Model Checking Dynamic and Hierarchical UML State Machines

Toni Jussila[1], Jori Dubrovin[2], Tommi Junttila[2], Timo Latvala[3], and Ivan Porres[4]

[1] Johannes Kepler Universität Linz
[2] Helsinki University of Technology
[3] University of Illinois at Urbana-Champaign
[4] Åbo Akademi University

Abstract. This paper presents a technique to model check UML specifications by translating UML models to the model checker SPIN. Our models consist of active UML classes, whose behavior is defined by hierarchical state machines. The intended application is to find errors in protocols communicating using asynchronous message passing. Compared to previous efforts using a similar approach, our novel points are the following. First, we consider a subset of UML that in our opinion is expressive enough for protocol models but allows a simpler translation to SPIN than existing work. Preliminary analysis of simple industrial models support our conclusions on the expressivity of our UML subset. Second, we present a powerful action language that is still amenable to automatic analysis. The action language is used to specify the effects of transitions, which may include dynamic creation of new objects. Finally, we discuss an even simpler SPIN translation for flattened UML state machines and compare it to the translation that supports hierarchy.

1 Introduction

Model-based approaches for system design have been studied for a long time. Advantages associated with model-based approaches are several. Models give designs a restricted implementation independence, and they also provide a convenient form of documentation. However, arguably the most important benefit is that the level of abstraction of the design is raised. This has many implications. Abstract models allow efficient communication of the design, since unnecessary details are hidden. They also facilitate testing and verification of the design at an early stage, the topic of this paper. The widely acknowledged benefit of this is that it is much cheaper to detect and correct software errors early in the design process.

We report preliminary results from a project, where the goal is to find errors in protocol designs using *model checking*. In our approach, the protocols are modeled using UML class diagrams and state machines. The Unified Modeling Language (UML) is a standardized graphical notation for modeling and documenting object-oriented software and business processes. UML is also the most widespread software modeling language, and it is accepted in the industry as the standard language for software analysis and design. UML state machine models can fairly naturally capture protocol designs, where the communication is asynchronous and data can be abstracted with the class subtyping mechanism.

Our approach to model checking UML designs is based on using the state-of-the-art model checking tool, SPIN [1]. Models with assert specifications are automatically translated to SPIN's input language, and counterexamples can be simulated. More advanced properties can be specified in the SPIN model as temporal logic formulae.

Our work improves or differs from previous work in the following ways. The subset of UML we support has specifically been chosen to be expressive enough for our intended application, modelling protocols, yet it allows a precise and fairly simple formal semantics. We present an action language, which is used to specify effects of transitions, that is powerful and amenable to automatic analysis. Supported features include dynamic creation of objects, the usual flow constructs, and arithmetic expressions. We also discuss a simpler SPIN translation for flattened state machines and compare it to the translation that supports hierarchy.

2 Related Work

The idea of applying model checking on UML-type state machine models is not new. Latella et al. [2] present a translation from UML state machines to PROMELA, the input language of SPIN. They only allow a model to contain a single state machine. In another translation by Mikk [3], the input language is not UML but statecharts, which is a similar formalism with different semantics. Perhaps the closest work to ours is [4] which presents a tool called vUML that translates UML to PROMELA. vUML supports a larger subset of UML than our approach. This, however, has the effect that the translation of a UML state machine is more complex; it requires a code block that chooses the transitions to be fired and another block that models the effects of transitions. All these works have the limitation that no data attributes can be associated with objects or state machines. Consequently, there is no action language, and the only possible effect of a transition is to send signals with no parameters.

The Hugo project [5] also supports SPIN as a back-end to verify UML models. Their initial PROMELA translation was only feasible for very small models, and the current version of the tool follows ideas similar to those in vUML. To our knowledge, the translation is undocumented. In [6], SPIN is used to generate test cases from abstract state machines. The OMEGA project [7] has created a tool set focusing on real-time properties. Their approach is based on translating UML to the IF intermediate language that has several model checking back ends. The Rhapsody UML verification environment [8] supports model checking of UML state machines by translating the models to the input language of the VIS symbolic model checker. Most UML constructs are supported, and the action language is a restricted subset of C++. Features that are not supported include deferred events and do activities.

3 UML Subset and Semantics

In our framework, a UML model contains classes, state machines, and deployment diagrams. Classes may contain instance attributes, i.e. data values associated with objects, and there may be associations between classes. Operation calls are not supported. Also, a class cannot be a subtype of another class, but we are planning to incorporate support for subtype relationships. We assume that all classes are active classes whose behavior is defined by behavioral state machines, discussed below. At run-time, objects communicate with each other asynchronously using signals. Signals may have associated parameter values. A deployment diagram is used to specify the initial configuration of objects.

3.1 A Well-Defined UML State Machine Language

Compared to standard UML state machines, we consider a subset. This is motivated by the need of a precise behavioral design language that can be verified efficiently.

UML offers two mechanisms for modeling concurrency: active objects communicating with signals, and orthogonal regions in the state machine of an object. We argue that the first approach is often preferable in an object oriented setting, where it is more natural to put emphasis on communicating objects (which can be dynamically created) instead of concurrent behavior within a single object. Indeed, the commercial UML tool Telelogic Tau [9] does not even allow orthogonal regions in state machines.

