Formal Verification of Java Code Generation from UML Models

Jan Olaf Blech   Sabine Glesner   Johannes Leitner

Institute for Program Structures and Data Organization
University of Karlsruhe, 76128 Karlsruhe, Germany

ABSTRACT

UML specifications offer the advantage to describe software systems while the actual task of implementing code for them is passed to code generators that automatically produce e.g. Java code. For safety reasons, it is necessary that the generated code is semantically equivalent to the original UML specification. In this paper, we present our approach to formally verify within the Isabelle/HOL theorem prover that a certain algorithm for Java code generation from UML specifications is semantically correct. This proof is part of more extensive ongoing work aiming to verify UML transformations and code generation within the Fujaba tool suite.

Keywords

Statecharts, Fujaba, Isabelle/HOL, verification, semantics, transformations, code generation.

1. INTRODUCTION

The generation of code from specification languages like UML is an important aspect of the Model Driven Architecture (MDA). State-of-the-art Computer Aided Software Engineering (CASE) tools like the Fujaba tool suite come with such code generation mechanisms. To ensure correct software and system behavior, it is a necessary prerequisite that the semantics of the original UML systems is preserved during code generation.

To ensure that such transformations are correct, formal verification is necessary. In this paper, we consider an algorithm that transforms simplified Statecharts into a Java-like language. We verify that this transformation algorithm is correct. Therefore we formalize the semantics of Statecharts and the targeted Java subset as well as the transformation algorithm within Isabelle/HOL. Furthermore, we prove, also within Isabelle/HOL, this code generation algorithm correct by showing that each Statechart specification and the corresponding program code are bisimilar. The use of bisimulation as equivalence criterion between Statecharts and the Java-like programming language ensures an adequate semantics formalism even for non-terminating systems and programs. In order to obtain machine-checked proofs as well as reusability of proofs, our proofs and formalizations are conducted within Isabelle/HOL.

This paper describes ongoing work that is part of a larger project aiming to verify real life transformations from UML specifications of CASE tools to Java code. Apart from our already finished formalization and proof work, we give a detailed plan for our future work concerning verification of MDA transformations. We believe that verification of Fujaba transformations can be a major and highly interesting research area in the future.

In Section 2, we discuss various approaches for the formalization of Statecharts. After that, we explain basic foundations for behavioural equivalence proofs in Isabelle/HOL in Section 3. Our formalization of the Java-like programming language as well as our Statechart formalization together with the actual correctness proof is discussed in Section 4. We discuss related work in Section 5. In Section 6, we conclude and present our future workplan.

2. SEMANTICS OF STATECHARTS

Statecharts are an extension of finite state machines which are implemented in many tools and widely used in practice. Nevertheless, the definintion of their semantics poses subtle difficulties due to inherent ambiguities. In this section, we first introduce Statecharts in Subsection 2.1. Afterwards, in Subsection 2.2, we summarize how executable code, e.g. Java code, can be generated from them automatically. In Subsection 2.3, we explain the difficulties when formally defining the semantics of Statecharts. Finally, in Subsection 2.4, we discuss methods to prove equivalence of Statecharts.

2.1 The Statechart Language

Statecharts [12] are a visual language which enhance finite state machines by hierarchical and parallel composition of states (and broadcast communication between parallel transtitions). They have found wide-spread use in the modelling of complex dynamic behaviour and are part of the UML standard [20] with advanced features such as history mechanisms and extended transition trigger conditions. In UML Statecharts, transitions as well as states can be decorated with actions, i.e. statements in some imperative language, thereby significantly increasing the expressive power of statecharts.

2.2 Code Generation From Statecharts

There are numerous tools providing code generation from Statecharts. Most of these employ one of the following three strategies of code generation which are almost directly derived from code generation strategies for finite automata.

- *(Hierarchical) Switch/Case Loop* This most simple approach creates a nested switch/case statement that branches according to the current state and the current event. Within a branch, transition-specific code, i.e. the action associated with the transition, is executed and the current state is set to the target state of the transition. Hierarchical and concurrent structures can be achieved using recursion.

