Electronic Notes in Theoretical Computer Science 47 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume47.html pp. 1 – 13

Model Checking UML State Machines and Collaborations

Timm Schäfer and Alexander Knapp and Stephan Merz

Ludwig-Maximilians-Universität München
Institut für Informatik, Oettingenstraße 67, 80 538 München, Germany
+(49) 89 2178 2179
majnu@gmx.net, {knapp, merz}@informatik.uni-muenchen.de
http://www.pst.informatik.uni-muenchen.de/projekte/hugo/

Abstract

The Unified Modeling Language provides two complementary notations, state machines and collaborations, for the specification of dynamic system behavior. We describe a prototype tool, HUGO, that is designed to automatically verify whether the interactions expressed by a collaboration can indeed be realized by a set of state machines. We compile state machines into a PROMELA model and collaborations into sets of Büchi automata ("never claims"). The model checker SPIN is called upon to verify the model against the automata.

1 Introduction

It is general consensus that verification techniques such as model checking are best applied in the early stages of system design when the costs are relatively low and the potential benefits are high. Adoption of formal methods will be eased when they can be applied within standard development processes and when they are based on standard notation. The Unified Modeling Language (UML [8]) has become accepted as the de facto standard notation for the design of object-oriented software. Beyond diagrams that represent the static structure of a system, it also defines diagrams to model the dynamic behavior of systems. First, state machines may be associated with UML classes to specify the states an object can be in and to describe its reactions to incoming events. Second, interaction diagrams such as sequence and collaboration diagrams describe how the objects of a system may interact by exchanging events over links. These two types of diagrams represent complementary views of system behavior and are often used in different phases of system design. In this paper we describe HUGO, a prototype tool designed to automatically verify whether the interactions expressed by a collaboration diagram can indeed be realized by a set of state machines. Technically, this is achieved by compiling state machines into a PROMELA [1] model, and collaborations into sets of Büchi

automata ("never claims"). The model checker SPIN is then called upon to verify the model against the automata.

The idea to analyze UML state machines and other variants of Statecharts using model checking has been suggested before [2,3,4,5,6], although we are not aware of other tools that verify state machines against collaboration diagrams. In contrast to most other encodings of Statecharts, ours is based on a dynamic computation of Statechart behavior rather than a pre-determined, static calculation of possible state transitions in response to input events. Our approach has the advantage of being more modular, more flexible, and easier to adapt to variants of Statechart semantics, including possible changes to the semantics of UML state machines. In this way, we are able to handle more constructs present in UML state machines than are supported by most other encodings, and we expect to add the remaining constructs, except for time and change events, to a forthcoming version of the tool. While a naïve programming approach to Statechart semantics might be expected to result in a prohibitive overhead in the number of states and transitions that need to be considered during model checking, our experience has so far indicated that this overhead can largely be avoided without compromising the result of the verification.

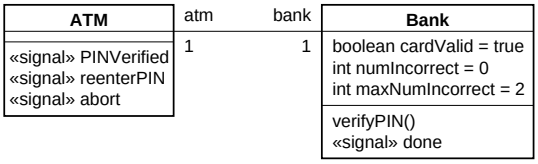
Besides model checking, HUGO also supports animation and the generation of Java code from UML state machine models, based on the same structure of implementation. Our aim is to ensure the correctness of the generated code with respect to the properties verified from the PROMELA model. The present version is still somewhat limited in this respect because guards and statements have to be expressed in the respective host languages (PROMELA or Java), and because the model checker lacks some of the constructs that are supported by code generation.

The structure of this paper is as follows: section 2 introduces the relevant UML notations. The translation of state machines to PROMELA is explained in section 3, whereas the verification of collaborations is described in section 4. Section 5 gives a more detailed comparison with related work, and section 6 concludes the paper.