Our subset allows orthogonality, but we propose a restriction that no two transitions in orthogonal regions can be enabled by the same event. We argue that this violates the idea of orthogonality and creates complicated dependencies that are hard to understand and analyze. Together with execution semantics in which at most one completion transition (transition without an explicit trigger) is fired at a time, the restriction guarantees that at most one region changes its state in each step. This corresponds to interleaving execution semantics suitable for the SPIN model checker, and results in a simple implementation of the transition selection algorithm.

UML allows continuous do activities in states, but we omit them because the model checker executes models in discrete steps. Furthermore, we do not currently support history pseudostates, fork or join pseudostates, or entry or exit activities in states. These are advanced modeling concepts that could later be incorporated to our framework.

In the following, the structure of state machines is formalized.

3.2 Structure of State Machines

A state machine contains *states* and *transitions* between them. A *composite state* contains one or more orthogonal *regions*, which in turn contain substates. During execution, one or more states are *active*. If a composite state is active, then exactly one direct substate in each region is active. Whenever a substate is active, its containing composite state is also active.

Let Σ be a finite set of states consisting of simple states Σ_{simple}, initial pseudostates $\Sigma_{initial}$, choice pseudostates Σ_{choice}, final states Σ_{final}, and composite states $\Sigma_{composite}$, and let $\mathcal{R}$ be a finite set of regions. We define a *child relation* $\searrow$ such that if a region r of a composite state c directly contains a state s, then $c \searrow r$ and $r \searrow s$.

Definition 1. *A tuple $H = \langle \Sigma_{simple}, \Sigma_{initial}, \Sigma_{choice}, \Sigma_{final}, \Sigma_{composite}, \mathcal{R}, top, \searrow \rangle$ is a* state hierarchy *iff $\searrow \subseteq \Sigma_{composite} \times \mathcal{R} \cup \mathcal{R} \times \Sigma$, and $\langle \Sigma \cup \mathcal{R}, \searrow \rangle$ is a tree rooted at $top \in \mathcal{R}$ such that the set of leaves of the tree is $\Sigma \setminus \Sigma_{composite}$.*

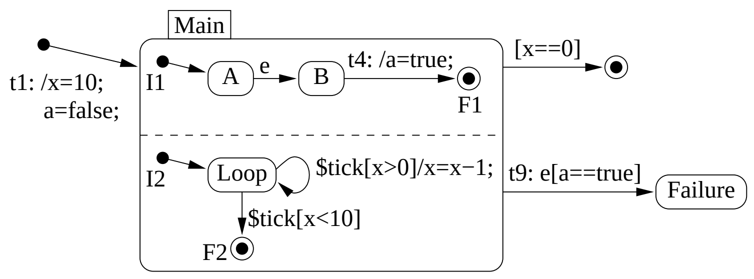


Fig. 1. Running example.

The inverse relation of $\searrow$ is the function $parent : \Sigma \cup \mathcal{R} \setminus \{top\} \rightarrow \Sigma_{composite} \cup \mathcal{R}$, giving the parent region of a state or the parent composite state of a non-top region. Thus, the direct substates of a composite state c are the states s such that $parent^2(s) = c$. The set of proper descendants of a state or region v is defined by $descendants(v) = \{v' \in \Sigma \cup \mathcal{R} \mid v \searrow^+ v'\}$, where $\searrow^+$ denotes the irreflexive transitive closure of $\searrow$.

Two states $s_1, s_2 \in \Sigma$ are *orthogonal*, denoted $s_1 \perp s_2$, iff there are regions $r_1, r_2 \in \mathcal{R}$ such that $r_1 \neq r_2$, $parent(r_1) = parent(r_2)$, $s_1 \in descendants(r_1)$, and $s_2 \in descendants(r_2)$. A set $S \subseteq \Sigma$ is *consistent* iff for any two distinct states $s_1, s_2 \in S$ either $s_1 \perp s_2$, $s_1 \in descendants(s_2)$, or $s_2 \in descendants(s_1)$. In particular, the set of active states in a state machine is always a maximal consistent set, i.e. a consistent set that is not a proper subset of any other consistent set.

Consider the state machine diagram shown in Fig. 1. States A and Loop are orthogonal, thus they can be active at the same time. In this situation, the entire set of active states would be $\{\text{Main}, \text{A}, \text{Loop}\}$. Notice that regions are not explicitly drawn in diagrams. Instead, the regions of a composite state are separated by a dashed line, and the top region contains the entire diagram.

A state machine transition is defined as follows. First, we assume the existence of a finite set of signals E, an expression language $\mathcal{L}_g$ for expressing guards, and an action language $\mathcal{L}_a$ for expressing effects. The languages are discussed in Section 3.5.