- *Table-driven approach* The second approach stems from a well-known method to implement finite state machines in compiler construction (e.g. scanner generation by the well-known unix tool "lex"). The actions caused by an event in a specific state are stored in a (nested) state/input table. In its most basic form, entries in this table might only consist of output symbols and successor states. When more complicated actions are used, more complex structures are necessary for the representation of state table entries, as demonstrated in [27]. In principle, a table-based approach is also suited for hardware implementation in embedded systems.

- *Virtual Methods* Deeply nested switch/case blocks may not be desirable in an object oriented system. This is especially true when code generated from a Statechart is subject to manual modification and maintenance ("round-trip engneering"). An alternative method of code generation from Statecharts makes use of an extension of the state pattern [7]. In this method, each state becomes a class in an inheritance hierarchy created in parallel to the substate hierarchy of the statechart. The events consumed by these states are realized as virtual method calls to the respective state classes.

These are the basic strategies for code generation from Statecharts. A more detailed overview can be found in [27]. [26] shows how hierarchical structuring information can be exploited to obtain smaller and more efficient code following the table-based strategy.

In this paper, we present our verification of code generation that follows the first strategy. In future work, we aim to extend our correctness proof to also allow for the verification of other generation algorithms as well.

2.3 Formal Semantics of Statecharts

The behaviour of a Statechart is modeled by a transition system whose states correspond to the configurations of the represented Statechart. A configuration itself is a maximum set of Statechart states that can be active at the same time. The actual behavior of the Statechart lies in the state transition function, describing how one configuration leads to the next, depending on the current input symbol. A formal definition of this step transition function has proven somewhat challenging. A multitude of approaches has been taken to define such a semantics. A comparison between 17 of these approaches can be found in [25].

One reason for the difficulties in defining a state transition function is the desired property of *synchrony.* Synchrony means that the system should react immediately to incoming events. In particular, incoming events and resulting actions should happen without delay at the same time. Most of the time, this does not pose a problem physically, since the sampling rate of the real system is typically limited and significantly longer than the reaction time of the system. However, using synchrony, concurrent states and broadcast communications allow for a single event to trigger a chain reaction of multiple transitions, called *microsteps*, in "zero time", creating a situation that is inconsistent with the notion of *causality* (i.e. that original actions and the actions triggered by them cannot be distinguished).

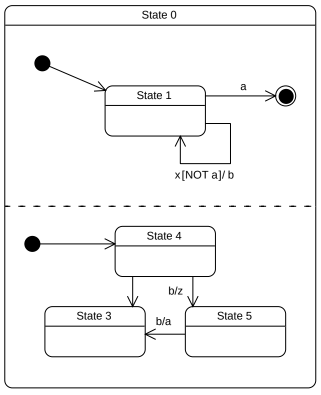

Figure 1: Event Conflict

As an example, consider the event `x` passed to the Statechart depicted in Figure 1 in the initial configuration (State 1, State 4). The system will take the loop at State 1, and also go from State 4 to State 5, since the first transition emits the event `b` and synchrony causes the system to react immediately. In the next step, if the input symbol is again `x`, this leads to a contradiction, since the transition from State 5 to State 3 causes the event `a`, in which case the original transition (the loop at State 1) should not have been taken.

The conflicts involving synchrony and negated conditions can be resolved in different ways. One way is to consider only *globally consistent* steps, which is intended in [22]. Global consistency is difficult to achieve, since their might be multiple sets of transitions for a signal even if the state machine is deterministic. Another, more direct solution is to take the *causal order* of microsteps into account and require that conditions guarding a transition only apply to events that occurred *before* the transition was taken. This kind of behavior is sometimes called *local consistency.* Local consistency leads to the paradox situation that although all microsteps take place at the same time, the order in which they occur is not irrelevant.

Since synchrony, although a desirable property for the modeling of real-time systems, causes these problems and often leads to counter-intuitive behaviour (especially when more complicated actions are allowed), more "practical" semantics, including the STATEMATE [13] and the UML [3] semantics, disregard this property and realize a step-by-step behaviour, in which events and actions generated in one step do not become visible until the next step. In UML, this is sometimes called *run-to-completion-semantics* [3].

