An Object-Oriented Framework for Explicit-State Model Checking

Mark Kattenbelt[1] and **Theo C. Ruys**[2] and **Arend Rensink**[2]

Abstract. This paper presents a conceptual architecture for an object-oriented framework to support the development of formal verification tools (i.e. model checkers). The objective of the architecture is to support the reuse of algorithms and to encourage a modular design of tools. The conceptual framework is accompanied by a C++ implementation which provides reusable algorithms for the simulation and verification of explicit-state models as well as a model representation for simple models based on guard-based process descriptions. The framework has been successfully used to develop a model checker for a subset of PROMELA.

1 INTRODUCTION

Model checking is the application of an automated process to formally verify whether a *model* conforms to a *specification* [7, 3]. There are numerous ways in which one could express a model, but typically the model can be interpreted as some sort of automaton. The level of abstraction that is used to describe models in tools varies significantly depending on the model checker, and ranges from low-level automata-based representations (such as the timed automata in UPPAAL [1]) to high-level specification languages that resemble programming languages (such as BIR in Bogor [12]). The specification can also be expressed in various ways, but is usually formulated in terms of properties in some type of temporal logic. The nature of the verification process used in model checkers is heavily dependent on the types of models and specifications it can verify.

Most model checkers are very specialised, and support only a single type of model. Additionally, it is not uncommon for model checkers to introduce their own specification language. Although this specialisation enables tools to optimise their verification algorithms, it does not encourage a reusable design. In order to reuse the functionality contained within model checkers one often has to resort to using the model specification language prescribed by this model checker. As a result, many transformations between input languages of tools currently exist and interaction between tools can only be achieved with considerable effort.

To emphasise the need for reuse, consider the great advancements of model checking in recent years [6]. The aspiration to apply model checking to systems of an industrial scale has led to the introduction of many new complex techniques and algorithms (i.e. partial-order reduction, symmetry reduction, predicate abstraction, slicing algorithms). Implementing a state-of-the-art model checker is not a trivial task, and therefore any opportunity to reuse functionality should be considered beneficial.

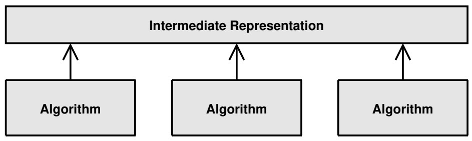

Figure 1 – Model checking frameworks usually have a single intermediate representation. In order to use the framework the model under consideration will have to be expressed in this intermediate representation.

The need for reuse and interoperability has been acknowledged by several others. For example, the *model-checking framework* BOGOR [12], the IF TOOLSET [4], the MODEL-CHECKING KIT [20] and the NCSU CONCURRENCY WORKBENCH [8] all offer a framework to enable reuse in verfication tools, and often employ a layered architecture. Similar to modern compiler suites, most of these frameworks use an *intermediate representation* to which high-level models are translated (see Figure 1). This representation can be a textual description in a modelling specification language, or a programmatic representation. For the frameworks mentioned previously, these intermediate representations are BIR, IF *specification*, *1-Safe Petri Nets* and *Labelled Transition Systems*, respectively.

The verification functionality of these frameworks is realised by algorithms that use this intermediate representation directly. Having a single intermediate representation is advantageous for the optimisation of verification algorithms. However, a drawback of this approach is that the applicability of the framework is limited by the expressiveness of the intermediate representation. Furthermore, a transformation of models to this intermediate representation is not always optimal. The verification algorithms cannot be reused for anything other than the intermediate representation used in the framework.

We have developed a framework that is not limited by a single intermediate representation. We provide a means of describing algorithms such that they can be used by many different intermediate representations. Related to our approach is the MÖBIUS MODELLING ENVIRONMENT [9, 11], which uses the same principle for performance analysis of stochastic models.

The goal of our framework is to enable the development of generic functionality that can be used in several verification tools directly, not necessarily limited to model checkers, and to improve the interoperability of tools. In the remainder of this article we will describe the core essentials of this framework. Details can be found in [16]. The meaning of 'framework' is two-fold in this article:

- *Conceptual architecture.* A conceptual architecture for a model checking framework which enables reuse of code. This architecture enables us to define algorithms that can be reused for different

[1] School of Computer Science, University of Birmingham, United Kingdom. http://www.cs.bham.ac.uk/~mxk/. (Supported by EPSRC grant EP/D07956X/1 during the authoring of this extended abstract.)

[2] Formal Methods and Tools group, Faculty of EEMCS, University of Twente, The Netherlands. http://fmt.cs.utwente.nl/.

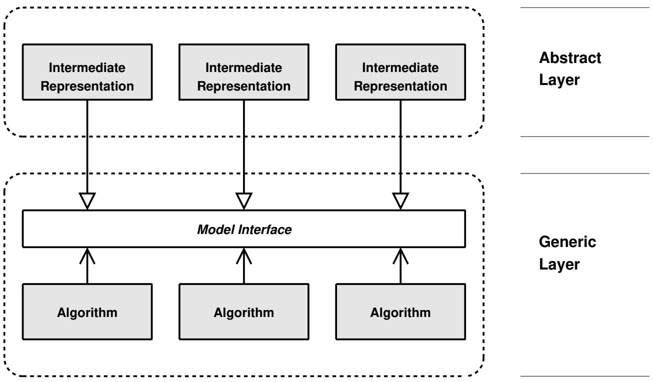

Figure 2 – The conceptual architecture of the framework, divided into a *generic layer* and an *abstract layer*.

intermediate representations. In Section 2 we will introduce this architecture.
- *Concrete architecture.* A proof-of-concept implementation of the conceptual architecture. On a low level, it consists of reusable algorithms for explicit-state verification techniques. On a high level, it provides a graph-based intermediate representation which represents models with guard-based process descriptions. This library is introduced in Section 3.