Definition 2. *A* transition *over a given state hierarchy is a tuple* $t = \langle s, e, g, a, S' \rangle \in (\Sigma \setminus \Sigma_{final}) \times (E \cup \{\tau\}) \times \mathcal{L}_g \times \mathcal{L}_a \times 2^{\Sigma}$ *such that there exists a region* $r \in \mathcal{R}$ *for which* S' *is a maximal consistent subset of* $descendants(r)$ *and* $s \in descendants(r)$, *and if* $s \in \Sigma_{initial} \cup \Sigma_{choice}$ *then* $e = \tau$. *We define* $source(t) = s$, $trigger(t) = e$, $guard(t) = g$, $effect(t) = a$, $targets(t) = S'$, *and* $container(t) = r$.

In the graphical UML notation, a transition is shown as an arrow from the source state to the main target state. A transition has a text label *name*: *trigger* [*guard*] /*effect*. Any of the four parts may be omitted.

A transition t has a source state $source(t)$ and a set of target states $targets(t)$. In the simple case, $targets(t)$ is a singleton set, but if the main target is a composite state, then we assume that $targets(t)$ also contains the initial states entered by the transition. For example, $source(\text{t4}) = \text{B}$ in Fig. 1, $targets(\text{t4}) = \{\text{F1}\}$, $source(\text{t9}) = \text{Main}$, and $targets(\text{t1}) = \{\text{Main}, \text{I1}, \text{I2}\}$. The intuition behind $container(t)$ is that it is the smallest region containing all the states exited or entered by the transition.

A transition can be fired only if an occurrence of its triggering event is dispatched. The trigger $trigger(t)$ is a signal name if the transition is triggered by the reception of a signal, or the special symbol τ if the transition is a completion transition. The symbol τ is omitted in diagrams. A transition also has an associated guard $guard(t)$, which defaults to `true` if it is omitted. The guard is a side-effect free Boolean expression giving a precondition for firing the transition. A transition may have an effect $\mathit{effect}(t)$ that is executed upon firing the transition. The default effect is to do nothing.

If an event occurrence is dispatched but it does not cause any transitions to be fired, the event can be *deferred* and dispatched again later. This happens if the event is designated deferrable by one of the active states. Thus, we define a mapping $\mathit{deferrable}$ from states to subsets of the set of signals E.

Definition 3. *A* UML state machine *is a tuple $\langle H, \Phi, \mathit{deferrable} \rangle$, where H is a state hierarchy, Φ is a set of transitions over H, and $\mathit{deferrable} : \Sigma \to 2^E$.*

3.3 State Configurations

A completion transition is triggered by an implicit completion event that is generated when the source state finishes all internal activity. For a pseudostate or simple state, this happens (in our UML subset) immediately when the state becomes active. For a composite state, a completion event is generated when all regions have reached their final states.

Instead of maintaining a queue of completion transitions, we associate equivalent information with each active state. We mark an active state *busy* if a completion event for the state has not yet been dispatched. An active state is *quiescent* iff it is not busy, i.e., iff a completion event has been dispatched without firing any completion transitions.

Definition 4. *A* state configuration *over a state hierarchy is a pair $\langle A, Q \rangle$, where the set of active states A is a maximal consistent subset of Σ and $Q \subseteq A$ is the set of quiescent states.*

A busy state is *completed* if, conceptually, a completion event for the state has been generated but not yet dispatched. Thus, a busy state is completed iff it is a non-composite state, or a composite state whose active substates are all quiescent final states.

Definition 5. *The set of* completed *states in a state configuration $C = \langle A, Q \rangle$ is*

$$\mathit{completed}(C) = \{ s \in A \setminus Q \mid \mathit{descendants}(s) \cap A \subseteq \Sigma_{\mathit{final}} \cap Q \}.$$

When a composite state c becomes active during execution, it is first busy and not completed. If all regions of the state reach their final states, then c becomes busy and completed and, conceptually, a completion event is generated. After that, it is possible to either (i) fire a completion transition whose source is c and whose guard is `true`, or, if there is no such transition, (ii) consume the completion event and make c quiescent. If c has become quiescent, no completion transitions from c will be fired because the completion event has already been consumed. This behavior fulfills the requirement of UML that the guards of completion transitions leaving a state are evaluated only once after the state has become completed.

If c above is replaced by a non-composite state s, the behavior is similar except that the phase of s being busy and not completed is skipped.

The initial state configuration of a state machine is $\langle\{s\},\{\}\rangle$, where $s \in \Sigma_{initial}$ and $parent(s) = top$. The model is ill-formed unless there is exactly one such s.

3.4 Transition Firing Dynamics

In general, a UML state machine instance moves from a state configuration to another by firing a maximal conflict-free set of enabled transitions. However, we have chosen to make the restriction, as discussed in Sect. 3.1, that orthogonal regions may not contain transitions triggered with the same signal. Formally, if $t_1, t_2 \in \Phi$ and $trigger(t_1) = trigger(t_2) \neq \tau$, then $\neg(source(t_1) \perp source(t_2))$. It follows that the maximal conflict-free set contains at most one transition.

A transition t is *enabled* by an event occurrence iff the event matches the trigger of t, the source state of t is active, and the guard of t evaluates to `true`. If an enabled transition t is fired, the resulting state configuration is obtained by first removing $source(t)$ and any other descendant states of $container(t)$ from the sets of active states and quiescent states, and then adding $targets(t)$ to active states. Nothing is added to the set of quiescent states because all new active states are busy.

