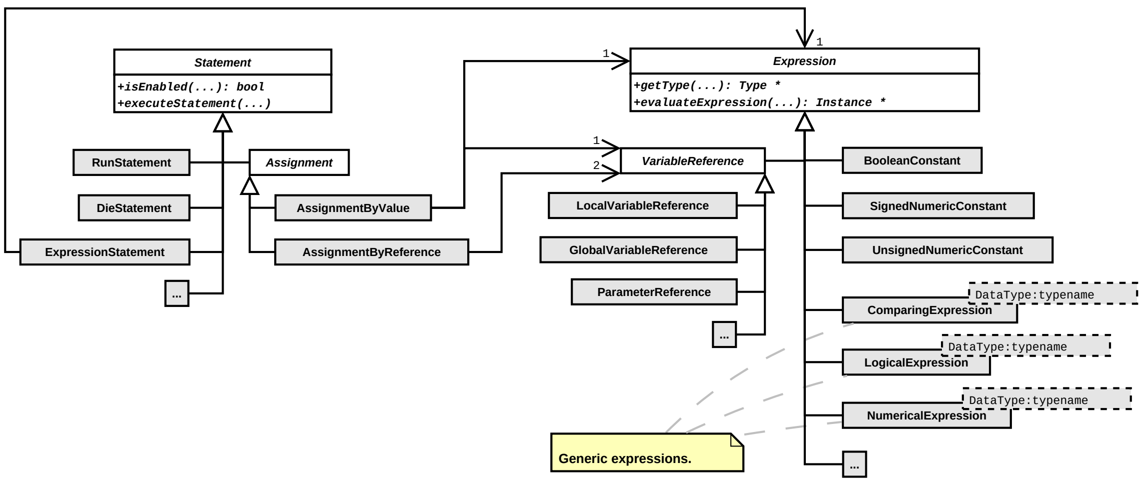


Figure 4.8 – Some statements and expressions in the framework. Note that the diagram is not complete, many associations have been left out for clarity.

Figure 4.8 shows how Statements and Expressions fit in the framework. It goes beyond the scope of this thesis to explain the semantics of each and every statement that was implemented. The documentation of the source code will explain the individual classes in more detail. The idea is that new Statements and Expressions can be introduced to model the semantics of the target language to be used by the abstract layer.

One thing that does stand out in the design in the figure is the 'generic expressions'. These expressions are meant to model mathematical operations, but abstract from the data type they are being used upon. For instance the ComparingExpression is meant to provide the operators $=$, $\neq$, $>$, $\geq$, $<$ and $\leq$. If used on a particular DataType it is required that this type implements a ComparingInterface, which has some abstract functions to realise the functionality. In a similar fashion, LogicalExpression is a generic expression for the logical operators $\vee$, $\wedge$ and $\neg$, and NumericalExpression for the operators $+$, $-$, $*$, / and $\%$.

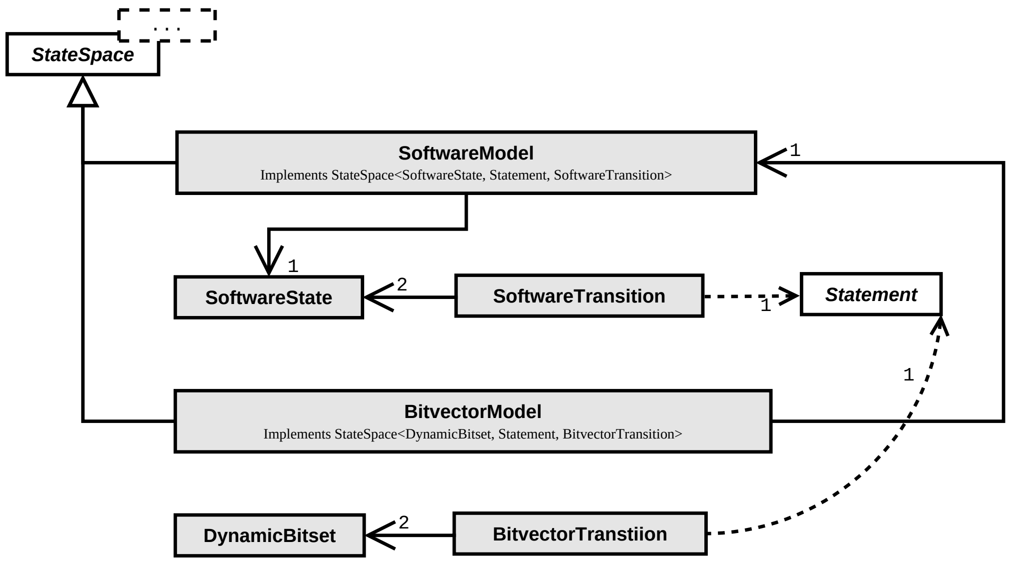
In this way adding a new data type is simplified, simply choose the type of operations that are valid for the data type and implement the interface related to these operators. In this way there is no need to write dedicated expressions for the boolean type. For instance, the BooleanInstance implements both ComparingInterface and LogicalInterface. Casting expressions are also implemented in a similar fashion.

4.2 Transformation to Linearised Representation

This section introduces a transformation of `SoftwareModels`, and in particular the transformation of `SoftwareStates` to a linearised bitvector representation. The reason for introducing a tranformation of the state space is two-fold:

- The state graphs described in the first part this chapter can become very large. Every node in the state graph is an instance of a class and resides somewhere in the main memory. If one was to model check using these graph representations as states, and one would verify a model with a few million states, the memory used by the state graphs would be the bottle-neck of the verification. A more compact representation is necessary if the memory requirements of the framework are to be within acceptable limits.

- A common operation during verification is comparing states to see whether they are the same. Comparing graphs for equality is trickier than comparing bitvectors.



...
StateSpace
SoftwareModel
Implements StateSpace<SoftwareState, Statement, SoftwareTransition>
1
1
SoftwareState
2
SoftwareTransition
1
Statement
1
BitvectorModel
Implements StateSpace<DynamicBitset, Statement, BitvectorTransition>
DynamicBitset
2
BitvectorTranstiion

Figure 4.9 – Transformation of `SoftwareModels` to `BitvectorModels`, essentially only the the state type is changed into a bitvector. As `Statements` are included in the typing information rather than for each state, it is unlikely that transforming them to a bitvector representation will be very profitable.

4.2.1 Linearising Rooted Deterministic Typed Graphs

In this section it is presented how rooted deterministic typed graphs can be linearised. Credit is due to Rensink who has kindly provided this algorithm; see also [57, 58, 60] about how graphs can be used as state representation in the context of model checking. The limitation of this linearisation procedure is that graphs have to be rooted, deterministic and typed.

Consider a graph G as presented by definition 4.1, and G is typed by means of a typing function $\tau_G : S_G \longrightarrow S_H$ onto type graph H. Furthermore, consider there is an arbitrary total ordering $\sqsubseteq_{L_H} \subseteq L_H \times L_H$ over labels in the type graph. Also,– we need a root node $\rho \in S_G$ from which every other node in S_G is reachable.

To help with the linearisation we introduce two functions, $LabelOut_H$ and $LabelIndex_s$. $LabelOut_H$ is a function that maps nodes in the type graph to labels on the outgoing edges of that node. The function $LabelIndex_s$ maps a set of labels (from $LabelOut_H$) to natural numbers using the total ordering function $\sqsubseteq_{L_H}$.

$$\begin{aligned} LabelOut_H &: S_H \longrightarrow \mathcal{P}(L_H) \\ LabelOut_H(s) &= \{l \mid (s,l,s') \in R_H\} \end{aligned}$$

$$\begin{aligned} LabelIndex_s &: LabelOut_H(s) \longrightarrow \{1,\ldots,|LabelOut_H(s)|\} \\ LabelIndex_s(l) &= |\{l' \mid l' \in LabelOut_H(s) \cdot l' \sqsubseteq_{L_H} l\}| \end{aligned}$$

The function $\mathsf{encode}(i)$ denotes the linearised representation of i. We assume we can linearise individual nodes of the state graph, nodes of the type graph and natural numbers. Furthermore, let $list$ be an indexed list of nodes (S_G) starting with index 1. Algorithm 4.1 descibes how graphs can be linearised.

Algorithm 4.1: Linearisation of a graph.

```
1:  $k \leftarrow 1$
2:  put $\rho$ in $list$ with index 1
3:  $enc \leftarrow$ empty list of linearised items
4:
5:  while $(k \leq |list|)$ do
6:    $s \leftarrow$ element of $list$ with index $k$
7:    add encode$(\tau_G(s))$ to $enc$
8:    add encode$(s)$ to $enc$
9:    $n \leftarrow 1$
10:   while $(n \leq |LabelOut_H(\tau_G(s))|)$ do
11:     $l \leftarrow LabelIndex^{-1}_{\tau_G(s)}(n)$
12:     if (there exists $s'$ such that $(s,l,s') \in R_G$) then
13:       if $(list$ contains $s')$ then
14:         $k' \leftarrow$ index of $s'$ in $list$
15:         add encode$(k')$ to $enc$
16:       else
17:         put $s'$ in $list$ with index $|list| + 1$
18:         add encode$(|list| + 1)$ to $enc$
19:       end if
20:     else
21:       add encode$(0)$ to $enc$
22:     end if
23:     $n \leftarrow n + 1$
24:   end while
25:   $k \leftarrow k + 1$
26: end while
27: return $enc$
```

Note that it could be considered unnecessary to ‘save’ the type of each node in the graph that were included by means of a label that is no cause for non-determinism in the type graph, as the type can be derived by means of the deterministic edge in the type graph. However, our type graph is not fully deterministic (see figure 4.2 on page 4.2). Also with an eye on possible future extensions of data types to include subtyping it might be wise to keep track of the type of each node in the graph.

4.2.2 Linearising State Graphs

In order to use the algorithm in the previous section to linearise state graphs, we need to fulfill the requirement that state graphs are rooted, deterministic and typed. From the formal definition introduced in 4.1.3 it should be clear that each state graph σ has a typing function τ_σ which maps nodes of the state graph onto type graph $\Gamma_{\mathcal{M}}$, and therefore fulfills the typing requirement. State graphs are also deterministic, as the only edges in state graphs are ones representing variable values and function calls. A variable can only have one value at a time and function calls are also no cause of non-determinism because call stacks are stacks of function instances and each function instance has at most one successor in the call stack.

The only requirement left to fulfill is the requirement that a state graph has to be rooted. Unfortunately this is a much harder requirement to fulfill. As figure 4.4 on page 44 shows, a typical state graph has no obvious root node. However, every node, or *instance*, is always reachable from either the global instance or a process instance. This is where we cheat a bit, rather than having one root node, we start several nodes in the list of nodes to be linearised.

In particular we start with the global instances and the process instances in the $list$. However, we cannot add them in arbitrary order as this might mean that two states that have the same state graph can be linearised to different bitvectors. Therefore, they will be ordered by means of their process identifier. Each process instance has a unique process idenfier as given by $ProcessIdentifier_\sigma : (\sigma_{\underline{Proc}}) \longrightarrow \mathbb{N}$. We introduce a total order of process instances $\sqsubseteq_{\sigma_{\underline{Proc}}} \subseteq (\sigma_{\underline{Proc}} \times \sigma_{\underline{Proc}})$ which is used find out the desired order in which process instances should be linearised. For $p, p' \in \sigma_{\underline{Proc}}$ it should hold that $ProcessIdentifier_\sigma(p) \leq ProcessIdentifier_\sigma(p') \iff p \sqsubseteq_{\sigma_{\underline{Proc}}} p'$. The function $ProcessIndex_\sigma$ is the order in which the processes are included in the linearisation algorithm.

$$ProcessIndex_\sigma : \sigma_{\underline{Proc}} \longrightarrow \{1, \ldots, |\sigma_{\underline{Proc}}|\}$$
$$ProcessIndex_\sigma(p) = |\{p' |\ p' \in \sigma_{\underline{Proc}} \cdot p' \sqsubseteq_{\sigma_{\underline{Proc}}} p\}|$$

Furthermore, the total ordering over labels in the type graph is simply the alphabetical ordering of the identifiers of the variables, which label the edges (and $\forall_{l \in L_H} \cdot l \sqsubseteq_{L_H} *$).

To illustrate the intention of the linearisation algorithm consider the state graph of the dining philosophers problem in figure 4.4 on page 44. It will be shown informally what a linearisation of this state graph would look like. The processes instances in this example are $Init$, $Phil1$, $Phil2$ and $Phil3$. Consider an aribitrary mapping:

$$ProcessIdentifier_\sigma = \{(Init, 0), (Phil1, 3), (Phil2, 7), (Phil3, 2)\}$$

This in turn enables the calculation of the total order relation $\sqsubseteq_{\sigma_{\underline{Proc}}}$, which will finally result into the following indexing relation:

$$ProcessIndex_\sigma = \{(Init, 1), (Phil1, 3), (Phil2, 4), (Phil3, 2)\}$$

This implies that before the first iteration of the linearisation algorithm begins, the $list$ variable contains the following nodes (1) ι, (2) $Init$,(3) $Phil3$, (4) $Phil1$, (5) $Phil2$.

Table 4.1 shows how this results into the linearisation of the state using algorithm 4.2. Each row in the table represents an instance. If we put all the elements in the table in a sequence a bitvector is formed that is a linearisation of the state. It is perhaps unclear what the `encode()` function actually encodes.

Encoding types A simple search over the type graph will provide us with all types that we could possibly encounter in all state graphs. Consider there are n types, then it is sufficient to use $\lceil log_2(n) \rceil$ bits to encode types. This functionality can be realised by the `BitvectorModel`.

Encoding instances Where the types could be encoded by means without help of the `SoftwareModel` instances cannot. Instances need to encode their internal state. This is all information *besides* variable values and function calls (as these are edges). We hold each `Type` responsible for knowing how to encode instances.

- A *global instance* has no internal state, and therefore its bitvector representation is empty. If ever there is a need to 'save' state-wide information, the global instance would be the designated place to do so.

Algorithm 4.2: Linearisation of a state graph σ.

```
1: put $\iota$ in $list$ with index 1
2:
3: $n \leftarrow 1$
4: while ($n \leq |\sigma_{\underline{Proc}}|$) do
5:    $process \leftarrow ProcessIndex_{\sigma}^{-1}(n)$
6:    put $process$ in $list$ with index $|list| + 1$
7:    $n \leftarrow n + 1$
8: end while
9:
10: $k \leftarrow 1$
11: $enc \leftarrow$ empty list of linearised items
12:
13: while ($k \leq |list|$) do
14:    $s \leftarrow$ element of $list$ with index $k$
15:    add encode($\tau_G(s)$) to $enc$
16:    add encode($s$) to $enc$
17:    $n \leftarrow 1$
18:    while ($n \leq |LabelOut_H(\tau_G(s))|$) do
19:       $l \leftarrow LabelIndex_{\tau_G(s)}^{-1}(n)$
20:       if (there exists $s'$ such that $(s, l, s') \in R_G$) then
21:          if ($list$ contains $s'$) then
22:             $k' \leftarrow$ index of $s'$ in $list$
23:             add encode($k'$) to $enc$
24:          else
25:             put $s'$ in $list$ with index $|list| + 1$
26:             add encode($|list| + 1$) to $enc$
27:          end if
28:       else
29:          add encode($0$) to $enc$
30:       end if
31:       $n \leftarrow n + 1$
32:    end while
33:    $k \leftarrow k + 1$
34: end while
35: return $enc$
```

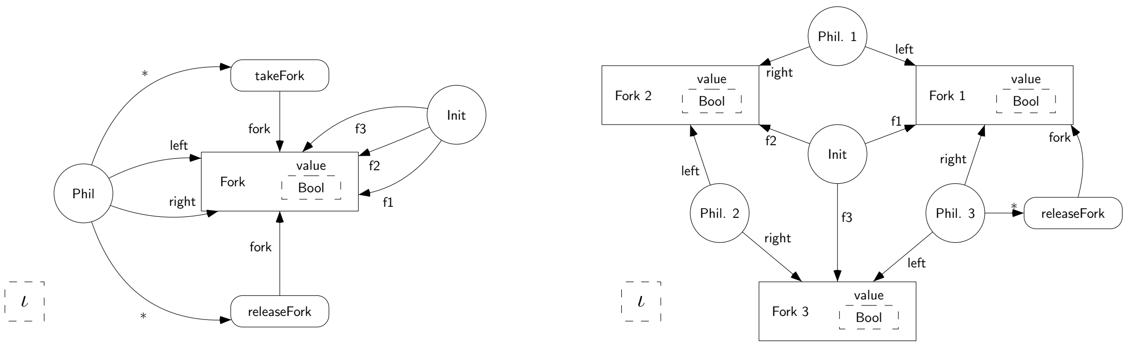
- A *process instances* consists at least of a ‘process identifier’ which is a natural number. Ideally, each process instance in each state uses the same length of bitvector to encode their identifier. As we do not exactly know what the biggest identifier would be in the model, an upper bound would have to be chosen. Furthermore, a process instance needs to remember the control-flow state it is in. For this the `ControlFlow` class will have to provide a mapping of control-flow states to bitvectors.