A proof-of-concept tool is built on top of our concrete architecture and is capable of verifying PROM$^+$ (a subset of PROMELA, see Section 3.4). It combines our intermediate representation with our reusable verification algorithms to realise its functionality.

2 CONCEPTUAL ARCHITECTURE

The conceptual architecture should enable reusable algorithms to be defined over multiple intermediate representations. Our architecture is based on a layered design as depicted in Figure 2, similar to other frameworks. In contrast to other frameworks, algorithms do not refer to the intermediate representation directly (Figure 1), but refer to a model interface instead. We distinguish two layers, a *generic layer* and an *abstract layer*.

Note that we use a slightly informal notation in our architectural diagrams. In general, white blocks are interfaces, whereas grey blocks actually contain some sort of implementation. *Associations* and *specialisation* relationships between blocks are shown using the notation commonly used in UML class diagrams.

2.1 Generic layer

The generic layer contains reusable *algorithms*, as well as a *model interface*. This model interface defines a number of operations to facilitate the algorithms. Additionally, we abstract from the types that are used in the model interface by means of type parametrisation (e.g. generics in JAVA, templates in C++).

The idea is that the model interface abstracts over the most elementary types used in the algorithms, which are likely to be different for different intermediate representations. In this way the algorithms need not to be concerned with the implementation of these types, and intermediate representations can provide their own custom implementation of these types. The model interface defines operations over these types such that the algorithms can efficiently realise their functionality using these operations, but it is the intermediate representations that actually implement these operations.

The most obvious choice of a generic layer would be one to facilitate *explicit-state model checking*. In this type of model checking each state is explicitly represented, and the verification process can usually be reduced to some type of exhaustive search over the state space. Candidate types for type parameters are elementary types such as *states* and *transitions*, whereas operations are likely to facilitate the on-the-fly construction of the state space (i.e. an operation to retrieve successors of a state). A generic layer for explicit-state model checking is discussed in Section 3. Other possible generic layers could facilitate *symbolic* or *bounded model checking*, where candidates for type parameters would include *sets of states* or *clauses*, respectively [16].

Generally speaking, anything contained within the generic layer is meant for use with any intermediate representation, and therefore uses type parameters. Items in the abstract layer are specific to an intermediate representation and therefore do not apply type parameters. Any specialisation relationship between the generic and the abstract layer therefore also implies a specialisation of types.

2.2 Abstract layer

The abstract layer contains intermediate representations of a programmatic form. The basic idea is that such an intermediate representation *specialises* the model interface in a generic layer. In other words, an intermediate representation implements the operations of the model interface for a particular set of types. In the context of explicit-state model checking, intermediate representations in the abstract layer can be very diverse, ranging from ‘low-level’ representations such as Labelled Transition Systems (LTS), and Graph Transition Systems (GTS) [18] to ‘high-level’ representations such as Process Algebras (PA) or those used in SPIN [13] and BOGOR [19].

The benefit of using type parameters is that an intermediate representation can implement its own elementary types. For instance, an intermediate representation that implements a model interface of a generic layer for explicit-state model checking can define its own state type. This is useful because the information contained within a state is significantly different for different intermediate representations. For instance, the information contained within a state of a PA model is very different from the state of a PROMELA model. In terms of an intermediate representation of an abstract layer for symbolic model checking, this type specialisation could be used to implement different ways of representing a *set of states*, such as BDDs [17, 5] or MDDs [15]. Arguably, the same effect can be accomplished with subtyping, but this introduces more flexibility (and overhead) than is necessary. The MÖBIUS tool uses a similar approach, and applies subtyping [10] as well as type parametrisation [11].

An alternative conceptual architecture is employed in the NCSU CONCURRENCY WORKBENCH [8]. In this framework intermediate representations can be translated into a LTS automatically by using the Structured Operational Semantics (SOS) of these intermediate representations.

3 CONCRETE ARCHITECTURE

In Figure 3 an overview of our library is shown. The generic layer consists of an explicit-state model interface and algorithms for simulation and verification. The motivation for this library originated from the desire to offer a modular alternative to state-of-the-art tool SPIN [13], which is reflected in the abstract layer. The ‘software model’ intermediate interpretation is meant for targeting a subset

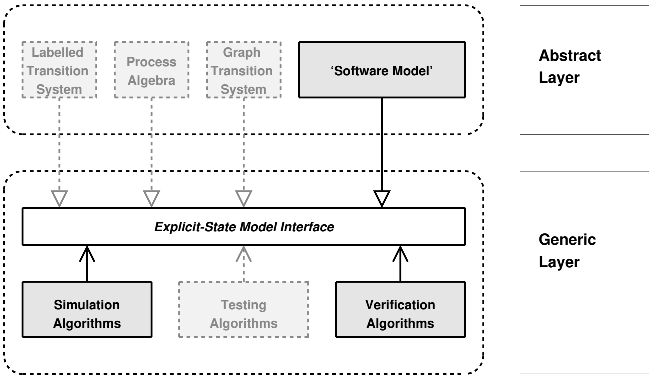

Figure 3 – The concrete architecture of the framework as implemented in our library. Elements that were not implemented, but are shown in the figure to provide a context, have dashed borders.

```
template <typename S, typename L, typename T>
class ExplicitStateModelInterface
{
  public:
  virtual S* getInitialState() =0;
  virtual T* getFirstTransition(S* s) =0;
  virtual T* getNextTransition(T* tr) =0;

  virtual S* getSource(T* tr) =0;
  virtual L* getLabel(T* tr) =0;
  virtual S* getTarget(T* tr) =0;
};
```

Listing 1 – The model interface of our library consists of a single C++ class called `ExplicitStateModelInterface`.

of PROMELA called PROM^{+}, and is the intermediate representation used in our proof-of-concept model checker. This representation could be extended to support other model specification languages such as BIR, and is therefore not dedicated to a single tool.