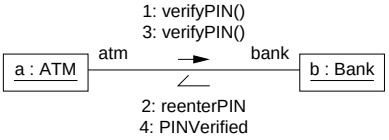
2 UML State Machines and Collaborations

We use a simple UML model of an automatic teller machine (ATM), shown in fig. 1, as our running example: The class diagram in fig. 1(a) specifies two (active) classes ATM and Bank connected by an association such that instances of ATM can refer to an instance of Bank via the role name bank, and, vice versa, instances of Bank can refer to an instance of ATM via atm. Classes define *attributes*, i.e., local variables of its instances, and *operations* and *signals* that may be invoked on instances by call or send actions, respectively.

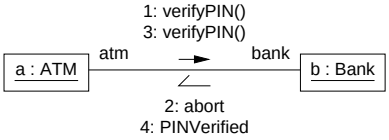
State machines for the classes ATM and Bank are shown in fig. 1(d) and 1(e), consisting of *states* and *transitions* between states. States can be *simple* (such as the states PINEntry and AmountEntry) or *composite* (states GivingMoney and Verifying); a *concurrent* composite state contains several *orthogonal regions*, separated by dashed lines. Moreover, *fork* and *join* (pseudo-)states, shown as bars, synchronize several transitions to and from orthogonal regions; *junction* (pseudo-)



