Transformation of UML State Machines for Direct Execution

Tim Schattkowsky, Wolfgang Müller
C-LAB, Paderborn University,
{tim,wolfgang}@c-lab.de

Abstract

Executable UML models are nowadays gaining interest in embedded systems design. This domain is strongly devoted to the modeling of reactive behavior using StateChart variants. In this context, the direct execution of UML State Machines is an interesting alternative to native code generation approaches since it significantly increases portability. However, fully featured UML 2.0 State Machines may contain a broad set of features with complex execution semantics that differ significantly from other StateChart variants. This makes their direct execution complex and inefficient. In this paper, we demonstrate how such State Machines can be represented using a small subset of the UML State Machine features that enables efficient execution. We describe the necessary model transformations in terms of graph transformations and discuss the underlying semantics and implications for execution.

1. Introduction

Some years ago, UML tools where mainly applied in early phases of the software development process like functional specification and for final documentation. Over the years, UML tools like Rose, Together, and Enterprise Architect became more advanced for a more complete coverage of the software design process from requirements over functional specification to implementation. Nevertheless, most of the general software design environments still do not provide sufficient means for code generation fully retargetable to different hardware platforms, programming languages, and operating systems. Most tools support just a small subset of the UML for generating templates, which still have to be completed, e.g., by coding the bodies of methods.

However, there already exist some application specific integrated solutions like XITUML and Telelogic Tau which simulate models or generate working code up to complete applications from these models. Nevertheless, they still distinguish between model simulation and code generation for the individually selected micro controller platform.

In [16], we have introduced our concept of a UML subset for execution on a UML Virtual Machine (UVM) which provides a generic and portable runtime platform for the direct execution of our UML subset [15][17]. In this paper, we describe the transformation of UML state machines applied in our approach to enable efficient execution through the virtual machine.

The remainder of this paper is structured as follows. The next section discusses related works. Section 3 introduces our basic concept of executable UML models with a focus on the use of state machines in the design and their equivalent executable counterparts, the Executable State Machines (ESMs). The underlying transformation is described in section 4. Section 5 gives a short illustrating example before section 6 finally closes with a conclusion.

2. Related Work

The transformation of StateChart variants has been a subject to research for some time. However, a mostly complete coverage of UML State Machines with its specific semantics seems to be missing. Existing work often only deals with small subsets and covers variants with significantly different execution semantics.

Harel and Naamad originally defined the operational semantics of StateCharts [7]. Several variants of StateCharts with different semantics where introduced thereafter. Excellent overviews are given by Levi [10] and von der Beeck [19], who also did significant work in formalizing UML statechart semantics [20]. Additionally, Damm et al. [2], for instance, introduced the compositional semantics of a larger statechart subset based on synchronous transition systems.

However, the dynamic semantics of UML state diagrams differs from the classical StateCharts semantics. Many approaches have been published on formalizations of subsets of UML State Diagrams applying various formal means over the last recent years. Examples are definitions of an abstract interpreter by [11] and by the means of ASMs by [1]. David and Möller [3] give a translation of restricted UML State Diagrams to flat UPPAAL timed automata. Metamodelling based approaches are given by Kleppe
and Warmer [8] in the context of the pUML (preciseUML) approach and by Engels et al. through the means of graph grammars [5]. In the latter approach, collaboration diagrams are interpreted as graph transformations and give operational semantics for a small subset of state machines. Varro applied graph grammars to formally define the code generation from a small UML state diagram subset for model checking [18]. Kuske [9] and Gogolla [6] also applied similar graph grammar for defining the formal semantics of certain parts of UML state diagrams. However, all of the latter approaches cover just small UML 1.x subsets. Other approaches cover the dynamic semantics by a mapping to formal means rather than define the mapping by formal means.

Existing approaches on executable UML like xUML [1] and X_UML [11] are currently based on the UML 1.x specification and rely on code generation for traditional programming languages like C++ or Ada.

3. State Machines in Our Executable UML Approach

Our approach on executable UML models is based on using a fully featured UML state machine to describe the behavior of an operation. The direct execution of such state machines used in our approach would be still inefficient, as examining the state hierarchy and orthogonal regions is time-consuming and requires complex execution logic, especially in hardware UVM implementations. Thus, we decided to transform the state machines into equivalent simpler state machines, which are more suitable for efficient execution. These are the Executable State Machines (ESMs) that are processed directly by our UVM.