Definition 6. *Let $C = \langle A, Q\rangle$ be a state configuration and let $t \in \Phi$ be a transition such that $source(t) \in A$. The* next state configuration *associated with t and C is*

$$nextstateconf(t, C) = \langle (A \setminus S) \cup targets(t), Q \setminus S\rangle,$$

where $S = descendants(container(t))$.

A completion transition is enabled iff its source state is active, busy, and completed, and the guard condition is true. If there is an active, busy, and completed state s that is not the source state of any enabled completion transition, we say that a *quiescing step* QUIESCE(s) is enabled. The only effect of firing the quiescing step is to make s quiescent, which corresponds to implicit consumption of the completion event for s. If there are any enabled completion transitions or quiescing steps, one of them is chosen nondeterministically for firing and (according to UML) only if there are none, signal-triggered transitions are considered.

Given a state configuration and an event occurrence, it is possible that several transitions are enabled. However, some of these are ruled out based on UML semantics, which states that transitions deeper in the hierarchy have priority. Priority also applies to deferral of events, but not to completion transitions because no busy, completed state can be a descendant of another one.

Definition 7. *Let T be the set of transitions enabled by an event e in a state configuration $\langle A, Q\rangle$ and let $S = \{source(t') \mid t' \in T\} \cup \{s \in A \mid e \in deferrable(s)\}$. The set of* prioritized *transitions is*

$$prioritized(T, A) = \{t \in T \mid descendants(source(t)) \cap S = \emptyset\}.$$

Because A is a consistent set of active states and the restriction on triggers in orthogonal regions holds, every $t \in prioritized(T, A)$ has in fact the same source state. One transition in the set is chosen nondeterministically for firing.

3.5 Action Language

In our subset of UML state machines, an action language is used in two roles, namely to specify the guard constraints and the effects of transitions.

The choice of an action language is connected to the level of support for various UML model elements. The minimal level is to allow sending signals to objects. Our action language supports more than this, e.g. attributes of objects and dynamic creation of new objects. The action language supported by our PROMELA translation is a subset of the Jumbala action language [10]. Jumbala is an object-oriented language that could be characterized as simplified Java tailored to the UML framework. The language is strongly typed with `int` (32-bit signed integer) and `boolean` primitive types and object reference types.

In state machines, the effects of transitions are lists of Jumbala statements, and the transition guards are `boolean` expressions. Below is a list of the kinds of statements supported by the PROMELA translation. The syntax and semantics follow the conventions of the Java programming language, with an added `send` statement.

– Assignments of the form '*lhs* = *rhs*`;`'.
– If statements of the form '`if (`*condition*`) {` *truestmt* `} else {` *falsestmt* `}`', where *condition* is a `boolean` expression. The `else` part may be omitted.
– Iteration statements of the form '`while (`*condition*`) {` *stmt* `}`'.
– Send statements of the form '`send` *signalname*`(`*paramvalues*`) to` *object*`;`'. A send statement places a signal event *signalname* in the input queue of *object*. Values for signal parameters are given as a comma-separated list.
– Assertions of the form '`assert` *condition*`;`'.

The following kinds of Jumbala expressions are supported. Below we assume that obj is the object in whose context the guard or action is evaluated.

– 32-bit decimal integer literals.
– The `boolean` literals `true` and `false`.
– The expression `this`, which is a reference to obj.
– Names of the form *identifier* or *identifier.identifier*. A name can resolve to either an object reachable from obj by following links (association instances), or an attribute.
– Infix expressions of the form *leftexpr op rightexpr*. The binary operator *op* can be one of `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `>`, `<`, `>=`, `<=`, `==`, `!=`, `<<`, or `>>`. The semantics of operators is the same as in PROMELA.
– Instance creation expressions of the form `new` *classname*`()`. The state machine of the newly created object begins executing automatically.

3.6 Execution of Models

At a given moment in time, the system is in a state that, as a whole, conforms to the UML model. We call this state the *global configuration* of the system. A global configuration consists of a set of objects, where each object obj contains the following information.

```
**while** true:
    pick an object $obj$
    $\langle A, Q\rangle := obj.stateconf$
    $sources := completed(obj.stateconf)$
    $compl := \{t \in \Phi \mid source(t) \in sources \wedge trigger(t) = \tau \wedge evalguard(obj, t, \langle\rangle)\}$
    $enabled := compl \cup \{\text{QUIESCE}(s) \mid s \in sources \wedge \nexists t \in compl$ such that $s = source(t)\}$
    **if** $enabled \neq \emptyset$:
        **if** $enabled \cap \Phi_{pseudostate} \neq \emptyset$:
            $enabled := enabled \cap \Phi_{pseudostate}$
        pick $t \in enabled$
        **if** $t = \text{QUIESCE}(s)$ for some $s$:
            $obj.stateconf := \langle A, Q \cup \{s\}\rangle$
        **else**:
            execute $effect(t)$ in the context $obj$
            $obj.stateconf := nextstateconf(t, obj.stateconf)$
    **else if** $obj.inputqueue$ is not empty:
        remove the first element $\langle e, params\rangle$ from $obj.inputqueue$
        $enabled := \{t \in \Phi \mid source(t) \in A \wedge trigger(t) = e \wedge evalguard(obj, t, params)\}$
        **if** $prioritized(enabled, A) \neq \emptyset$:
            pick $t \in prioritized(enabled, A)$
            $assign(obj, t, params)$
            execute $effect(t)$ in the context $obj$
            $obj.stateconf := nextstateconf(t, obj.stateconf)$
            push $obj.deferredqueue$ in front of $obj.inputqueue$
            $obj.deferredqueue :=$ empty
        **else if** $\exists s \in A$ such that $e \in deferrable(s)$:
            append $\langle e, params\rangle$ to $obj.deferredqueue$
```