Defining a step function becomes even more problematic when we aim to develop a *compositional* semantics. In general, compositional semantics means that the semantics of a larger program or system can be derived from the semantics of its parts. Especially in the case of parallel composition, Statecharts are "noncompositional" in nature – the behaviour of the concurrent state can differ significantly from that of its substates when parallel transitions have overlapping actions and conditions. When being concerned with the equivalence of Statecharts, it is important to define an equivalence as a congruence with respect to the Statechart constructors. This specifically means that states that are regarded as equivalent should again yield an equivalent Statechart when composed with the same state and Statechart constructor. [24] and [17] demonstrate that such an equivalence relation cannot be defined by regarding only complete steps. Instead, the causal ordering of microsteps needs to become part of the semantics. [24] achieves this by constructing more complex labels in the resulting transition system, which then contain information on how the respective step was constructed out of microsteps. [17] use two different kinds of transitions, microstep transitions and σ-transitions, which determine the beginning and completion of a step, to incorporate information about causal ordering directly into the transition system.

2.4 Equivalences of Statecharts

The ability to prove the equivalence of Statecharts allows us to prove the correctness of elementary Statechart-to-Statechart-transformations. [6] present a collection of 23 such transformations, e.g. splitting a state or shifting a transition up and down the substate hierarchy. To prove the semantical correctness of such a transformation, a suitable Statechart semantics is needed. Since we are interested in local Statechart transformations, compositionality becomes an even more important issue here. In [18], a number of equivalences (which are originally introduced in [5]) is applied to the labelled transition systems defined by a slight variation of the semantics in [24]. Moreover, congruence properties are studied with respect to Statechart constructor operations. They show that (of the six investigated relations), the weakest relation between Statecharts that is still a congruence is bisimilarity between their corresponding labelled transition systems.

In this paper, we consider a restricted subset of Statecharts consisting of non-hierarchical automata without concurrency and show how their equivalence can be defined by bisimulation in Isabelle/HOL. In the following section, we introduce some basics about the theorem prover Isabelle/ HOL as well as our formulation of bisimulation. Afterwards, in Section 4, we use this formalization to prove the equivalence between the considered restricted set of Statecharts and the Java code generated from them.

3. PROOF FOUNDATIONS: BISIMULATION AND ISABELLE/HOL

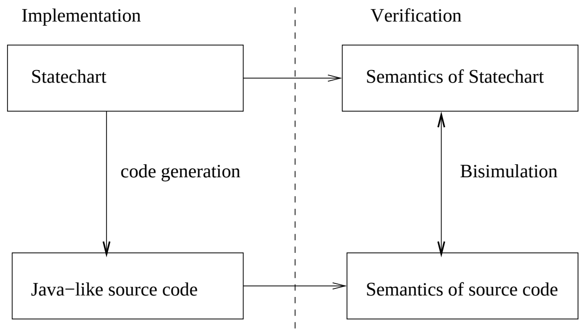

Figure 2: Verification of Code Generation

This section describes bisimulation as the method of choice for transformation verification. Moreover, we desribe some Isabelle/HOL related aspects.

3.1 Bisimulation

Our principle idea is to regard a Statechart and its transformation –a program in a higher programming language– as semantically equivalent if they denote the same observable behaviour. For example in a deterministic specification, they must have the same sequences of observable states. Figure 2 shows the principle to prove such a transformation correct. One needs a Statechart semantics and a target language semantics as well as mappings from concrete Statecharts and programs to their respective semantics. To verify a transformation, one has to show that the semantics of the original system is preserved during its transformation. In our case, this means that the sequences of observable states are the same for both Statechart and generated program. On the semantics side, this means that they have to bisimulate each other, i.e. that their semantics are in a bisimulation relation.

As a basic prerequisite, the semantics of Statecharts and Java-like programs must be comparable. For this purpose, we express their semantics as a Kripke structure.

DEFINITION 1 (KRIPKE STRUCTURES). *A Kripke structure is a five tuple (AP, S, R, S_0, L) where AP is a set of atomic propositions, S is a set of states, R is a transition relation, S_0 is the initial state, and L is a labeling function mapping states to sets of atomic propositions.* $\diamond$

Hence, a Kripke structure is equivalent to an annotated state transition system.