ESMs are a subset of UML state machines. The model for these ESMs is shown in Fig. 1. The major changes are the removal of composite states and the limitation to only the doActivity. After removing the hierarchy, the entry and exit activities become meaningless anyway. A model compiler can easily resolve all supported Pseudostates. The first state in the compiled model is inherently the initial state. The final state is detected by the UVM because it has no outgoing transitions. However, it is not explicitly represented in the model.

However, we found it necessary to change the semantics of the doActivity for our application. The UML defines that activity as some activity starting after the state entry and stopping immediately when a transition is triggered. Thus, it must be possible to abort the doActivity at any point without side effects. This is unrealistic for software designs. Therefore, we assume that the doActivity is always completed before a State can be left. This usually meets the designer’s assumption and enables the transformation of concurrent states as described later in this paper.

The transformation of UML state machines into ESMs is nontrivial, as the behavior and semantics of the original model have to be preserved. The next section will describe this transformation in terms of graph transformation rules also defining the behavioral semantics of the UML state machines in our approach in the context of our UML Virtual Machine.

4. Model Transformation

UML state machines can be seen as instances of the metamodel for UML State Machines. Thus, we can represent such state machines as a graph with objects and links as vertices and edges. We can also do this for our Executable State Machines (ESMs), since they also are a subset of UML state machines. Thus, we can use graph transformation [4] rules to formally describe the desired transformation from UML state machines to ESMs. We introduce these rules in a way that allows their application also for a less restricted state machine model by applying only a subset of these rules.

We specify the transformation rules by giving pre- and postcondition for inplace replacement in the model graph. The precondition consists of a positive and an optional negative part. The positive part (left-hand side, ‘L’) defines a subgraph that must exist in the model graph and is subject to modification by the rule. The optional negative part (‘N’) must not exist to allow application of the rule. For the precondition, all depicted model elements must be matched. This includes not only the nodes and edges, but also their labels including identifiers, roles etc. Multiplicities on links are employed to match several different instances.
of an association. If more than one instance may exist, this is depicted using a multiobject. Including a lower bound of 0 depicts optional parts. If a link or object must not exist, it is stroked out. Variables in brackets are used to bind identifiers. The negative part of the precondition may include references to such bound variables from the positive precondition.

Finally, the postcondition (right hand side, ‘R’) is given to specify the actual transformation to be applied to the L subgraph. Elements existing in both graphs are maintained. Elements that only exist on the R subgraph are added to the graph. Elements that only exist on the L side and dangling edges (links) are removed.

We give no particular application order for the transformation rules based. Thus, the design of the rules themselves has to ensure that the application of the rule set terminates when no more rules can be applied. In our application scenario we ensure this by removing a significant part of the precondition subgraph in each rule and avoiding circular dependencies in between rules for correct (w.r.t. our UML state machine model) input graphs. However, malformed input could cause such a situation.

![Fig. 2: Transforming the Initial Substate](image)

Ensuring the semantic correctness of all rules is quite complex. Proving its correctness is even harder, as it would involve the creation of complete formal specifications of the UML StateMachine behavior and the transformations described herein. This finally raises the issue of correctness again. Thus, we will focus here on reasoning about the rules itself and discuss why their application is supposed to yield correct results.

![Fig. 3: Move outgoing Superstate Transitions to Substates](image)
The observable behavior of the ESM being the transformation result should equal the behavior of the original UML state machine. The consumption of events and the ordering of the triggered activities determine this behavior, which are both the essential properties we have to check for each of the transformation rules. It is important to notice that these properties may be temporarily harmed by a transformation rule, as long as it is ensured that there exists another rule fixing the issue that will be applied later in all possible scenarios.

The transformation from UML state machines to ESMs has to compensate the limitations of ESMs. Thus, the essential goals of this transformation are eliminating composite states, removing entry/exit activities from States, and effects and guards from transitions. Although we do not support deferred triggers in our approach, this transformation can preserve these triggers. Thus, we left the deferred triggers in here to extend the applicability of our transformation approach in other scenarios.

We assume that the initial model is valid and contains complete sets of initial and final states for all levels. Composite states must contain at least one substate.

Our strategy for eliminating composite states is to replace all transitions on the composite State with complete sets of initial and final states for at all levels. Although we do not support deferred activities from States, and effects and guards from transitions. Although we do not support deferred triggers in here to extend the applicability of our transformation approach in other scenarios. Thus, the essential goals of this transformation are eliminating composite states, removing entry/exit transitions. Although we do not support deferred triggers in here to extend the applicability of our transformation approach in other scenarios.

We assume that the initial model is valid and contains complete sets of initial and final states for at all levels. Composite states must contain at least one substate. Our strategy for eliminating composite states is to replace all transitions on the composite State with complete sets of initial and final states for at all levels. Once the transitions are replaced, we move the substates from the lowest level one level up and push the according entry/exit behavior into the transition effect. Exhaustive application of the respective rules eliminates the hierarchy in the state machine.