  Finally, for some semantics in languages it is necessary to store additional information in the process instances. For example to realise the atomic execution of a process, each process instance would need a bit flag in its representation to to indicate it is executing exclusively. These bit flags would have to be encoded in the bitvector too.
- Similar to process instances, *function instances* will need to remember their control-flow state, for which they can use the functionality introduced for process instances.
- *Data instances* simply need to remember their value. For instance, a `BooleanInstance` will need 1 bit to encode whether its value is $true$ or $false$.

Encoding edges Encoding edges is nothing more than referring to the row in which the target instance is encoded. This is done by means of a natural number. Obviously, we cannot

Table 4.1 – Linearisation of the dining philosophers state depicted in figure 4.4. Each row represents an instance in the state graph.

Iteration	Encoding of type	Encoding of instance	Encoding of edges		
$k = 1$	encode(ι)	encode(ι)			
$k = 2$	encode($Init$)	encode($init$)	encode(6) ‘f1’	encode(7) ‘f2’	encode(8) ‘f3’
$k = 3$	encode($Philolospher$)	encode($phil3$)	encode(8) ‘left’	encode(6) ‘right’	encode(9) ‘*’
$k = 4$	encode($Philolospher$)	encode($phil1$)	encode(6) ‘left’	encode(7) ‘right’	encode(0) ‘*’
$k = 5$	encode($Philolospher$)	encode($phil2$)	encode(7) ‘left’	encode(8) ‘right’	encode(0) ‘*’
$k = 6$	encode($Fork$)	encode($fork1$)	encode(10) ‘value’		
$k = 7$	encode($Fork$)	encode($fork2$)	encode(11) ‘value’		
$k = 8$	encode($Fork$)	encode($fork3$)	encode(12) ‘value’		
$k = 9$	encode($releaseFork$)	encode($rFork$)	encode(6) ‘fork’		
$k = 10$	encode($Boolean$)	encode($bool1$)			
$k = 11$	encode($Boolean$)	encode($bool2$)			
$k = 12$	encode($Boolean$)	encode($bool3$)			



(a) Optimised type graph (b) Optimised state graph

Figure 4.10 – Optimisation of state and type graph. The boolean $value$ of forks is now encoded as an internal value of a $Fork$.

encode all natural numbers by means of a statically sized bitvector, nor do we know the largest possible number of rows that could occur for all states in the model. Therefore we require another upper bound.

4.2.3 Optimisation of State Linearisation

The size of the bitvector that represents a state should be as small as possible. For this reason this section introduces a means of reducing the state vector. The optimisation is discussed informally. This is done by designating certain types in the type graph and considering it as part of another type. Figure 4.10 shows how the value of a $Fork$ is considered directly as part of a $Fork$ rather than a separate instance.

The $value$ of a Fork is no longer represented by a row, but is included in the encoded representation of $Fork$. The reduction in the bitvector is three-fold:

- The type $Bool$ can be omitted as a type all together, possibly reducing the number of bits required to encode types.
- There are fewer edges, as the $value$ edge can be omitted.

- There are fewer 'rows' or instances in the state graph, avoiding overhead such as the type designation. Also the reduction in rows means that fewer bits could be used to represent the edges of the state graph.

Table 4.2 shows how much the state vector is reduced. However, the bitvectors representing the internal value of $fork1$, $fork2$ and $fork3$ have increased by 1 bit to encode the boolean value.

Table 4.2 – Optimised linearisation of the dining philosophers state depicted in figure 4.4. Note how there are fewer rows and edges than in table 4.1.

Iteration	Encoding of type	Encoding of instance	Encoding of edges		
$k = 1$	encode(ι)	encode(ι)			
$k = 2$	encode($Init$)	encode($init$)	encode(6) 'f1'	encode(7) 'f2'	encode(8) 'f3'
$k = 3$	encode($Philolospher$)	encode($phil3$)	encode(8) 'left'	encode(6) 'right'	encode(9) '*'
$k = 4$	encode($Philolospher$)	encode($phil1$)	encode(6) 'left'	encode(7) 'right'	encode(0) '*'
$k = 5$	encode($Philolospher$)	encode($phil2$)	encode(7) 'left'	encode(8) 'right'	encode(0) '*'
$k = 6$	encode($Fork$)	encode($fork1$) encode($bool1$)			
$k = 7$	encode($Fork$)	encode($fork2$) encode($bool2$)			
$k = 8$	encode($Fork$)	encode($fork3$) encode($bool3$)			
$k = 9$	encode($releaseFork$)	encode($rFork$)	encode(6) 'fork'		

This reduction is not always possible, as there are two conditions that must hold in order for an instance to be eligible for optimisation:

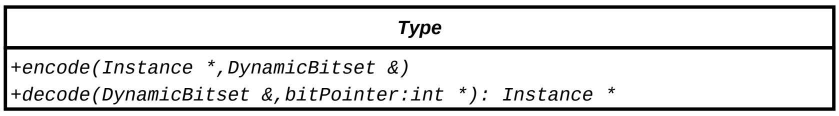
- The instances that are encapsulated by a 'parent instance' should never be referenced from other instances in the state graph. In the analogy of the tables, it is not possible to refer to a row that isn't there.

- The encapsulated instances should always have a value. This requirement is useful when decoding a bitvector, as otherwise there is no means of knowing whether the next few bits represent an internal instance or some edges.

It is not always possible to designate certain instances as suitable for optimisation, as in order to validate the requirements one would have to look at each state graph in the state space. However, one could argue that 'stack variables' in programming languages such as C++ or JAVA would almost always fulfill the requirements (unless they can be pointed to by reference or pointer variables). This is where we argue the reduction would be most useful. In the context of the additional decisions made for pointer variables in section 6.1.2 we could argue that any normal variable is suitable for optimisation. It should be noted that this optimisation is currently not implemented.

4.2.4 Functionality to Support Bitvector Representation

Most functionality can be provided by the `BitvectorModel` itself, such as the encoding of edges and types. However, the encoding of instances is something it cannot do, and for this we need to add some functionality to the design of `SoftwareModels`. 4.11 shows the abstract *encode* and *decode* methods that are added to `Type`.


Type

*+encode(Instance *,DynamicBitset &)*
*+decode(DynamicBitset &,bitPointer:int *): Instance **

Figure 4.11 – The `Type` class is responsible for encoding and decoding `Instances` for which two abstract methods have been added *encode* and *decode*.

The *encode* function adds its representation to the back of an existing bitvector, whereas the *decode* function reconstructs its instance from a given bitvector.

Note that we have purposely not mentioned any decoding functionality, as this does not add much too the understanding of the transformation. However, it should be taken into account that any linearised representation can still be decoded to a `SoftwareState`.

Chapter 5

Including Features

Chapter 1 introduced a number of features that were encountered in tools (see section 1.2.5 in particular). This chapter provides suggestions as to how these model checking techniques can be implemented in the framework described in the previous sections. Although these suggestions have been composed with much consideration, none of these features have been implemented, and the implementation of such features would require a more in-depth study of these features.

5.1 Change of Generic Layer

In the introductory chapter it was explained that ————it is possible to categorise algorithms into explicit-state, symbolic and bounded categories. Chapter 3 directly addresses the inclusion of explicit-state verification algorithms in the framework. However, the symbolic and bounded categories have not yet been addressed. The introduction of these approaches would require a change of the generic layer.

5.1.1 Symbolic Model Checking

The current implementation of the framework does not support symbolic model checking. The state space representation in the 'generic layer' is deliberately chosen to support explicit-state verification. If symbolic model checking is to be included in the framework then a new 'generic layer' for symbolic model checking is to be introduced with the same conceptual structure as the current layers, but which defines state spaces symbolically.

As this project did not include an extensive study on symbolic model checking, this section will only show how one would define a 'generic layer' for symbolic model checking conceptually. Consider the incomplete interface 5.1. The idea is that all or most elements of a symbolic interface would consist of sets of states, rather than individual states. Rather than enforcing a particular representation of these sets, we use generics to abstract from the type of objects that represents these sets. This is similar to how the explicit-state 'generic layer' abstracted from individual states. When implementing the symbolic interface, **BDD**s can be used as the type to represent sets of states. To perform model checking of temporal formulae, one would need additional operations on these sets. For instance a common operation in **CTL** model checking algorithms is to find out in which states a particular atomic proposition from AP holds. This could be realised by means of a function $Hold : AP \longrightarrow \mathcal{P}(S)$. Besides a symbolic representation of states, it is also possible

$$States : \mathcal{P}(S)$$
$$\ldots : \ldots$$

Interface 5.1 – Incomplete symbolic STATESPACE

to use symbolic representations of the transition relation. The complete definition of a symbolic interface would require an extensive knowledge of symbolic model checking algorithms, and is not considered to be in the scope of this project.

5.1.2 Bounded Model Checking

Bounded model checking assumes a model can be expressed as a propositional formula, and verification algorithms are in fact satisfiability solvers. To include bounded model checking in our framework, we would need to redefine a ‘generic layer’ for bounded model checking. This would most likely result in an interface which abstracts from the type of object that represents *propositional formulae*, *clauses* and *literals*. The interface could consist of a single function that returns the total formula representing the system. Functions could be included to make queries about the structure of the formula, such as extracting the clauses from a formula or testing whether a clause is a unit clause. Also, functions could perform manipulations on the formula, such as removing clauses. Again, defining such an interface would require extensive knowledge about bounded model checking, which is not within the scope of this project.

5.2 Search Strategies

In section 3.3.2 it was explained how a depth-first search can be realised by means of a `DepthFirstStrategy`. The idea is that also breadth-first and directed searches are to be included by the framework by means of implementations of `SearchStrategy`

5.2.1 Breadth-First Search Strategy

Figure 3.5 already shows how a breadth-first strategy could be included in the framework. It is very similar to the depth-first strategy but has a *queue of states* rather than a *stack of transitions*. The idea is that in each iteration a state is extracted from the queue, and for all outgoing transitions the EXPLORE event is triggered. Feedback in the form of GOAL would terminate the search, SKIP would do nothing, and NEW would add the target state to the queue.

Another undefined area is the problem of generating counter-examples. Arguably, there are two ways to extract such a counter example. One could implement a queue with the sufficient information to extract the counter example, or one could add sufficient information to a store to extract the required information. Either solution would require a specialisation of some abstract class in the framework, e.g. `Queue` or `Condition`, respectively.

5.2.2 Directed Search Strategy

Directed searches are to be implemented with a search strategy that uses a priority queue. Consider a theoretical catagorisation of states in the state space. A state is a member of the *open* set if it is in the queue, which means it was visited, but not yet expanded. The *closed* set is the set of visited states that have been expanded, these could be stored in a state store via the search adapter.

A large number directed strategies could be described by means of a *best-first* principle [52]. Each iteration the most promising state of the *open* set is extracted from the *open* set and added to *closed*, and new successors are added to *open* set. Which state is most promising is determined by heuristics, which could be implemented by some function $Heuristics : S \longrightarrow \mathbb{R}^+$. Heuristics can be based on the cost of the path to the state as well as the estimated cost to a goal state. More important is that for these strategies the heuristics are only evaluated for states in the *open* set. As the priority queue directly resembles the *open* set, the information required for heuristics can be stored in the priority queue.

Some other strategies, like A^*, allow states in the *closed* set to be reopened into the *open* set if a less costly path to this state is found. This implies that the cost of the cheapest path to states in *closed* must be stored. Not only does this mean that a specialistic store has to be built to deal with these costs, it also implies that the interaction between the store and the search strategy is more elaborate than simple feedback. The advantage of this approach is that it is guaranteed that the goal state with the lowest cost is encountered first.

It is arguable whether the latter class of directed strategies is very desirable to be used in model checking. Although it does present the advantage of finding the 'shortest' path, it does come at the the expense of reopening states and storing additional information in the state store. Figure 5.1 shows how a best-first strategy could see fit in the framework. It could be argued that the `BreadthFirstStrategy` is the same as the `BestFirstStrategy` except for that the former retrieves its states from a normal queue and the latter from a priority queue which happens to have priorities based on heuristics. The most natural way of combining the two strategies would be to consider a breadth-first search a heuristic best-first search with a constant cost of 1 for each transition and the estimated cost of 0 to reach a goal state. The most promising state would be the state with the lowest cost. This cost would be solely defined by the cost of the transitions on the path to the state. As the cost of transitions is constant the most promising state would be the one with the shortest path from the initial state.

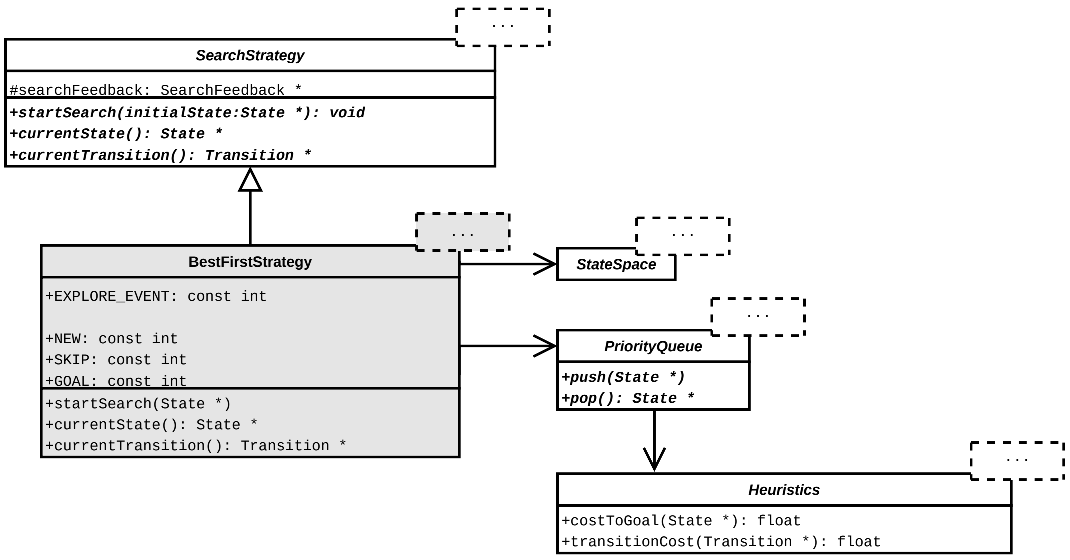

SearchStrategy
#searchFeedback: SearchFeedback *
+startSearch(initialState:State *): void
+currentState(): State *
+currentTransition(): Transition *
...
BestFirstStrategy
+EXPLORE_EVENT: const int
+NEW: const int
+SKIP: const int
+GOAL: const int
+startSearch(State *)
+currentState(): State *
+currentTransition(): Transition *
StateSpace
PriorityQueue
+push(State *)
+pop(): State *
Heuristics
+costToGoal(State *): float
+transitionCost(Transition *): float

Figure 5.1 – The inclusion of `BestFirstStrategy` in the design of the framework.

