Towards Verification of Model Transformations Via Goal-Directed Certification

Gabor Karsai and Anantha Narayanan

Institute for Software Integrated Systems,
Vanderbilt University, P.O. Box 1829 Sta. B.
Nashville, TN 37235, USA
gabor.karsai@vanderbilt.edu, ananth@isis.vanderbilt.edu

Abstract. Embedded software is widely used in automotive applications, often in critical situations where reliability of the system is extremely important. Such systems often use model based development approaches. Model transformation is an important step in such scenarios. This includes generating code from models, transforming design models into analysis models, or transforming a model between variants of a formalism (such as variants of Statecharts). It becomes important to verify that the transformation was correct, and the transformed model or code preserved the semantics of the design model. In this paper, we will look at a technique called "goal-directed certification" that provides a pragmatic solution to the verification problem. We will see how we can use concepts of bisimulation to verify whether a certain transformation instance preserved certain properties. We will then extend this idea using weak bisimulation and semantic anchoring, to a more general class of transformations.

Keywords: Behavior Preservation, Bisimulation, Weak Bisimulation, Semantic Anchoring.

1 Introduction

Model-driven development of embedded systems relies on the use of model transformations that translate and establish linkage between different modeling formalisms, design artifacts produced during the development process, and possibly executable code. The validity of these transformations is crucial for the correct functioning of the system. To prove that the system will work as predicted, we must be able to assure that the model transformations preserved the semantics of the models. For instance, the control logic of an automotive application (say, an Anti-lock Braking System) may be developed using a model-based approach (say, using Stateflow). We may transform this model to verify its control logic (say, using NuSMV), or generate code from it (say, using Mathworks' Real-time Workshop). In a high-consequence application (like the ABS) it is essential that the results of the verification hold true for the generated code. The question at the heart of any model-based development process is: for applications where

M. Broy, I.H. Krüger, and M. Meisinger (Eds.): ASWSD 2006, LNCS 4922, pp. 67–83, 2008.

safety is essential, do the transformations on the models provide verifiable assurances for the preservation of properties across the transformation?

Observe that the problem is similar to the verification of compilers for high-level languages. Compiler verification, in general, is currently not solved: we don't have a full formal "proof" of a production-quality compiler. The pragmatic approach is to use a "certification process" which validates the behavior of a compiler using a large set of test examples and observing the results of the compilation (often through execution). Unfortunately, this approach may not be feasible for model transformations, as model transformations often do not produce executable code. It may not be economical to develop a large set of test examples for certifying a highly specialized transformation tool, and the correctness of the transformations' results is difficult to establish only via reading.

2 Background

2.1 Goal-Directed Certification

If the requirement for total verification can be relaxed, some verification problems could be solved using techniques from program synthesis tools. NASA ARC [8] has recently developed an approach to generating assurances for code produced by automatic program synthesis tools. Their approach is based on

1. adding annotations to the generated code that capture certain pre-conditions and post-conditions on each statement,
2. capturing safety properties that need to be verified in some logic notation,
3. translating the conditions from (1) and the safety properties from (2) into a set of verification conditions that are simplified,
4. using a symbolic (and automatic) theorem prover to generate a formal proof that the selected safety conditions hold for the generated code, and
5. using a proof checker to check that the proof is valid.

The result of this process is a "certificate": a formal proof that the generated code does satisfy the desired safety properties.

This approach does not solve the verification problem in general. It answers the question for a specific generated code, and for a selected set of safety properties. This simplifies the process greatly, and makes it technically feasible. We will call this approach "goal-directed certification".

Note that this approach removes the need for *trust* in the generator, as the certificate is evaluated after the generation. This means that the certification will capture any errors in the generated code, irrespective of faults in the generator.

The work of Denney and Fischer in [8] is similar to the Proof-Carrying Code (PCC) work of Necula [16]. In PCC, a compiler is extended to produce object code accompanied with proofs for safety policies that can be independently verified on a host system. In Certifiable Program Generation, the idea is extended to code generators, to provide assurances about the generated source code. Since they operate at the level of source code (as opposed to object code), the formulation of the safety properties is changed appropriately. In this paper, we

extend this idea to model transformations, where we wish to provide assurances about the generated models. Consequently, the formulation of the safety policies and the verification methods will be different, but the basic architecture is comparable.

2.2 GReAT

GReAT [5] is a language for specifying model transformations graphically using elements of the meta-models of the constituent domains. The meta-models of the domains are specified using UML and OCL. GReAT belongs to the class of practical graph transformation systems such as AGG[9], PROGRES[17] and FUJABA[18].

One of the features of GReAT is the ability to link elements from the different meta-models, to create temporary cross-domain links. These links are called *cross-links*, and can be used to trace relations between model elements belonging to different domains during the course of the transformation.