The components of the library are written in C++, and feature a modular object-oriented design. Functionality in the generic layer includes simulation and reachability algorithms. The 'software model' intermediate representation comprises the largest part of the library, as it is aimed to be as general as possible.

3.1 Explicit-state model interface

The definition of a model interface has two important features, a set of type parameters and a set of operations. These types and operations should be chosen carefully because all intermediate representation that use this generic layer will have to conform to this interface. Additionally, the operations are to enable all prospective algorithms of this generic layer to realise their functionality efficiently.

The model interface of our prototype can be found in Listing 1. This listing shows that our implementation language is C++. Although it is not necessary to understand C++ in order to understand the principles of our design, we use code samples to illustrate our design. These principles could also be implemented in another language, such as JAVA. We will provide a brief explanation with each code sample, but we refer to [21] for a more concise reference on C++.

Note that we do not define the model interface for any specific type of model representation (e.g. LTS or Kripke structures) but attempt to provide an interface for a large class of automata-based models. In our implementation we chose to abstract from the type of states (`S`), type of labels (`L`), and type of transitions (`T`) used in the intermediate representations. The set of operations is defined such that model information can be retrieved on-the-fly. These functions are abstract (e.g. pure and virtual in C++), and will need to be implemented by any intermediate representation. The initial state object of a model can be retrieved using the `getInitialState` function. Given a state of the model, we can retrieve all outgoing transitions objects of this state in a total order using the `getFirstTransition` and `getNextTransition` functions.

Note that our choice of operations has already limited the type of intermediate representations that can use this generic layer (i.e. precisely *one* initial state is required and all outgoing transitions of a state are required to be in some total order). This is a compromise between generalising the model interface to be compatible with a large number of intermediate representations and providing a set of operations through which explicit-state model checking can be achieved efficiently.

To complete the interface we add methods that map transition objects to the source state object (`getSource`), to the target state object (`getTarget`) and to a label object (`getLabel`). All operations are conveniently gathered in the model interface such that there are no restrictions on the implementation of the state, label and transition objects.

Note that all operations work with pointers to elementary types, to facilitate the need for sharing instances. For example, labels are likely to label multiple transitions of the model, and it might be useful for these to be represented by the same label instance.

The prototype implementation of this generic layer actually uses reference counting pointers to keep track of all instances that were provided through the model interface. This arises from the fact that it is written in unmanaged C++, and any created instance will need to be deleted somewhere. As instances might be shared, it is not obvious where this deletion should happen. Reference counting pointers provide additional flexibility to avoid this problem. We use regular pointers in our code listings to make them easier to understand.

3.2 Generic Algorithms

To illustrate how one can define reusable algorithms over the model interface we use the example of a basic depth-first search, as provided in [14]. Although this algorithm is not a very realistic example of an algorithm used in explicit-state model checkers, it is useful to illustrate how this algorithm can be implemented generically (i.e. for all intermediate representations). A more realistic example can be found in [16].

In an idealistic scenario the model interface itself would provide sufficient functionality for any algorithm that we wish to implement in the generic layer. In practice this is not feasible. For example, in the case of our basic depth-first search algorithm, we are looking for erroneous states. As we cannot assume anything about the state type, and this information is not present in the model interface, we will need to get this information elsewhere. Furthermore, instead of simply looking for erroneous states, we would like to generalise the depth-first algorithm to look for any type of 'goal state'. This results in the introduction an additional interface called `GoalCondition`, which contains a single abstract function `isGoalState` that can be used to determine whether a state is a goal state or not (see Listing 2). Note that this interface also uses type parameters, and that if an intermediate representation wishes to use the depth-first algorithm then it will also have to provide an implementation of the

```
template <typename S, typename L, typename T>
class GoalCondition
{
  public:
  virtual bool isGoalState(S* s) =0;
};
```

Listing 2 – The `GoalCondition` interface has a single function `isGoalState` which identifies states of interest. This function is typically implemented in the abstract layer.

`GoalCondition` interface, specialised with the same types (note that type parameters `T` and `L` are not essential for this particular interface, but throughout our implementation we have included all type parameters in all interfaces for consistency).

The definition of the `GoalCondition` enables a search for an arbitrary set of states. This set will typically be specific to an intermediate representation, and therefore will be implemented in the abstract layer. Examples are *accepting states* for automata, *erroneous states* for programs, or the *solved state* for Rubiks cubes. Alternatively, the set of states could be identifiable in a generic way (i.e. for all intermediate representations). As we cannot assume anything about the types of states, transitions and labels, this is not very common. Examples are the *initial state* and *deadlock states*. Deadlocks states can be found generically by checking whether a state has any outgoing transitions.

Now that the issue of detecting erroneous states has been addressed we can implement the basic depth-first algorithm generically. An implementation of this algorithm inside an encapsulating class is shown in Listing 3. This encapsulating class, `BasicDepthFirstSearch`, also abstracts over the type of state, label and transitions used in the algorithm. It has two fields, `m` is an implementation of an `ExplicitStateModelInterface` and `g` is an implementation of `GoalCondition`, both specialised with the paramethised types of the encapsulating class. The `dfs` function is a direct translation the algorithm in [14] to C++ code, but implemented generically.