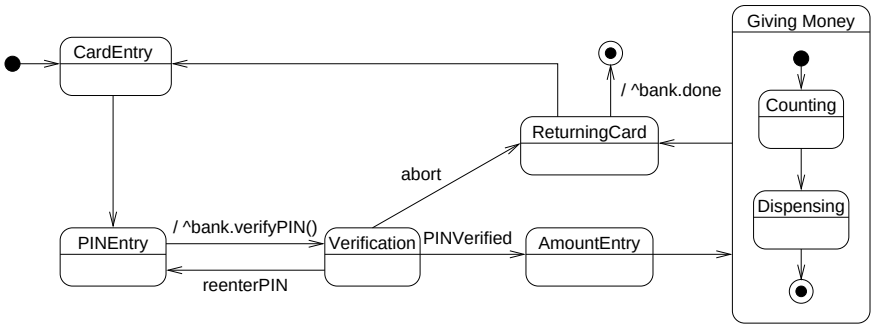
(a) Class diagram



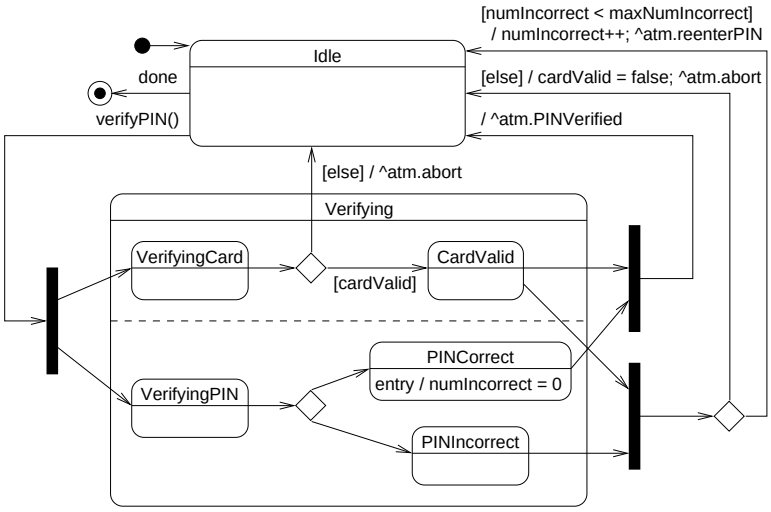
(b) Collaboration diagram 1



(c) Collaboration diagram 2



(d) State machine diagram for class ATM



(e) State machine diagram for class Bank

Fig. 1. UML model of an ATM

states, represented as lozenges, chain together multiple transitions. Finally, *history* (pseudo-)states (not contained in the state machines shown in fig. 1) record previous state information of a composite state. Transitions between states are triggered by *events*. Transitions may also be guarded by conditions and specify actions to be executed or events to be emitted when the transition is fired. (The UML does not specify a concrete syntax for guards and actions; HUGO allows PROMELA code.) For example, the transition leading from state Verification to state ReturningCard requires signal abort to be present; the transition branch from VerifyingCard to CardValid requires the guard cardValid to be true; the transition from PINEntry to Verification causes a call event verifyPIN to be sent to the instance referred to by bank. Events may also be emitted by *entry* and *exit* actions that are executed when a state is activated or deactivated, and by *(do-)activities* that are performed as long as the state is active. Transitions without an explicit trigger (e.g. the transition leaving AmountEntry), are called *completion transitions* and are triggered by *completion events* which are emitted when a state completes all its internal activities.

The actual state of a state machine is given by its *active state configuration* and by the contents of its *event queue*. The active state configuration is the tree of active states; in particular, for every concurrent composite state each of its orthogonal regions is active. The event queue holds the events that have not yet been handled by the machine. The *event dispatcher* dequeues the first event from the queue; the event is then processed in a *run-to-completion* (RTC) step. First, a maximally consistent set of enabled transitions is chosen: a transition is *enabled* if all of its source states are contained in the active state configuration, if its trigger is matched by the current event, and if its guard is true; two enabled transitions are *consistent* if they do not share a source state. For each transition in the set, its *least common ancestor* (LCA) is determined, i.e. the lowest composite state that contains all the transition's source and target states. The transition's main source state, that is the direct substate of the LCA containing the source states, is deactivated, the transition's actions are executed, and its target states are activated.

The example state machines simulate the interaction of an ATM with a single hypothetical user and a bank computer. The simulation focuses on card and PIN validation and we abstract from all other interactions by using completion transitions on most of the states, serving to sustain the progress of the simulation. After the user has entered his bank card, the ATM requests a PIN to be entered and then asks the bank to verify the entry, waiting for a reply. If both card and PIN are valid, the ATM may proceed to dispense money; if the PIN is invalid the ATM will have the user reenter the PIN; if the card is invalid the ATM will be requested to abort the transaction and return the card immediately (this ATM does not keep invalid cards). After having retrieved his card, the user may reenter the same card as many times as he wishes or end the interaction. As shown in fig. 1(e), the bank computer validates the bank card concurrently to the PIN code. If the card is not valid, the concurrent validation is exited immediately and the ATM is requested to abort the transaction. The completion transition leaving VerifyingPIN simulates any possible PIN entry by branching non-deterministically into the states PINCorrect and PIN-

Incorrect. The two join transitions evaluate the results of the concurrent validations. If an incorrect PIN has been entered and the card is valid, the counter of invalid PIN entries is incremented; however, if the counter has exceeded a maximum value, the card is invalidated and the transaction aborted. In contrast, if a correct PIN has been entered, the counter is reset to zero.

The collaboration diagrams shown in figs. 1(b) and 1(c) introduce instances a of ATM and b of Bank that are connected by a link. The collaboration in fig. 1(b) specifies the following expected interaction: in a first message the ATM a asks its corresponding bank b to verify a PIN by sending the call event verifyPIN. The bank b replies with signal reenterPIN, thus requiring the ATM to ask the user for another PIN entry. The third message, again of type verifyPIN, is finally acknowledged by the bank b via the signal PINVerified. The collaboration diagram specifies messages of type verifyPIN to be transmitted synchronously, meaning that the caller should wait for a reply from the callee before proceeding. All other messages are exchanged via asynchronous signals, i.e., without waiting for a reply. The collaboration in fig. 1(c) specifies undesirable behavior: After the card has been invalidated by the bank, as acknowledged by sending abort, no PIN entry should be valid.

3 Representation of UML State Machines in PROMELA

We now describe our encoding of a system of state machines in SPIN's input language PROMELA. The code fragments shown in the following are simplified versions of the actual translations produced by HUGO.