2.3 Extended Hierarchical Automata

To demonstrate our idea of goal-directed certification, we will use Statecharts as the design language, and Extended Hybrid Automata (EHA) [14] as the analysis language. EHA has been chosen as the analysis language as it has a straightforward mapping into the PROMELA language used in SPIN[15]. EHA were used to give formal operational semantics for Statecharts, and they offer a simple hierarchical representation for Statecharts that was used in correctness proofs [19].

EHA models are composed of one or more *Sequential Automata*, which are non-hierarchical finite automata. The states of a Sequential Automaton (called *Basic States*) may be *refined* into further Sequential Automata, to express hierarchy in a flat notation. A Statechart model can be represented by a Sequential Automaton, with a finite automaton representing the top level states of the Statechart. Compound states in the Statechart must be represented as individual Sequential Automata, and marked as *refinements* of the corresponding Basic States in the EHA. The entire Statechart can be represented this way, using a set of Sequential Automata and a series of refinements.

Some transitions in the Statechart may cut across levels of hierarchy. Such transitions are said to be inter-level. Transitions in an EHA model, however, are always contained within one Sequential Automaton, and cannot cut across levels of hierarchy. Inter-level transitions my therefore be elevated based on the scope of the transition. An inter-level transition is placed in the Sequential Automaton corresponding to the Statechart state containing it, and is drawn between the Basic States corresponding to the top-most ancestors of the source and target states in the Statechart. The transition in the EHA is also annotated with special attributes called *source restriction* and *target determinator*, which keep track of the actual source and target states of the transition. Figure 1 shows the meta-model for EHA. The complete transformation from Statechart to EHA will be explained later.

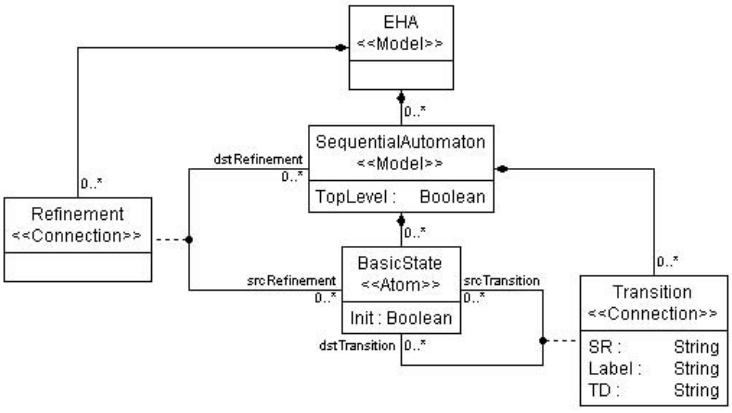


Fig. 1. EHA meta-model in UML

2.4 Bisimulation

Bisimulation is an equivalence relation between Labeled Transition Systems (LTS), which can conclude whether the two systems will behave identically. In other words, if two systems have a bisimulation relation, then one system simulates the other and vice versa. Given an LTS (S, Λ, $\rightarrow$), a relation R over S is a *bisimulation* if:

$$(p,\, q) \in R \text{ and } p \xrightarrow{\alpha} p' \text{ implies that there exists a } q' \in \mathrm{S},$$
$$\text{such that } q \xrightarrow{\alpha} q' \text{ and } (p',\, q') \in R,$$

and conversely,

$$q \xrightarrow{\alpha} q' \text{ implies that there exists a } p' \in \mathrm{S},$$
$$\text{such that } p \xrightarrow{\alpha} p' \text{ and } (p',\, q') \in R.$$

If we considered the union two transition systems representing a Statechart model and an EHA model, and find a relation R relating each Statechart state to an EHA state, and proved that R is a bisimulation, we can conclude that the Statechart model and the EHA model will behave identically.

3 Verifying Model Transformations by Goal-Directed Certification

In goal-directed certification, we do not wish to provide a general correctness proof for the transformation. We try to solve the more tractable and useful question of trying to prove that a particular instance of a transformation preserved certain properties of interest for the instance models involved. Suppose we have a design modeling language that has convenient features for representing complex controller behaviors and designs, and we have a simpler analysis language that comes with sophisticated verification tools. In our case study, we

will consider Stateflow [1] as our design language, and PROMELA (of the SPIN model checker [12]) as be the analysis language. We use the design language for expressing controller designs, and then translate the design models into the analysis language where the actual verification is done. We then try to answer the question, do the results of the verification on the PROMELA instance model hold on the Stateflow instance model? In other words, how can we be sure that the model transformation that maps design models into analysis models preserves the semantics of those models?

We wish to note that in some simple cases, we may be able to specify a transformation using a sequence of declaratively specified steps, and provide an argument for its correctness by construction. However, we do not yet have a model transformation tool that can take a purely declarative specification and produce an automated transformation. In a production scenario, it is usually not feasible to provide such assurances by construction, due to implementation complexity issues. We propose an automatable and reusable method to verify the correctness of the implementations of model transformations, which can be integrated into the transformation, but will perform the verification independent of the transformation itself. Thus, an incorrect implementation will not produce a valid certificate of correctness.

