Model Checking and Code Generation for UML State Machines and Collaborations

Alexander Knapp and Stephan Merz

Institut für Informatik, Ludwig-Maximilians-Universität München
{knapp,merz}@informatik.uni-muenchen.de
http://www.pst.informatik.uni-muenchen.de/projekte/hugo/

1 Supporting the UML by Formal Methods

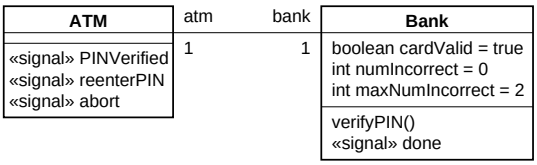
The "Unified Modeling Language" (UML [1]) is generally accepted as the de facto standard notation for the analysis and design of object-oriented software systems. It provides diagrams for the description of static, dynamic, and architectural aspects of systems at different levels of detail. In particular, dynamic aspects of system behavior can be specified with the help of interaction (i.e., collaboration or sequence) diagrams that describe single system runs. A more operational view is provided by UML state machines, a variant of the Statechart notation introduced by Harel [2], that are associated with instances of classes.

The UML deliberately encourages the use of redundant descriptions of the same aspects of a system, for example during different phases of software development. This redundancy generates an obvious opportunity for verification and validation techniques to ensure the consistency of these descriptions. Moreover, formal methods are generally most beneficial when applied to abstract descriptions. We describe an ongoing project to develop a set of tools, tentatively called HUGO, where model checking technology is applied to relate UML state machines and interaction diagrams. Considering the state machine view as the "model" and the interaction view as the "property", model checking can be used to ensure that a system run as specified by the interaction diagram can indeed be realised by a set of interacting state machines. In some cases, the absence of errors can be expressed as the impossibility to realise certain "erroneous" interactions. As is typical for applications of model checking, we concentrate on the control part of UML models and largely abstract from the data manipulations.

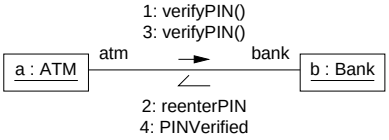
While verification technology such as model checking can reveal errors in system designs, coding errors during later implementation stages may still occur. Since state machines can specify an object's behavior in full detail, we propose to generate code directly from the UML model. Ideally, formal analysis and code generation are applied to the same model, raising the confidence in the correctness of the resulting system.

2 ATM Example

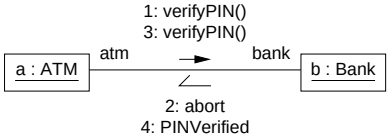
We will illustrate our approach at the hand of a simple UML model shown in Fig. 1 that describes the interaction of an automatic teller machine (ATM), a bank computer, and a single user (left implicit). The simulation focuses on the validation of the user's



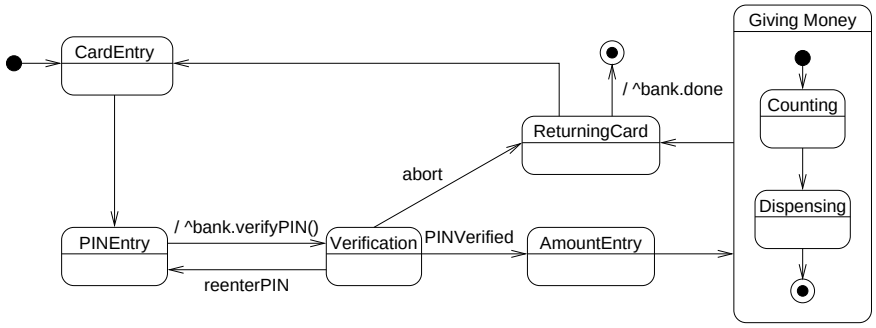
(a) Class diagram



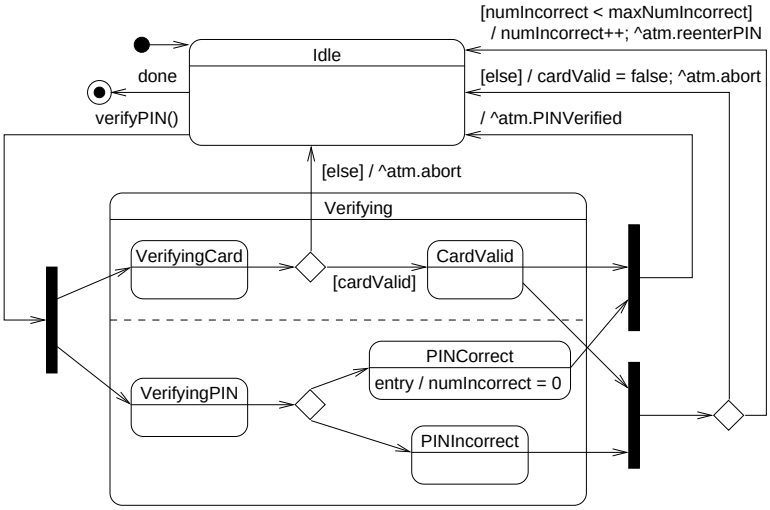
(b) Expected collaboration



(c) Erroneous collaboration



(d) State machine diagram for class ATM



(e) State machine diagram for class Bank

Fig. 1. UML model of an ATM

card and PIN, abstracting from the actual data and computation. The class diagram in Fig. 1(a) lays out the static structure of the system. The dynamic behavior is specified by state machines for the classes ATM and Bank in Fig. 1(d) and 1(e).