To remove incoming transitions on a composite State, we replace those with direct transitions to the state marked by the initial Pseudostate and remove that Pseudostate (see Fig. 2). Since it is the only Pseudostate in our approach, this also ensures that the region afterwards only contains states and final states.

Fig. 4: Resolve conflicting Triggers and integrate Triggers from Superstate
superstate transition guard with the negated disjunction of all guards from substate transitions having the conflicting trigger (Rules 4&5). These rules do not account for conflicting completion transitions. Those can be processed using similar rules.

Once no transitions to and from a composite State exist, we are sure that all the substates are properly connected to the world using direct (possibly inter-level) transitions replicating the composite State transitions. At this point, the State is ready to be propagated one level up in the hierarchy. However, the composite state still affects the entry/exit behavior of these transitions. Thus, the rule for moving the state one level up (not shown here) has to account for these activities. As the rule requires a bottom-level State, all inter-level transitions are inherently leading to a higher level and involve the according entry and exit for this state depending on their orientation. We create a linked list for the entry and exit activities containing the activities in their hierarchical order for later processing. Each application of the rule adds new activities accordingly. Transitions at the same level are marked as InLevelTrans. This avoids false detection as an inter-level transition after moving the State up. The mark is removed when both source and target State approach at the same level. The according rule matches for InLevelTrans linked to two same-level States.

The rules in Fig. 3 transfer all transitions from a superstate to the contained states. However, before these rules are applied, all outgoing transitions of each state are marked for copy and each final state is replaced by a normal state and has all completion transitions (these may have different guards) from the superstate marked to be copied to that state. It is important to notice, that at this point all Vertices in the Region are exactly States. The second rule in Fig. 3 replicates such a marked transition for a state and marks it as copied. The third rule deletes transitions that have been copied to all relevant substates. Again, this requires a different rule for completion transitions.

The copy rule marks all copied transitions from the superstate with the ‘parent’ role, as they need further treatment. We have to resolve conflicting triggers between these copied transitions and transitions on the State. As this has to be done per trigger, it is useful to enforce transitions with at most a single trigger. The respective rules (not presented here) do that by creating a per trigger copy of each transition placing the ‘single’ role to indicate a superstate transition with a single trigger. The original transition is deleted once it is copied it for all triggers. Only the rule for triggered transitions is shown here. A similar rule exists for completion transitions.

The rules in Fig. 4 resolve conflicting triggers one at a time. The first rule deletes the superstate transition if the trigger is deferred in the substate, as the substate has precedence. The second rule allows a superstate transition that is non-conflicting to become valid for the substate. In case of a conflict, a non-guarded substate transition is consuming the trigger. Thus, rule three deletes the conflicting superstate transition. However, if a guarded substate transition is in conflict, it depends on the actual result of the guard evaluation, which transition has to be taken. Thus, we have to create a new guard for the conflicting superstate transitions that only evaluates to true when none of the conflicting guard conditions evaluates to true. Thus, we construct such a guard as the conjunction of the

![Fig. 5: Enqueue State Activities in Effect](image)

![Fig. 6: Move Exit Activity to Transition Effect](image)
The exit activity. Once all activities are copied, the original composition is deleted.

This enables uniform processing of these activities through rules like the one in Fig. 6. That rule appends the first exit activity from the list to the effect. A similar rule is used for entry activities (but not shown here). The rule for entry activities only additionally checks that no more exit activities are in the list and appends the first entry activity from the list. Furthermore, there exists another pair of rules like these dealing with transitions that have no effect. These rules are equal to the ones shown, but have no need to create a new activity. The entry/exit activity is directly assigned as transition effect there. However, there may have been ActivityReferences appended to the list that point to no activity (e.g., because the State had no entry activity). These can be eliminated by a simple rule.

After the application of the rules discussed so far, we yield a state machine where all hierarchical States have been pushed up to the highest level. But there are still the remains of the composite states. As we removed all incoming transitions to those States, they can be easily removed using a rule eliminating all States with no incoming transitions. This results in a flat state machine where all Activities from composite States are encoded into Transition effects.