The above questions are difficult to answer in general, but we can possibly answer them if we restrict ourselves to single instances and specific properties. One such model property is reachability: what states are reachable/unreachable in the design model? We would like to answer this by translating the design model into an analysis model, executing the reachability analysis on the analysis model, and deducing that the reachability holds for the original design model given the model transformation is correct.

Reachability is checked by a model checker via state-space exploration. Thus, if we can somehow show that the state-space of the design model has an isomorphic mapping into the state-space of the analysis model, then the reachability properties checked on the analysis model have the same logical truth-value for some equivalent reachability properties in the design model. We can use our definition of bisimulation here, to find the relation R between the elements of the two instance models, and check if R is a bisimulation. If R is a bisimulation, we can conclude that the two models behave identically, when it comes to the property of reachability.

Figure 2 shows the basic architecture for this evaluation. The model transformation generates the target model from the source model, and the target model is verified by the model checker (SPIN). As described earlier, our model transformation language (GReAT) allows us to link source and target elements using *cross-links*. We will use these cross-links to trace the relation R which links states in the source (Stateflow) model to the corresponding items in the target (EHA) model. A straightforward *bisimilarity checker* is used to trace these relations and find whether R is a bisimulation. If the bisimilarity checker determines R to be a bisimulation, we can conclude that the results of the model checker for the analysis (EHA) model will be valid on the design (Stateflow) model.

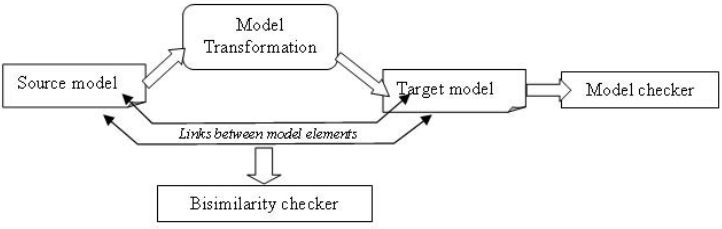


Fig. 2. Architecture for verifying reachability preservation in a transformation

Please note that there exist solutions that can perform reachability analysis on Stateflow/Simulink models, such as the OSC Embedded *Validator* [2] and TNI's Safety-Checker Blockset [4] that are available in the industry. Our aim is not to provide such a solution. We also do not wish to provide a method for defining the semantics of these languages. We simply wish to consider a simple model transformation as a case study to demonstrate our methodology.

The following subsections will describe in detail the transformation, construction of the cross-links and checking for bisimilarity.

3.1 Transforming Statechart Models to EHA

The model transformation was built using GReAT and GME [13]. We first defined meta-models in GME for the two languages, Statechart and EHA. The transformation uses the following steps (which are automatically executed by the GReAT transformation engine):

1. Every Statechart model is transformed into an EHA model, with one top level Sequential Automaton in the EHA model.
2. For every (primitive or compound) state in the Statechart (except for regions of concurrent states), a corresponding Basic State is created in the EHA.
3. For every composite state in the Statechart model, a Sequential Automaton is created in the EHA model, and a "refinement" link is added that connects the Basic State in the EHA corresponding to the state in the Statechart, to the Sequential Automaton in the EHA that it is refined to.
4. All the contained states in the composite state are further transformed by repeating steps (1) and (2). The top level states in the Statechart are added to the top level Sequential Automaton in the EHA.
5. For every non-interlevel transition in the Statechart model a transition is created in the EHA between the Basic States corresponding to the start and end states of the transition in the Statechart model.
6. For every inter-level transition in the Statechart model, we trace the scope of the transition to find the lowest parent state s_P that contains both the source and the target of the transition. A transition is created in the EHA, in the Sequential Automaton corresponding to s_P. The source of the transition in the EHA is the Basic State corresponding to the highest parent of the source in the Statechart that is within s_P, and the target in the EHA is the Basic

State corresponding to the highest parent of the target in the Statechart that is within s_P. The transition in the EHA is further annotated, with the *source restriction* attribute set to the Basic State corresponding to the actual source in the Statechart, and the *target determinator* set to the basic state corresponding to the actual target in the Statechart.

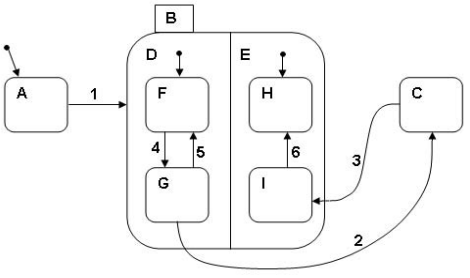
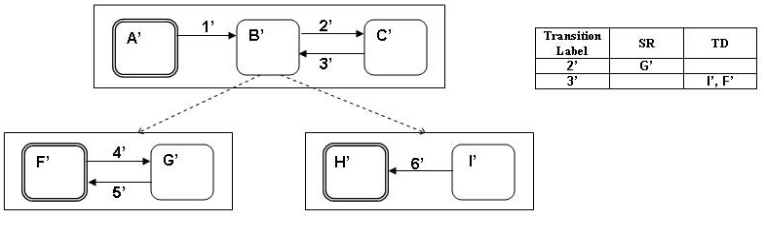


Fig. 3. A sample Statechart model

Figure 3 shows a sample Statechart model and Figure 4 shows the transformed EHA model. Transitions *2* and *3* are inter-level, and the *source restriction* and *target determinator* values for the EHA model are shown in the table on the top right.



