Translating Statecharts to B

Emil Sekerinski and Rafik Zurob

McMaster University, Department of Computing and Software
Hamilton, Ontario, Canada
{emil,zurobrs}@mcmaster.ca

Abstract. We present algorithms for the translation of statecharts to the Abstract Machine Notation of the B method. These algorithms have been implemented in *iState*, a tool for translating statecharts to various programming languages. The translation proceeds in several phases. We give a model of statecharts, a model of the code in AMN, as well as the intermediate representations in terms of class diagrams and their textual counterpart. The translation algorithms are expressed in terms of these models. We also discuss optimizations of the generated code. The translation scheme is motivated by making the generated code comprehensible.

1 Introduction

Statecharts, an extension of finite state diagrams by *hierarchy*, *concurrency*, and *communication* were conceived as a visual formalism for the design of reactive systems [4]. Because of the appeal of the graphical notation, statecharts are now part of object-oriented modeling techniques [2,12,13].

In this paper we present algorithms for translating statecharts to the Abstract Machine Notation of the B method [1]. These algorithms have been implemented in *iState*, a tool for translating statecharts to various programming languages. While *iState* can generate code in various languages, AMN is used as a reference for several reasons: First, AMN supports nondeterminism. Nondeterminism between transitions can arise in statecharts, hence can be reflected directly in AMN. Secondly, AMN supports parallel (independent) composition of statements. This turns out to be essential for the translation of concurrent states. Additionally, invariants can be expressed in AMN, allowing statecharts to be analyzed for safety properties.

Our goal with *iState* is that the resulting code is not only executable, but is also comprehensible. The original motivation is its use for teaching statecharts. However, having comprehensible code allows us to get confidence in the translator and is a prerequisite for the generated code to be further analyzed.

A translation scheme for statecharts into AMN that supports hierarchy, concurrency and communication was proposed in [15]. In [14] the structure of its implementation in *iState* is discussed: a translation in phases is presented and the intermediate representations are formally defined and the notions of representable, normalized, and legal statecharts are introduced, see Figure 1. Normalized statecharts appear as an intermediate representation and code is generated only for legal statecharts. These classes of statecharts are defined in terms of an abstract representation given by class diagrams and in textual form. The refinement of this abstract representation is also discussed.

M. Butler, L. Petre, and K. Sere (Eds.): IFM 2002, LNCS 2335, pp. 128–144, 2002.

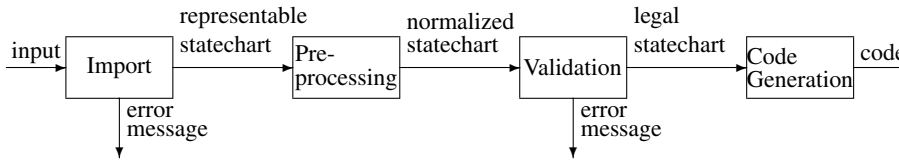


Fig. 1. Phases and intermediate representations of *iState*.

In this paper we present the translation algorithms. The algorithms are expressed in terms of an abstract representation of statecharts [14] and an abstract representation of AMN machines. Both abstract representations are given by class diagrams and in textual form. We also discuss the optimization done during and after the translation. To the best of our knowledge these algorithms are new.

An interpretation of statecharts in B for the purpose of structuring reactive systems was proposed by Lano et al [6]. In this approach concurrent states can only be on the outermost level such that each concurrent state can be mapped to a B machine. We allow arbitrarily composed statecharts and translate the statechart to a single B machine. Nguyen [11] studies the translation of both class diagrams and state transition diagrams to B. This generalizes our approach in specifying a state transition for every object of a class, but does not include nested and concurrent states. Laleau and Mammar [5] describe a tool for translating UML class diagrams, state diagrams, and collaboration diagrams to B. Building on [11], their approach is to define states in a state diagram by predicates over the attributes of an object, i.e. typically statechart states partition the states of objects. Our approach is to represent states directly by variables. Also, they do not mention nested or concurrent states.