Every state of a state machine is modeled by an individual PROMELA process. For every state machine, two additional processes serve to dispatch events stored in the event queue and to handle transitions. Communication among the processes that are associated with a single state machine occurs via unbuffered channels, whereas the event queue is modeled as a buffered channel.

Basic definitions. For every state, we record its currently active substate and its container. A state process listens for requests to arrive on channel `chanActionRequest` and sends results along channel `chanDone`.

```
typedef ActionRequestType {
   short action;
   EventType event;
}
typedef RetvalType {
   byte val;
}
typedef StateType {
   chan  chanActionRequest = [0] of {ActionRequestType};
   chan  chanDone = [0] of {RetvalType};
   short activeSubstate;    // id of active substate
   short container;         // id of container
}
```

A state machine maintains an array of its states and two queues of pending ordinary and completion events. The length of the event queue is set to the constant `eventQueueCapacity`, which can be increased if SPIN reports a channel overflow. In case of a synchronous call event, the sending machine includes its id with the event and waits for a notification from the receiving machine along channel `chanSCDone`, indicating that the call event has been processed. The array `sm` holds the data for all state machines of the model.

```
typedef EventType {
   short id;
   short synchronousCaller; // id of calling machine (call events)
   short completedState;    // id of completed state (compl. evts)
}
typedef StateMachineType {
   StateType state[numStates];   // numStates inferred from UML model
   chan eventQueue = [eventQueueCapacity] of {EventType};
   chan eventQueueCompletion[numCompletionEvents] of {EventType};
   chan chanSCDone = [1] of {bool};
}
StateMachineType sm[numStateMachines];
```

A transition is represented as a record that contains its triggering event and its source and target states. (The compiler first decomposes any compound transitions that contain conditional branches into several simple transitions by considering all possible paths.) The actions associated with a transition are determined by the transition's id and are coded into the states and the transition handler, as explained below. Each state process maintains an array of its outgoing transitions.

Dispatching events. The event dispatcher process dequeues events and passes them on to the active state configuration, thus triggering an RTC step. If there are any pending completion events, they will be handled before ordinary events. An ordinary event is passed on to the top state of the state machine. A completion event is directly passed on to the completed state if that state is still active; another completion event may have already deactivated it. If, however, the completed state is the top state, all processes of the state machine will be terminated. The event dispatcher waits for a return value to be sent from the handling state. If the current event was a synchronous call event, the calling machine is informed of the fact that its request has been processed. At this point the RTC cycle resumes. The initialization of the state machine is similarly performed by passing the special action request `act_ensureInitialization` to the top state.

Because different state machines communicate exclusively by message passing, their internal operations cannot interfere with each other. We take advantage of this observation by having such internal operations execute atomically, guarded by a global semaphore (cf. the macros `enterMutex` and `exitMutex` in the code fragment below). The semaphore is released after the execution of an RTC step and whenever events are generated during the execution of actions. This strategy leads to a significant reduction of the number of states that SPIN needs to generate

during verification runs, without affecting the result of the analysis.

```
proctype eventDispatcher(byte smID, byte topID) {
    StateType top = sm[smID].state[topID];
    ...
    enterMutex
    actionRequest.action = act_ensureInitialization;
    top.chanActionRequest!actionRequest;
    top.chanDone?retval;
 continueProcess:
    exitMutex
    if  // wait for event to arrive, prioritize completion events
    :: nempty(sm[smID].eventQueueCompletion) ->
       sm[smID].eventQueueCompletion?currentEvent;
       enterMutex
       handlingState = sm[smID].state[currentEvent.completedState];
       if
       :: handlingState == top ->
          goto terminate
       :: handlingState != top && handlingState.isActive ->
          actionRequest.action = act_handleEvent;
          actionRequest.event = currentEvent;
          handlingState.chanActionRequest!actionRequest;
          handlingState.chanDone?retval;
       :: else ->
       fi;
    :: (empty(sm[smID].eventQueueCompletion) &&
        nempty(sm[smID].eventQueue)) ->
       sm[smID].eventQueue?currentEvent;
       enterMutex
       actionRequest.action = act_handleEvent;
       actionRequest.event = currentEvent;
       top.chanActionRequest!actionRequest;
       top.chanDone?retval;
    fi;
    if
    :: currentEvent.synchronousCaller != -1 ->
       sm[currentEvent.synchronousCaller].chanSCDone!true;
    :: else -> skip
    fi;
    goto continueProcess
 terminate: // terminate all processes of state machine
    ...
}
```