Fig. 2. Execution Algorithm for UML Models

1. The values of the instance attributes of obj.
2. The links that are navigable from obj, pointing to other objects.
3. The state configuration of the state machine of obj, denoted $obj.stateconf$.
4. The input queue of obj, which we denote by $obj.inputqueue$.
5. The deferred queue of obj, denoted $obj.deferredqueue$.

The last two elements are FIFO queues whose elements are signal event occurrences represented as pairs $\langle e, params\rangle$, where $e \in E$ is a signal name and $params$ is a tuple of signal parameter values.

The algorithm in Fig. 2 illustrates the execution of a model. In one step of execution, one object is nondeterministically chosen. If any completion transitions or quiescing steps are enabled, then one of them is fired. Otherwise, an event occurrence is removed from the input queue and a prioritized enabled transition is fired. If no such transition exists and the event is not deferrable, the event occurrence is implicitly consumed. In transition selection, transitions whose source state is an initial or choice pseudostate (the set $\Phi_{pseudostate}$) are preferred to other completion transitions.

In order to handle signal parameters, the algorithm uses the following auxiliary procedures.

– $assign(obj, t, params)$ modifies the global configuration by assigning the values in the tuple $params$ to the instance attributes of obj. The attributes receiving the new values are named in the trigger of the transition t.
– $evalguard(obj, t, params)$ evaluates $guard(t)$ in the context of object obj. The values of $params$ are assigned as if $assign(obj, t, params)$ had been executed, for the duration of guard evaluation. After evaluation, the original attributes are restored and a truth value is returned.

The execution algorithm is such that in one step a single transition in any object is fired, and in the next step a transition in another object might be fired. We call this *transition segment granularity*. Alternatively, it would be relatively straightforward to modify the algorithm to use *compound transition granularity*, so that one step of execution would correspond to a compound transition, i.e. a sequence of transition segments with only pseudostates between them. A third possibility would be *run-to-completion granularity*, where the firing of a signal-triggered transition in an object is followed by as many completion transitions or quiescing steps as possible in the same object before executing transitions in other objects.

4 Translation to PROMELA

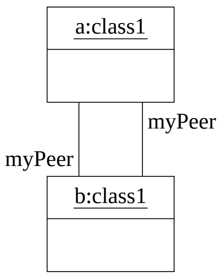
This section presents our main contribution, the translation to PROMELA. It translates the active classes and their state machines to corresponding PROMELA processes, and uses the deployment diagram to infer the initially active objects and how their associations to other active objects are set. The resulting PROMELA program can be checked for deadlocks or assertion violations in the model. If an error is found, the error trace can be simulated in the UML model by using a separate model simulator.

Our translation requires the user to supply information about the model that is otherwise hard to infer: (i) the size of the input and deferred queues (QSIZE), and (ii) the maximum number of instances of a each class (MAXIDS). It is relatively easy to augment the PROMELA translation to check whether these limits are exceeded, and give an error requesting the user to increase them if this happens.

4.1 SPIN and PROMELA: Brief Introduction

PROMELA (PROcess MEta LAnguage) is the input language of the tool SPIN [1] initially developed in Bell Labs by Gerard Holzmann et al. The language allows for the dynamic creation of processes and both synchronous (rendezvous) and asynchronous communication through communication channels.

The PROMELA language is rather rich, however our translation does not need most of the features. The elements that we use are briefly presented. The queue (asynchronous channel) operations are as follows. The send command `q!v1,...,vn` appends the queue `q` with the message comprised of the values `v1,...,vn`. Similarly, the receive command `q?v1,...,vn` reads the first message from the queue (or blocks if the queue is empty). A central part of our translation is the `if...fi` compound statement that can for instance be as follows:



```
#define MAXIDS 2
int class1_myPeer[MAXIDS];
int class1_pid = 0;
init {
   atomic {
      class1_myPeer[0] = 1;
      run class1(0); /* Start active object a */
      class1_pid = class1_pid + 1;
      class1_myPeer[1] = 0;
      run class1(1); /* Start active object b */
      class1_pid = class1_pid + 1;
   }
}
```

Fig. 3. An initial configuration and its PROMELA translation.

```
if
  :: (a == 1) -> a = a + 1;
  :: (a == 0) -> a = 1;
  :: else -> a = 0;
fi
```

The statement above defines a simple selection construct with three option sequences, each starting with a double-colon. Each option sequence starts with a guard (in our example, the first guard is (a == 1)). This guard must evaluate to true so that this option sequence can be executed. If several guards evaluate to true, then one of them is chosen nondeterministically and the associated commands are executed. The else sequence (if present) is chosen iff the guards of every other sequence evaluate to false.