DEFINITION 2 (BISIMULATION RELATION [4]). *Let $M = (AP, S, R, S_0, L)$ and $M' = (AP, S', R', S'_0, L')$ be two Kripke structures with the same set of atomic propositions AP. A relation $B \subseteq S \times S'$ is a bisimulation relation between M and M' if and only if for all s and s', if $B(s, s')$ then the following conditions hold:*

1. $L(s) = L'(s')$

2. *For every state s_1 such that $R(s, s_1)$ there is s'_1 such that $R'(s', s'_1)$ and $B(s_1, s'_1)$*

3. *For every state s'_1 such that $R(s', s'_1)$ there is s_1 such that $R'(s, s_1)$ and $B(s_1, s'_1)$* ⋄

We get a Kripke structure representing the semantics of a Statechart by unrolling its configuration transitions. These correspond to the transition relation R in the Kripke structure. The states S of a Kripke structure correspond to configurations in Statecharts with S_0 being the initial configuration. We encode a stream of upcoming input events as well as the current execution time in the Kripke structure states, too. A Kripke structure may have infinitely many states. This corresponds to a non-terminating Statechart.

We can describe the semantics of a program in a higher programming language as a Kripke structure M: If we specify its semantics such that the execution of a single instruction is atomic, the semantics of a program is specified by a state and a state transition function. Each state may consist of the current execution state and memory and variable mappings. Kripke structures are used as follows for the specification of program semantics: The atomic propositions represent the variable mapping, memory etc. S is the set of states reachable within the execution of the program M. R represents possible state transitions and the conditions under which they appear. S_0 is the initial state. L is a labeling function mapping states to their observable parts. This is appropriate for both Statecharts and programming language. Two programs, or a Statechart and a Java-like program, resp., are bisimilar if there exists a bisimulation relation B such that the initial states of both programs are within the relation.

If we describe the semantics of a Statechart as a Kripke structure M and the semantics of a corresponding program in a higher programming language as a Kripke structure M', then the bisimulation relation B expresses behavioral equivalence, with an equivalence criterion that we can choose freely: E.g. we can restrict the variables that appear in states – as well as in the atomic propositions – to input/output values. Then $L(s) = L'(s')$ checks state equivalence. With the notion of bisimulation, we have a formal criterion under which a program and a Statechart show the same behavior.

In the case of deterministic systems that we examine in this paper, the requirements for a bisimulation get even simpler: We regard two programs as semantically equivalent iff:

- They start with equivalent initial states s and s'. This is denoted $s \simeq_O s'$ where O is some set of observable actions. By equivalence we mean that the observable parts of the states must be the same, corresponding to the requirement $L(s) = L'(s')$ in Definition 2.

- For two states s and s' in the bisimulation relation, we require that the succeeding states are equivalent again. This is formalized in Isabelle/HOL as:
  $\forall\ s\ s'\ .s \simeq_O\ s' \ \longrightarrow \mathit{next}\ s \simeq_O \mathit{next}\ s'$
  where *next* returns the succeeding state.

This notion of program and Statechart equivalence captures very elegantly the semantics of both terminating and non-terminating programs and Statecharts. With its state abstraction, it is flexible enough to prove most transformations and optimizations correct. If we want to prove the correctness of a code generation algorithm from Statecharts, we have to show that Statechart and generated program(code) denote the same sequence of observable states which is exactly what a bisimulation proof does.

Bismulation may be defined on Kripke structures. A mathematically more elegant approach is to use coalgebraic data types instead of Kripke structures [14].

3.2 Isabelle/HOL-Specific Aspects

Isabelle/HOL is a generic higher order logic (HOL) theorem prover ensuring a very high expressivness of specifications. Theorem provers can be used to create specifications, formulate lemmata and theorems on them and prove them correct. Unlike model checking, working with theorem provers, especially those using higher order logics, is highly interactive. Specifications have to be designed very carefully in order to be able to prove them correct. The process of proving a system specification correct takes some effort but often reveals errors in the specification that would have been overlooked otherwise.

In Isabelle/HOL, bisimulation can be formulated in multiple ways. A very elegant way is to use coalgebraic datatypes, e.g. lazy lists [21], which in most cases of practical relevance come automatically with a bisimulation principle. In contrast to model checking, our Isabelle proofs verify complete semantical equivalence and not just certain aspects or conditions.