State machines consist of states, which may be simple (like CardEntry), composite (GivingMoney), or concurrent composite (Verifying) showing orthogonal regions that represent the parallel composition of submachines. Each state can have entry and exit actions (entry / numIncorrect = 0), which are executed when a state is activated or deactivated, as well as a (do-)activities performed as long as the state is active. Transitions between states are triggered by events (abort), show guards ([cardValid]), and may specify actions to be executed or events to be emitted when a transition is fired (^atm.abort). Completion transitions (transition leaving CardEntry), are triggered by an implicit completion event emitted when a state completes all its internal activities. Fork and join (pseudo-)states (bars) synchronize transitions entering or exiting orthogonal regions, junction (pseudo-)states (lozenges) can be used for case splits.

The actual state of a state machine is given by its active state configuration, i.e., the tree of active states where an active concurrent composite state contains one active substate per orthogonal region, plus the contents of its event queue, containing the events received but not yet dispatched. The event dispatcher dequeues the first event from the queue, which 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 containing all the transition's source and target states. The transition's main source state, i.e. 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 collaboration diagram in Fig. 1(b) specifies an expected interaction between an ATM a and a bank b: the bank reacts to an incorrect PIN (sent synchronously) by requesting another PIN to be entered (sent asynchronously), which is then acknowledged by the bank. In contrast, the collaboration in Fig. 1(c) describes an undesired behaviour: when the bank aborts a transaction, the card should have been invalidated, and no subsequent PIN entry should be valid.

3 Code generation in HUGO

The code generation component of HUGO produces Java code that behaves as prescribed by the state machines of a UML model. A generic set of Java classes (collected in the package `hugo.rt.java`) provides a standard runtime component state for UML state machines. These classes are organized along the UML meta model in order to ensure adherence to the UML semantics. For example, every state of a state machine is represented by a separate object that provides methods for activation, deactivation, initialization, and event handling. Similarly, objects representing the event queue and the event dispatcher are generated for every state machine; they implement the RTC semantics. Since the UML calls for completion events to be prioritized, there are actu-

ally two queues for completion events and ordinary events, and any completion events are dispatched first. The event dispatcher hands the event to the top state of the state machine. Transition selection is implemented by a greedy algorithm as suggested by the UML semantics: the event traverses the state hierarchy until one or more states are found that consume the event. Simple states determine whether to fire one of their transitions, whereas composite states first let their active substate(s) handle the event before trying to fire their own transitions. This policy ensures that innermost transitions are prioritized, as required by the UML. In the case of concurrent composite states, the orthogonal regions are traversed in a random permutation to ensure fair selection.

These generic classes are complemented by code that is specific to the given model. In particular, HUGO generates a Java class for each class defined in the UML model that contains method bodies for the specified operations and signal receptions, as well as a `run` method to set up and initialize the associated state machine. Besides, auxiliary classes are generated for all declared events as well as for guards and actions that appear as annotations of transitions (we assume that Java syntax is used in the UML model). A default implementation for do activities sets up a separate thread and handles the interaction with the state machine concerning (de-)activation and completion; the user should define a subclass that performs the actual task. For the ATM example about 800 lines of code in 39 classes are generated.

HUGO code generation, interpretative in nature, is not intended to produce product-quality, optimized code, but rather to represent the UML semantics as faithfully as possible. The code generator supports all features of UML state machines except for time and change events; support for time events is currently being implemented.

4 Model Checking

The main focus of the model checking component of HUGO is to verify the consistency of UML state machines against specifications expressed as collaboration or sequence diagrams. HUGO can also check for absence of deadlocks. More sophisticated verification (e.g., against properties expressed in temporal logic) is possible, but requires some knowledge of the underlying model checker and the structure of the translation.

The first implementation of the HUGO back end for model checking compiled UML state machines into PROMELA, the input language of the SPIN model checker [3], and attempted to closely follow the structure of the code generator. Instead of objects, different processes are generated for each (sub-)state of a state machine, and the transition selection algorithm relied on passing messages between these processes. Collaborations are compiled into observer automata that may synchronize on the messages exchanged between the interacting instances. The feasibility of an interaction can thus be reduced to a reachability problem for the observer automaton. A successful run produces a "counter-example" that can be displayed to the user and contains detailed information about the states entered and transitions taken by the participating state machines.

This compilation scheme still adheres closely to the UML meta model and made us confident about the validity of the analysis. However, a naïve implementation would have resulted in state spaces far beyond the reach of SPIN. Our implementation therefore made use of (informal) symmetry arguments to reduce the amount of non-determinism,

and relied on the compression algorithm implemented in SPIN to reduce the memory requirements [9]. Even with these optimizations, a UML model such as the ATM example of Fig. 1 would generate about half a million states and transitions, taking SPIN about a minute to analyze. Moreover, the size of the intermediate PROMELA and C codes could be on the order of one megabyte or more, often causing the time taken by the C compiler to dominate the running time of the model checker.