5.3 Specialised Stores

Efficient use of memory is essential in model checking, because the memory requirements can be the bottleneck of the verification process. Typically, most memory is consumed to remember the set of visisted states in a store. The functionality of a store is realised by means of `Actions` in the generic layer, as is shown in figure 5.2.

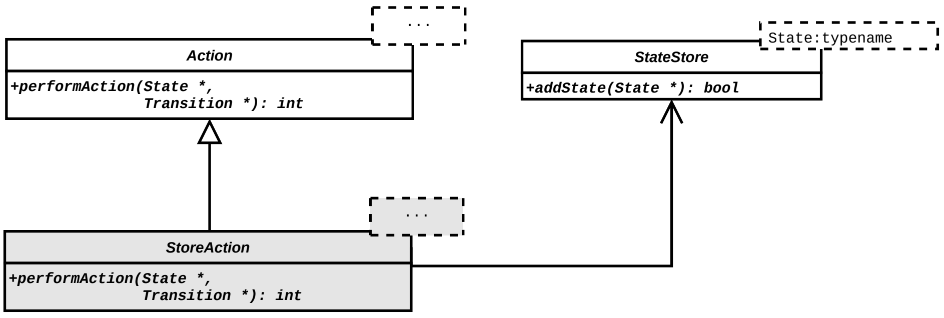

Action
+performAction(State *, Transition *): int
...
StateStore
State:typename
+addState(State *): bool
StoreAction
+performAction(State *, Transition *): int

Figure 5.2 – The implementation of stores in relation to the search architecture described in section 3.3.

Specialising stores to achieve a more efficient use of memory is an important aspect of model checking. In the generic layer we abstract from the type of states being used. Most specialised stores cannot maintain this principle and specialise the store interface for a specific type of states.

5.3.1 State Compression

If states are represented by bitvectors, a more efficient store can be realised by compressing individual states. For instance a Huffman encoding could provide a significant improvement in the significant improvement in the size of the state vector [35].

As our abstract layer represents states as bitvectors after linearisation of the state graphs, stores specialised for bitvectors could be applied to the state resulting from the linearisation procedure given in section 4.2.

5.3.2 State Caching

A store would be more efficient if it didn't have to store every single visited state. The principle of caching could be applied [33]. This means the state store requires functionality for deciding which state is the least interesting and removing this state from the store to free memory.

Most functionality of a caching store could be realised by the store itself. However, if one wants to ensure some states are always cached, such as the states on the current search stack, then additional information is required. The store could be notified by actions hooked on the EXPLORE as well as the BACKTRACK event of the depth-first search strategy by means of actions.

5.3.3 State Collapsing

By using knowledge about the internal structure of states, it is possible to achieve a more efficient use of memory [35]. For instance the software models defined in chapter 4 have a notion of instances. It is possible to collapse states by using these instances. Collapsing could be realised by creating a separate store for instances and by storing states by means of references to this store. This means that identical instances are stored only once in the 'instance store', rather than multiple times as part of states in a normal state store.

One could argue it is possible to implement a collapsing store generically, by abstracting from the type of states as well as the type of components being collapsed. This would require knowledge about the components that can be identified in a state as well as providing methods to extract these components and substitute them for a reference to the 'component store' in the state representation.

It is much easier to implement a collapsing store for a specific type of state and component. An example of this approach is shown in figure 5.3, which collapses `FooStates` by means of extracting `FooComponents`. As this store knows it only ever has to deal with `FooStates`, it can use the internal structure of these states to find components to collapse.

5.3.4 Minimised Automata

If states are represented by bitvectors, then the set of visited states is a set of bitvectors. It is possible to use an automaton with the alphabet $\{0, 1\}$ to represent this set of states [37, 36]. The automaton only accepts the words (e.g. bitvectors) that represent states that have been visited. To see whether a state was already visited is the check of whether the automaton accepts the state's bitvector. Also, algorithms can be defined which insert new bitvectors in the automaton, such that new states can be added. This specialisation would implement a `StateStore` with the state type specialised to bitvectors.

5.3.5 Bit-State Hashing and Hash Compaction

Approximate techniques such as bit-state hashing and hash compaction are also specialisations of stores. They are approximative in the sense that sometimes they can suggest a state was already visited, due to a hash-collision, although in reality the state is explored for the first time [48].

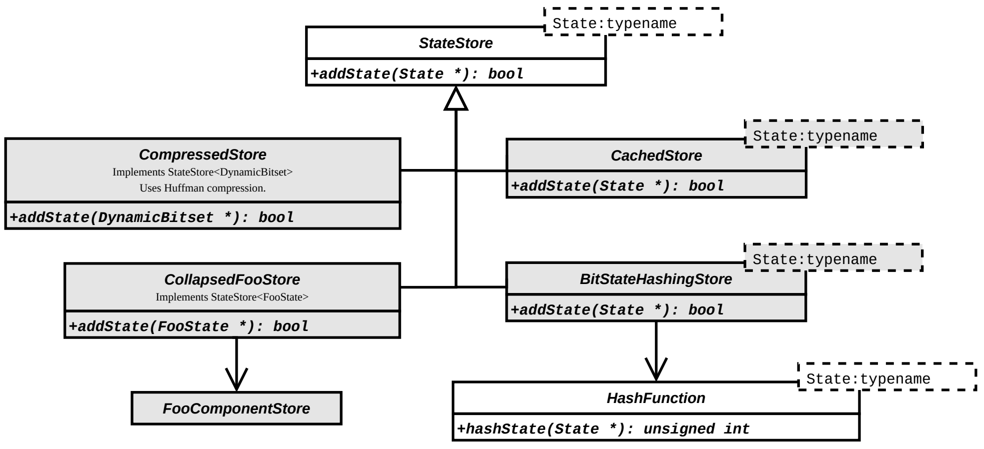


Figure 5.3 – Several techniques in model checking that are to be implemented as specialisations of `StateStore`.

Although these techniques do not require make any assumptions about state types themselves, they do require hash functions (e.g. $HashState : S \longrightarrow \mathbb{N}^{+}$). In figure 5.3 a simple `BitStateHashing` store is presented.

Currently, the framework contains a store specialised for bitvectors that has an internal hash-function and a staticly sized hashtable of 2^{16} elements.

5.4 Reduction Methods

Reduction methods such as partial-order reduction and symmetry reduction have no obvious place in the framework, as did the new search strategies and specialised stores.

5.4.1 Partial-Order Reduction

Partial-order reduction is a means of reducing the state space by exploiting the commutativity of transitions in models with asynchronously executing components. See chapter 10 of [19] and [49, 28, 32, 55] for a discussion of this technique in literature.

Partial-order reduction strategies require information about processes, dependencies of transitions, control-flow of processes and other high-level features that do not exist in the generic layer. We categorise strategies for partial-order reduction as follows:

Static partial-order reduction Reduction strategies that only use the information present in the abstract layer, without the help of run-time information in the generic layer, for example [49].

Dynamic partial-order reduction Reduction strategies that do require information from the generic layer, such as the states on the depth-first search stack. An example of this category is described in chapter 10 of [19].

The need to use run-time information, such as the search stack, is a result of the **C3** condition [19], which excludes certain cycles from the reduced state space. Dynamic algorithms use the search stack for cycle detection whereas static algorithms statically satisfy stricter requirements that guarantee **C3** is never violated [49].

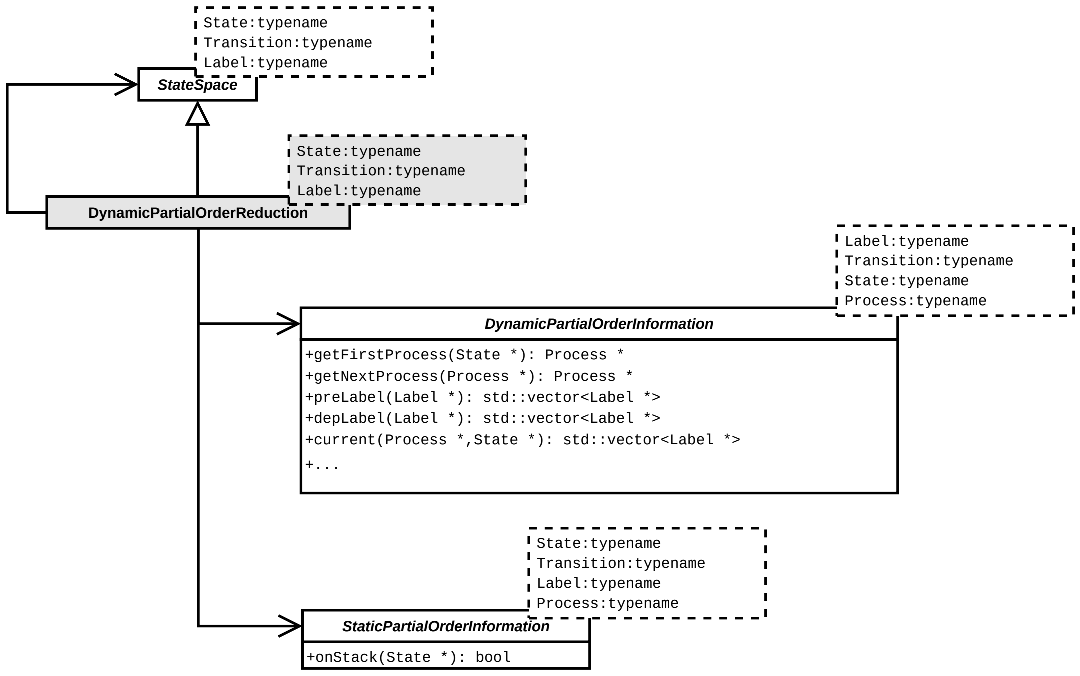
A partial-order reduction results in a reduced state space, which consists of the same type of states, labels and transitions as the original state space. We could offer information about the reduced state space in several ways:

- Provide means of querying the reduced state space in the STATESPACE interface in addition to the existing methods that query the normal state space. The generic algorithms in the generic layer would have to be adapted to be able to exploit the reduced state space information.

- Implement the state space interface with the reduced state space information. This implies all algorithms simply work on the reduced state space, and require no alteration.

Ideally, one would like to implement each partial-order reduction algorithm generically, rather than on the level of abstract layers. Consider the algorithm described in [19], a generic implementation would require a lot of information, such as the processes in per state, dependency relations of transitions, information about control-flow, the property being checked etc. This implies that it is also required to abstract from new types, such as process types.

A possible generic design of the partial-order method described in [19] is presented in figure 5.4. The `DynamicPartialOrderReductionInformation` will have to be implemented somewhere in the generic layer, for instance by the depth-first stack.



State:typename
Transition:typename
Label:typename
StateSpace
State:typename
Transition:typename
Label:typename
DynamicPartialOrderReduction
Label:typename
Transition:typename
State:typename
Process:typename
DynamicPartialOrderInformation
+getFirstProcess(State *): Process *
+getNextProcess(Process *): Process *
+preLabel(Label *): std::vector<Label *>
+depLabel(Label *): std::vector<Label *>
+current(Process *,State *): std::vector<Label *>
+...
State:typename
Transition:typename
Label:typename
Process:typename
StaticPartialOrderInformation
+onStack(State *): bool

Figure 5.4 – Design of a generic implementation of the partial-order reduction algorithm as described in [19].

Note that it is much simpler to implement partial-order on the level of abstract layers, as these would have information regarding the processes and control-flow readily available. However, this approach would not allow reuse of the partial-order functionality for other abstract layers.

5.4.2 Symmetry Reduction

The symmetries that can be found in the state spaces of models depend on the model type, and can range from heap and process symmetry in models based on software to symmetry defined by rotations and permutations of states in the state space of a Rubik's cube.

Lets consider the formal definition of symmetries as defined in [52] which uses group theory [31]. There can be a number of symmetries in a state space $\langle L,\ S,\ I,\ R\rangle$, which are defined as bijection on S. For instance, software-based model could define separate bijections to realise both heap symmetry and thread symmetry. Symmetries defined by bijections form a group under function composition and thus allow the composition of multiple symmetries. A symmetry defined by a bijection induces a new state space, or *quotient system*, where each state is a class of symmetric states (e.g. equivalence class) defined by the orbit of the bijection.

The most common way of model checking quotient systems is by using representative functions $Rep : S \longrightarrow S$ which maps each state in the state space to a representative state of its equivalence class. Ideally, one would like Rep to be canonical, which means that $Rep(s) = Rep(s')$ implies that s and s' are symmetric.

It is not always possible to define *Rep* canonically, and if it is possible, it might still be preferable to settle for a underestimation of symmetry if the computational complexity of *Rep* is too costly. In [11, 52] it is explained how the inclusion of a representative function changes a depth-first strategy. To include symmetry into the design we present two alternative solutions:

- One could consider the quotient system as the state space to be verified. In our framework this implies a transformation is applied from original state space to a quotient system. These transformations were explained in section 3.4. An example setup to realise such a transformation is shown in 5.5.

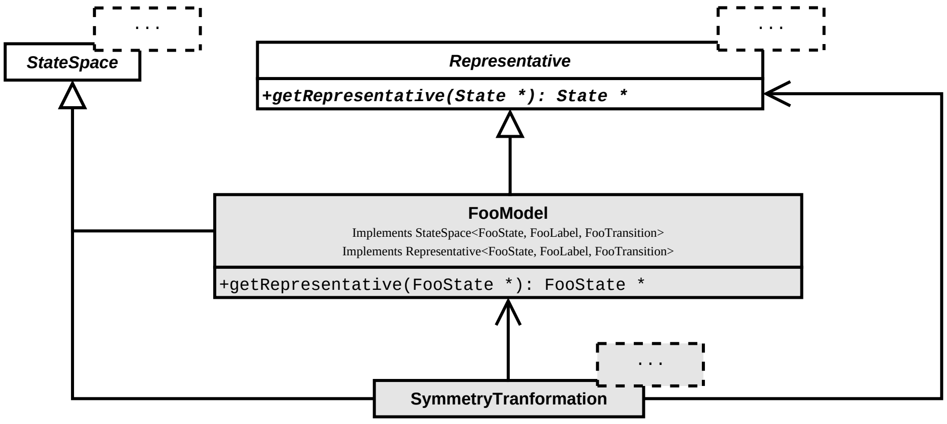

...
StateSpace
...
Representative
+getRepresentative(State *): State *
FooModel
Implements StateSpace<FooState, FooLabel, FooTransition>
Implements Representative<FooState, FooLabel, FooTransition>
+getRepresentative(FooState *): FooState *
...
SymmetryTranformation

Figure 5.5 – Implementation of symmetry reduction by means of a state space transformation using a representative function.

  There is a downside to this setup. Consider a verification run where a mutual exclusion property is being checked using thread-symmetry. If the property is violated a counter-example can be generated. Due to the transformation the counter-example will be based of representative states. As the representatives are based on thread symmetry it will be difficult to find out which process is responsible for each transition in the counter-example.

- The alternative option is to use the algorithm presented in [11]. Instead of a verifying a quotient system the knowledge of symmetry is only used when deciding whether a state was previously visited or not. Although the search strategy and search feedback will still work on the original states, a state store is responsible for storing representatives rather than the original states. Figure 5.6 shows how this principle could look in practice.

Our software models use the last alternative. As the introductory chapter introduced three types of symmetry, we will explain how these can be applied when using `SoftwareModels`. The idea is that we do not provide a representative in the form of a `SoftwareState`, but rather present a symmetric version of the linearised state. These are bitvectors where the order of process instances does not depend on their process identifiers, but on other heuristics instead.

Heap symmetry As our state representation has no notion of memory addresses or object identifiers, the heap is solely defined by the structure of the state graph and the internal data of instances. As our representation abstracts from memory addresses anyway, there is no additional functionality needed to implement heap symmetry.