Transition Label	SR	TD
2'	G'	
3'		I', F'

Fig. 4. Sample EHA model

3.2 Verifying Behavior Equivalence

In order to define the verification problem we introduce the following concepts. A *state configuration* in a Statechart is a valid set of states that the system can be active in. If a state is part of an active configuration, then all its parents are also part of the active configuration. A transition in the Statechart can take the system from one state configuration to another state configuration, where the source and target states of the transition are subsets of the initial and final state configurations. Similarly, a state configuration in an EHA model is a set of Basic States. If a Basic State is part of an active configuration, and is part of a non-top-level Sequential Automaton, then the Basic State that is refined into this Sequential Automaton is also a part of the active configuration.

An EHA model truly represents the reachability behavior of a Statechart model, if every reachable state configuration in the Statechart has an equivalent reachable state configuration in the EHA and vice versa.

Definition. We can define a relation R between the Statechart and the EHA models, and check if the relation is a bisimulation as follows:

1. Given a state configuration S_A in the Statechart model, there exists an equivalent state configuration S_B in the EHA model
2. Given a transition $\boldsymbol{t}$: $S_A \rightarrow S'_A$ in the Statechart, there exists an equivalent transition $\boldsymbol{t'}$: $S_B \rightarrow S'_B$ in the EHA
3. If for any two equivalent state configurations (S_A, S_B), there exist equivalent transitions ($\boldsymbol{t}$: $S_A \rightarrow S'_A$, $\boldsymbol{t'}$: $S_B \rightarrow S'_B$) such that S'_A and S'_B are equivalent (and vice-versa), then the relation R is a bisimulation.

If the R is a bisimulation, then verifying the EHA model for reachability will be equivalent to verifying the Statechart model for reachability. If not, it means that the models do not behave identically with respect to reachability, and that could be due to an error in the transformation.

When the model transformation algorithm outlined earlier is implemented, it is known which of the Basic States and Transitions in the EHA were created corresponding to which of the states and transitions in the Statechart. However, it is not certain whether all the states were represented, all compound states were refined correctly, all the transitions were connected correctly, and all the inter-level transitions were annotated correctly. To verify this, we need to keep track of the relation R between the two models, and check if there is a bisimulation.

3.3 Checking for Bisimilarity

We built a tool that checks the bisimilarity between the design and analysis models by validating the equivalence relation between the states and transitions of the two sides. The checking of bisimilarity is of linear complexity in the number of states and transitions considered. The checking was made possible by the model transformation approach we have used that uses graph rewriting with explicit links between source and target elements maintained throughout the transformation, as explained below.

During the model transformation, our model transformation tool allows us to create *cross-links* that link model elements in the source model to those in the target model. During the transformation process, when a transformation rule matches a state or a transition in the Statechart and creates the equivalent Basic State or transition in the EHA, a cross-link is created, which marks the relation R between the two elements. Figure 5 shows a sample GReAT rule which achieves this. The top part of the rule matches the Statechart element, and the bottom part creates the corresponding EHA element. The ✓mark in *BasicState* indicates that it is newly created, along with the associated links (these appear in blue when seen in color). The link from *BasicState* to *PrimitiveState* is a *cross-link*. The *AttributeMapping* block allows us to add code to the rule, to perform some

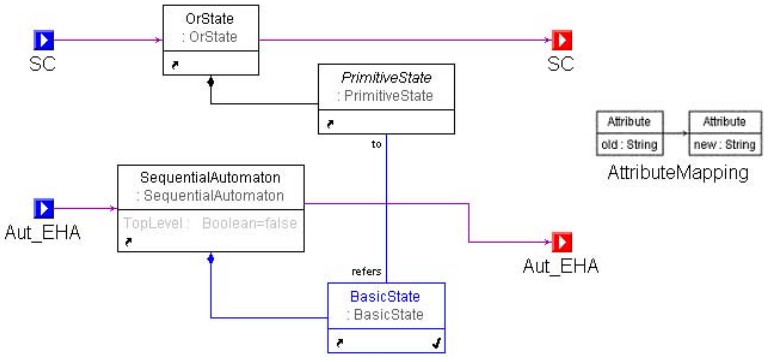


Fig. 5. Sample GReAT rule with cross-link

special actions such as set the attribute values for the newly created elements. When the transformation is complete, the equivalence relations can be accurately traced using these cross-links.