Mikk et al. [10] and Lilius and Paltor [7,8] discuss the translation of statecharts to Promela, the input language of the Spin model checker. Promela is related to AMN in the sense that both are extensions of the guarded command language. The translation of Lilius and Paltor relies on a universal algorithm for each step. By comparison we generate code directly for each event. Also, they consider queuing of events according to UML whereas we follow Harel's statecharts in generating and consuming events instantaneously. In order to eliminate inter-level transitions, Mikk et al. use extended hierarchical automata as an intermediate step in the translation. In our translation scheme inter-level transitions do not cause any additional complications. In their translation scheme copies of the pre-state of all variables is kept in order to ensure that a state change is sensed only in the subsequent state. In our translation scheme this is done simply using the parallel composition of AMN (though a refinement of the generated machine may need to introduce such variables). In their translation scheme not only the states but also the events are represented by variables in order to maintain a set of simultaneously generated events. In our approach we do not introduce variables for the events but have the (syntactic) restriction that in any step an event cannot be broadcast simultaneously more than once. Our restriction is motivated by keeping the generated code simple and comprehensible.

Section 2 presents the abstract model of statecharts in terms of class diagrams and in textual form. Section 3 gives the normalization algorithms carried out on that model. Section 4 presents the abstract model of the AMN code in terms of class diagrams and

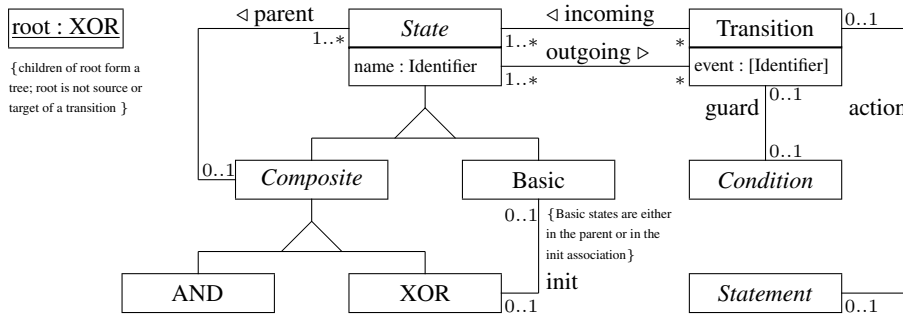


Fig. 2. Representable statecharts defined by a class diagram.

in textual form. Section 5 presents the translation algorithms and Section 6 discusses further processing. We conclude with a discussion in Section 7.

2 The Statechart Model

We define *representable* statecharts in two ways, graphically by the class diagram in Figure 2 and in an equivalent textual form. We follow the presentation of [14] but lift some restrictions. Our model is also related to the model of statecharts in [9]. Besides the difference in style which arises from starting with a graphical model and the fact that we do not consider entry and exit actions, the differences are that we do not introduce configurations and we do not consider sets of simultaneous events.

Let us introduce *Object* to be the set of all objects. The class of states is a subset of objects. Every *State* object has an attribute *name* of type *Identifier*. We let $S \to T$ denote the set of all total functions from S to T.

$$State \subseteq Object$$
$$name \in State \to Identifier$$

States are either composite states or basic states, but no state can be both basic and composite. Furthermore, *State* is an abstract class, meaning that all objects of class *State* must belong to one of its subclasses.

$$Composite \subseteq State \wedge Basic \subseteq State$$
$$Composite \cap Basic = \emptyset \wedge Composite \cup Basic = State$$

Likewise, composite states are either AND states or XOR states, but no state can be both an AND state and an XOR state. The class *Composite* is also abstract.

$$AND \subseteq Composite \wedge XOR \subseteq Composite$$
$$AND \cap XOR = \emptyset \wedge AND \cup XOR = Composite$$

Transitions are also objects. Each transition has an optional attribute *event* of type *Identifier*. Spontaneous transitions have no event name attached to them. We let $S \nrightarrow T$ denote the set of all partial functions from S to T.