Symmetric data types The introduction of scalar sets in our framework would require an implementation of a `DataType` and `DataInstance`. The symmetry in a scalar sets could be directly implemented in the linearisation function of the scalar type, as a scalar set is by nature always symmetrical.

Thread symmetry Thread or process symmetry is based on a symmetry of process instances. We consider two states in the process layer as thread-symmetrical if the structure of the state

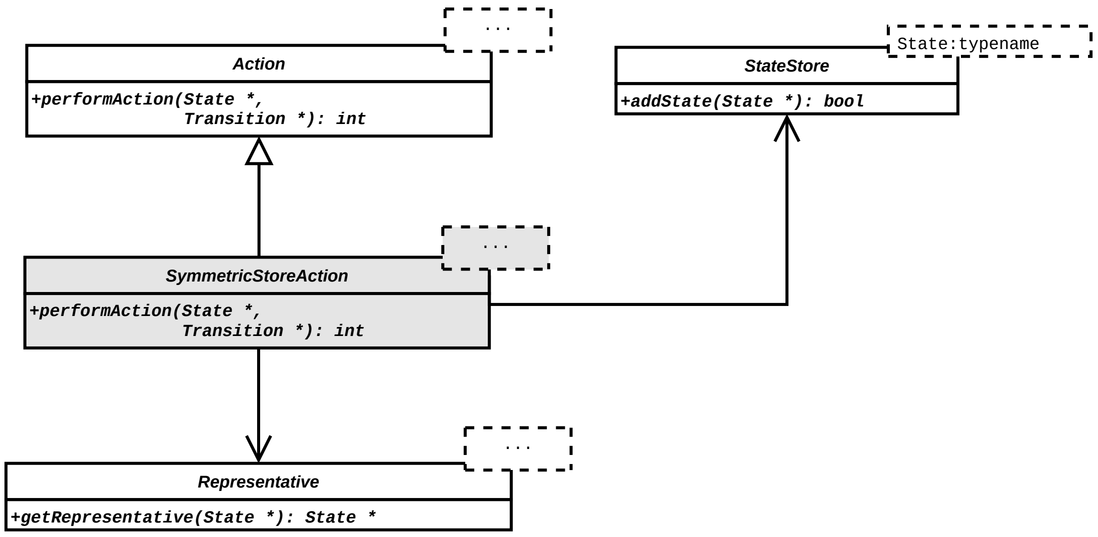

...
Action
+performAction(State *, Transition *): int
State:typename
StateStore
+addState(State *): bool
...
SymmetricStoreAction
+performAction(State *, Transition *): int
...
Representative
+getRepresentative(State *): State *

Figure 5.6 – A symmetric store action using a representative function.

graphs and the internal data of all instances are identical, with the exception of the process identifiers in the process instances.

As a representative function for thread-symmetry we try and order the process instances in such a way that many state graphs that are thread-symmetrical are linearised in the same way. Therefore we need to change the $ProcessIndex_\sigma$ function as process identifiers are no longer relevant to the order of inclusion of the process instances. The process identifier was the only means of distinguishing process instances that were identical in both type and control-flow state.

Firstly, we try and sort process instances by type, and then by other heuristics such as process counter (e.g. current control-flow state). If these do not distinguish between two process instances, then heuristics based on the position of the instance in state graph can be used, such as the number of frames on the call stack (represented by the $*$ edges), and the **κBOTS** algorithm [60].

The current implementation compares only process types and control-flow states. To realise this we need an arbitrary total order on process types $\sqsubseteq_{\mathcal{M}_{Proc}} \subseteq (\mathcal{M}_{Proc} \times \mathcal{M}_{Proc})$. The redefinition of bijection $ProcessIndex_\sigma$ assigns numbers $1, \ldots, |\sigma_{\underline{Proc}}|$ to process instances and with the following constraint with respect to the order of types.

$$ProcessIndex_\sigma(p) < ProcessIndex_\sigma(p') \implies \tau_{\sigma_{\underline{Proc}}}(p) \sqsubseteq_{\mathcal{M}_{Proc}} \tau_{\sigma_{\underline{Proc}}}(p')$$

Furthermore, if $\tau_{\sigma_{\underline{Proc}}}(p) = \tau_{\sigma_{\underline{Proc}}}(p')$ then $ProcessIndex_\sigma(p) < ProcessIndex_\sigma(p')$ is restricted by a total order on the reachable control-flow states of the process type at hand. As we have not introduced any formal means of describing the control-flow, we presume that this informal desciption is sufficient to understand change in the order of inclusion of process instances in the linearisation process.

When encoding the process instances we omit the linearisation of the process identifiers. This results in a bitvector that is strictly speaking not a state of the original model, but it is a representative of the equivalence class.

It should be taken into account that in order for this reduction to be correct the process identifiers should not have a significant influence on the semantics of the model. For instance, if the order in which the process instances can die is dependent on the order of processes in the state then it can be the case that this reduction is not correct. To be more specific, it should never be the case that by changing the order of the process instances we can reach a goal state that could not be reached before.

5.5 Additional Resources

As was explained in section 1.2.5, performance improvements in tools can be accomplished by finding alternative resources such as hard drives or by using distributed verification techniques. Unfortunately, using these principles has a profound impact on the search architecture.

5.5.1 Use of External Storage

In terms of design the use of external storage is easily implemented as a specialisation of `StateStore`, which could use some external storage device to realise its functionality. However, in practice a problem arises when considering that these external devices are often slow, which makes a store action to external devices expensive.

Rather than storing one state at a time, one could combine the store action of several states into one action [62]. For instance, a breadth-first strategy could combine the store action of each level of states, rather than storing them individually. The problem is that our current strategies request feedback for each single explorative step.

A solution is to introduce a new strategy, with a new event (say BATCH_EXPLORE) that triggers each time the strategy wants a group of states checked. The arguments of the feedback function would not no longer provide sufficient information, as a set of states is required, rather than a single state. Also the feedback given by the search adapter is insufficient because feedback is required for each of the states in the batch rather than feedback for a single state in terms of SKIP or CONTINUE. This could be realised by means of some additional communication between the search strategy and the state store. The current design of the search module does not provide a means of directing this communication through the search adapter.

5.5.2 Distributed Verification

It is possible to verify large state spaces using distributed model checking. By distributing the memory and computational requirements of the verification process over several processing units, larger state spaces can be verified.

Unfortunately, the current search strategies and in particular the depth-first strategy are not easily converted into their distributed variant. Especially the distributed search for acceptance cycles is an active field of research, as is illustrated by [6, 5, 4, 14].

The inclusion of distributed algorithms in the framework is addressed only conceptually, as a more detailed description would require an extensive study of distributed algorithms.

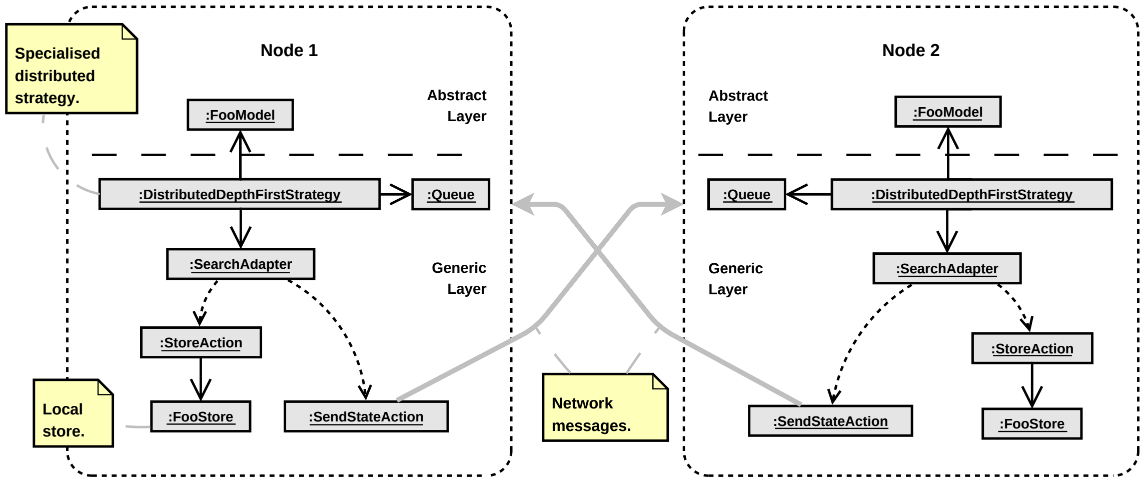


Figure 5.7 – Conceptual setup of a distributed search.

Typically, all distributed algorithms divide the state space over the nodes in the network. Consider the distributed depth-first algorithm presented in [51]. If there are n nodes in the network, a function $Node : S \longrightarrow \{1, \ldots, n\}$ assigns the states in the state space to the nodes. Each node has a local queue of states to explore. If during exploration a state is found that does not belong to the current node, then a network message is used to send this state to the correct node.

Figure 5.7 introduces a conceptual object diagram where two nodes are verifying the same state space. A special distributed search strategy is used as well as special `SendStateActions` to send network messages to the other node. This diagram does not take into account the details that would be required for distributed verification, such as a manager process, or how the network messages result in a state being added to the local queue.

The two nodes do not physically share the objects in the abstract layer, such as state graphs. It should be noted that the abstract layer is usually a means of constructing the state space on-the-fly, and does not impose significant memory requirements.

Chapter 6

The Tool Layer: Model Checking Prom$^+$

In this chapter we will discuss the usage of the framework to model check a simple model specification language called **Prom**$^+$. **Prom**$^+$ is meant to illustrate the usage of the generic and abstract layers presented in this thesis. It is in fact a tool in the 'tool layer' as depicted in figure 2.3 in chapter 2.

6.1 The Modelling Language Prom$^+$

As the name implies **Prom**$^+$ is based on a very small subset of **Promela** with a few changes and additions. This section will explain the syntax and semantics of **Prom**$^+$, and relates the tool layer to the abstract layer presented in chapter 4.

6.1.1 Syntax

The syntax of **Prom**$^+$ is given in the **EBNF** in figure 6.1. It is based the syntax of **Promela** as defined in [36]. A few items in the grammar require further explanation:

- Declarations and statements are strictly separate. Each process definition should start with the declaration of any local variables. It is not allowed to explicitly intialise variables in the declaration. The reason for this restriction is that intialisation is seen as an action (or rather a transition in the control-flow), whereas declarations are not.
- **Prom**$^+$ only supports primitive types. Potential extensions of **Prom**$^+$ could introduce new types, such as channels, arrays, mtypes, typedefs or other types.
- **Prom**$^+$ supports the dynamic creation of objects. These 'heap' objects can only be assigned to pointer variables. Similarly pointer variables can only refer to dynamically allocated objects. Both pointer variables and dynamic object creation do not exist in **Promela**, therefore section 6.1.2 discusses this subject in more detail.

Besides the semantics of pointer variables and the '`new`' and '`reset`' statements, the semantics are identical to the semantics of **Promela** as they are described in [36]. To illustrate the usage of **Prom**$^+$-models, listing 6.1 provides an example of a **Prom**$^+$ specification of Peterson's mutex algorithm [8, 56].

6.1.2 Pointer Variables and Dynamic Object Creation

The **Prom**$^+$ language includes the notion of pointer variables similar to **dSPIN**'s 'pointers' [41, 26]. This section describes how pointers are included in the state representation, which was introduced in section 4.1.3. Consider a state where the global instance ι has two variable values associated with it, namely a **dSPIN**-style pointer variable `int & x_ptr` which currently points to some integer instance, and a normal variable `int x`. This could be represented in two ways:

```
        prom ::= (mult_decl ‘;’)* (proctype ‘;’)+
        decl ::= type (‘*’)? ident
   mult_decl ::= type (‘*’)? ident ( ‘,’ (‘*’)? ident )*
    proctype ::= ‘active’ ‘[’ number ‘]’ ‘proctype’ ident ‘(’ ( params )? ‘)’
                 ‘{’ (mult_decl ‘;’)* (stmnt ‘;’)+ ‘}’
      params ::= decl ( ‘;’ decl )*
        type ::= ‘bit’ | ‘bool’ | ‘byte’ | ‘short’ | ‘int’
       stmnt ::= do_stmnt | if_stmnt | assgn_stmnt | new_stmnt | reset_stmnt | run_stmnt |
                 expr | assert_stmnt | ‘skip’
    do_stmnt ::= ‘do’ (branch)+ ‘od’
    if_stmnt ::= ‘if’ (branch)+ ‘fi’
      branch ::= ‘::’(‘else’ ‘;’)? (stmnt ‘;’)* (‘break’ ‘;’)?
 assgn_stmnt ::= (‘*’)? ident ‘=’ expr
   new_stmnt ::= ident ‘=’ ‘new’ type
 reset_stmnt ::= ‘reset’ ident
    run_stmnt ::= ‘run’ ident ‘(’ ( args )? ‘)’
        args ::= expr ( ‘,’ expr )*
        expr ::= expr ( ‘<’ |‘<=’ | ‘>’ |‘>=’ | ‘==’ |‘!=’ | ‘&&’ |‘||’ | ‘+’ |‘-’ | ‘*’ |
                 ‘/’ | ‘%’ ) expr | ( ‘!’ |‘-’ ) expr | ‘(’ expr ‘)’ | ‘true’ | ‘false’ |
                 number | ( * )? ident
assert_stmnt ::= ‘assert’ ‘(’ expr ‘)’
       ident ::= ( ‘a’ | ... | ‘z’ | ‘A’ | ... | ‘Z’ | ‘_’ )+
      number ::= ( ‘0’ | ... | ‘9’ )+
```

Figure 6.1 – The grammar of **Prom**$^+$ in **EBNF** style. The syntax and semantics of Prom$^+$ are based on **Promela** [36].

- Pointers could be types, as is shown in figure 6.2(a). The advantage of this approach is that the state graph representation does not need to be altered. Modelling pointers as types would enable pointers to pointers, and expressions of pointers (pointer arithmetics). However, this implementation would introduce a lot of new concepts such as pointer types, dereferencing expression, and the left-value operator [41], which would complicate the abstract layer. Also adding an additional node per pointer variable would enlarge the state representation significantly, although the optimisation given in section 4.2.3 would be capable of minimising the cost.
- Pointers could be special edges, as is shown in figure 6.2(b). The reasoning is that an edge in a graph is already a ‘reference’ to an instance, and therefore there is no need to introduce new types for pointers. It would also avoid making the state graph bigger due to pointer usage, and would not introduce a significant amount of new functionality in the abstract layer.

The current implementation uses the latter approach, as it keeps the state representation at a lower size. Also there is no need for pointer arithmics (as we have to memory addresses to perform arithmics with), and if ever a pointer to a pointer was required, this could be modelled by means of a ‘typedef’ type with a pointer field. A typedef type in **Promela** is similar to a ‘struct’ in C++.

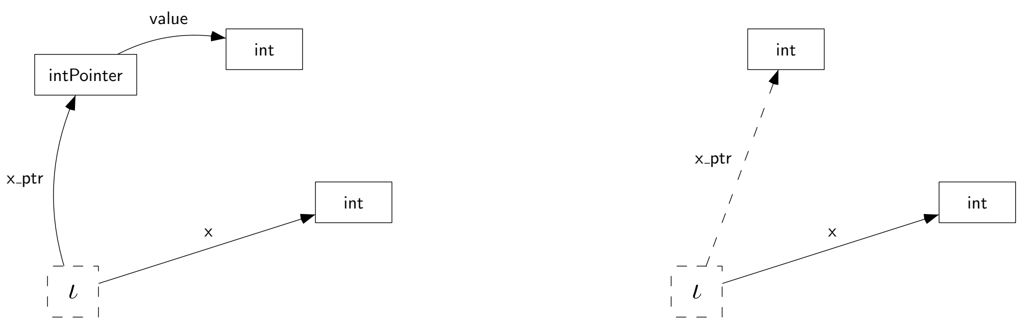
However, besides state representation there are also a lot of semantics involved when considering pointer variables. Note that the state representation we use, opposed to **dSPIN**, does not have a notion of heap or stack instances, nor does it have a notion of memory locations. A further limitation is that the linearisation algorithm implies that only reachable instances in the state graph are taken into account. This introduces some problems when trying to implement the same semantics as **dSPIN**, which follows the semantics of unmanaged C++.