At the end of the transformation, we can check if the equivalence relation is a bisimulation. Rather than checking for all possible state configurations in the Statechart, it is more efficient to consider every transition in the Statechart and the minimal source state configuration. If we can confirm that every transition in the Statechart model has an equivalent transition in the EHA model for which the source and target state configurations are equivalent (and similarly from transitions in the EHA model to the Statechart model), then we can conclude that a state configuration in the Statechart is reachable if and only if its equivalent state configuration in the EHA model is reachable.

In our implementation of the bisimulation checker, we collect the set of all the transitions from the source graph. For each transition in this set, we find the equivalent transition in the EHA by following the cross-link. Now we can compute the minimal source state configuration S_A for the transition in the Statechart model, and the source state configuration S_B for the EHA model. We check the equivalence of S_A and S_B by taking every state s in S_A, finding its equivalent state s' from the EHA, and checking if s' is in S_B, and vice versa. The target states are checked similarly. If this check succeeds for all transitions in the Statechart, and there are no more transitions in the EHA, then the two systems can be said to be bisimilar with respect to reachability. In other words, if bisimilarity holds then we can determine reachability in the Statechart model by verifying it in the EHA model. If this check fails, then there may be errors in the transformation, and the generated EHA model does not truly represent the input Statechart model.

3.4 Conclusions from the First Case Study

The experimental setup used to demonstrate the approach was as follows. We created a set of models in the design language and translated them into the

analysis language, marking the equivalence relations during the transformation using cross-links. After finishing the translation, the bisimulation checker was called, with the cross-links between the source and target models preserved. If this checker verified that there is a bisimulation relation between the models, then reachability properties verified on the analysis model will be guaranteed to hold for the source model.

Note that in this approach, the verification done on the analysis models *together* with the bisimilarity check provides the certification. Either one of them alone is not sufficient. Furthermore, the certificate is valid *only* for the particular model and *not* for the model transformation in general.

To illustrate this point, we deliberately introduced a small error in the transformation process that caused problems only if hierarchical states were used. This resulted in transformations that checked correctly for specific models (with no state hierarchy), but failed to check for others (i.e. hierarchical ones). Hence, models without hierarchical states where transformed correctly, while models with hierarchical states were not. The result of the verification could be accepted only if the bisimilarity check has succeeded. It is interesting to note that though the transformation had an error, in the instances where the check succeeded, the target models did truly represent the source models with respect to reachability. Thus, there would be no error in performing reachability analysis on the target models in these instances. In other words, our approach will capture instances where a transformation fails, rather than capture errors in the transformation itself.

The question that arises is whether a strict bisimulation relation is provable in a wider range of transformations, where there may not be a one-to-one mapping between the models that are representable in the source and the target languages. An important question is whether we even need to prove that a strict bisimulation exists. It would be useful to explore other less strong forms of equivalence, that can be applied to more generic cases.

To answer these questions, we will go to our next case study, where we will consider two Statechart variants that differ in certain features, and thus in the set of systems they can represent. We will study a transformation from one variant to the other, and try to prove that it preserved certain properties of interest. We will use *Semantic Anchoring* [6] and a slightly modified notion of bisimulation, called *weak bisimulation*.

4 More Background

4.1 Semantic Anchoring

The meta-model of a domain Specific Modeling Language (DSML) specifies its syntax and static semantics. Semantic Anchoring [7] [6] is a method for specifying the dynamic semantics of DSMLs. It relies on the observation that a broad category of behaviors can be represented by a small set of behavioral abstractions. The behavior of certain abstractions such as Finite State Machines or Timed Automata has been studied over several years in a wide range of applications.

Their behaviors are well understood and precisely defined. Such abstractions are called *Semantic Units*. Semantic Anchoring is the specification of the behavior of a DSML as a transformation from the DSML to a chosen semantic unit. The semantic unit is usually represented in some formalism, such as Microsoft's Abstract State Machine Language (AsmL) [3], for performing verification.

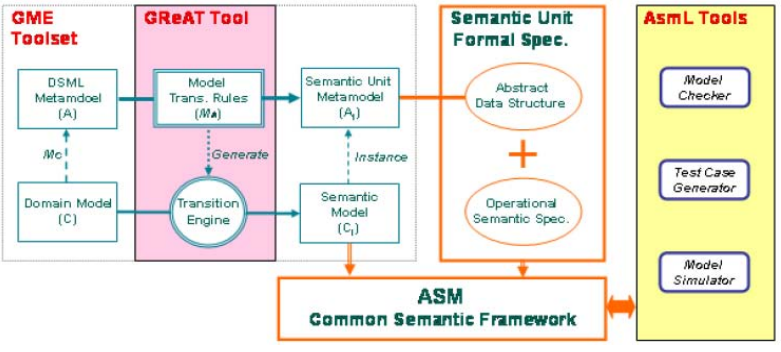


Fig. 6. Tool Architecture for Semantic Anchoring

Figure 6 shows the architecture used for semantic anchoring. In our case study, we will use FSMs as the semantic unit.