If we were to include `BasicDepthFirstSearch` in our architectural diagram, this would result in a generic layer as depicted in Figure 4. The `BasicDepthFirstSearch` block has an association with the `ExplicitStateModelInterface` and with the new interface `GoalCondition`, because these are fields used in the algorithm. The `GoalCondition` has two generic implementations, and is potentially implemented for some intermediate representations in the abstract layer.

The introduction of another interface (`GoalCondition`) does not add significant requirements to the abstract layer. Firstly, implementing an interface other than the model interface should be fairly

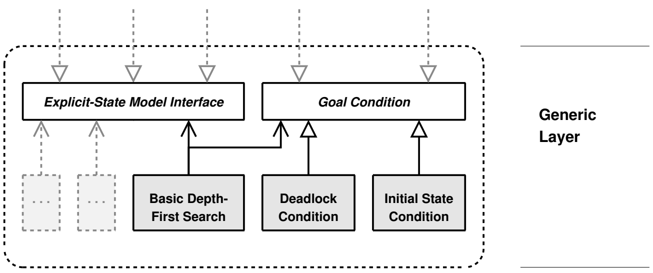

Figure 4 – The generic layer of the framework as it would look if we encorporated `BasicDepthFirstSearch` and `GoalCondition`.

straight-forward. For example, if there exists an intermediate representation for automata, then checking whether a state is accepting (e.g. implementing a `GoalCondition` for *accepting states*) should be a trivial task. Additionally, these interfaces do not have to be implemented unless the algorithm that uses these interfaces is used. Finally, it is not impossible that there already exists a generic implementation with the desired functionality.

Although we used a very simple example to illustrate the implementation of generic functions in our framework, we argue that this technique is scalable and can be applied to realistic algorithms that are used in model checking today. The actual algorithms implemented in our framework provide both simulation and reachability algorithms. Rather than providing a number of distinctly separate algorithms, we chose to apply a more modular approach. We would like to emphasise that our implementation of simulation and verification functionality is just one of many possible approaches. A simplified overview of the implemented generic layer is presented in Figure 5. As is evident from the figure, algorithms are no longer represented by a single block, but are divided into several blocks to provide a greater degree of flexibility.

The `Simulation` class is associated with both a `SimulationStrategy` and a `SimulationObserver`. These are both interfaces, and can be implemented generically or can be specialised to suit a specific intermediate representation.

```
template <typename S, typename L, typename T>
class BasicDepthFirstSearch
{
  private:
  /* model under consideration */
  ExplicitStateModelInterface<S, L, T>* m;
  /* the goal of this search */
  GoalCondition<S, L, T>* g;

  ...

  public:
  void dfs(std::set<S*>& Statespace, S* s)
  {
    /* if s is a goal state */
    if (g->isGoalState(s)) {
      /* report goal */
    }
    else {
      /* add s to state space */
      Statespace.insert(s);

      /* iterate over transitions of s */
      T* tr = m->getFirstTransition(s);
      while (tr != 0) {

        /* get target state of tr */
        S* t = m->getTarget(tr);

        /* if t is not in Statespace, then dfs */
        if (Statespace.find(t) == Statespace.end())
          dfs(Statespace, t);

        /* get next transition of s*/
        tr = m->getNextTransition(tr);
      }
    }
  }
};
```

Listing 3 – A generic implementation of the basic depth-first search algorithm in [14]. Requires an implementation of an `ExplicitStateModelInterface` and a `GoalCondition`.

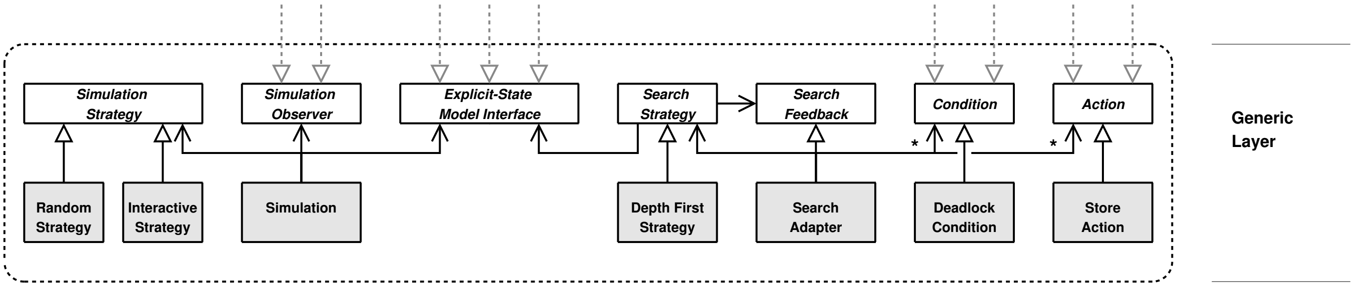

Figure 5 – The architecture of the generic layer, as implemented in our framework. The left-hand side facilitates a simulation algorithm, whereas the right-hand side shows a modular implementation of a reachability algorithm.

The `SimulationStrategy` is responsible for choosing a path through the model, and has generic implementations for random, interactive and guided strategies. Specialised implementations could include random strategies that take into account the probabilities associated with transitions, if it is a specialisation for an intermediate representation that has such a notion of probabilities. The `SimulationObserver` provides a way for tools to observe the simulation, and would most likely consist of specialised implementations to update user interfaces.

The search functionality offered by our framework is slightly more complex. The `SearchStrategy` is an interface for search strategies, whose implementations will have full control over the order of traversal of the states in the model. Currently the only implementation available is a depth-first strategy. Any strategy relies on feedback from `SearchFeedback` such as '*this state was previously visited*', '*this is a new state*' or '*this is a goal state*'. `SearchAdapter` implements this feedback procedure by maintaining pairs of *conditions* and *actions*. `Condition` identifies certain states or transitions, and is in fact very similar to `GoalCondition`. When such a condition holds then an `Action` is executed. Examples of such actions could include storing a state in a store, starting a nested search or reporting a goal state. The feedback given by the `SearchAdapter` is dependent on the actions that were executed. Simple searches can be constructed by combining conditions and actions in a simple fashion, e.g. '*always store a state*' and '*if this state is in the store, report that this state was previously visited*' and '*if this is a goal state, report this goal state*'. The simulation and search functionality of our framework is explained in more detail in [16].