- We have no *direct* means of detecting whether an object was allocated on the heap or on the stack. Listings 6.2 and 6.3 show how this detection is necessary during deletion. Obviously

Listing 6.1 – A Prom$^+$ model of Peterson's mutex algorithm, unsuitable for symmetry reductions due to the turn values and the `Init` process.

```
byte mutex; bit * flag_1, * flag_2, turn;

active [1] proctype Property() { assert(mutex < 2); }

active [0] proctype Process(
    bit * my_flag;
    bit * other_flag;
    bit turn_value)
{
    do
    ::  *my_flag = 1;
        turn = turn_value;
        (*other_flag == 0 || turn != turn_value);

        /* BEGIN CRITICAL SECTION. */
        mutex = mutex + 1;
        mutex = mutex - 1;
        /* END CRITICAL SECTION. */

        *my_flag = 0;
    od;
}

active [1] proctype Init()
{
    mutex = 0;
    turn = 0;
    startGuard = false;

    flag_1 = new bit;   *flag_1 = 0;
    flag_2 = new bit;   *flag_2 = 0;

    run Process(flag_1, flag_2, 0);
    run Process(flag_2, flag_1, 1);
}
```

the presented code results in a run-time exception as one is not allowed to delete instances that are not on the heap. To realise this with our state representation one would need to be able to distinguish between stack and heap objects.



(a) Pointers as types. (b) Pointers as special edges (dashed).

Figure 6.2 – Representation of pointers in the state graph, either as a separate type or as a special edge.

Listing 6.2 – Deleting stack variables in C++

```
int main() {
    int x;
    int * x_ptr;

    x = 1;
    x_ptr = &x;

    delete x_ptr;
}
```

Listing 6.3 – Deleting stack variables in DSPIN

```
active [1] proctype Main() {
    int x;
    int & x_ptr;

    x = 1;
    x_ptr = &x;

    delete x_ptr;
}
```

One solution is to denote some instance as a stack instance if it has a variable referring to it that is not marked as a pointer variable. Note that this would be an expensive operation with the current implementation, as we do not maintain back-edges for variable values. Other solutions involve adding additional information to the state graph to mark data instances as heap instances or stack instances. This could be done in the internal representation of the data instances, or by globally maintaining a list of heap instances.

- If multiple pointers reference the same data instance, an this heap instance is deleted, this would resolve in 'dangling pointer'. These are pointers that refer to an invalid memory address. Our state representation is simply a graph, we cannot refer to a node that isn't in the graph. An alternative would be to include an additional node to which pointers point once their right-hand value is destructed. Note that without back-edges, it is an expensive operation to find all pointers that refer to a certain instance.

Besides adding new functionality to the framework by solving the problems of pointer semantics above, there is also the option of adopting different semantics. Rather than modelling C++-like semantics, like DSPIN, one could also adopt the JAVA approach. Heap and stack are kept totally separate in the sense that pointer variables can only refer to heap objects, not to stack objects, and normal variables can only refer to stack objects. Also, there is no possibility to explicitly delete heap objects, as a heap object is deallocated by the garbage collector only when there are no more pointer variables that refer to it. These semantics are much easier to implement in our framework, as it already has a 'garbage collector' (only reachable instances are taken into account in the state graph).

This does mean have different semantics compared to DSPIN; in particular we have no left-value operator, as we don't want pointer variables to point to stack variables. Also we have no `delete` statement, as this would imply a heap variable would be destructed. Instead we have a `reset` statement, which makes the pointer variable change into a null pointer, which is synonymous with the removal of an edge in the state graph. The garbage collector is responsible for removing any unreferenced instances. Listing 6.4 shows how the semantics of PROM$^+$'s pointers differ from DSPIN's.

6.1.3 Semantics

It might be unclear how PROM$^+$ models can be mapped to our abstract layer. We will discuss how we can create a type graph of a PROM$^+$ specification by means of the grammar in figure 6.1.

- The process types ($\mathcal{M}_{Proc}$) are synonymous with the `proctype` definitions defined by the *proctype* rule of the grammar. Each process type is a node in the type graph. The typename associated with a process type is defined by the identifier (*ident*).

- The function types ($\mathcal{M}_{Func}$) are non-existent in PROM$^+$, as it does not contain a notion of functions. This implies that also $\mathcal{M}_{Call}$ is empty.

Listing 6.4 – A PROM$^+$ model illustrating the pointer semantics.

```
int * x, * y; int * null;

active [1] proctype Main() {
    x = new int;

    *x = 4;      /* Notice the C++ syntax. */
    y = x;

    reset x;

    *y == 4;     /* No run-time error, y still exists! */
    reset y;     /* The int instance will be destroyed. */
    y == null;   /* y equals null. */
}
```

- The data types ($\mathcal{M}_{Data}$) are defined by the *type* rule in the syntax. The range of these primitive types are same as their counterparts in PROMELA [36]. For each of these types there is exactly one node in the type graph.

- The variables ($\mathcal{M}_{Var}$) are the labels of the type graph. They are defined by the rules *decl* and *mult_decl*. Note that his includes the parameters in process definitions. Note that the inclusion of '`*`' in a declaration marks the variable as a pointer variable.

  The scope of the variables ($\mathcal{M}_{Scope}$) is determined by the location in which they are defined. If the variable was directly declared from within a *mult_decl* in *prom*, then it maps to ι as it is a global variable. If it is a result from the *mult_decl* or *params* in *proctype*, then the scope of this variable is the process type. The type of the variable ($\mathcal{M}_{VarType}$) is the data type of the *type* field in the *decl* or *mult_decl* rule that declared the variable.

Besides constructing the type graph, the typing information also contains a control-flow for each process type. These control-flow graphs should not be considered as a subgraphs of the type graph, but rather as the internal information of process types, as depicted in figure 4.7. Each transition in the control-flow graph can be enabled or disabled depending on the global state of the model. In section 4.1.6 it was explained how a control-flow relates to the `SoftwareModel`, however, it is yet to be explained how such a control-flow is created from a PROM$^+$ specification.

Each process type has an initial control-flow state. Normal statements such as those defined by grammar rules *assgn_stmnt*, *new_stmnt*, *reset_stmnt*, *run_stmnt*, *expr*, *assert_stmnt* and the terminal '`skip`' are simply edges in the control-flow graph, which follow each other sequentially. Each of these statements maps to a subclass of `Statement` in our abstract layer. We will briefly discuss the semantics of each of these statements:

- The assignment statement (*assgn_stmnt*) can either assign an expression to a variable by value, or assign a pointer variable to another pointer variable by reference. This depends on how the variable on the left-hand side is defined. If this is declared as a pointer, then it can be dereferenced by using an additional *. An assignment is always enabled.

- The new statement (*new_stmnt*) allocates a new instance of a data type to a pointer variable with the default value of this type. This statement is always enabled.

- The reset statement (*reset_stmnt*) removes the value of a pointer variable. If this results in a unreferenced data instance, then this instances is removed from the state graph. A reset statement is also always executable.

- The run statement (*run_stmnt*) adds a process instance of a certain type to the state graph. The variable values receive a default initialisation and the arguments are evaluated. A run statement is only executable if there is a process identifier available within the bound dictated by the number of bits reserved to represent the process identifier. Normally 8 bits are reserved and therefore 256 process instances can be created. The number of bits reserved for this purpose can be changed by a command-line argument in the PROM$^+$ tool.

- The expression statement (*expr*) does not change the state. However, an expression statement is only executable if the expression evaluates to a value that is not 0. Note that all expression are evaluated as integers, just like in PROMELA.
- The assertion statement (*assert_stmnt*) does not influence the semantics of the model, but is only useful for the verification.
- The skip statement (`'skip'`) is always executable, and changes nothing in the state graph.

The *if_stmnt* and *do_stmnt* are less obvious. They are not a single transition in the control-flow, but are constructs that introduce non-determinism. Each 'if' and 'do-statement' consists of a number of branches, which in turn consist of a sequence of statements. Both the 'if' and the 'do-statement' give the process a choice as to which branch to go into, as long as the executability of the transitions in the branches is not violated. Therefore, 'if' and 'do' statements are not executable if the first statements of all their branches are not executable. The `'else'` construct can be used in the last branch of an 'if' or 'do-statement' to enable a branch if all other branches are not executable. The difference between the 'if' and 'do-statement' is that after the sequences in the branches the 'do-statement' routes the last transition of this branch to the beginning of the 'do-statement', thereby creating a loop, whereas an branch of an 'if-statement' does not form a loop. The `'break'` construct can be used to 'break' out of a 'do-loop'.

Finally, the control-flow of processes is always implicitly concluded with a statement that kills the process instance. This statement is only enabled if the process instance has the highest process identifier of all process instances in the state. Note that this statement is not always reachable.

The initial state of a PROM$^+$ model consists of a global instance (ι). Also the `'active'` construct provides the number of instances of a certain process type that are to be present in the initial state. These are given process identifier according to the order in which the processes are specified in the PROM$^+$ specification. Note that as the global instance and process instances can also contain variables, these variabels are given a default value in the initial state. This is '0' or 'false' for normal variables of a primitive type, whereas pointer variables initially have no value.

6.1.4 Prom$^+$'s Language Features in Comparison

In terms of the language features described in section 1.2.3, PROM$^+$ is quite limited, as is illustrated by table 6.1. Although PROM$^+$ itself only exhibits only a small number of features, it does not use the full potential of the abstract layer. For instance, the design of the abstract layer included the notion of functions, but these are not included in PROM$^+$. It should be noted that the features that are ascribed to the abstract layer have not all been implemented, but are considered compatible with the current design.

It might be unclear how these features map to the principles described in previous chapters, therefore the features will be briefly discussed per category.

Control-flow The features in the control-flow section are means to define a control-flow graph in the typing information of the abstract layer. In the abstract layer this information is only present in the form of a local state space. Given a global state and a process instance a transition in the control-flow can be enabled or not. Jumps are simply transitions in this control-flow state space and could be added quite easily. Functions allow the hierarchical composition of control-flow on a function stack, and as we included the notion of functions our state graph these are also are compatible with out abstract layer. Note that in the implementation of the abstract layer, functions have not been included in order to speed up the implementation of this layer. Exception handling would require additional functionality in the abstract layer to simulate exceptions being thrown down function stacks. Parallelism of processes is very clearly present in our abstract layer, as is illustrated by the possibility of having multiple process instances in a state.

Table 6.1 – An overview of Prom$^+$'s language features by means of the features described in section 1.2.3.

		Control Flow				Comm.			Data Abstraction						Miscellaneous				
	Semantic Model	Jumps	Function Stacks	Exception Handling	Parallel Threads	Atomic	Locks	Channels	Composite Types	Classes	Inheritance of Data	Method Overriding	Custom Data Types	Symmetric Data Types	Dynamic Data Creation	Dynamic Thread Creation	Assertions	Clocks	Probabilities
Prom$^+$	LTS				✓										✓	✓	✓		
Abstract Layer	LTS	✓	✓		✓		✓	✓	✓	✓	✓		✓	✓	✓	✓	✓		
Promela	LTS	✓			✓	✓		✓	✓							✓	✓		

Communication The only means of communication between parallel components in our abstract layer is currently by means of variables. Atomicy could be added by adding a single bit flag to each process instances to indicate atomic execution, and by including this knowledge in the $getFirstTransition$ and $getNextTransition$ of the `SoftwareModel`. Locks and channels could be included by defining additional data types, and by creating special statements which enforce the semantics of these objects.

Data Abstraction This category maps to the data types in the abstract layer. Composite types could simply be specialised data types, whereas classes could map to a specialised composite type and some function types in the type graph. Inheritance could be simulated by means of associations between composite types. However, dynamic method overriding would require additional run-time information and thus a change in the state graph definition. Symmetric data types have been discussed in section 5.4.2, and can be implemented by means of a custom data type.

Finally, the dynamic creation of objects and threads are simply statements that add additional data instances and process instances to a state graph. Assertions are a special type of statements that don't change the state graph, but that do trigger `AssertionConditions`, such that assertion violations can be detected during a search.

The inclusion of probabilities or clocks is not possible in the abstract layer as the generic layer does not include these notions. They would require changes in the generic layer as did symbolic and bounded model checking.

6.2 The Tool

As might be clear from the previous section most functionality in the PROM$^+$-tool is there to to construct the correct type graph from the syntax of the model. This includes constructing the control-flow of process types, and the declaration of global as well as local variables. Most of this functionality is implemented in a parser and in a tree walker that walks over the **AST** of the PROM$^+$ specification.

6.2.1 Tool Implementation

The PROM$^+$ tool illustrates how a tool can use the framework. Both the framework and the tool are c++ projects. The framework implements both the generic layer (`mcf::generic_layer`) and the abstract layer for software models (`mcf::software_layer`). The framework is then compiled as a static library, and used in the tool. The PROM$^+$ tool contains a parser that creates a model, in terms of the abstract layer in the framework, from a **Prom**$^+$ specification. The simulation and verification functionality provided in the generic layer of the framework is used to provide simulations and verifications for these models. The framework consists of approximately **6 KLOC**, and the **Prom**$^+$ tool consists of approximately **6 KLOC** of which most is code that is generated by a parser generator.

The tool is capable of simulating a **Prom**$^+$ model, but can also simulate the control-flow of process types. The verification can target assertion violations as well as deadlocks. Finally, the tools also realises thread-symmetry reductions.

6.2.2 Performance

To put the performance of our tool into a context it is compared to the state-of-the-art tool **SPIN**. In order to compare the two tools we specify some models that are both valid **Prom**$^+$ and **promela** specifications. Besides this we test the effectiveness of the symmetry reductions by comparing verification runs of the **Prom**$^+$-tool with symmetry disables to runs where symmetry reduction was enabled. To do this we require **Prom**$^+$-models that exhibit thread-symmetries.

We developed models of three mutex algorithms, namely Peterson's, Dekker's and Dijkstra's algorithms [8, 56], as well as models of the dining philosophers problem. For these we have created two sets of models, one that can be checked by both **Promela** and **Prom**$^+$ and an equivalent model suitable for thread-symmetry reductions for just **Prom**$^+$. They can be found in appendix D, ranging from code listings D.1 up to D.8. The dining philosophers were verified for both 3 and 4 philosophers.

Note that in order to create **Prom**$^+$ models that are suitable for symmetry reduction we need to make sure that the permutation of symmetric processes does not influence the result of the linearisation of the state graph. In terms of mutex algorithms this means that the initial process needs to reset all the pointer variables used in the other processes, as it would otherwise include them in its linearisation. Also, instead of using a turn value, we use a pointer variable to model the 'turns' in mutex algorithms. In this way the linearisation process does not distinguish between the two process due to the value of the turn variable as was the case in listing 6.1.

The mutex algorithms were verified for deadlocks and assertions, whereas the dining philosopher models were only checked for dummy assertions (to make sure the complete state space was visited). The models in appendices were compared by means of the following metrics:

States visited The number of unique states in the state space that was visited during the verification run.

States revisited The number of times a state was revisited during the verification run. This means that the number of transitions explored during the verification run is equal to the number of *states visited* plus the number of *states revisited*.

Verification time The time it took the tool to execute the verification algorithm. This excludes initialisation work such as parsing, code generation or intialisation of the state store. By changing the source code of both the generated **SPIN**-models and the **Prom**$^+$-tool the time required by the verification procedure was measured. The time presented in table 6.2 is the average of 5 verification runs.

Statevector size The average size of the statevectors in the model rounded to the nearest byte. This is based on the default settings of both tools. In the **Prom**$^+$-tool this implies a default setting of 8 bits per process identifier and 8 bits per reference to an instance in the state graph and no optimisations (see 4.2.3).