Implementation of states. States wait for action requests to arrive on the channel `chanActionRequest`. There are different types of requests, including the initialization of a state and its substates, handling of events, and activation or deactiva-

tion. After an action request has been processed, a return value is passed to the requesting process via channel `chanDone`.

Requests to handle non-completion events traverse the active state configuration as follows: A non-concurrent composite state passes the event to its active substate, giving it a chance to consume the event first. Similarly, a concurrent composite state lets its orthogonal regions handle the event one by one. If some region consumes the event and fires a transition that leaves the concurrent composite state, thus causing it to be deactivated, the remaining orthogonal regions will not handle the event, since these transitions would be in conflict. Symmetrically, if some region fires a transition that does not leave the concurrent composite state, the remaining regions will also be constrained to firing transitions that remain within the concurrent state.

If the event has not been consumed by a substate (in particular, if the current state is a simple state), the state will try to fire one of its own transitions. If some outgoing transitions are enabled, one of them is chosen non-deterministically and passed on to the transition handler. The traversal algorithm outlined above ensures that transitions from substates have higher priority than transitions originating at containing states, as stipulated by the UML semantics, and also that transitions originating from states in separate orthogonal regions have the same priority. Moreover, the algorithm computes precisely the maximally consistent sets of transitions [8, p. 2-173f.]. Note that the ordering of regions is unspecified by the UML. The PROMELA model will therefore by default non-deterministically choose some permutation of the orthogonal regions. While this non-determinism is irrelevant for many models, it carries a hefty performance penalty during verification, because SPIN has to consider all possible orderings. A run-time switch therefore allows to restrict verification to a fixed ordering of orthogonal regions.

States handle a completion event by firing one of their enabled completion transitions; we consider a completion transition with more than one source state (involving a join pseudo-state) to be enabled when all of its source states are completed.

Transition handler. The transition handler first determines the LCA L of the transition to be fired. It will then request the main source state of the transition to deactivate itself and, recursively in a bottom-up order, all of its active substates. The actions associated with the transition are executed and any emitted events are sent to the event queues of the receiving state machines. It is at this point that the global mutex mentioned above is yielded so that some other state machine may proceed. If a synchronous call action is to be executed, the sending state machine will only proceed after the receiving state machine has signaled that the call has been processed. Subsequently, the transition handler requests the transition's target states to activate themselves and their containers up to the LCA L. Finally, proper initialization of the newly activated substate of L is ensured. This is necessary, as some target states may be composite states, or may be nested in orthogonal regions, thus requiring other orthogonal regions to be initialized.

Activities. Activities are modeled as separate processes that run while their cor-

responding state is active. Although activities run concurrently to a state machine, simulating an entirely concurrent execution leads to an unnecessarily high verification complexity, forcing SPIN to analyze every possible interleaving of execution sequences. Therefore we have activities take part in the global mutex. The user is required to indicate at which points the activity should be interruptible by inserting the special macro `yieldDoActivity` in its body. The activity will be aborted when its containing state is deactivated.

Completion. A completion event is raised when a state completes all its internal activities, as follows: a simple state completes internal activity when both its entry action and its activity (if present) have terminated. A non-concurrent composite state completes when its active substate is a final state and its activity has terminated. Finally, a concurrent composite state completes when all its regions have completed and its activity has terminated.