$$Transition \subseteq Object$$

$$event \in Transition \nrightarrow Identifier$$

Conditions and statements are objects as well.

$$Condition \subseteq Object \land Statement \subseteq Object$$

The *guard* association relates every transition to at most one condition. Likewise, the *action* association relates every transition to at most one statement. We do not require that every condition and every statement relate to exactly one transition, as conditions and statements may appear as part of other conditions and statements, respectively. We let $S \rightarrowtail\!\!\!\!\!\!\rightarrow T$ denote the set of partial, injective functions from S to T.

$$guard \in Transition \rightarrowtail\!\!\!\!\!\!\rightarrow Condition$$

$$action \in Transition \rightarrowtail\!\!\!\!\!\!\rightarrow Statement$$

The *outgoing* association relates every state to all the transitions leaving it. Any state may have zero or more transitions leaving it but every transition must have at least one state as origin. We let $S \leftrightarrow T$ denote the set of relations from S to T and $ran(R)$ the range of relation R.

$$outgoing \in State \leftrightarrow Transition$$

$$ran(outgoing) = Transition$$

The *incoming* association relates every transition to all the states to which it leads. Any state may have zero or more transitions leading to it but every transition must have at least one state as destination. We let $dom(R)$ denote the domain of relation R.

$$incoming \in Transition \leftrightarrow State$$

$$dom(outgoing) = Transition$$

The *init* association relates an XOR state to at most one basic state, which we call its *init* state. This is the state from which all the initializing transitions are leaving, the destinations of which are the initial states. *Init* states do not appear graphically in the statecharts, or perhaps just as fat dots. They are added here for allowing initializing and proper transitions to be treated uniformly. Not every XOR state must have an *init* state.

$$init \in XOR \rightarrowtail\!\!\!\!\!\!\rightarrow Basic$$

The *parent* association relates states to their parent states, which must be composite states. Every state has at most one parent and every composite state must have at least one child.

$$parent \in State \nrightarrow Composite$$

$$ran(parent) = Composite$$

We define the relation *children* to be the inverse of the function *parent*. We let R^{-1} denote the inverse of relation R.

$$children \mathrel{\widehat{=}} parent^{-1}$$

All basic states are either in the *init* or *parent* association.

$$ran(init) \cup dom(parent) = Basic$$

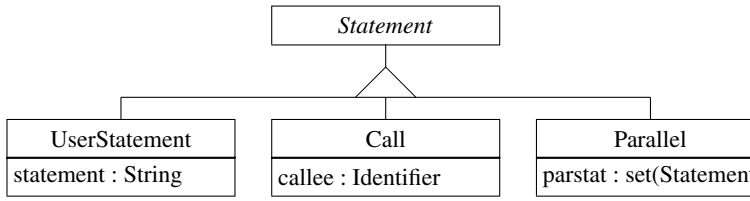


Fig. 3. Statements defined by a class diagram.

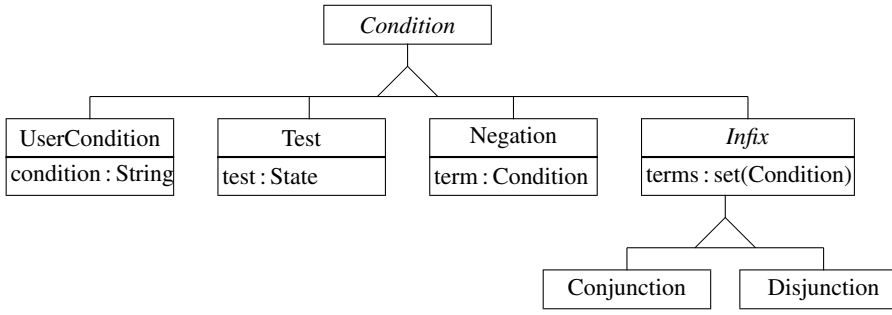


Fig. 4. Conditions defined by a class diagram.