Memory usage The maximum amount of memory used by the tool, measured in megabytes. The measurements obtained are only indicative.

The experiments were performed on a computer with an **AMD XP 2000+** processor and **512MB RAM**. Both the **Prom**$^+$-tool and the generated code by **SPIN** were compiled in **GCC 3.4.4** (**MinGW**) with identical optimisation flags. Both tools were run in their default settings, with the exception that the partial-order reductions in **SPIN** were turned off. Table 6.2 presents the results of the measurements.

Table 6.2 – A comparison in performance of **PROM$^+$** against **SPIN**. Both the default settings (‘Prom$^+$’) and the results of verification with symmetry reduction (‘Prom$^+$ (with thread symmetry reductions)’) is shown.

			States Visited	States Revisited	Verification Time (seconds)	Statevector Size (bytes)	Memory Usage (mega bytes)
Peterson	D.1	Prom$^+$	**64**	**88**	**0.08**	**13**	**0.7**
		Prom$^+$ (with thread symmetry reductions)	**64**	**88**	**0.09**	**10**	**0.7**
		SPIN	**64**	**88**	**< 0.01**	**20**	**2.6**
	D.2	Prom$^+$	**126**	**138**	**0.23**	**23**	**0.7**
		Prom$^+$ (with thread symmetry reductions)	**82**	**82**	**0.18**	**19**	**0.7**
Dekker	D.3	Prom$^+$	**288**	**398**	**0.35**	**13**	**0.7**
		Prom$^+$ (with thread symmetry reductions)	**288**	**398**	**0.48**	**10**	**0.7**
		SPIN	**288**	**398**	**< 0.01**	**20**	**2.6**
	D.4	Prom$^+$	**356**	**468**	**0.81**	**26**	**0.8**
		Prom$^+$ (with thread symmetry reductions)	**206**	**260**	**0.55**	**21**	**0.7**
Dijkstra	D.5	Prom$^+$	**860**	**1248**	**1.39**	**16**	**0.8**
		Prom$^+$ (with thread symmetry reductions)	**860**	**1248**	**1.84**	**13**	**0.8**
		SPIN	**860**	**1248**	**< 0.01**	**24**	**2.6**
	D.6	Prom$^+$	**942**	**1338**	**2.34**	**26**	**0.8**
		Prom$^+$ (with thread symmetry reductions)	**498**	**694**	**1.83**	**23**	**0.8**
Philosophers (3)	D.7	Prom$^+$	**885**	**1490**	**0.93**	**11**	**0.7**
		Prom$^+$ (with thread symmetry reductions)	**885**	**1490**	**1.18**	**8**	**0.7**
		SPIN	**885**	**1470**	**< 0.01**	**20**	**2.6**
	D.8	Prom$^+$	**1082**	**1961**	**1.93**	**18**	**0.8**
		Prom$^+$ (with thread symmetry reductions)	**462**	**888**	**1.03**	**14**	**0.8**
Philosophers (4)	D.7	Prom$^+$	**8545**	**22288**	**16.29**	**14**	**1.7**
		Prom$^+$ (with thread symmetry reductions)	**8545**	**22288**	**20.43**	**10**	**2.0**
		SPIN	**8545**	**22288**	**0.03**	**24**	**2.7**
	D.8	Prom$^+$	**11124**	**29539**	**40.40**	**24**	**1.9**
		Prom$^+$ (with thread symmetry reductions)	**4464**	**11819**	**20.12**	**19**	**1.4**

Chapter 7

Conclusion

This chapter concludes the thesis. It briefly discusses the findings of this thesis and evaluates the results. Finally a brief discussion on related work and future work is given.

7.1 Summary

This section discusses the conceptual architecture, as well as the implementation of our generic layer, abstract layer and the tool. The design decisions made during the development of the framework and their impact on end result are also discussed.

Conceptual architecture In chapter 2 the conceptual architecture was discussed. In this chapter we state this conceptual architecture is applicable for more than just explicit-state model checking. In later chapters only one possible implementation of this conceptual architecture was discussed. More specifically, the generic layer was based on explicit-state model checking and the abstract layer based on graph-based representations of software-based models. We argue this illustrates the conceptual architecture is suitable for explicit-state model checking, but to show it is also suitable for symbolic, bounded or probabilistic would require a similar discussion leading to generic layers in these fields. We do argue that for explicit-state model checking the generic layer provides the ability to provide algorithms for simulation and verification that are independent of the model type, which was one of our objectives.

The library In chapter 3 until 4 we presented a possible implementation of the conceptual architecture for explicit-state model checking. Not everything that was presented in this thesis was actually implemented. For instance, the mappings discussed in section 3.4.2 have not been implemented, nor have function types, function instances and function calls. The optimisation of the linearisation process described in section 4.2.3 has also not been implemented. Finally, all features described in chapter 5 have not been implemented, except for the thread-symmetry reductions.

Generic layer The generic layer provides the abstraction of a StateSpace interface, which enables the definition of generic simulation and verification algorithm. One point of discussion concerns this interface. A total order of the outgoing transitions per state is imposed by the $FirstTransition$ and $NextTransition$ functions, this might be too strict for some models. Also, the programmatic interface with reference counting pointers comes at a cost, a significant amount of verification time is spend on keeping track of the references. The design of the search module provides great flexibility, but this comes at the cost of the overhead of additional function calls.

It is important to note that the type abstraction principle in the generic layer does not necessarily means that the framework is per definition slower than other tools with specialised algorithms. The modular design of the search module, and the choice to use pointers with reference counting are the design choices that came at performance hits.

Abstract layer The graph representation of states as offers a formal basis which can be useful for constructing algorithms, such as linearisation and symmetry reductions. Also, the graph representation is intuitive to understand, as is the design of state representation and typing information (figure 4.3 and 4.5). The transitions in our abstract layer have no real formal basis. This implies that framework is very liberal with enforcing any semantics of the models. This can be considered an advantage as this implies we can use the abstract layer to model a wider range of model specification languages, but also a disadvantage as it is the responsibility of the tool developer to ensure the semantics are correct.

Additionally, we'd like to address the inclusion of pointers to the state graphs discussed in section 6.1.2. Arguably, the decision to model pointers as data types would have been more intuitive. Combined with the optimisation of the linearisation procedure the argument that statevectors would be significantly larger would no longer be valid.

Tool layer The **PROM**$^+$ tool provides a model checker for a simple specification language called **PROM**$^+$. The verification result of a few models have been presented in table 6.2. There are a few interesting points regarding these measurements:

- The verification time measurements shows the **PROM**$^+$ tool is inferior to **SPIN** in terms of the time required to verify a model. The **PROM**$^+$ tool is slower by an approximate factor of 10^3, which is obviously a very significant amount. This illustrates that our modular approach cannot compete with the optimised code generated by **SPIN** with respect to time.
  It is not clearcut which parts of the framework significantly contribute to the slow verification times of the **PROM**$^+$. It is the combination of using reference counting pointers, a modular search module which requires a lot of function calls, and also the use of state graphs proves to be expensive. Not only encoding and decoding state graphs to and from bitvectors is expensive, also copying the states before applying a statement is expensive. For instance, each `Instance` in a the `SoftwareState` is a c++-object that needs to be allocated on the heap, which takes time.
  It should be noted that the framework is not fully optimised, and it is not unreasonable to assume that some performance enhancements could still be made. However, to be competitive with **SPIN** would require a massive improvement.
- The memory usage of the **PROM**$^+$ tool *is* competitive with **SPIN**. Although our state store is a hash table that has an overhead in of approximately 50 bytes per element (compared to **SPIN**'s 8 bytes), and our stack is extremely inefficient as it stores all states, the average size of the state vector provides some opportunities. Given an implementation of the stack and the state store with **SPIN**'s efficiency, the **PROM**$^+$ tool should have a smaller overhead.
  Additionally, the current linearisation procedures do not implement the optimisation presented in section 4.2.3, nor have the settings of the verification runs been optimised. For instance, for most models it would have been sufficient to use just 2 bits for the process identifier instead of the default 8. This would result in even smaller bitvectors. However, before any optimisations in state vectors is noticeable, a more efficient implementation of the stack and state store would be required.
- The thread symmetry reductions of our 'symmetric' **PROM**$^+$ models also are a reason for discussion. The results of the verification runs show a 35%, 42% and 47% reduction in the number of visited states of the Peterson, Dekker and Dijkstra mutex models, respectively. The dining philosophers model resulted in a 57% and 60% reduction in the number of visited states, for scenarios with 3 and 4 philosophers respectively.
  In the most ideal case a mutex model for two processes could achieve a 50% reduction, and for the dining philosophers this would be 67% and 75%, respectively.
  The reason why our symmetry reduction does not achieve this ideal reduction is two-fold. Firstly, in order to exploit symmetry reduction we need an initiating process, during the execution of this process no symmetry reduction can be applied.

Secondly, our representatives are not canonical. The heuristics we use to find representatives, namely by process type and by control-flow state, is not always sufficient to achieve a good reduction. For instance, consider the specification in listing 7.1. There are 5 independent processes of the same type in this model, and one would expect a symmetry reduction to reduce the number of visited states significantly. For the current implementation of symmetry reduction this is not the case. The heuristics cannot distinguish between the process instances as they all have the same type type and also are always in the same control-flow state, because there is only one reachable control-flow state. Verifying this model in Prom$^+$ with thread-symmetry enabled does not result in any reduction.

In the mutex model we can see that as the state spaces grow larger, a more efficient reduction is achieved. This can be ascribed to the fact that the overhead of the initiating process is relatively smaller for larger state spaces. However, it should be noted that the nature of the models also has an influence. It could be argued that for mutex models less likely that the process instances are not distinguishable by means of their control-flow state. For example, if the mutual exclusion is enforced then the two process instances can never be in a control-flow state in the critical section at the same time.

Finally, thread-symmetry might not always preserve correctness of the verification result. If the semantics of are dependent on the process identifiers then it is possible that certain error states are omitted. For instance, the '`_pid`' variable in Promela might introduce problems. Also, the order in which processes die depends on the process identifiers. In our mutex and dining philosophers models we have processes that never terminate, and therefore this is not an issue.

Chapter 5 shows how certain features, that were originally introduced in section 1.2.5, are related to the framework. These features are included on different levels of abstraction, therefore they see fit in respectively the generic or abstract layer. Although some of these features will require extensive effort to add to the current implementation fo the framework, such as for instance symbolic model checking, they are a logical and natural extension of the framework. Only the inclusion of external storage and distributed verification do not seamlessly extend the philosophy of the search functionality in our generic layer, due to their need for specialised search strategies.

7.2 Evaluation

We argue that the principle of using a layered architecture in combination with type abstraction to provide generic algorithms for explicit-state verification is the primary new concept that was presented in this thesis. This concept realises the objectives presented in section 1.3, at least to some extend. Firstly, the division in layers provides the ability to use the same code-base in multiple tools. Not only can the algorithms in the generic layer be reused for different implementations of abstract layers, the abstract layer could, at least in theory, be used for multiple specification languages.

However, in order to really accomplish the objectives, the framework would have to be applied in several tools. For this to be realistic, the framework would have to be competitive in terms of its performance. We argue that the type abstraction used in the generic layer is not the cause

Listing 7.1 – A worst-case Prom$^+$ model for thread-symmetry reduction.

```
active [5] proctype A() {
    byte x;

    do
    :: x = x + 1;
    od;
}
```

of the performance issues. For instance, the standard c++ libraries such as `std::vector` and `std::map` are also based on type abstraction, but are very competitive in terms of performance. The performance issues are a result of design decisions made in both the generic layer and the abstract layer. These decisions provide flexibility in terms of the flexibility of the framework, but compromise on performance.

In hindsight, the trade-off between modularity and performance is perhaps too biased towards modularity. Some decisions, such as the decision to use reference counting pointers, and the decision to use a flexible search module rather than a dedicated solution, might have been taken differently if the performance of the tool was known at the time the decision was taken.

Also, the current abstract layer might be slightly too complex to really get across the essential message of our conceptual architecture. Perhaps the concept of reuse of algorithms should have been illustrated with two less complex abstract layers. For example, an abstract layer for process algebras might have suited the scope of this project better.

7.3 Related Work

Compared to other verification frameworks, and particulary the model checking kit (section 2.2.3) and the **IF** toolkit (section 2.2.4), the conceptual architecture of this framework is not essentially different. All frameworks feature a layered architecture. The main difference is that our generic layer uses type abstraction to abstract from the type of states, labels and transitions.

This type abstraction offers some advantages compared to the approach in the other frameworks. The information kept within the states, labels and transitions is not lost when mapping the abstract layer on the generic layer. When outputting a counter-example of a verification run this information can be used to print a more informative trace. Also the abstraction principle allows specialised implementations of, for instance, state stores.

It terms of the functionality and performance of our framework, it is not yet competitive with other frameworks. In order to be more competitive, more time would have to be spend optimising the performance of the framework, possibly by sacrificing some modularity in both the generic and abstract layer. Also, it would be beneficial to provide a wider range of functionality, such as additional search strategies, stores, etc. Compared to other frameworks, the functionality that is offered is very limited.

7.4 Future Work

Future work on the framework would elaborate on the conceptual architecture presented in chapter 2, and can be very diverse.

- Firstly, a different group of model types could be supported by means of a new generic layer. For instance, by developing a layer for symbolic or bounded model checking.
- Secondly, one could develop a new abstract layer that uses the current generic layer for explicit-state model checking.
- Thirdly, new the current generic and abstract layer could be extended with additional search strategies, stores, or even transformations of state spaces to support **LTL** model checking as well as data types, statements, etc.
- Also, the current abstract layer could be extended to support new language features, such as atomic constructs or data inheritance. Also, one could implement function types and instances which were discussed in chapter 4, but were not included in the implementation of the framework, or change the pointer variable representation such that pointers are represented as types.
- A more basic continuation of the framework would be to investigate methods to improve the performance of the framework. This could include reconsidering some design decisions that were made during the creation of the current framework, such as the use of reference counting pointers as well as redesigning the verification functionality in the generic layer.

- Finally, it would be interesting to considering alternative ‘abstract layers’, and to compare their performance to the current graph-based implementation. This would give more insight into which layer is mostly responsible for the slow verification times.

Bibliography

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Model-checking continuous-time Markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.

[4] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model-checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 106–115. IEEE Computer Society, 2003.

[5] Jiri Barnat, Lubos Brim, and Jakub Chaloupka. From distributed memory cycle detection to parallel LTL model checking. *Electronic Notes in Theoretical Computer Science*, 133:21–39, 2005.

[6] Jiri Barnat, Lubos Brim, and Jitka Stribrná. Distributed LTL model-checking in SPIN. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, pages 200–216, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[7] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL implementation secrets. In *Proceedings of 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002.

[8] M. Ben-Ari. *Principles of concurrent and distributed programming*. Prentice Hall international series in computer science. Prentice Hall, 1990.

[9] Johan Bengtsson and Wang Yi. Timed automata: semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *In Lecture Notes on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*. Springer–Verlag, 2004.

[10] Dragan Bosnacki. A light-weight algorithm for model checking with symmetry reduction and weak fairness. In Thomas Ball and Sriram K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 2003.

[11] Dragan Bosnacki, Dennis Dams, and Leszek Holenderski. Symmetric SPIN. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 1–19, London, UK, 2000. Springer-Verlag.

[12] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. Tools and applications: the IF toolset. In M. Bernanrdo and F. Corradini, editors, *Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Real Time*, volume 3185 of *LNCS*. Springer, 2004.

[13] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE conference on Design Automation*, pages 40–45, New York, NY, USA, 1990. ACM Press.

[14] Lubos Brim, Ivana Cerná, Pavel Krcál, and Radek Pelánek. Distributed LTL model checking based on negative cycle detection. In *Proceedings of the 21st conference on Foundations of Software Technology and Theoretical Computer Science*, pages 96–107, London, UK, 2001. Springer-Verlag.