The usage of type parameters in algorithms does not necessarily have an impact on performance. The abstraction is resolved at compile-time, and does not add significant run-time cost in modern compilers. For instance, the standard library of C++ (`std`) is also based on type parametrisation and is generally considered to be very efficient.

3.3 Graph-based intermediate representation

We have explained how generic functionality can be defined in the generic layer, but have not yet addressed any implementation of the abstract layer. In this section we will discuss the intermediate representation that was implemented in our prototype tool. We would like to emphasise that this implementation is only one of many possible intermediate representations that could be defined.

The type of models we will be trying to target are simple software-based models with guard-based process descriptions, global and local variables with primitive and pointer types, as well as dynamic process and data creation. We will use this intermediate representation to verify a subset of PROMELA in Section 3.4. Listing 4 shows that we have a `SoftwareModel` which implements the `ExplicitStateModelInterface` and specialises the type parameters with `SoftwareStates`, `Statements` and `SoftwareTransitions`. The remainder of this section will elaborate on the implementation of `SoftwareModels`.

Due to the dynamic nature of our target models, we will use a graph-based representation of states in our intermediate representation. Our graph-based state representation is based on the representation used in BOGOR [19]. Data values and process instances are *nodes*, whereas variables induce *edges* in our *state graphs*. If a variable has a value then it is represented as an edge originating from the scope in which it is defined (typically a process instance) to the data value this variable evaluates to in the current state of the model. Additionally, we have a global node which acts as the start node for global variables.

We chose to model pointer variables as special kinds of variables, rather than introducing an additional level of indirection. Stategraphs annotate edges that are induced by these pointer variables. Typically, pointer variables model heap data, whereas normal variables model stack data. We require heap and stack data values to be strictly separate (i.e. a pointer variable can never point to the value of a normal variable).

By using the state graph representation our intermediate representation is a simplification of real-life software, because we do not model concepts such as memory location, functions and classes. We

```
class SoftwareModel
  : public ExplicitStateModelInterface
  <
    SoftwareState,
    Statement,
    SoftwareTransition
  >
{
  virtual SoftwareState*
    getInitialState();
  virtual SoftwareTransition*
    getFirstTransition(SoftwareState* s);
  virtual SoftwareTransition*
    getNextTransition(SoftwareTransition* tr);

  virtual SoftwareState*
    getSource(SoftwareTransition* tr);
  virtual Statement*
    getLabel(SoftwareTransition* tr);
  virtual SoftwareState*
    getTarget(SoftwareTransition* tr);
};
```

Listing 4 – An implementation of the `ExplicitStateModelInterface` by an intermediate representation of `SoftwareModels`.

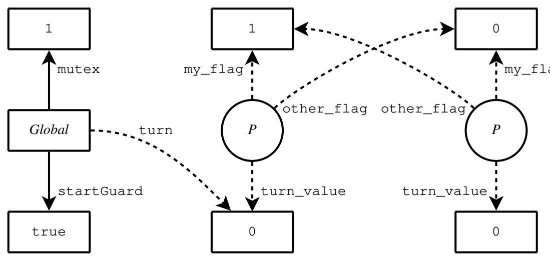

Figure 6 – A graphical representation of a state in our intermediate representation.

consider abstraction over memory locations to be a good thing, as this means detecting heap symmetry reduces to checking whether two state graphs are isomorphic. The other simplifications have been made due to time limitations, and would be welcome additions to our intermediate representation. We informally address the inclusion of features such as functions and classes in [16].

Figure 6 shows a state graph of a model, which is actually a reachable state of the PROM$^+$ model shown in Listing 5. The formal definition of state graphs has been explained in [16], we shall just explain them informally. Circles are process instances, whereas rectangles are data instances. Edges induced by variable values are labelled with the variable name and are dashed only if the variable is a pointer variable.

The implementation of state graphs is relatively straight-forward (see the top left portion of Figure 8). A `SoftwareState` has an association with a `GlobalInstance` and some `ProcessInstances`. We presume that every other node in the state graph is reachable from either the global instance or a process instance.

It is clear from the example that the models we try to target are very dynamic in nature. For instance, we cannot determine how many process instances are going to be created during runtime by means of static analysis, nor can we predict what state graphs we will encounter. This implies that it is sensible to construct the state space on-the-fly (alternatively one could construct the whole state space at once, but just feed the model interface this information on-the-fly).

To facilitate the on-the-fly creation of our models, we will need to implement the semantics of our model through our transition and label type. We mentioned previously that a *statement* is a suitable candidate for a label type. As is evident from Listing 5, statements are part of the control-flow of process types. Multiple process instances can share the same process type, and this process type can be shared by multiple `SoftwareStates`. To facilitate the notion of type, we introduce a type graph to our intermediate representation (which is a type graph for every state graph of the model). This type graph includes nodes for process types, data types, and shows possible variable relations between these types. It is here that we store model-wide information such as the control-flow, the types of variables, statements, etc. This type graph *can* be extracted by means of static analysis. Figure 7 shows the type graph extracted from Listing 5.