The root state is an XOR state. Every composite state is a descendant of root. We let R^* denote the transitive and reflexive closure of relation R.

$$root \in XOR$$
$$Composite \subseteq children^*[\{root\}]$$

The root state must not be the source or target of a transition.

$$root \notin dom(outgoing)$$
$$root \notin ran(incoming)$$

This completes the textual definition of statecharts. Compared to [14] we do not require that every AND state has two children and all children of AND states are XOR states. While our tool enforces this in order to make the graphical and textual representation interchangeable, our translation algorithms can cope with the more general case. For brevity, we define the conditions of guards and the statements of actions only graphically by the class diagrams in Figures 3 and 4. Statements are either user defined, are broadcasts, or are compositions of statements. Broadcasts are referred to as calls and the composition is referred to as parallel.

3 Preprocessing

Normalization adds those transition arrows to representable statecharts that can be left out. Normalization is the first step to translation. A statechart is normalized if two conditions hold, *targetsProper* and *transitionsComplete*.

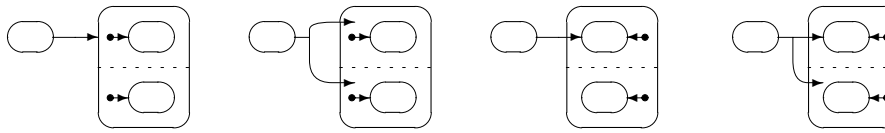


Fig. 5. (a) A statechart that violates *targetsProper*. (b) Normalization of the statechart in (a): the AND target of the transition is replaced by its two XOR children. (c) A statechart that violates *transitionsComplete*. (d) Normalization of the statechart in (c): the XOR child of the AND state that is not entered by the transition is added to its targets.

Targets of transitions must be either Basic or XOR states. If a target is an AND state, then that transition can be replaced by one that forks to all the children of that AND state, see Fig. 5(a).

$$targetsProper \;\widehat{=}\; ran(incoming) \subseteq Basic \cup XOR$$

If an AND state is entered by a transition, then all its children must be entered by that transition as well, see Fig. 5(b). We define the *closest common ancestor* of a set ss of states to be that state that is an ancestor of each state in ss and all other common ancestors are also its ancestor, where each state is also its own ancestor. We let $x\ R\ y$ denote that the pair of x and y is in relation R. For any $ss \subseteq State$ we define $cca(ss)$ by:

$$c = cca(ss) \Leftrightarrow \forall s \in ss \,.\, (c\ parent^*\ s \wedge \forall a \in State \,.\, (a\ parent^*\ s \Rightarrow a\ parent^*\ c))$$

The closest common ancestor exists for any set of states that consists of non-*init* states. The *path* from state s to a set ss of descendants of s is the set of all states that are on the paths from s to a state of ss. Formally, $path(s,ss)$ is defined as those states that are descendants of s and ancestors of states in ss, excluding s but including the states of ss. We let R^+ denote the transitive closure of relation R.

$$path(s,ss) \;\widehat{=}\; children^+[\{s\}] \cap parent^*[ss]$$

The set $to(tr)$ of transition tr is the set of all target states of that transition. Dually, the set $from(tr)$ is the set of all source states of tr:

$$to(tr) \;\widehat{=}\; outgoing^{-1}[\{tr\}]$$
$$from(tr) \;\widehat{=}\; incoming[\{tr\}]$$

Following [3], the *scope* of a transition is the state closest to the root through which the transition passes.

$$scope(tr) \;\widehat{=}\; cca(from(tr) \cup to(tr))$$

The states *entered* by a transition are all the states on the path from the scope of the transition to the targets of the transition. For symmetry, we define the states *exited* by a transition as all the states on the path from the scope of the transition to the sources of the transition.

$$entered(tr) \;\widehat{=}\; path(scope(tr), to(tr))$$
$$exited(tr) \;\widehat{=}\; path(scope(tr), from(tr))$$

This finally allows us to state the requirement that for all states entered by a transition, if the state is an AND state, then all children of that state must be entered by the transition as well. We let $R \rhd S$ denote the restriction of the range of relation R to set S, formally defined as $R; id(S)$.