4. VERIFYING THE TRANSFORMATION FROM UML TO JAVA

In this section, we present our verification of Java code generation from UML models. Therefore, we consider a simplified subset of UML Statecharts, namely finite state machines (FSMs). To verify code generation, i.e. the transformation from an FSM to a target language program, a semantics for both FSMs and the target language with the same semantic domain is required. As already explained in the previous section, we concentrate on observational equalivalence by modelling semantics as the state transition sequences that can be observed during execution. In Subsection 4.1, we show how we represent FSMs and their semantics in Isabelle/HOL. Then, in Subsection 4.2, we introduce our target programming language WSC that contains while, switch, and case statements. The code generation algorithm is given in Subsection 4.3. Its correctness proof is presented in Subsection 4.4.

4.1 Formalization of Finite State Machines

Finite state machines are formalized as tuples (S, E). S is a list of *states* having arbitrary type. This allows in particular for hierarchical FSMs since the type of a state can be again an FSM. (Note that in the work presented here, we have not dealt with hierarchical FSMs. But since we want to so soon, we have already adjusted the specifications for this purpose.) E is a list of transitions connecting the states in S. A transition consists of its *source* and *target* (which are represented by the position of the source and target states in the state list S), a *trigger* symbol and an *action* symbol. We enhance the set of action symbols by the new symbol `silence`. With this extension, we make sure that every transition has an action associated with it.

Our decision to use a list to store the states of a Statechart has many benefits. It eliminates the need to find an ordering of states when generating code. Moreover, it is suitable for hierarchical automata since lists are inductive types. Active states can now be referenced by their position in the state list. We call the pairing of state machine and currently active state an *FSM configuration*. For example, (((State 4, State 3, State 5), ((1,2), (1,3), (3,2))), 1) is the initial configuration of the lower substate of the statechart in Figure 1.

The semantics of an FSM is the potentially infinite sequence of output symbols it produces given a sequence of input symbols. If the state machine is deterministic, i.e. there is at most one transition for each trigger and source state, a partially defined *step function* exists that selects this transition (if it exists) for a given state and input symbol. The output sequence is then calculated by the corecursive application of this function. (For simplicity, one might think of this corecursively defined output sequence as the potentially infinite list of transitions that are performed during the run of the FSM.)

4.2 The WSC Language

As target language, we consider a simplified subset of Java called WSC (while-switch-case) that contains switch and case statements and a specialized while loop. The language has only two variables of type integer, without the ability to define new ones. WSC only incorporates the limited functionality needed for the code generated from state machines, and can be easily transformed into real Java. Our definition of its syntax within Isabelle/HOL is given in Figure 3.

```
datatype WSC_Variable =
  CURRENTSYMBOL | STATE
  /— This language has only two variables/

datatype WSC_Expression =
  CONST nat | VAR WSC_Variable

datatype WSC_Statement =
  WHILE_NEXT_SYMBOL WSC_Statement |
  SWITCHCASE WSC_Expression
    "( WSC_Expression × WSC_Statement ) list"
    WSC_Statement
        ( "SWITCH _ CASES { _ } DEFAULT _" ) |
  ASSIGN WSC_Variable WSC_Expression |
  CONS WSC_Statement WSC_Statement ("_;_") |
  OUTPUT ActionType |
  SKIP
```

Figure 3: Syntax definition of the WSC language

The semantics of WSC is defined by specifying for each program a potentially infinite sequence of ouput symbols. For each pair consisting of a WSC program and its current state, we define the observable *output*, the *successor state*, and the *continuation*, i.e. the remainder of the program to be executed. Thereby, a state is a function mapping variables to values. We can then corecursively apply these functions, thus yielding a potentially infinite sequence of successor states and output symbols. For a more detailed description of this procedure, see [16]. Since we are only interested in output equivalence, we can disregard the sequence of states. Obviously most WSC statements do not produce any output, thus the output sequence will contain a lot of empty events which are not generated by the corresponding statechart semantics. We therefore define a *cleaned sequence* which contains only the elements that are not empty.