The implementation of the type graph is shown on the top right portion of Figure 8. Similarly to the state graphs, all nodes in the type graph are reachable from the `GlobalType` or a `ProcessType`. As this information is model-wide, a `SoftwareModel` has associations with the `GlobalType` and all `ProcessTypes`. As can be seen in Figure 8, `ProcessTypes` implement the model interface too, because their control-flow is considered to be a type of explicit-state model too. This makes it possible to query the control-flow of process types in an on-the-fly manner.

A `SoftwareModel` normally only has an initial state graph and a type graph at its disposal to realise the operations in the model interface, which are extracted using static analysis. We will informally explain how a `SoftwareModel` implements the model interface using only this information. The `getInitialState` is simply a trivial operation to retrieve the initial state. The `getFirstTransition` and `getNextTransition` operations are responsible for constructing all enabled `SoftwareTransitions` originating from a `SoftwareState`. Although this information is retrieved in several steps, here we will suffice with explaining how one can extract all enabled transitions from a `SoftwareState` (which is given as an argument) using Figure 8.

The idea is that each `SoftwareState` contains a certain number of `ProcessInstances`. Each of these `ProcessInstances` has a `ControlFlowState` which represents the program counter of this process. For each of these `ProcessInstances`, we look up the corresponding

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

active [0] proctype P(
  bit * my_flag;
  bit * other_flag;
  bit * turn_value)
{
  /* Wait for initialisation */
  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 P(flag_1, flag_2, turn_1);
  run P(flag_2, flag_1, turn_2);

  /* Do not break symmetry */
  reset flag_1;
  reset flag_2;
  reset turn_1;
  reset turn_2;

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

Listing 5 – An implementation of Petersons mutual exclusion algorithm [2] in PROM$^+$.

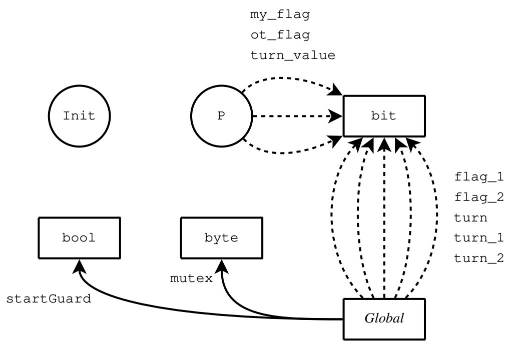

Figure 7 – The type graph of all state graphs in the model described by Listing 5.

ProcessType. Using the getFirstTransition and getNextTransition of the ProcessType we can retrieve all possible ControlFlowTransitions from the current ControlFlowState. An expression in the Statement associated with this transition (i.e. the guard) enables us to see whether this transition is available for the current state. If it is, then we can construct a SoftwareTransition using the information that we have just found. A SoftwareTransition is basically a tuple of the SoftwareState, a reference to the executing ProcessInstance and the ControlFlowTransition that is associated with this step. The getSource and getLabel functions of the model interface are therefore trivial to implement, and if at some point the getTarget function is called, then the ControlFlowTransition along with the Statement are responsible for copying and modifying the SoftwareState into a new state. Statements are therefore basically programmatic implementations of graph morphisms, and contain implementations such as assignments, expressions, assertions etc. Due to the dynamic nature of our states implementing these statements is not entirely straight-forward. For instance, in order to assign to a variable, its edge must first be located in the graph.

In addition to the functionality shown in Figure 8 we have also implemented a means of linearising SoftwareStates to bit vectors. Representing states as a sequence of bits is much more efficient than representing them as a number of object instances and is the typical approach undertaken by model checkers. Our linearisation uses the fact that SoftwareStates are actually programmatic representations of state graphs. By means of a simple algorithm we encode our graphs, using the fact that there exists a type graph, that every node is reachable from the global instance or a process instance, and that the state graphs are deterministic. Details of our encoding algorithm can be found in [16] and is different to the method used in [19]. Heap symmetry is achieved automatically, as isomophic graphs are encoded to the same bit vector with our algorithm. Additionally, process symmetry can be achieved by ignoring the process identifier of process instances during the encoding procedure, and by letting the encoding algorithm look only at the type of the processes and their program counters. This creates representatives that are not necessarily canonical (i.e. some thread-symmetrical states have different representatives), but offers a reasonable reduction with a low run-time overhead.