Normally, the scheduled object in SPIN can change after each command. However, if this is not desired, it can be prevented by enclosing several commands inside an atomic block (with the keyword atomic). We also use this feature.

4.2 Global Variables and Initialization Block

The global variables of the PROMELA specification are the input queues and deferred queues for each class. These are arrays with MAXIDS slots. The associations for each class are also stored in a similar table. Finally, for each class, there is an integer that stores the process number of the last created instance of that class.

The initialization of the active processes (the structure of the PROMELA init block) is as follows. Each active object (declared in the deployment diagram) is processed in turn. First, its associations are configured by setting the values in the global association table and then the object is started with the command run with its process id as the argument. In order to maintain scheduling in the init-process, these commands are enclosed inside an atomic block. For instance, the left-hand side of Fig. 3 shows a deployment diagram describing an initial configuration of a model. It has two objects, a and b, that are instances of the active class “class1” and the initializations of the association “myPeer” from “class1” to itself. The right hand-side of Fig. 3 gives the PROMELA translation of the diagram.

4.3 Translation of State Machines

Figure 5 gives a skeletal translation of the state machine in Fig. 1. Each class is translated to a PROMELA process (a proctype declaration, like line 4 in Fig. 5). This process has one argument, the instance number of the created object. The first instance gets the number zero and the maximum number of instances is user specified. The instance attributes of the class and integer variables encoding the state configuration of each region in state machine of the class are declared first.

The main loop encoding the state machine is divided into two parts, evalcompletions and evaltriggers for completion and signal-triggered transitions, respectively (lines 11 and 34 in Fig. 5). This division is due to the fact that according to UML semantics, completion transitions have priority over signal-triggered transitions. Therefore the PROMELA code follows the following idea. First, fire completion transitions as long as possible. Then, consume a signal event from the event queue (or wait until the queue becomes non-empty), fire a signal-triggered transition, and go back to trying to fire completion transitions.

In order to handle completion transitions correctly without actually generating completion events and having a queue (with priority over the normal signal event queue) for them, we use the concept of busy and quiescent states introduced in Sect. 3.3. That is, for each simple and composite state that has outgoing completion transitions, we have two possible values in the PROMELA state vector (e.g. s_Top_Main_busy and s_Top_Main in the code in Fig. 5). Completion transitions are only evaluated if the state is marked busy. As pseudostates only have outgoing completion transitions, there is no need for such additional information for them.

The general structure for the block evaluating whether a completion transition can be fired is fairly simple as there is no need to take the transition priority caused by the hierarchy into account. In order to first fire completion transitions leaving from a pseudostate, the block consists of two consecutive, similar sub-blocks: the first only considers pseudostates (lines 12–21 in Fig. 5) while the second (lines 22–33) takes care of the simple and composite states. Both sub-blocks are just large if. . . fi blocks with one option sequence for each (pseudo)state with outgoing completion transitions. Each option sequence first checks whether the (pseudo)state in question is active (and busy and completed if it is a simple or composite state), and then non-deterministically chooses a completion transition whose guard is true. If there was no such completion transition, then (i) a quiescing step is taken if the state was simple or composite, or (ii) following UML specifications, an error is reported if the state was a pseudostate.

The code for signal-triggered transitions (lines 35–57) starts with a consumption of a signal event from the queue, or non-deterministic generation of an external signal if the state configuration is in a state that can consume an external signal event.[5] The general structure for evaluating whether a signal-triggered transition can be fired is a large if. . . fi block (lines 40–57) with one option sequence for each state in the top level state machine. The guards for the option sequences are simply checks of whether a particular state is active. The option sequence then depends on whether the state is

[5] An *external* signal is a signal with no parameters and whose name starts with a $-sign. It can be non-deterministically generated by the environment of the UML model. Such signals are convenient when modeling open or underspecified systems.

simple or composite. If the state s is composite, a nested if. . . fi block follows (for example lines 42–50 in Fig. 5), this time containing an option sequence for all the children of s. After this (possibly empty) block, a subsequent if. . . fi block evaluates for each outgoing signal-triggered transition whether (i) the trigger matches the consumed signal event, and (ii) the guard evaluates to true.

In both blocks described above, the code for firing a transition is similar. The encoding of the effect of the transition (a sequence of Jumbala statements) is presented in detail in Sect. 4.4. The translation of each transition is finished with PROMELA code that sets the new active state to be the target state of the transition. For composite states, we also have to set its regions to their initial states. If the target state has outgoing completion transitions, then the state is marked busy. After this, the control flow is transferred to the beginning of the block evaluating completion transitions.

4.4 Translation of Action Language

The supported constructs are presented in Section 3.5. Now, we describe their translation to PROMELA. The translation of simple assignments is simple, both variable declarations and assignments are syntactically similar in PROMELA and Jumbala.