4.3 The Code Generation Algorithm

From an initial statechart configuration $(((s_1, \ldots, s_n), (t_1, \ldots, t_n)), n)$ we generate WSC code according to the algorithm depicted in Figure 4.

```
while_next_symbol {
 switch ( STATE ) {
   ⋮ – for j = 1...n
   case ( j ) {
    switch ( CURRENTSSYMBOL ) {
      ⋮ – for all transtions $t_k$ with source($t_k$) = j
      case( trigger($t_k$) ){
       STATE := target($t_k$);
       output action($t_k$)
      }
    }
   }
 }
}
```

Figure 4: Overview of Code Generation Algorithm

The default cases of the two switch statements are not shown – they are both empty: For the inner switch statement, this means that an input symbol has occurred that is not the trigger of any transition in the current state. The outer switch statement is never reached if the original state machine is reasonably well-formed, since this would require the STATE variable to point to a non-existing state.

4.4 Correctness Proof

This transformation from FSM to WSC is considered semantically correct iff the semantics of source (i.e. FSM) and target (i.e. WSC program) are always the same. Thus, we have to compare the output sequences of a finite state machine and its generated WSC code. We use out_{WSC}, out_{FSM} resp. to denote these output sequences. Apart from the FSM or WSC program they need an input parameter I – a stream of external events. Using the bisimulation principle from section 3 to show that two sequences are equal, we have to find and define a *bisimulation relation* in which they are contained. In this case, this means a relation $\sim$ such that:

(1) $out_{WSC}(CodeGen(A), I) \sim out_{FSM}(A, I)$

(2) If $X \sim Y$, then either X and Y are both empty, or

  (a) the first elements of X and Y are equal, and

  (b) for the remaining sequences X' and Y', $X' \sim Y'$ holds.

```
lemma c2 :
  assumes a1
     : "X = WSC_seqOut_clean ( FA_CodeGen C ) ( i ⇝ I )"
  shows "X = Leaf ( Some ( takeStepIO C i ) )
        ⇝ WSC_seqOut_clean ( FA_CodeGen ( nextConfig C i ) ) I"
proof ( cases "i ∈ set ( map Trigger ( Relevant C ) )" )
assume  "i ∉ set ( map Trigger ( Relevant C ) )"
hence " WSC_seqOut ( FA_CodeGen C ) ( i ⇝ I ) =
    Leaf None ⇝ Leaf None ⇝ Leaf None ⇝
    Leaf ( Some ( takeStepIO C i ) )  ⇝ FollowCodeGen i C I"
proof ...( 29 proof steps omitted ) ...done
thus "X =  Leaf ( Some ( takeStepIO C i ) )
    ⇝ WSC_seqOut_clean ( FA_CodeGen ( nextConfig C i ) ) I" using a1
by ( simp  )  ( unfold WSC_seqOut_clean_def  FollowCodeGen_def ,  auto )
next
assume  "i ∈ set ( map Trigger ( Relevant C ) )"
hence "WSC_seqOut ( FA_CodeGen C ) ( i ⇝ I )
    = Leaf None ⇝ Leaf None ⇝ Leaf None ⇝
    Leaf ( Some ( takeStepIO C i ) )   ⇝  Leaf None ⇝ FollowCodeGen i C I"
proof ...( 38 proof steps omitted ) ...done
thus "X =  Leaf ( Some ( takeStepIO C i ) ) ⇝
    WSC_seqOut_clean ( FA_CodeGen ( nextConfig C i ) ) I" using a1
by ( simp ) ( unfold WSC_seqOut_clean_def  FollowCodeGen_def ,  auto )
qed
```

Figure 5: Some Details of the Correctness Proof in Isabelle/HOL

```
theorem "state_sequenceIO A I = WSC_seqOut_clean ( FA_CodeGen A ) I"
proof -
have "bisimulation ( ⋃ I A .
    { ( state_sequenceIO A I , WSC_seqOut_clean ( FA_CodeGen A ) I ) } )"
apply ( unfold bisimulation_def , rule ballI , simp , ( erule exE )+ )
      ...( 17 proof steps omitted ) ...
    apply ( auto , simp add: c1 , simp add: c2 )
done
thus ?thesis
```

Figure 6: Main Correctness Theorem

For this particular case we now define a relation by

$$X \sim Y :\Longleftrightarrow \exists A\ I.\ \begin{array}{l} X = out_{WSC}(CodeGen(A), I) \\ Y = out_{FSM}(A, I) \end{array}$$

This definition trivially fullfills requirement (1). Note that the definition of a bisimulation relation is an artificial construct for conducting proofs. Thus, it remains to be shown that for any non-empty input sequence:

(a) *The first output symbols from a state machine and its generated WSC code are equal.* This can be easily shown by symbolically executing the first steps of our semantics definiton for both WSC and FSMs.