3.4 PROM$^+$ model checker

As a proof of concept, the framework has been used to build a model checker for PROM$^+$, which is a subset of PROMELA [13] aug-

```
        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 9 – The grammar of PROM$^+$ in EBNF style. The syntax and semantics of PROM$^+$ are based on PROMELA [13].

mented with features for dynamic memory allocation. The grammar of PROM$^+$ is depicted in Figure 9.

The syntax of PROM$^+$ is almost all interpretable as PROMELA, and the although PROM$^+$ syntax is much more restricted, the syntax that is permitted has the same semantics as in PROMELA. The semantics of PROMELA are described in detail in [13]. In contrast to PROMELA, PROM$^+$ only allows the declaration of variables of primitive types, and requires these to be either prior to all process declarations or prior to any statement in a process declaration. There is no means of explicitly giving these variables an initialisation value, and all variables are initialised with 0. Although PROM$^+$ lacks many features of PROMELA (i.e. channels, arrays, typedefs, mtypes), it does have facilities for dynamic object creation that PROMELA does not have. We will briefly explain the semantics of the newly introduced syntax.

Pointer variables can be declared like normal variables by using an additional ‘*’, similar to C (pointer variables are initialised as null-pointers). These pointer variables are only allowed to refer to heap data (data created by a new statement), and cannot point to the same data instances as normal variables. The reset statement resets a pointer variable to a null-pointer. The assignment and comparison operators work similar to how they do in C (i.e. whether an assignment is an assignment by reference or by value can depend on the variable declarations). Note that if a data instance allocated by a new statement is no longer reachable in the state graph, then it is destructed. Therefore one could say we employ some form of garbage collection, although this is implicit as non-reachable instances in the state graph are simply not encoded.

It turned out to be relatively easy to combine the two layers of the concrete architecture to construct a model checker for PROM$^+$. In the previous section we mentioned that all that is needed for SoftwareModels was an initial state and all the typing infor-

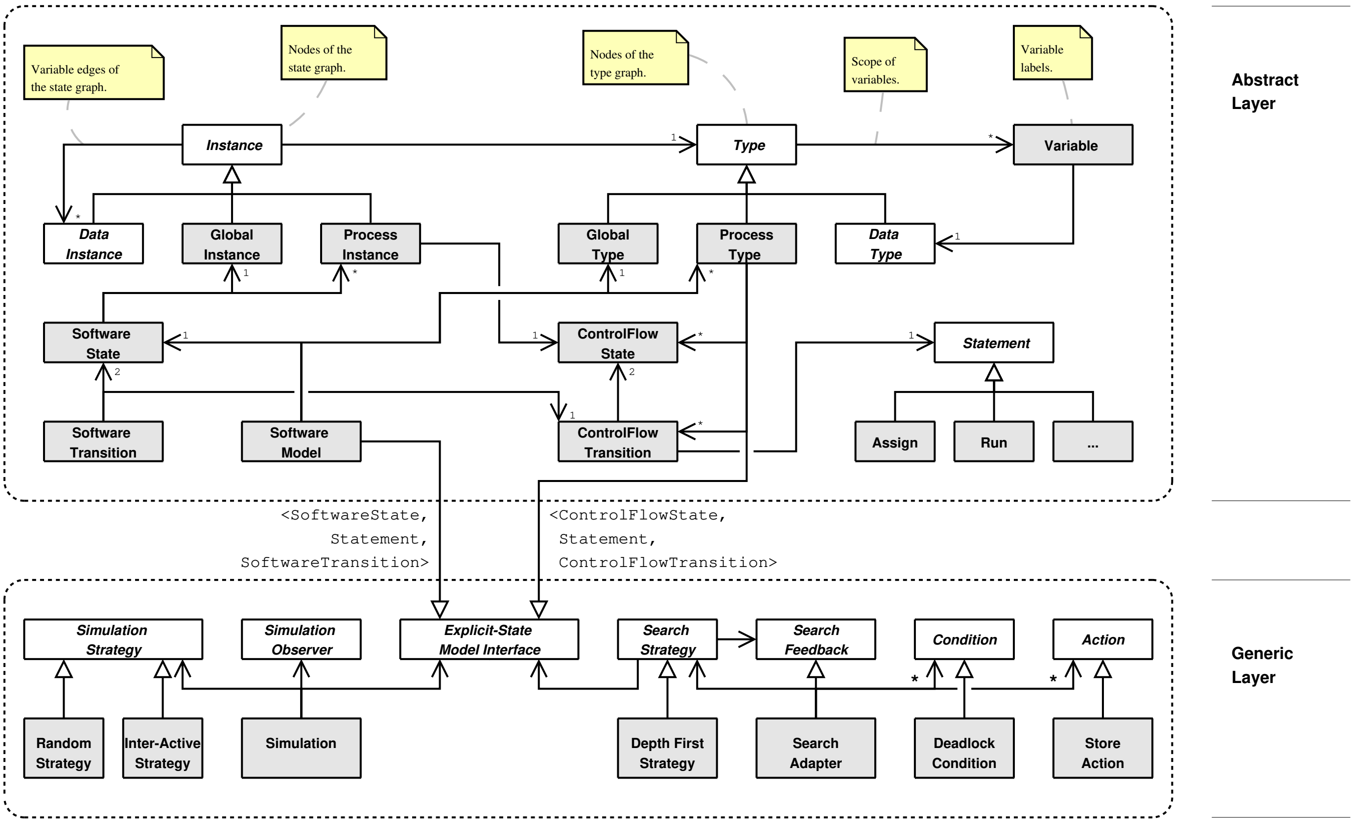

Figure 8 – The conceptual architecture of the framework, divided into a *abstract layer* and an *abstract layer*.

mation. By means of a parser we can generate this information in a straight-forward manner. Once the `SoftwareModel` is constructed, one only has to instantiate the desired algorithms in the generic layer with the appropriate types in order to use them.

An example of a PROM$^+$ model can be found in Listing 5. This is a model specification of the mutual exclusion algorithm by Peterson (as described in [2]). This particular model enables the exploitation of thread-symmetry as the parametrisation of the processes with pointer variables creates state graphs that are thread-symmetrical. In [16] we have developed several models in both PROM$^+$ and equivalent models in a subset of both PROM$^+$ and PROMELA such that we could analyse the effectiveness of our thread-symmetry reduction and to compare the performance of our prototype tool to SPIN.

3.5 RESULTS

The primary new concept of this work is the use of a *layered architecture* in combination with *type parametrisation* to provide reusable algorithms for explicit-state verification. We argue that most of the functionality of our prototype implementation is indeed reusable, and therefore the conceptual architecture does enable reuse in the way we have intended. Not only can reuse be achieved by using the same algorithms for different intermediate representations, different tools could also use the same intermediate representation. For instance, if we have a testing tool and a verification tool for PAs then it makes sense to use the same intermediate representation. Sharing an intermediate representation would improve the interoperability of tools.

The preliminary experiments with the prototype (see [16]) have shown that, with respect to memory consumption (the average size of the bit sequences that represent states), the prototype is comparable or at times even more efficient than SPIN. With respect to time, however, SPIN is still three orders of magnitude faster. Obviously, the design philosophies behind SPIN are directly opposite to those of ours, and it is therefore not surprising that the performance of our tool is worse. SPIN has been continuously optimised to verify PROMELA models as efficiently as possible, thereby making it very difficult to reuse SPIN. In contrast, we sacrifice performance in order to enable reuse. Despite this difference, it is our expectation that we can improve the prototype implementation to achieve performance nearer one order of magnitude slower than SPIN, without sacrificing the principles of our conceptual architecture.

There are a few design choices that have seriously impacted the performance of our tool. Firstly, the choice to use reference counting pointers has placed a significant overhead on everything in the framework. Secondly, the choice to implement reachability algorithms in a modular fashion comes at the cost of a lot of overhead in the form of function calls, which could be avoided by means of more specialised algorithms. Finally, the choice to use graphs to represent states in the intermediate representation comes at the cost of expensive graph operations (such as linearisation). These issues could be improved without changing the principles of our conceptual architecture.