When a final state F is activated a completion event for its container C_F is generated if C_F has any outgoing completion transitions. However, if C_F is an orthogonal region inside a concurrent composite state C_C with outgoing completion transitions, a counter indicating the number of C_C's orthogonal regions that have completed is incremented, and a completion event is generated when that counter equals the number of orthogonal regions contained in C_C. The counter will be decremented upon deactivation of C_F.

Analogously, an activity raises a completion event upon termination, provided its corresponding state has an outgoing completion transition. In particular, in our implementation, a composite state with an activity only fires a completion transition when completion events have been raised for its final states as well as for its activity.

History states. History states contain a composite state's history information. Like ordinary states, they are modeled as processes, yet they only handle action requests to update and retrieve history data. Whenever a composite state is deactivated it requests its shallow and deep history states to update their history data. When the transition handler fires a transition that targets a history state, it first retrieves its history data. If a history does not contain any history data, or only contains final states, it returns the states that are targeted by its default transition.

4 Verifying Collaborations

HUGO is mainly intended to verify whether certain specified collaborations are indeed feasible for a set of UML state machines. To do so, it generates Büchi automata that accept all executions that conform to the collaboration, and calls on SPIN to verify that no execution of the model is accepted by these "never claims". If the collaboration is possible, SPIN will produce a "counter example" that allows the successful execution to be replayed.

Generation of never claims. Simple sequential collaborations such as the one shown in fig. 1(b) could be represented by LTL formulae such as

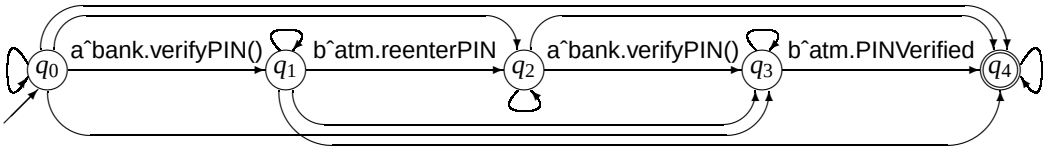


Fig. 2. Never claim for sample collaboration 1.

$$\Diamond(\text{a\^{}bank.verifyPIN} \wedge \Diamond(\text{b\^{}atm.reenterPIN} \wedge \Diamond(\text{a\^{}bank.verifyPIN} \wedge \Diamond(\text{b\^{}atm.PINVerified}))))$$

However, the algorithm built into SPIN to produce never claims from LTL formulae becomes ineffective for collaborations that specify more than about five action occurrences, and HUGO therefore generates its own never claims. The never claim generated for collaboration 1 of our running example is shown in fig. 2 where, e.g., the transition from state q_0 to q_2 is labeled by a^bank.verifyPIN() $\wedge$ b^atm.reenterPIN and self loops are labeled by true—the UML semantics allows arbitrary actions to occur in between those explicitly shown in the collaboration diagram.

Beyond simple sequential collaborations, HUGO can also verify activations due to synchronous call actions. Collaborations that include concurrent actions will be translated into a set of never claims; such a collaboration is satisfied if SPIN produces a counter example for at least one never claim in the set. However, instantiation of objects at run-time is presently not supported, and therefore the «create» stereotype defined by the UML cannot be handled by HUGO.

Verification. As outlined in section 3, HUGO first compiles the UML model into PROMELA code. Given a configuration of instances, each with its corresponding state machine, it will use SPIN to check whether the model contains any deadlocks. If the user has also specified a collaboration to be satisfied, HUGO generates never claims as described above and calls on SPIN to generate an analyzer and run the verification. If SPIN finds a way to satisfy the collaboration, it will generate a trail, and HUGO causes that trail to be executed. The operation of the state machines can be traced with the help of `printf` statements embedded into the PROMELA code, each state machine being represented in a different column. Figure 3(a) contains the first few lines from the trail that is generated for collaboration 1 of our running example, giving an example of how this interaction is possible; fig. 3(b) shows SPIN's statistics for the exhaustive search proving that the interaction specified in collaboration 2 is indeed impossible.