Once all hierarchy is removed, a rule is employed to merge two regions from the orthogonal state into one. The respective rules are only described here as they are quite large. However, the methods applied have been already shown here. The rules essentially delete one of the two regions and create a cross-product of the states from both regions while fixing the Activities and Transitions accordingly. This is done by first transforming all the simple States temporarily into composite states holding a complete copy of the state machine from the removed region. These are later again flattened leading to the cross-product. Conflicting orthogonal transitions are resolved by creating up to three versions of these transitions from a cross-product state, one for each possible scenario where one or both original states use the trigger. The guards of these transitions are extended accordingly. All State activities are encoded directly into the transitions as we did to resolve the hierarchy. As ESMs do not support transition effects, we have to move these effects from the transitions into States. Thus, we have to encode the effect into the target State of each transition. As that target state is likely to have several incoming transitions, it is necessary to replicate the State for each incoming transition. An additional state will be introduced to apply these rules to transitions to a final state. The rule in Fig. 7 is for that purpose. A copy of the target State including deferred triggers and outgoing transitions is made using the copy mechanism.
for transitions. A new doActivity is created from the transition effect and the State doActivity. Again, if the state has none, a similar simpler rule can be used, but has been omitted here. Furthermore, we left out the part of the rule detecting that the new State has been already created.

As those rules remove the transition from the original target State, that State may later be removed by the rule detecting States with no incoming transitions in cases where all transitions moved to separate copies of the State. Note that the entry and exit activities are already removed at this point.

After exhaustive application of the rules considered so far, we yield a flat state machine where States have only a doActivity and transitions have no effect. However, a State still may have deferred triggers.

We finally have to eliminate the guards by replacing all outgoing transitions using the same trigger with a single transition to a new State evaluating the respective guards in the doActivity. This doActivity then may directly trigger the according transition. If none of the guards evaluates to true, we go directly to a new State embedding the same set of outgoing none of the guards evaluates to true, we go directly to a new State embedding the same set of outgoing transitions.

As those rules remove the transition from the original target State, that State may later be removed by the rule detecting States with no incoming transitions in cases where all transitions moved to separate copies of the State. Note that the entry and exit activities are already removed at this point.

After exhaustive application of the rules considered so far, we yield a flat state machine where States have only a doActivity and transitions have no effect. However, a State still may have deferred triggers.

We finally have to eliminate the guards by replacing all outgoing transitions using the same trigger with a single transition to a new State evaluating the respective guards in the doActivity. This doActivity then may directly trigger the according transition. If none of the guards evaluates to true, we go directly to a new State embedding the same set of outgoing transitions.

However, a State still may have deferred triggers.

We finally have to eliminate the guards by replacing all outgoing transitions using the same trigger with a single transition to a new State evaluating the respective guards in the doActivity. This doActivity then may directly trigger the according transition. If none of the guards evaluates to true, we go directly to a new State embedding the same set of outgoing transitions.

After the exhaustive application of all the presented rules, a fully featured UML state machine with all kinds of activities, composite and orthogonal states is transformed into an ESM having at most a single Activity per State and just a trigger per transition. This ESM is equivalent to the UML State Machine with respect to the described semantics.

## 5. Example

In this section, we will outline the transformation of an UML state machine into an ESM to illustrate the transformation approach. Fig. 8 shows the original state machine and three snapshots during the transformation process. The first snapshot shows the state machine after removing the initial and final state and all the transitions from the composite state. The next snapshot shows the state machine after the removal of the composite state where the entry and exit activities have been moved to the transition effects for all inter-level transitions from the composite state. Furthermore, trigger conflicts have been resolved resulting in the removal of the composite state transition from C to A and the extended guard condition of the transition from D to A. The last snapshot shows the state machine after moving the transitions effects into states by introducing new states for every incoming transition.
Finally, the guards have to be eliminated. Fig. 9 shows a cutout of the resulting final ESM showing how guard conditions are evaluated using additional states. It is important to note that the figure is not very precise, but more intuitive than an object diagram depicting the true situation. The decisions in that figure are actually states in the ESM containing bytecode for activities that evaluate the guards and directly trigger the according transitions.

6. Conclusion

We introduced the non-trivial translation of our previously introduced executable UML subset to executable state machines by formal graph transformations. The graph transformations provide a well-defined semantics for UML state machines by the translation to well defined non-hierarchical state machines, which are equivalent to annotated finite state machines. The graph transformations thus define a formal behavioral semantics for UML state machines.

We have applied that approach to formally cover our concepts of directly executable UML in the context of our previously introduced UML Virtual Machines. In that context, graph transformation provides us with additional means to (i) reason about the code generation to executable state machines and to (ii) provide a complete abstract specification for the implementation of our code generator. The given graph transformations are implemented in the model compiler we use to create binary executable models for our UVM. These models can be executed using either a software UVM implementation [15] or a hardware implementation in an FPGA [17].

References