4.2 Statechart Variants

Statecharts [10] were first proposed by Harel to model the reactive behavior of systems. Since then, different variants of the formalism have been proposed to address specific problems. The result is that today we have several variants of the Statecharts formalism, such as iLogix Statecharts and MATLAB Stateflow. A number of such variants and the differences is studied in [20]. On some occasions, such as during tool integration, we may need to transform models from one variant of the formalism to another. It is very important in these cases that the behavior is preserved by the transformation.

Since the common commercial Statecharts variants vary in subtle issues, we will look at two hypothetical variants that will vary on a small but significant set of features. Let us call these hypothetical variants *SCA* and *SCB*. We will now look at the differences between *SCA* and *SCB*.

Compositional Semantics. Compositional semantics is the property of being able to define the semantics of a compound component completely from the semantics of its subcomponents, without looking at its internal syntactical structure. Having compositional semantics simplifies verification in many cases. Having transitions that cut across levels of hierarchy violates compositional semantics. Thus, if a Statechart variant allows inter-level transitions, then it will

not have compositional semantics. For our case study, *SCA* will permit inter-level semantics, while *SCB* will not.

Note that *SCA* models which have inter-level transitions can also be represented in *SCB*, by using *self termination* and *self start* states [20]. These will be explained with examples later.

Instantaneous States. Some Statechart variants allow states to be entered and exited in a single time step (In the current case study we only consider the synchronous model). Such states are called *instantaneous*. Such a Statechart can step through a series of instantaneous states in a single time step, until a *non*-instantaneous state is reached. Such a series of transitions is called a *macro step*. For our case study, *SCA* will not permit instantaneous states, while *SCB* will permit them.

State References. In some Statechart variants, transitions may be guarded by referencing the activity of other parallel states. This is done using state references, which are special conditions denoting the activity of states. The condition *in(S)*, *en(S)* and *ex(S)* will be true in the time steps when the state S is active, entered and exited respectively. In our case study, *SCA* will permit state references, while *SCB* will not.

These are significant differences in the features offered by the variants. Representing a model in one variant in terms of the other variant will require some significant changes. This will be explained with an example later. It must be understood that all *SCA* models may not be representable in *SCB* (and vie versa), but our goal is to verify whether the transformation was correct in the specific cases that *were* representable.

4.3 Weak Bisimulation

Let us go back to our earlier definition of bisimulation. Given an LTS (S, Λ, $\rightarrow$), a relation R over S is a *bisimulation* if:

$$(p,\, q) \in R \text{ and } p \xrightarrow{\alpha} p' \text{ implies that there exists a } q' \in \mathrm{S},$$
$$\text{such that } q \xrightarrow{\alpha} q' \text{ and } (p',\, q') \in R,$$

and conversely,

$$q \xrightarrow{\alpha} q' \text{ implies that there exists a } p' \in \mathrm{S},$$
$$\text{such that } p \xrightarrow{\alpha} p' \text{ and } (p',\, q') \in R.$$

This defines a *strict* bisimulation, which enforces a strict one-to-one mapping of the state space. In some cases, two systems may have essentially the same behavior, but differ in their state spaces. One system may have intermediate states that are not observable externally, but truthfully reproduce the observable behavior of another system. If we modified our notions of what constitutes states, transitions and labels, we may find that the two systems are bisimilar. This leads us to the notion of *weak bisimilarity* [11]. For instance, in a system that allows

instantaneous states, we can define weak bisimulation by considering only non-instantaneous states in the bisimulation definition, and by considering macro steps as a single transition. This gives a more practical and usable method to compare the behavior of two systems.

5 Verifying Transformations Using Semantic Anchoring

We now look at our modified architecture for goal-directed certification, using semantic anchoring and weak bisimulation.

We first define the behavior of the DSMLs of the transformation by semantic anchoring. This is used to generate the behavior model from the instance models. The behavior models will be in a common semantic unit (Finite State Machines). A tool is used to check if the generated behavior models are weakly bisimilar, based on a suitable definition of weak bisimulation which we will see later. Figure 7 shows an overview of this framework.

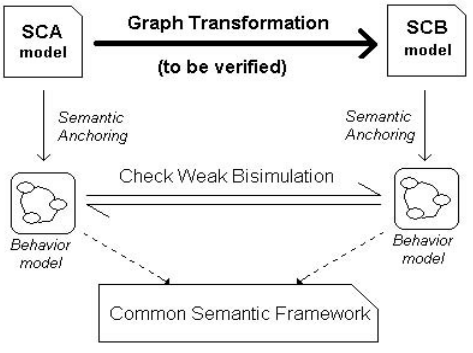


Fig. 7. Framework for verifying behavior preservation

5.1 Transformation from *SCA* to *SCB*

Figure 8 shows an *SCA* Statechart model, and Figure 9 shows the transformed *SCB* model. The transformation from *SCA* to *SCB* first represents all the states in the *SCA* model. The first issue to address is the presence of inter-level transitions. These are represented using self-termination and self-start states. The transition $\boldsymbol{T_2}$*:b* in the *SCA* model is an inter-level transition, triggered by an event *b*. This is represented in the transformed *SCB* model by adding a self-termination state *D*. The transition is broken into two parts, one from the start state to D, and another from the parent state *Q* to *C*. Neither of these transitions are inter-level, but their combined effect is similar to the inter-level transition in the *SCA* model.