4 FUTURE WORK

The proof-of-concept framework already shows significant potential, but to meet our objectives the framework should be extended in several directions. New generic layers (e.g. for symbolic or bounded model checking) are anticipated. Different intermediate representations (other than the current graph implementation) should be developed for the current explicit-state model interface. Additionally, the functionality in the generic layer should be extended by adding new reusable algorithms. This could include new search strategies or verification of liveness properties (informally addressed in [16]).

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.

We expect that the architecture and library will develop into a useful and reusable generic library for formal verification.

REFERENCES

[1] Gerd Behrmann, Alexandre David, and Kim G. Larsen, 'A tutorial on UPPAAL', in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT 2004)*, eds., Marco Bernardo and Flavio Corradini, volume 3185 of *LNCS*, pp. 200–236. Springer–Verlag, (2004).

[2] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, Prentice-Hall International Series in Computer Science, Prentice-Hall, 1990.

[3] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petruccie, Ph. Schnoebelen, and P. McKenzie, *Systems and Software Verification: Model-checking techniques and tools*, Springer-Verlag, 2001.

[4] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis, 'The IF TOOLSET', in *International School on Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, eds., Marco Bernardo and Flavio Corradini, volume 3185 of *LNCS*, pp. 237–267. Springer-Verlag, (2004).

[5] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang, 'Symbolic model checking: 10^{20} states and beyond', in *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS 1990)*, ed., John Mitchell, pp. 428–439. IEEE Computer Society Press, (1990).

[6] 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*, ed., Reinhard Wilhelm, volume 2000 of *LNCS*, pp. 176–194. Springer-Verlag, (2001).

[7] Edmund M. Jr. Clarke, Orna Grumberg, and Doron A. Peled, *Model Checking*, MIT Press, 1999.

[8] Rance Cleaveland and Steve Sims, 'The NCSU Concurrency Workbench', in *8th International Conference on Computer Aided Verification (CAV 1996)*, eds., Rajeev Alur and Thomas A. Henzinger, volume 1102 of *LNCS*, pp. 394–397. Springer-Verlag, (1996).

[9] T. Courtney, D. Daly, S. Derisavi, V. Lam, and W. H. Sanders, 'The MÖBIUS modeling environment', in *Tools of the 2003 Illinois International Multiconference on Measurement, Modelling, and Evaluation of Computer-Communication Systems*, Research report no. 781/2003, pp. 34–37. Universität Dortmund Fachbereich Informatik, (2003).

[10] Daniel D. Deavours, Graham Clark, Tod Courtney, David Daly, Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick G. Webster, 'The MÖBIUS framework and its implementation', *IEEE Trans. Softw. Eng.*, **28**(10), 956–969, (2002).

[11] Salem Derisavi, Peter Kemper, William H. Sanders, and Tod Courtney, 'The MÖBIUS state-level abstract functional interface', *Perform. Eval.*, **54**(2), 105–128, (2003).

[12] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, and Robby, 'Building your own software model checker using the BOGOR extensible model checking framework', in *17th International Conference on Computer Aided Verification (CAV 2005)*, eds., Kousha Etessami and Sriram K. Rajamani, volume 3576 of *LNCS*, pp. 148–152. Springer-Verlag, (2005).

[13] Gerard J. Holzmann, *The* SPIN *Model Checker – Primer and Reference Manual*, Addison-Wesley, 2004.

[14] Gerard J. Holzmann, Doron Peled, and Mihalis Yannakakis, 'On nested depth-first search', in *The Spin Verification System*, eds., Jean-Charles Grégoire, Gerard J. Holzmann, and Doron A. Peled, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, (1996).

[15] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli, 'Multi-valued decision diagrams: theory and applications', *International Journal on Multiple-Valued Logic*, **4**(1-2), 9–62, (1998).

[16] Mark Kattenbelt, *Towards an explicit-state model checking framework*, Master's thesis, University of Twente, Enschede, The Netherlands, 2006. (available from `http://www.cs.bham.ac.uk/~mxk`).

[17] K. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[18] Arend Rensink, 'Towards model checking graph grammars', in *Workshop on Automated Verification of Critical Systems (AVoCS 2003)*, eds., Michael Leuschel, Stefan Gruner, and Stphane Lo Presti, Technical Report DSSE–TR–2003–2, pp. 150–160. University of Southampton, (2003).

[19] Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif, 'Space-reduction strategies for model checking dynamic software', *Electronic Notes in Theoretical Computer Science*, **89**(3), (2003).

[20] Claus Schröter, Stefan Schwoon, and Javier Esparza, 'The Model-Checking Kit', in *24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003)*, eds., Wil M. P. van der Aalst and Eike Best, volume 2679 of *LNCS*, pp. 463–472. Springer-Verlag, (2003).

[21] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, third edn., 2000.