(b) *For the remaining output sequence pair, we can find a finite state machine A such that its output is the left entry of the pair and its generated WSC code outputs the right entry.* For A, we can choose the follow configuration $Follow_i(A)$, which is the same state machine with a different initial state, namely the state in which the original machine is in after the symbol i has occurred. By proving some basic properties of the semantics of both FSM and WSC, we have been able to show within Isabelle/HOL that the code generated from $Follow_i(A)$ produces as output the original sequence minus the first symbol.

Some details of our Isabelle/HOL proof are shown in Isabelle/HOL syntax in Figures 5 and 6. They are used for the proof of the final theorem from Figure 6. It proves equality of FSM A and generated code *FA_CodeGen A* by generating two state sequences and defining a bisimulation containing them. Note that Isabelle implicitly quantifies over all automata A and all kinds of input I. The preceeding lemma (Figure 5) shows that the cleaned output of the generated code, i.e. output of the program with all empty actions removed, is the same as the the action emitted by the first step of the state machine, `takeStepIO C i`, followed by the output of the code generated from the follow configuration. This is the core of the proof as described in Figure 6.

5. RELATED WORK

Apart from the work on semantics of Statecharts discussed in Section 2, there is more related work on the verification of code generation techniques. Verified code generation for Statecharts is closly related to compiler verification since one can regard such a code generator as a special compiler.

In the area of compiler verification, a two-fold notion of correctness has been established: One distinguishes between the correctness of the translation algorithm itself and the correctness of its implementation. To ensure the first kind of correctness, the correctness of the algorithm, one needs to verify a given translation algorithm as we have verified the Java code generation algorithm in this paper. For the second kind of correctness, a very promising approach is to use program result checkers [10, 11]. Here one does not verify the code generator itself but only its result. An independent checker takes the source and target program, which would be a Statechart and a Java-like program in the context of this paper, and checks whether they have the same semantics. This may ensure correct code generation for each distinct run of the code generator. This technique has also become known as translation validation [23].

Recently our own work has concentrated on verifying compiler optimizations. In [1], we have verified dead code elimination which is a popular compiler optimization. This work also uses bisimulation to define semantical equivalence of programs. In [2], we introduce a principle to model data dependencies with partial orders in order to ease verification. We hope to reuse this concept to further improve our semantics formalism adequate for transformation verification on Statecharts. Summaries can be found in [8, 9].

It should be noted that languages like the Object Constraint Language (OCL) [19] that are frequently used in the UML context may only be used to formulate certain properties such as invariants and pre- and postconditions of UML specifications. In extension, our approach covers the complete semantics of a (Statechart) specification. Hence it is possible to completely verify code generation instead of validating only certain properties.

6. CONCLUSIONS & FUTURE WORK

In this paper, we have demonstrated that it is feasible to specify and verify the transformation from restricted Statecharts to executable program code within the Isabelle/HOL theorem prover. We have specified the semantics of this restricted set of Statecharts as well as of the target programming langugage. Moreover, we have verified a simple code generation algorithm. For this purpose, we have introduced some basic verification principles like bisimulation and explained how they can be used to verify code generation from UML specifications or transformations on UML specifications themselves.

For our future work, we see a large research potential in two directions. First we want to complete our Statechart and Java language specification, thereby in particular verifying more complex code generation techniques. Secondly, we want to verify transformations on Statecharts themselves.

The completion of the Java semantics should be straightforward since formalizations of the semantics of Java in Isabelle/HOL have become very mature in recent years, see e.g. [15]. The authors of this paper regard the establishment of an adequate Statecharts semantics and its formalization in Isabelle/HOL as the most challenging task. The specification of the code generation technique is also a very complex task, especially when one regards optimizations and parallelism. Another area of future research is the verification of transformations on Statecharts themselves. Formal verification of flattening Statecharts might be an actual task for our very near future work.