5 Related Work

The application of model checking techniques to variants of Statecharts has, among others, been described in [2,3,4,5,6]. Usually, the encoding is based on a static precomputation of the possible transitions between state configurations. For example, Lilius and Paltor [4] have also defined an operational semantics of UML state machines in PROMELA as a basis for vUML, also based on the SPIN model checker. Their compilation relies on a statically pre-computed table of possible transitions and conflicts. Our motivations for computing the transitions dynamically have been

```
atm:
         bank:
...
CardEntry
PINEntry
^bank.verifyPIN()
         Idle
         VerifyingPIN
         VerifyingCard
         PINIncorrect
         CardValid
         ^atm.reenterPIN
         Idle
Verification
PINEntry
^bank.verifyPIN()
...
```

(a) Trail for collaboration 1

```
State-vector 3780 byte,
depth reached 33904, ...
  456478 states, stored
  114229 states, matched
  570707 transitions
         (= stored+matched)
     478 atomic steps
...
Stats on memory usage (in Megabytes):
1730.965 equivalent memory usage ...
25.006 actual memory usage for states
       State-vector as stored =
       43 byte + 12 byte overhead
40.905 total actual memory usage
...
67.110user 0.150system 1:07.27elapsed
```

(b) Statistics for collaboration 2

Fig. 3. Results of model checking.

to arrive at a clear structure of the translation whose correctness should be justifiable by inspection, and to be able to adapt easily to changes in state machine semantics, as can be anticipated for the forthcoming version 2.0 of the UML. A programming approach to implementing operational semantics should also make it easier to support the full range of UML constructs. For example, although HUGO presently does not support choice states (as does none of the other encodings), it is obvious how they can be implemented, while they pose non-trivial problems for encodings based on static pre-computation.

Interestingly, the overhead incurred by our approach does not seem to be prohibitive. We avoid state explosion by having state machines compute their RTC steps atomically, guarded by a global mutex. Therefore, the number of states generated during verification runs increases only by a linear factor. As far as state representation is concerned, we use a rather naïve approach, but rely on SPIN's state space compression algorithm to weed out the irrelevant part of the state vector, cf. the statistics on memory consumption shown in fig. 3(b). We have compared vUML and HUGO on the dining philosophers example contained in the vUML distribution. Both tools find the deadlock in less than one second of verification time. For larger models, however, even a linear factor can make a noticeable difference. We have found that vUML compiles the model from our running example into much more compact PROMELA code, which can be used to analyse the feasibility of the collaborations in a couple of seconds, compared to well over a minute. We are currently working on optimizing HUGO's compiler. Unfortunately, PROMELA does not allow synchronous communication between processes to occur inside a `d_step`, unlike inside an `atomic` block.

The most distinctive difference between vUML and HUGO lies in their respective verification capabilities: whereas vUML is restricted to deadlock checking (collaboration diagrams are used only to define the links between objects), we focus on ensuring that the different views represented by collaborations and state machines are indeed coherent. For better usability, HUGO imports state machines from models produced by standard UML editors using NovoSoft's XMI (XML metadata interchange) parser [7], whereas vUML requires a proprietary, Python-based input format.

The format of extended hierarchical automata has been very popular to define the semantics of Statechart variants [3,5,6], and Latella, Majzik, and Massink [3] have provided a semantics for a behavioral subset of UML state machines, implemented in PROMELA. However, activities, entry and exit actions, completion events and transitions, history states, and context are missing in their paper; transition effects may only generate new events. Moreover, this approach does not directly support the verification of collaborations, as it is limited to a single state machine. We have not been able to compare their model checker with HUGO because it appears to be no longer available.