[15] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: a new symbolic model verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification*, pages 495–499, London, UK, 1999. Springer-Verlag.

[16] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[17] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solvings. *Formal Methods in System Design*, 19(1):7–34, 2001.

[18] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194, London, UK, 2001. Springer-Verlag.

[19] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[20] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[21] R. Cleaveland and S. Sims. Generic tools for verifying concurrent systems. In I. Lovrek, editor, *Second International Workshop on Applied Formal Methods in System Design*, pages 3–8, Zagreb, Croatia, June 1997. University of Zagreb, Faculty of Electrical Engineering and Computing. 953-184-004-0.

[22] Patrick Cousot. Abstract interpretation based formal methods and future challenges. In Reinhard Wilhelm, editor, *Informatics. 10 Years Back. 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[23] Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.

[24] Pedro R. D'Argenio, Holger Hermanns, Joost-Pieter Katoen, and Ric Klaren. MoDeST - a modelling and description language for stochastic timed systems. In *Proceedings of the Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification*, pages 87–104, London, UK, 2001. Springer-Verlag.

[25] P.R. D'Argenio, J.-P. Katoen, and E. Brinksma. An algebraic approach to the specification of stochastic systems (extended abstract). In *Proceedings of the IFIP Working conference on Programming Concepts and Methods*, pages 126–147.

[26] Claudio Demartini, Radu Iosif, and Riccardo Sisto. dSPIN: a dynamic extension of SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 261–276, London, UK, 1999. Springer-Verlag.

[27] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, and Robby. Building your own software model checker using the bogor extensible model checking framework. In *Computer Aided Verification: 17th International Conference*, volume 3576 of *Lecture Notes in Computer Science*, pages 148–152. Springer-Verlag, 2005.

[28] S. Edelkamp, S. Leue, and A. Lafuente. Partial-order reduction and trail improvement in directed model checking. *International Journal on Software Tools for Technology Transfer*, 6(4):277–301, 2004.

[29] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed explicit model checking with HSF-SPIN. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, pages 57–79, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[30] Javier Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2-3):151–195, 1994.

[31] Joseph A. Gallian. *Contemporary abstract algebra*. Houghton Mifflin Company, fifth edition, 2002.

[32] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems*. PhD thesis, University of Liège, Liège, 1995.

[33] Patrice Godefroid, Gerard J. Holzmann, and Didier Pirottin. State-space caching revisited. *Formal Methods in System Design*, 7(3):227–241, 1995.

[34] Keijo Heljanko and Ilkka Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3(4&5):519–550, 2003.

[35] Gerard J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proceedings of the 3th International SPIN Workshop*, 1997.

[36] Gerard J. Holzmann. *The SPIN model checker*. Addison-Wesley, 2004.

[37] Gerard J. Holzmann and Anuj Puri. A minimized automaton representation of reachable states. *International Journal on Software Tools for Technology Transfer*, 2(3):270–278, 1999.

[38] G.J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *The SPIN Verification System*, pages 23–32. American Mathematical Society, 1996. Proceedings of the 2nd Spin Workshop.

[39] G.J. Holzmann and Doron Peled. An improvement in formal verification. In *Proceedings of Formal Description Techniques*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.

[40] Michael Huth and Mark Ryan. *Logic in computer science*. Cambridge University Press, 2004.

[41] Radu Iosif. The dSPIN user manual.
`http://www-verimag.imag.fr/~iosif/dspin/`.

[42] C. Norris Ip and David L. Dill. Efficient verification of symmetric concurrent systems. In *International Conference on Computer Design*, pages 230–234, 1993.

[43] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1-2):41–75, 1996.

[44] C. Norris Ip and David L. Dill. State reduction using reversible rules. In *Proceedings of the 33rd annual conference on Design Automation*, pages 564–567, New York, NY, USA, 1996. ACM Press.

[45] ISO 12207. *International standard, information technology software life cycle process. ISO 12207*, 1995.

[46] Harmen Kastenberg, Anneke Kleppe, and Arend Rensink. Engineering object-oriented semantics using graph transformations. CTIT Technical Report 06-12, University of Twente, 2006.

[47] Harmen Kastenberg and Arend Rensink. Model checking dynamic states in GROOVE. In Antti Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Software Model Checking*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305. Springer-Verlag, 2006.

[48] Matthias Kuntz and Kai Lampka. Probabilistic methods in state space analysis. In *Validation of Stochastic Systems*, volume 2925 of *Lecture Notes in Computer Science*, pages 339–383. Springer-Verlag, 2004.

[49] Robert P. Kurshan, Vladdimir Levin, Marius Minea, Doron Peled, and H. Yenigün. Static partial order reduction. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 345–357, London, UK, 1998. Springer-Verlag.

[50] Marta Kwiatkowska. Model checking for probability and time: from theory to practice. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, pages 351–360, Washington, DC, USA, 2003. IEEE Computer Society.

[51] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 22–39, London, UK, 1999. Springer-Verlag.

[52] A. Lluch-Lafuente. Symmetry reduction and heuristic search for error detection in model checking. In *2nd Workshop on Model Checking and Artificial Intelligence*, 2003.

[53] Rémi Morin and Brigitte Rozoy. On the semantics of place/transition nets. In *Proceedings of the 10th International Conference on Concurrency Theory*, pages 447–462, London, UK, 1999. Springer-Verlag.

[54] University of Stuttgart. Model-checking kit.
`http://www.fmi.uni-stuttgart.de/szs/tools/mckit/`.

[55] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

[56] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[57] Arend Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and S. Lo Presti, editors, *Workshop on Automated Verification of Critical Systems*, Technical Report DSSE–TR–2003–2, pages 150–160. University of Southampton, 2003.

[58] Arend Rensink. The GROOVE simulator: a tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.

[59] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 267–276, New York, NY, USA, 2003. ACM Press.

[60] Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif. State-space reduction strategies for model checking dynamic software. *Electronic Notes in Theoretical Computer Science*, 89:499–517, 2003.

[61] Ulrich Stern and David L. Dill. Parallelizing the murphi verifier. In *Proceedings of the 9th International Conference on Computer Aided Verification*, pages 256–278, London, UK, 1997. Springer-Verlag.

[62] Ulrich Stern and David L. Dill. Using magnatic disk instead of main memory in the murphi verifier. In *Proceedings of the 10th International Conference on Computer Aided Verification*, pages 172–183, London, UK, 1998. Springer-Verlag.

[63] J. Tretmans. *A formal approach to conformance testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[64] J. Tretmans and E. Brinksma. Côte de resyste – automated model based testing. In M. Schweizer, editor, *3rd Workshop on Embedded Systems*, pages 246–255, Utrecht, The Netherlands, October 2002. STW Technology Foundation.

[65] Edward C. Turner and Karen F. Gold. Rubik's groups. *The American Mathematical Monthly*, 92(9):617–629, 1985.

[66] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science*, pages 322–331, 1986.

[67] Carsten Weise. An incremental formal semantics for PROMELA. In *Proceedings of the 3rd SPIN workshop*, 1997.

[68] Sergio Yovine. Model-checking timed automata. In G. Rozenberg and F. Vaandrager, editors, *Lectures on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*, pages 114–152. Springer-Verlag, 1998.

Appendix A

Notation

The notation throughout this document is supposed to be intuitively clear. This appendix is meant as a quick reference only.

Concept	**Notation**	**Example(s)**
Set	Starts with capital letter	S, L or $Domain$
Element	Starts with small letter	s, l or d
Powerset	$\mathcal{P}(\ldots)$	$\mathcal{P}(S)$
Cartesian product	$Set_1 \times \ldots \times Set_n$	$S \times S$
Tuple	$\langle Element_1, \ldots, Element_n \rangle$	$\langle S, L, R \rangle$
Interface definition	$\textsc{Name}$	$\textsc{StateSpace}$
Interface implementation	$[\![Name]\!]$	$[\![Stack]\!]$
Function definition	$Function : (Dom_1 \times \ldots \times Dom_n) \longrightarrow Range$	$F : (R) \longrightarrow S$
Function call	$var \leftarrow Function(param_1, ..., param_n)$	$s' \leftarrow F(r_1)$
Bitarray	$\mathsf{encode}(Name)$	$\mathsf{encode}(State)$
Bitarray concatenation	$\mathsf{encode}(Name)\mathsf{encode}(Name)$	$\mathsf{encode}(Type)\mathsf{encode}(ObjectId)$

Sometimes this report applies a slightly lazy notation. For instance, consider the *functions*. If omitting the parenthesis around the domain does not introduce ambiguity, then they are omitted. Also in some cases, a function does not have a real range (return type), instead of denoting this with a dummy range, it is simply omitted (e.g. $Push : (R)$). Note that in this case the parenthesis are useful, as it is also possible to ommit the domain (e.g. $InitialState : S$). Sometimes it is also useful to omit both domain and range (e.g. $SimulationStarted : ()$).

Appendix B

Dedicated Solution for Depth-First Search

This appendix introduces a dedicated design for exhaustive depth-first searches over state spaces. Initially a recursive algorithm is presented, which is similar to the pseudo algorithms often presented in literature [38, 39, 10, 19]. However, for a more practical solution an iterative version is presented as well.

Required Interfaces for Recursive Depth-First Search

To provide a recursive search over the state space we require some additional functionality, namely a GOAL function and a STORE.

In a search we are almost always looking for a particular subset of states, such as an accepting states or a deadlock states. We could provide information about goal states by means of an interface GOAL. This interface allows the search algorithm to be oblivious to the type of states it is looking for.

$$Goal : S \longrightarrow \{\textbf{TRUE}, \textbf{FALSE}\}$$

Interface B.1 – GOAL.

Besides goal states, it is desirable for the search algorithm to avoid visiting states more often than is strictly necessary. For this purpose we define an interface that is supposed to keep track of the states we have visited, namely the STORE interface.

$$\begin{aligned} Add &: (S) \\ Contains &: S \longrightarrow \{\textbf{TRUE}, \textbf{FALSE}\} \end{aligned}$$

Interface B.2 – STORE.

There could be a lot of different implementations for stores, such as bit-state hashing stores [36], minimised automata [37], or stores that only store a canonical representation of a state, which would in effect would accomplish a symmetry reduction.

Recursive Depth-First Search Algorithm

The most intuitive way of denoting an exhaustive depth-first search algorithm is by means of recursion. For example, see algorithm B.1, which is similar to how the depth-first search algorithm is denoted in literature in the context of explicit-state model checking [38, 39, 10, 19], except for that these are usually *nested* depth-first search algorithms.

There are a few issues with algorithm B.1, which arise from the fact that a recursive function is used to formalise the algorithm. Actual implementations implementations of this algorithm

Algorithm B.1: $Explore(s' \in S) \longrightarrow \{\text{TRUE}, \text{FALSE}\}$

```
Require: $[\![StateSpace]\!]$, $[\![Goal]\!]$, $[\![Store]\!]$.
1:  if ($[\![StateStore]\!].Contains(s')$) then
2:      return FALSE
3:  else
4:      $[\![Store]\!].Add(s')$
5:      if ($[\![Goal]\!].Goal(s')$) then
6:          return TRUE
7:      else
8:          $tmp \leftarrow [\![StateSpace]\!].FirstTransition(s')$
9:          while ($tmp \neq \epsilon$) do
10:             if ($Explore([\![StateSpace]\!].Target(tmp))$) then
11:                 ...
12:                 return TRUE
13:             end if
14:             $tmp \leftarrow [\![StateSpace]\!].NextTransition(tmp)$
15:         end while
16:         return FALSE
17:     end if
18: end if
```

in model checkers such as **SPIN** do not use function recursion, and use an equivalent iterative algorithm.

- The call stack used to realise function recursion can be the bottleneck of the verification process. For each transition on a path of the model, the call stack would need to maintain a stack frame. As models can become very large, the number of stack frames that can fit on the call stack can be a serious limitation on most systems.
- The path to the counter-example is not explicitly present in algorithm B.1, but is implicit in the call stack.

In the rest of this appendix the iterative version of algorithm B.1 is introduced. But in order to omit the use of the call stack, this algorithm requires the use of a stack, which is introduced in the next section.

Additional Interfaces for the Iterative Depth-First Search

Interface B.3 introduces the functionality of stack. This is a stack of transitions rather than states, which might be intuitively confusing. Consider a situation where the search uses the stack for back tracking. If the algorithm uses a stack of states, without any other information, the algorithm has no means of finding out which path it just back-tracked from. Also if we end up with a stack of states as a counter-example, then it is unclear which transitions led to this sequence of states. A stack of transitions avoids these issues. The STACK interface consists of a $Push$ procedure and a Pop function. The Pop function returns an ϵ if the stack is empty.

$$Push : (R)$$
$$Pop : R \cup \{\epsilon\}$$

Interface B.3 – STACK.

Some model checkers do not physically maintain the complete stack of transitions, implicitly storing all the states on the current path (because the STATESPACE requires the ability to retrieve the source and target state of a transition). However, another strategy would be to only remember

Algorithm B.2: $TryTransition(r \in R)$

```
1: $target \leftarrow [\![StateSpace]\!].Target(r)$
2:
3: **if** $(\neg [\![Store]\!].Contains(target)$ **then**
4:     $s' \leftarrow [\![StateSpace]\!].Target(r)$
5:     $[\![Stack]\!].Push(r)$
6:     $foundNewState \leftarrow$ TRUE
7: **end if**
```

the change in states, or to employ reversible transitions [44]. This kind of optimisation can be realised by implementations of STACK.

Iterative Depth-First Search Algorithm

The iterative version of algorithm B.1 is shown in algorithm B.3. The general idea of this algorithm is that in each iteration s' is a newly discovered state. While there are new states to be found in the successors of s', the algorithm keeps exploring. When the algorithm cannot explore further, it starts to back-track. It pops a transition of the stack and attempts to find previously undiscovered states.

A common operation in this algorithm is to test a transition to see whether it leads to a new state. This is shown in algorithm B.2. This function be seen as a local function which has access to all variables and interfaces as algorithm B.3 does, and should just be seen a way of denoting the algorithm in a more compact fashion.

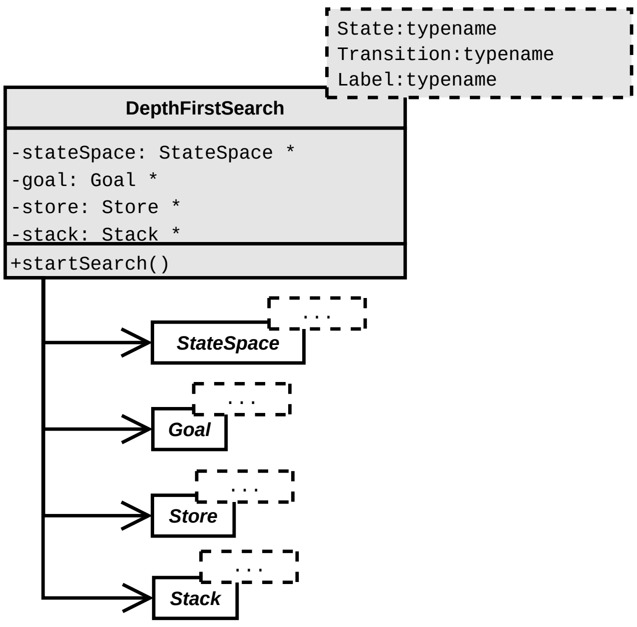


Figure B.1 – Design of a generic depth-first search algorithm using an $[\![StateSpace]\!]$, $[\![Stack]\!]$, $[\![Store]\!]$ and $[\![Goal]\!]$ interface.

In figure B.1 an overview of the depth-first solution is presented in an UML diagram.