To achieve verified code generation in practice, it is not sufficient to only verify the code generation algorithm. The implementation of the algorithm might introduce errors as well, cf. our discussion on compiler verification and checkers in Section 5. There are two different principles to guarantee a correct implementation of the code generation algorithm. One could verify the implementation itself. This seems like a rather tedious task which we believe is not yet feasible for real-life implementations. On the other hand, one can verify a simple program result checker that checks for each run of the code generation mechanism that the generated code is a correct translation of the original Statechart. Such a checker can be much simpler than the original transformation implementation. Hence it is easier to verify. Alternatively, one might even generate such a checker from its specification automatically, also a branch of our ongoing work. Such a checker generator could become part of the Fujaba tool suite. We want to tackle these problems in our future work.

7. REFERENCES

[1] J. O. Blech, L. Gesellensetter, and S. Glesner. Formal Verification of Dead Code Elimination in Isabelle/HOL. In *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods*, Koblenz, Germany, September 2005. IEEE Computer Society Press.

[2] J. O. Blech, S. Glesner, J. Leitner, and S. Mülling. Optimizing Code Generation from SSA Form: A Comparison Between Two Formal Correctness Proofs in Isabelle/HOL. In *Proceedings of the Workshop Compiler Optimization meets Compiler Verification (COCV 2005), 8th European Conferences on Theory and Practice of Software (ETAPS 2005)*, Edinburgh, UK, April 2005. Elsevier, Electronic Notes in Theoretical Computer Science (ENTCS).

[3] M. Born, E. Holz, and O. Kath. *Softwareentwicklung mit UML2 2.* Addison-Wesley, 2004.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.

[5] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24(2):211–237, 1987.

[6] H. Frank and J. Eder. Equivalence transformation on statecharts. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering, SEKE 2000*, pages 150–158, 2000.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] S. Glesner. Verification of Optimizing Compilers, 2004. Habilitationsschrift, Universität Karlsruhe.

[9] S. Glesner and J. O. Blech. Logische und softwaretechnische Herausforderungen bei der Verifikation optimierender Compiler. In *Proceedings der Software Engineering 2005 Tagung (SE 2005).* Lecture Notes in Informatics, März 2005.

[10] S. Glesner, G. Goos, and W. Zimmermann. Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers). *it - Information Technology*, 46:265–276, 2004. Print ISSN: 1611-2776.

[11] W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler Correctness and Implementation Verification: The Verifix Approach. In P. Fritzson, editor, *Poster Session of CC'96*. IDA Technical Report LiTH-IDA-R-96-12, Linkoeping, Sweden, 1996.

[12] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[13] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering Methodology*, 5(4):293–333, 1996.

[14] B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin*, 67:222–259, 1997.

[15] G. Klein and T. Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298:583–626, 2003.

[16] J. Leitner. Coalgebraic Methods in the Verification of Optimizing Program Transformations Using Theorem Provers. Minor Thesis (*Studienarbeit*), University of Karlsruhe, 2005.

[17] G. Lüttgen, M. von der Beeck, and R. Cleaveland. A compositional approach to statecharts semantics. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 120–129, New York, NY, USA, 2000. ACM Press.

[18] A. Maggiolo-Schettini, A. Peron, and S. Tini. Equivalences of statecharts. In *International Conference on Concurrency Theory*, pages 687–702, 1996. Springer-Verlag, LNCS.

[19] O. Object Management Group. OMG Unified Modeling Language Specification, March 2003. Version 1.5.

[20] O. Object Management Group. UML standard 2.0, 2005. available at http://www.uml.org.

[21] L. C. Paulson. A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definitions, 2004. available at www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle2004/doc/ind-defs.pdf.

[22] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *TACS '91: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 244–264, London, UK, 1991. Springer-Verlag.

[23] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, Lisbon, Portugal, April 1998. Springer Verlag, Lecture Notes in Computer Science, Vol. 1384.

[24] A. C. Uselton and S. A. Smolka. A compositional semantics for statecharts using labeled transition systems. In *CONCUR '94: Proceedings of the Concurrency Theory*, pages 2–17, London, UK, 1994. Springer-Verlag, LNCS.

[25] M. von der Beeck. A comparison of statecharts variants. In *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*, pages 128–148, London, UK, 1994. Springer-Verlag.

[26] A. Wasowski. On efficient program synthesis from statecharts. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 163–170, New York, NY, USA, 2003. ACM Press.

[27] A. Zündorf. Rigorous Object Oriented Software Development with Fujaba. Unpublished draft, 2002.