For the semantics of the transition to be identical, the system must transition from state *B* to state *C* in a single time step. To achieve this, *D* is made an instantaneous state, and an instantaneous event *i* is generated. Thus, the *D*

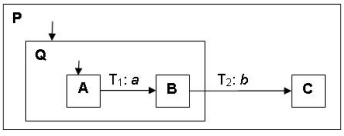


Fig. 8. A sample *SCA* model

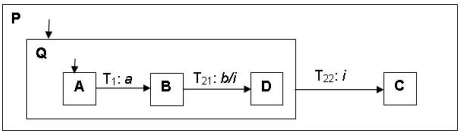


Fig. 9. A sample *SCB* model

can be exited and i will be available in the same time step. This will make the series of transitions $\boldsymbol{T}_{21}$ and $\boldsymbol{T}_{22}$ into a macro step that will be identical to the transition $\boldsymbol{T}_2$ in the *SCA* model, to an external observer.

After copying the states, the transformation copies all normal transitions between the corresponding states. Inter-level transitions are then elevated to the common parent of both the start and the end states. Self-termination and self-start states are added on the source and target sides of the transition as necessary, using unique instantaneous states and actions. When state references are encountered in the *SCA* model, unique actions representing the state activity are added to all transitions entering or exiting the state. For instance, when a state reference *en(S)* appears, all its occurrences are denoted in the *SCB* model by a specially named action, and this action is added to all transitions entering state *S*.

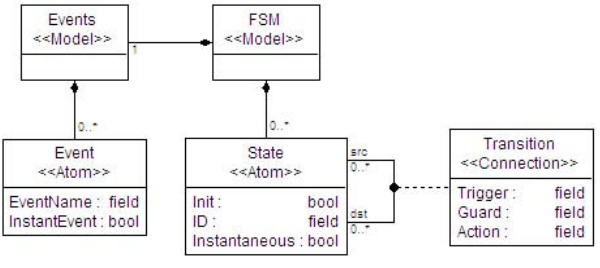


Fig. 10. Meta-model for FSM semantic unit

5.2 Behavior by Semantic Anchoring

We represent the behavior of both the Statechart variants using a common semantic unit, namely Finite State Machines. The meta-model for FSM is shown in the Figure 10.

The semantic anchoring is specified as a transformation from the Statechart variant to the FSM semantic unit. The FSM is enhanced with instantaneous states and actions, to represent the behavior of instantaneous states and actions of the *SCB* model. Figures 11 and 12 show the behavior models in terms of the FSM semantic unit, for the sample Statechart models described above. The FSM semantics are defined by an AsmL model generated from the FSM model. The AsmL model models the behavior of the states and transitions of the FSM, taking into account the instantaneous nature of some of the states. Here, we assume

that the semantic anchoring has been correctly specified, and does not need to be verified. However, it may be possible to use the bisimulation techniques described earlier to verify that the semantic model correctly represents the source model.

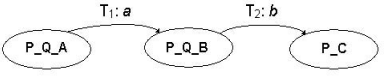


Fig. 11. FSM semantic model for the SCA model

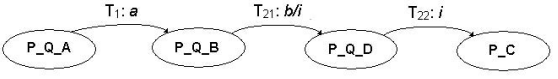


Fig. 12. FSM semantic model for the SCB model

5.3 Verifying Behavior Preservation

To provide a certificate for an instance of a transformation from *SCA* to *SCB*, we will compare the generated behavior models of the Statechart models, using the definition of weak bisimulation described below.

Weak Bisimulation. We establish the relation R between non-instantaneous states of the two transition systems. We then define a transition $\boldsymbol{T}$ as a transition from one non-instantaneous state to another, and its label as the aggregate of the events and actions of the constituent transitions, if instantaneous states are involved, ignoring the instantaneous actions (which will not be visible outside the macro step). Using these conditions, we redefine our bisimulation relation for weak bisimulation as follows. Given the relation R between non-instantaneous states p and q, R is a weak bisimulation if:

$$\forall\ (p,\ q) \in R \text{ and } \forall \alpha\text{: } p \overset{\alpha}{\Rightarrow} p',\ \exists\ q' \text{ such that } q \overset{\alpha}{\Rightarrow} q' \text{ and } (p',\ q') \in R,$$

and conversely,

$$\forall \alpha\text{: } q \overset{\alpha}{\Rightarrow} q',\ \exists\ p' \text{ such that } p \overset{\alpha}{\Rightarrow} p' \text{ and } (p',\ q') \in R.$$

The weak transition is represented by $\Rightarrow$, and its label α is represented as a comma-separated list of the triggers and actions involved. According to this definition, the FSMs shown in Figure 11 and Figure 12 are weakly bisimilar. Note that this notion of weak bisimilarity guarantees equivalence of behavior between the two models, for all practical purposes.