Queue operations are used to send messages to active associations of an object or to read messages from its input queue. In order to manage the dynamic creation of new instances we store the associations of objects as well as their input queues in a global array. Each entry in an array of a particular association is an instance number. Thus sending a message along an association requires accessing this array with the instance number of the process to obtain the target object. The message is then sent to the input queue of this object.

New objects are created as follows. There is a global variable that is one greater than the instance number of the last created instance. This number is used to set the associations of the new object (entries in the global arrays). Then the appropriate proctype is called with the instance number. Finally, this number is incremented.

We also support the if. . . else if structure of Jumbala by simulating this with the PROMELA if. . . fi structure. The constructs are different in that in Jumbala, the executed branch is the first one where the guard evaluates to true. In PROMELA, any branch whose guard evaluates to true can be chosen. However, this is easy to simulate by introducing Boolean flags that guarantee that if a particular guard evaluates to true, then all the preceding guards evaluate to false. The while statement is translated to a PROMELA do. . . od structure. This translation is straightforward.

Finally, it should be noted that the code in Fig. 5 does not accurately model scheduling of objects. Indeed, if this code were a part of a larger concurrent system, the system would have too many behaviors since SPIN could change the scheduled object in the middle of the execution of a UML transition. This omission is intentional due to lack of space and the fact that UML semantics does not define scheduling policy (see discussion at the end of Sect. 3.6). One can for instance allow an object to fire only a single transition or fire completion transitions until a stable configuration is reached. Both of these scheduling policies (and others) can be implemented using the PROMELA atomic statement.

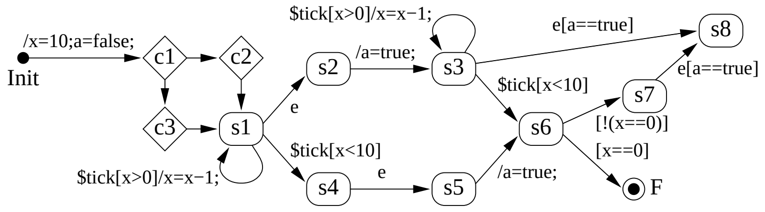


Fig. 4. The flattened version of the state machine in Fig. 1.

4.5 Flat State Machines

An interesting idea is to consider the case where UML state machines are flattened. Intuitively this means that hierarchical states are replaced with several simple states so that the behavior of the system is the same, Fig. 4 shows the flattened version of the state machine in Fig. 1. For flat state machines, we propose a translation where the state vector of SPIN is made shorter by removing the variables storing the component states. Instead, the PROMELA code has a label for each state of the flattened machine which is followed by an if . . . fi block for its outgoing transitions. If a transition is fi red, control flow is set to the transition's target state by using a goto statement to the corresponding label. Whether or not this added simplicity compensates the potential blowup in the state machine and PROMELA code sizes is a question we have yet to answer.

5 Evaluation

We have implemented our translation in a tool called PROCO [6]. Its input parameters are the UML (version 1.4) model in the XMI format supported by the Coral tool [11], the maximum number of active instances of a class, and the size of the input and deferred queues. The output is a PROMELA model.

We have tested the tool with several simple models. One of them models a protocol consisting of an environment and two protocol entities, a sender and a receiver. The environment initiates a session, after which the protocol entities shake hands. After the handshake, the protocol is running and the sender forwards data signals from the environment to the receiver. Our initial model contains a deadlock: the fi rst data signal may reach the sender before the handshaking is complete, and the data is lost. This can be fi xed by simply deferring the data signal in the state where the sender is waiting for the handshake. PROCO is able to detect the deadlock and it is possible to simulate the corresponding trace.

We have also applied PROCO to a simplifi ed model received from an industrial partner. The model portrays a client-server architecture at an abstract level, using 4 active classes and a total of 33 state machine states. PROCO and SPIN fi nd a deadlock in the model, or prove the absence of a deadlock if the model is modifi ed, in fractions of a second, regardless of whether the model is flattened or not.

[6] available at `http://www.tcs.hut.fi/SMUML/`

To better assess the scalability of the approach, we need to obtain larger models in XMI format and run experiments with them. We expect some of the big challenges to be the handling of data and polymorphism. Our tool does not currently support generalization of classes, and there are no arrays or passive classes representing data structures. Another possible issue is the efficient handling of advanced state machine concepts such as history states.

6 Conclusions

This paper outlines an approach to model check UML state machines. Although using SPIN as a back-end model checker has been tried before, our work differs from previous work in that it focuses on a UML subset for protocol models. We also support more action language features than some other previous work.

In the near future we plan to conduct case studies to evaluate our approach. We are especially interested in evaluating whether flattening of state machines can help analysis, and how PROMELA translations should be designed to increase the efficiency of partial order reductions. We also wish to investigate whether the subclassing mechanism of our action language can be used to support data abstraction.

Acknowledgements. This work has been financially supported by Tekes, Nokia, Conformiq, Mipro, the Academy of Finland, and the Emil Aaltonen Foundation.