We have therefore optimized our translation such that the generic processes of the original translation are replaced by processes tailored to the given model. We have roughly followed the ideas presented by Lilius and Porres [7], although we have found their implementation of the transition selection algorithm in vUML to be incorrect. With these optimizations, the ATM example can be analyzed in less than a second, and code size is reduced to about 900 lines. The SPIN back end of HUGO handles most features of UML state machines, excepting deep history states, sync states, internal transitions, and time as well as change events.

As a third component of HUGO, we have also implemented a back end for the real-time model checker UPPAAL [5] to support the analysis of models involving real-time constraints, expressed via time events after(d) that are raised *d* time units after the source state has been entered [4]. The lack of complex data structures in the UPPAAL modeling language and the absence of memory optimizations required yet a different compilation strategy where the hierarchical structure of UML state machines is flattened out to obtain a conventional finite-state machine model. Because the UML semantics does not prescribe a particular timing model, we had to take several design decisions. Essentially, we adopt a semantics where local computation takes zero time, whereas communication between instances is time-consuming, bounded by a user-defined constant. The UPPAAL back end is the least complete of the HUGO components in terms of supported features, lacking history states, deferred events, do activities, and change events. We have used it to analyze a UML model of the "Generalized Railroad Crossing", a standard benchmark problem for timed systems, in a couple of seconds.

5 Discussion

The UML is widely used for the description of object-oriented designs and therefore provides an excellent environment for applications of formal methods to increase the quality of systems. We believe that tools that provide added value while being least intrusive and as much automated as possible will find it easier to be accepted by software developers. HUGO takes as input standard XMI files that can be produced by off-the-shelf UML editors and allows both the model and the properties to be specified in terms of UML diagrams. Moreover, it provides both analysis and code generation features in order to eliminate or at least reduce the potential for coding errors to creep in.

The idea to apply model checking technology to variants of Statecharts is certainly not new. In the context of the UML, similar projects have been reported by Latella et al. [6] and by Lilius and Porres [7]. The main problem is to what degree to formalize the operational semantics of UML state machines, which is only described informally by the UML designers [8]. Latella et al., following previous work on the formalization of Harel Statecharts, encode UML state machines in the format of hierarchical automata,

which are then compiled to SPIN. Unfortunately, they do not support several important features of UML state machines such as do activities, entry and exit actions, completion events and transitions, history, and choice states, and it appears non-trivial to add some of these features in their framework. Lilius and Porres instead employ a custom format to describe the semantics of UML state machines, their vUML tool, however, shows several deviations from the semantics, in particular, as regards completion events and transition selection. In any case, the translations to such formats are non-trivial, and it is not obvious how to adapt them to changes in the UML semantics that can be anticipated for the forthcoming version 2.0 of the UML. We have therefore tried to ensure the correctness of our compilers by close adherence to the UML meta model, which should also make our compilers relatively straightforward to adapt to changes. Nevertheless, we are now investigating the use of a common representation that would allow better integration of the different components of HUGO and could be used to justify different optimizations performed by the back ends.

Development of HUGO is an ongoing project, and we intend to support some more advanced features of the UML. In particular, it is not always possible or convenient to express properties as collaborations without referring to the active state configuration of the state machines. We intend to support a limited set of OCL constraints when compiling the UML model. Moreover, example runs are now only presented in textual format (for the SPIN back end) or in the built-in simulation environment (for the UPPAAL back end). It would be desirable to present these again at the level of the UML model.

Acknowledgements. Timm Schäfer implemented the first version of HUGO, Christopher Rauh the back end for UPPAAL. Simon Bäumler optimized the SPIN back end.

References

1. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison–Wesley, Reading, Mass., &c., 1998.
2. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comp. Program.*, 8(3):231–274, 1987.
3. G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 5(4), 1997.
4. A. Knapp, S. Merz, and C. Rauh. Model Checking Timed UML State Machines and Collaborations. In *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002. To appear.
5. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. J. Softw. Tools for Techn. Transfer*, 1(1–2):134–152, 1997.
6. D. Latella, I. Majzik, and M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-Checker. *Formal Aspects Comp.*, 11(6):637–664, 1999.
7. J. Lilius and I. P. Paltor. Formalising UML State Machines for Model Checking. In R. B. France and B. Rumpe, editors, *Proc. 2nd Int. Conf. UML*, volume 1723 of *Lect. Notes Comp. Sci.*, pages 430–445. Springer, Berlin, 1999.
8. Object Management Group. Unified Modeling Language Specification, Version 1.4. Specification, OMG, 2001. `http://cgi.omg.org/cgi-bin/doc?formal/01-09-67`.
9. T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. In S. Stoller and W. Visser, editors, *Proc. Wsh. Software Model Checking*, volume 55(3) of *Elect. Notes Theo. Comp. Sci.*, Paris, 2001. 13 pages.