Algorithm B.3: Iterative depth-first search algorithm

```
Require: $[\![StateSpace]\!]$, $[\![Goal]\!]$, $[\![Store]\!]$, $[\![Stack]\!]$.
$foundTarget \leftarrow$ FALSE
$foundNewTarget \leftarrow$ FALSE

$s' \leftarrow [\![StateSpace]\!].InitialState()$

while ($s' \neq \epsilon \wedge \neg foundTarget$) do
  {* Invariant: State $s'$ is always a newly discovered state. *}
  $[\![StateStore]\!].AddState(s')$
  if ($[\![Goal]\!].Goal(s')$) then
    $foundTarget \leftarrow$ TRUE
  else
    $foundNewState \leftarrow$ FALSE

    $r' \leftarrow [\![StateSpace]\!].FirstTransition(s')$
    while ($\neg foundNewState \wedge r' \neq \epsilon$) do
      {* Iterate over outgoing transitions of $s'$. *}
      $TryTransition(r')$
      if ($\neg foundNewState$) then
        $r' \leftarrow [\![StateSpace]\!].NextTransition(r')$
      end if
    end while

    if ($\neg foundNewState$) then
      $r' \leftarrow [\![Stack]\!].Pop()$
      while ($\neg foundNewState \wedge r' \neq \epsilon$) do
        {* Iterate over stack. *}
        $alt \leftarrow [\![StateSpace]\!].NextTransition(r')$
        while ($\neg foundNewState \wedge alt \neq \epsilon$) do
          {* Iterate over alternative outgoing transitions of $r'$. *}
          $TryTransition(alt)$
          if ($\neg foundNewState$) then
            $alt \leftarrow [\![StateSpace]\!].NextTransition(alt)$
          end if
        end while
        if ($\neg foundNewState$) then
          $r' \leftarrow [\![Stack]\!].Pop()$
        end if
      end while
    end if

    if ($\neg foundNewState$) then
      $s' \leftarrow \epsilon$
    end if
  end if
end while
```

Appendix C

Alternative Abstract Layers

The abstract layer defined in chapter 4 is only one possible interpretation of the explicit-state state space. Alternative abstractly layers are presented graphically in this appendix.

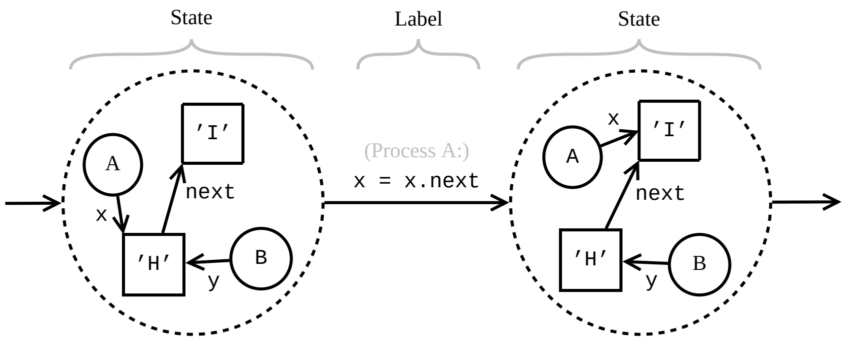


Figure C.1 – In this software-based model it is obvious that the states of the state space have some internal structure. Threads are denoted by circles, whereas heap objects are denoted by squares [60].

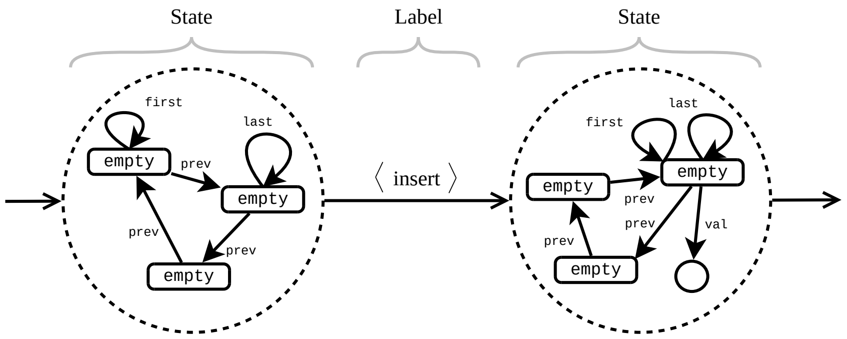


Figure C.2 – Another abstraction that can be seen as a state space is a graph transition system. Its states are graphs, and the transitions are graph morphism. The concept of graph transition systems in the context of model checking is explained in [57], including this example in its entirety. In this figure the specification of the morphism *insert* is omitted for the sake of brevity.

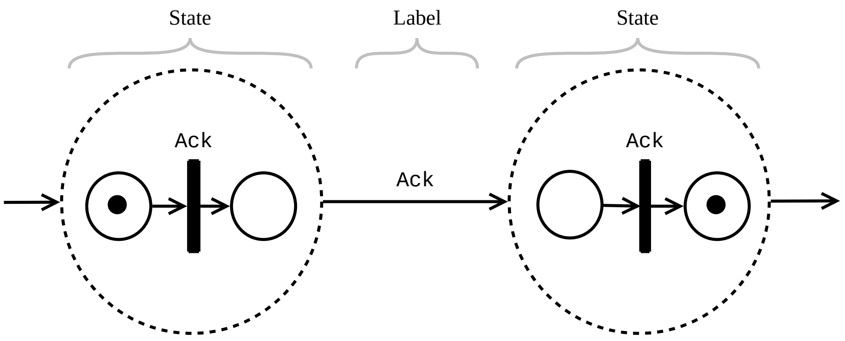


Figure C.3 – Petri nets could be considered as an abstract way of defining state spaces. Each state consists of a certain marking of tokens, whereas each transition is the firing of a transition in the actual Petri net. The initial state is some initial marking of tokens [30].

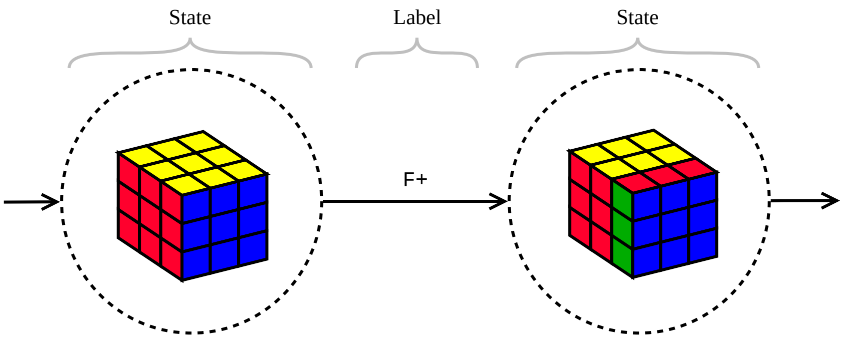


Figure C.4 – One could consider a Rubik's puzzle a state space too. Transitions are rotations of the (B)ottom, (T)op, (L)eft, (R)ight, (F)ront or (P)osterior face, in a (+) clockwise or (-) counter-clockwise fashion. The number of states in this state space is approximately $4.3 \cdot 10^{19}$ [65].

Appendix D

Example Prom$^+$ Models

Peterson's Mutex Algorithm

Listing D.1 – A **Promela** and **Prom**$^+$ model of Peterson's mutex algorithm.

```
byte mutex;
bit flag_a, flag_b, turn;

active [1] proctype Property() { assert(mutex < 2); }

active [1] proctype Process_A()
{
    do
    ::  flag_a = 1;
        turn = 0;
        (flag_b == 0 || turn != 0);

        /* BEGIN CRITICAL SECTION. */
        mutex = mutex + 1;
        mutex = mutex - 1;
        /* END CRITICAL SECTION. */

        flag_a = 0;
    od;
}

active [1] proctype Process_B()
{
    do
    ::  flag_b = 1;
        turn = 1;
        (flag_a == 0 || turn != 1);

        /* BEGIN CRITICAL SECTION. */
        mutex = mutex + 1;
        mutex = mutex - 1;
        /* END CRITICAL SECTION. */

        flag_b = 0;
    od;
}
```

Listing D.2 – A PROM^{+} model of Peterson's mutex algorithm suitable for symmetry reduction.

```
byte mutex;
bit * flag_1, * flag_2;
bit * turn;
bit * turn_1, * turn_2;
bool startGuard;

active [1] proctype Property() { assert(mutex < 2); }

active [0] proctype Process(
    bit * my_flag;
    bit * other_flag;
    bit * turn_value)
{
    startGuard;
    do
    ::  *my_flag = 1;
        turn = turn_value;
        (*other_flag == 0 || turn != turn_value);

        /* BEGIN CRITICAL SECTION. */
        mutex = mutex + 1;
        mutex = mutex - 1;
        /* END CRITICAL SECTION. */

        *my_flag = 0;
    od;
}

active [1] proctype Init()
{
    mutex = 0;
    startGuard = false;

    flag_1 = new bit;   *flag_1 = 0;
    flag_2 = new bit;   *flag_2 = 0;

    turn_1 = new bit;   turn = turn_1;
    turn_2 = new bit;

    run Process(flag_1, flag_2, turn_1);
    run Process(flag_2, flag_1, turn_2);

    /* Ensure this process (Init) does not cause symmetry to break. */
    reset flag_1;
    reset flag_2;

    reset turn_1;
    reset turn_2;

    /* Now start! */
    startGuard = true;
}
```

Dekker's Mutex Algorithm

Listing D.3 – A PROMELA and PROM$^+$ model of Dekker's mutex algorithm.

```
byte mutex;
bit flag_a, flag_b ,turn;

active [1] proctype Property() { assert(mutex < 2); }

active [1] proctype Process_A() {
    do ::   flag_a = 1;
        do ::   (flag_b == 0) -> break;
           ::   (flag_b == 1) ->
            if ::   (turn == 1) -> flag_a = 0;
                turn == 0;
                flag_a = 1;
               ::   (turn == 0) ->  skip;
            fi;
        od;

        /* BEGIN CRITICAL SECTION. */
        mutex = mutex + 1;
        mutex = mutex - 1;
        /* END CRITICAL SECTION. */

        turn = 1;
        flag_a = 0;
    od;
}

active [1] proctype Process_B() {
    do ::   flag_b = 1;
        do ::   (flag_a == 0) -> break;
           ::   (flag_a == 1) ->
            if ::   (turn == 0) -> flag_b = 0;
                turn == 1;
                flag_b = 1;
               ::   (turn == 1) -> skip;
            fi;
        od;

        /* BEGIN CRITICAL SECTION. */
        mutex = mutex + 1;
        mutex = mutex - 1;
        /* END CRITICAL SECTION. */

        turn = 0;
        flag_b = 0;
    od;
}
```

Listing D.4 – A PROM$^+$ model of Dekker's mutex algorithm suitable for symmetry reduction.

```
byte mutex; bool startGuard;
bit * flag_1, * flag_2, * turn_1, * turn_2, * turn;

active [1] proctype Property() { assert(mutex < 2); }

active [0] proctype Process(
    bit * my_flag;      bit * other_flag;
    bit * my_turn_value;    bit * other_turn_value)
{
    startGuard;
    do
    ::  *my_flag = 1;
        do
        ::  (*other_flag == 0) -> break;
        ::  (*other_flag == 1) ->
            if
            ::  (turn == other_turn_value) -> *my_flag = 0;
                (turn == my_turn_value); *my_flag = 1;
            ::  (turn == my_turn_value) -> skip;
            fi;
        od;

        /* BEGIN CRITICAL SECTION. */
        mutex = mutex + 1;
        mutex = mutex - 1;
        /* END CRITICAL SECTION. */

        turn = other_turn_value;
        *my_flag = 0;
    od;
}

active [1] proctype Init()
{
    startGuard = false;     mutex = 0;
    flag_1 = new bit;   *flag_1 = 0;
    flag_2 = new bit;   *flag_2 = 0;
    turn_1 = new bit;   turn = turn_1;
    turn_2 = new bit;

    run Process(flag_1, flag_2, turn_1, turn_2);
    run Process(flag_2, flag_1, turn_2, turn_1);

    reset flag_1;   reset turn_1;
    reset flag_2;   reset turn_2;

    startGuard = true;
}
```

Dijkstra's Mutex Algorithm

Listing D.5 – A **Promela** and **Prom**$^+$ model of Dijkstra's mutex algorithm.

```
byte mutex, status_a, status_b;
bit turn; bool startGuard;

active [1] proctype Property() { assert(mutex < 2); }

active [1] proctype Process_A() {
    do ::   status_a = 1;
        do ::   skip;
            do ::   (turn != 0) -> status_a = 0;
                (turn != 0 && status_b == 0) -> turn = 0;
               ::   (turn == 0) -> break;
            od;
            status_a = 2;
            if ::   (status_b != 2) -> break;
               ::   else -> skip;
            fi;
        od;

        /* BEGIN CRITICAL SECTION. */
        mutex = mutex + 1;
        mutex = mutex - 1;
        /* END CRITICAL SECTION. */
        status_a = 0;
    od;
}

active [1] proctype Process_B() {
    do ::   status_b = 1;
        do ::   skip;
            do ::   (turn != 1) -> status_b = 0;
                (turn != 1 && status_a == 0) -> turn = 1;
               ::   (turn == 1) -> break;
            od;
            status_b = 2;
            if ::   (status_a != 2) -> break;
               ::   else -> skip;
            fi;
        od;

        /* BEGIN CRITICAL SECTION. */
        mutex = mutex + 1;
        mutex = mutex - 1;
        /* END CRITICAL SECTION. */
        status_b = 0;
    od;
}
```

Listing D.6 – A PROM$^+$ model of Dijkstra's mutex algorithm suitable for symmetry reduction.

```
byte mutex, * status_1, * status_2;
bit * turn, * turn_1, * turn_2;
bool startGuard;

active [1] proctype Property() { assert(mutex < 2); }

active [0] proctype Process(byte *my_status; byte *other_status; bit *turn_value)
{
    startGuard;
    do
    ::  *my_status = 1;
        do
        ::  skip;
            do
            ::  (turn != turn_value) -> *my_status = 0;
                (turn != turn_value && *other_status == 0) -> turn = turn_value;
            ::  (turn == turn_value) -> break;
            od;

            *my_status = 2;

            if
            ::  (*other_status != 2) -> break;
            ::  else -> skip;
            fi;
        od;

        /* BEGIN CRITICAL SECTION. */
        mutex = mutex + 1;
        mutex = mutex - 1;
        /* END CRITICAL SECTION. */

        *my_status = 0;
    od;
}

active [1] proctype Init()
{
    startGuard = false;
    mutex = 0;

    status_1 = new byte;    *status_1 = 0;
    status_2 = new byte;    *status_2 = 0;

    turn_1 = new bit;
    turn_2 = new bit;

    turn = turn_1;

    run Process(status_1, status_2, turn_1);
    run Process(status_2, status_1, turn_2);

    reset turn_1;   reset status_1;
    reset turn_2;   reset status_2;
    startGuard = true;
}
```

Dining Philosophers in Prom$^+$

Listing D.7 – A **Promela** and **Prom**$^+$ model of two dining philosophers.

```
bool f1, f2;

active [1] proctype Philosopher1() {
    do ::
        !f1 -> f1 = true;
        !f2 -> f2 = true;
         f2 = false;
         f1 = false;
    od;
}

active [1] proctype Philosopher2() {
    do ::
        !f2 -> f2 = true;
        !f1 -> f1 = true;
         f1 = false;
         f2 = false;
    od;
}
```

Listing D.8 – A thread-symmetrical version of the dining philosophers in PROM$^+$.

```
bool * f1, * f2; bool startGuard;

active [0] proctype Philosopher(bool * left; bool * right) {
    startGuard;

    do ::
        !*left -> *left = true;
        !*right -> *right = true;
         *right = false;
         *left = false;
    od;
}

active [1] proctype Init() {
    startGuard = false;

    f1 = new bool;
    f2 = new bool;

    *f1 = false;
    *f2 = false;

    run Philosopher(f1, f2);
    run Philosopher(f2, f1);

    /* Ensure this process (Init) does not cause symmetry to break. */

    reset f1;
    reset f2;

    /* Now start! */

    startGuard = true;
}
```