References

1. Holzmann, G.J.: The Spin Model Checker. Addison Wesley (2004)
2. Latella, D., Majzik, I., Massink, M.: Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. Formal Aspects of Computing **11** (1999) 637–664
3. Mikk, E.: Semantics and Verification of Statecharts. PhD thesis, Christian-Albrechts-Universität (2000)
4. Porres, I.: Modeling and Analyzing Software Behavior in UML. PhD thesis, Åbo Akademi (2001)
5. Knapp, A., Merz, S.: Model checking and code generation for UML state machines and collaborations. In: 5th Workshop on Tools for System Design and Verification. Report 2002-11, Reisensburg, Germany, Institut für Informatik, Universität Augsburg (2002)
6. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using SPIN to generate tests from ASM specifications. In: Abstract State Machines. Number 2589 in LNCS, Springer-Verlag (2003)
7. Ober, I., Graf, S., Ober, I.: Validation of UML models via a mapping to communicating extended timed automata. In: 11th International SPIN Workshop on Model Checking of Software, 2004. Volume LNCS 2989. (2004)
8. Schinz, I., Toben, T., Mrugalla, C., Westphal, B.: The Rhapsody UML verification environment. In: SEFM, IEEE Computer Society (2004) 174–183
9. Telelogic: Telelogic Tau G2 v2.7 (2006) Software. `http://www.telelogic.com/`.
10. Dubrovin, J.: Jumbala — an action language for UML state machines. Research Report A101, Helsinki University of Technology, Lab. for Theoretical Computer Science (2006)
11. Alanen, M., Porres, I.: Coral: A metamodel kernel for transformation engines. In: Proc. Second European Workshop on Model Driven Architecture (MDA). Number 17-04 in Tech. Report, Computing Laboratory, Univ. of Kent (2004) 165–170

```
/* constant definitions for all states, signals etc. */
#define s_Top_Init 0
...
proctype M(int proc_id) {
      byte state_Top = s_Top_Init;
      byte state_Top_Main_R1 = s_Top_Main_R1_None;
      byte state_Top_Main_R2 = s_Top_Main_R2_None;
      byte x; bool a; /* Instance attributes of the owning class */
      byte trigger, p1, ...; /* for signal type & parameters */
      xr inputqueues[proc_id];

      evalcompletions:
      if /* Try to fire completion transitions from pseudostates */
      :: (state_Top == s_Top_Init) ->
        x = 10; a = false; state_Top = s_Top_Main_busy; state_Top_Main_R1 = s_Top_Main_R1_Init;
        state_Top_Main_R2 = s_Top_Main_R2_Init; goto evalcompletions;
      :: (state_Top_Main_R1 == s_Top_Main_R1_Init) ->
        state_Top_Main_R1 = s_Top_Main_R1_A; goto evalcompletions;
      :: (state_Top_Main_R2 == s_Top_Main_R2_Init) ->
        state_Top_Main_R2 = s_Top_Main_R2_Loop; goto evalcompletions;
      :: else -> skip;
      fi
      if /* Try to fire completion transitions from real states */
      :: (state_Top_Main_R1 == s_Top_Main_R1_B_busy) ->
        a = true; state_Top_Main_R1 = s_Top_Main_R1_Final; goto evalcompletions;
      :: (state_Top = s_Top_Main_busy && state_Top_Main_R1 == s_Top_Main_R1_final &&
        state_Top_Main_R2 == s_Top_Main_R2_final) ->
        if
        :: (x == 0) -> state_Top = s_Top_Final; state_Top_Main_R1 == s_Top_Main_R1_None;
          state_Top_Main_R2 == s_Top_Main_R2_None; goto evalcompletions;
        :: else -> state_Top = s_Top_Main; goto evalcompletions; /* Quiescing step */
        fi
      :: else -> skip; /* No completion transition was enabled */
      fi
      evaltriggers:
      if
      :: inputqueues[proc_id]?trigger,p1; /* Consume signal event (if any) */
      /* Non-deterministically create an external signal if in a state that can consume it */
      :: (state_Top_Main_R1 == s_Top_Main_R1_Loop) -> trigger = signal_$tick;
      fi
      if
      :: (state_Top == s_Top_Main_busy || state_Top == s_Top_Main) ->
        if
        :: (state_Top_Main_R1 == s_Top_Main_R1_A && trigger == signal_e && true) ->
          state_Top_Main_R1 = s_Top_Main_R1_B_busy; goto evalcompletions;
        :: (state_Top_Main_R2 == s_Top_Main_R2_Loop && trigger == signal_$tick && x > 0) ->
          x = x - 1; goto evalcompletions;
        :: (state_Top_Main_R2 == s_Top_Main_R2_Loop && trigger == signal_$tick && x < 10) ->
          state_Top_Main_R2 == s_Top_Main_R2_Final -> goto evalcompletions;
        :: else -> skip
        fi
        if /* Signal was not consumed by any substate of Main */
        :: (trigger == signal_e && a == true) -> state_Top_Main_R1 = s_Top_Main_R1_None;
          state_Top_Main_R2 = s_Top_Main_R2_None; state_Top = s_Top_Failure; goto evalcompletions;
        :: else -> skip
        fi
      :: else -> skip;
      fi
      goto evaltriggers; /* implicit consumption occurred */
}
```

Fig. 5. PROMELA code of running example.