6 Discussion

The UML provides an opportunity to apply model checking technology to abstract software design models. Because the semantics of UML state machines is rather non-trivial, we hope that HUGO will find applications in the design of control-intensive and distributed object-oriented applications.

Development of HUGO is an ongoing project whose intention is to allow users to apply model checking technology "behind the scenes" to object-oriented designs. Beyond optimizing the translation, we are working to remove some of the limitations of the current prototype: events cannot be parameterized, there can only be one instance of any given class, and some of the more advanced constructs of state machines, such as synch and choice states and deferred events need to be implemented. Collaborations must currently be specified in plain textual form; a future version will be able to input collaboration diagrams from XMI files. More ambitiously, we would like to make use of SPIN's additional verification capabilities based on appropriate OCL constraints in the model.

We have noticed a few ambiguities in the semantics of UML state machines as described in [8]. For example, the semantics is vague about completion transitions with several concurrent source states; we consider such a transition to be enabled when completion events have been generated for all of its source states. Similarly, it is not clear to us what the semantics prescribes in case the guards of all completion transitions leaving a given state are false when the completion event is being handled. Our implementation consumes the completion event without firing any completion transition. If completion events could be marked as deferred events, the user would be able to indicate that the event should be reconsidered later. Moreover, the semantics does not clearly specify when a composite state with an

activity completes; we require that all final states have been reached and that the activity has terminated. Concerning synchronous call events, we let the callee send a return signal when the RTC step that has consumed the synchronous call event has been completed (the same strategy has been adopted by Lilius and Paltor [4]). It could be argued instead that the return signal should be sent explicitly by the callee, possibly after several steps of internal computation, but it is not obvious to us how to denote such behavior in a UML model.

Beyond model checking, HUGO also features Java code generation, either for animation or for direct inclusion into any Java application. It is based on the same overall structure, but the implementation is more complete and encompasses full UML state machines except for time and change events. In particular, the ambiguities that we have noticed about the semantics of UML state machines have been resolved in the same way, and users may therefore expect the Java implementation generated from a UML model to behave correctly with respect to the PROMELA translation produced from the same model. The present version of HUGO may still require the use of different models because guards and actions have to be expressed in the respective host languages. We are considering to support at least a subset of UML's object constraint logic (OCL) instead of PROMELA or Java annotations.

References

[1] Holzmann, G. J., *The SPIN Model Checker*, IEEE Trans. Softw. Eng. **23** (1997), pp. 279–295.

[2] Kwon, G., *Rewrite Rules and Operational Semantics for Model Checking UML Statecharts*, in: A. Evans, S. Kent and B. Selic, editors, *Proc. 3nd Int. Conf. UML*, Lect. Notes Comp. Sci. **1939** (2000), pp. 528–540.

[3] Latella, D., I. Majzik and M. Massink, *Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-Checker*, Formal Aspects Comp. **11** (1999), pp. 637–664.

[4] Lilius, J. and I. P. Paltor, *Formalising UML State Machines for Model Checking*, in: R. B. France and B. Rumpe, editors, *Proc. 2nd Int. Conf. UML*, Lect. Notes Comp. Sci. **1723** (1999), pp. 430–445.

[5] Mikk, E., Y. Lakhnech and M. Siegel, *Hierarchical Automata as Model for Statecharts*, in: R. K. Shyamasundar and K. Ueda, editors, *Proc. 3rd Asian Computing Science Conf.*, Lect. Notes Comp. Sci. **1345** (1997), pp. 181–196.

[6] Mikk, E., Y. Lakhnech, M. Siegel and G. J. Holzmann, *Implementing Statecharts in Promela/SPIN*, in: *Proc. IEEE Wsh. Industrial-Strength Formal Specification Techniques* (1999).

[7] *Novosoft UML API*, Available at `http://nsuml.sourceforge.net`.

[8] Object Management Group, *Unified Modeling Language Specification, Version 1.4*, Draft, OMG (2001), `http://cgi.omg.org/cgi-bin/doc?ad/01-02-14`.