Checking for Weak Bisimulation. The relation R between corresponding (non-instantaneous) states of the two behavior models is traced by using specially coined labels to represent the states in the system. Having generated the behavior models, and given the relation R, we can modify our bisimulation checker to check for weak bisimulation. If the checker shows that the two behavior models are weakly bisimilar, we can conclude that the transformed *SCB* model truly represents the source *SCA* model.

6 Conclusions

We feel that goal-directed certification is more practical and achievable than providing a general correctness proof for a model transformation. Using the transformation itself to trace the equivalence relation R between the source and target elements makes it easier to check if the systems are bisimilar. We have also seen that weak bisimulation helps us to practically extend this approach to a wider range of systems. Semantic anchoring is a very powerful technique for specifying DSML behavior, and a combination of a well chosen semantic unit and a well defined weak bisimulation criterion can help us verify the semantic equivalence of models generated by a model transformation.

We may also choose to represent a small subset or a specific aspect of a system's behavior by semantic anchoring, and use suitably defined weak bisimulation criteria to verify the preservation of specific behaviors in model transformations between DSMLs that are otherwise very different. Further research in using this technique in a wide range of transformations will provide more insight into the nature of such behaviors. Other types of transformations that we wish to address in the future include abstractions and refinements, such as from a block diagram like representation (such as Simulink) to embedded code.

Acknowledgments. The research described in this paper has been supported by an NSF Grant, CNS-0509098, titled: Software Composition for Embedded Systems using Graph Transformations.

References

1. Matlab's Simulink/Stateflow, `http://www.mathworks.com/products/stateflow/`
2. OSC Embedded Validator, `http://www.osc-es.de/index.php?idcat=17`
3. The Abstract State Machine Language, `http://www.research.microsoft.com/fse/asml`
4. TNI Safety-Checker Blockset, `http://www.tni.fr/en/produits/safetycheckerblockset/index.php`
5. Agrawal, A., Karsai, G., Ledeczi, A.: An end-to-end domain-driven software development framework. In: OOPSLA 2003: 18th annual ACM SIGPLAN conference on OOP, systems, languages, and applications, pp. 8–15. ACM Press, New York (2003)
6. Chen, K., Sztipanovits, J., Abdelwahed, S., Jackson, E.K.: Semantic Anchoring with Model Transformations. In: ECMDA-FA, pp. 115–129 (2005)
7. Chen, K., Sztipanovits, J., Neema, S.: Toward a semantic anchoring infrastructure for domain-specific modeling languages. In: EMSOFT 2005: Proceedings of the 5th ACM international conference on Embedded software, pp. 35–43. ACM Press, New York (2005)
8. Denney, E., Fischer, B.: Certifiable Program Generation. In: GPCE, pp. 17–28 (2005)
9. Göttler, H.: Attributed graph grammars for graphics. In: Proceedings of the 2nd International Workshop on Graph-Grammars and Their Application to Computer Science, London, UK, pp. 130–142. Springer, Heidelberg (1983)

10. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
11. Harwood, W., Moller, F., Setzer, A.: Weak Bisimulation Approximants. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 365–379. Springer, Heidelberg (2006)
12. Holzmann, G.J.: The Model Checker SPIN. Software Engineering 23(5), 279–295 (1997)
13. Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. Computer 34(11), 44–51 (2001)
14. Mikk, E., Lakhnech, Y., Siegel, M.: Hierarchical Automata as Model for Statecharts. In: ASIAN 1997, pp. 181–196. Springer, Heidelberg (1997)
15. Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.J.: Implementing Statecharts in PROMELA/SPIN. In: WIFT 1998: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques, Washington, DC, USA, p. 90. IEEE Computer Society Press, Los Alamitos (1998)
16. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL 1997), January 1997, pp. 106–119 (1997)
17. Schürr, A., Winter, A., Zündorf, A.: The PROGRES approach: Language and environment. In: Rozenberg [21], ch. 13, pp. 487–550, 15.
18. Nickel, U., Niere, J., Zündorf, A.: Tool demonstration: The FUJABA environment. In: The 22nd International Conferenceon Software Engineering (ICSE), Limerick, Ireland, ACMPress, Limerick (2000)
19. Varró, D.: A Formal Semantics of UML Statecharts by Model Transition Systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 378–392. Springer, Heidelberg (2002)
20. von der Beeck, M.: A Comparison of Statecharts Variants. In: ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems, London, UK, pp. 128–148. Springer, Heidelberg (1994)