$$transitionsComplete \;\widehat{=}\; (entered \rhd AND); children \subseteq entered$$

We present an algorithm that makes all targets proper and all transitions complete. The algorithm iterates over all transitions. For each transition, a set *vs* of states to be visited is maintained. This is initially the set of all AND states entered by the transition. Each of these states is replaced by the set *rs* of all its children that are not entered by the transition. This continues until there are no more states to be visited. The children that are added in the replacement step can be either XOR, AND, or Basic states. Of these, the AND states have to be visited as well and are therefore added to *vs*. Assuming that the hierarchy is finite, the algorithm terminates and establishes *targetsProper* and *transitionsComplete*. We let $x :\in e$ denote the nondeterministic assignment of an element of the set e to x. We write $R[S] := T$ for modifying the relation R such that all elements of S relate to all elements of T, formally $R := R \oplus (S \times T)$, where $\oplus$ stands for the relational overwrite.

procedure *normalize*
  for $tr \in Transition$ **do**
    var $vs : set(State)$;
    begin $vs := entered(tr) \cap AND$;
      while $vs \neq \emptyset$ **do**
        var $s : State, rs : set(State)$;
        begin $s :\in vs$; $rs := children[\{s\}] - entered(tr)$;
          $incoming[\{tr\}] := incoming[\{tr\}] - \{s\} \cup rs$;
          $vs := vs - \{s\} \cup (rs \cap AND)$
        end
    end

Besides normalization, the preprocessing step also renames states with the same name but in different parts of the statechart by appending the names of the parent states. As states do not necessarily carry user-defined names, unique names are generated for those.

The validation step, which follows the preprocessing step, detects remaining name conflicts as well as a number of other errors: (1) Transitions must not be between concurrent states. (2) A transition can fork only to concurrent states. (3) A transition can join only from concurrent states. (4) There must not be a cycle in spontaneous transitions. (5) Every *init* state must have a transition leaving it. (6) The *init* transition must go to a child. (7) *init* states cannot be targets of transitions. (8) *init* transitions must not have an event or a guard. (9) No spontaneous transition can leave the target of an *init* transition. (10) Broadcasts don't lead to the same event being generated twice. These are further discussed in [14]. Violating these conditions would either cause the translation to fail or the generated code to be invalid.

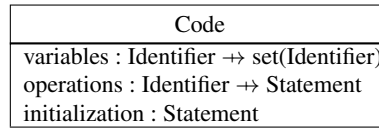


Fig. 6. Generated code defined by a class diagram.

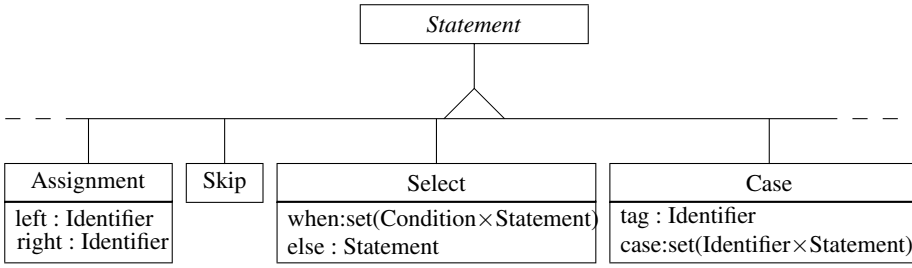


Fig. 7. Statements added to the class for the purpose of code generation.

4 The Code Model

The model of the code is given in Fig. 6. The attribute *variables* defines variables of enumerated set type by mapping variables to their possible values. The attribute *operations* maps operations to their body. The generated operations have no parameters. The initialization is a statement. All identifiers in *variables* and the domain of *operations* have to be distinct. In case the same state name appears in different parts of a statechart, this is resolved by appending the name of the pare nt states, otherwise an error is reported.

In order to model the generated statements, we extend the class of statements by those in Fig. 7. Generated statements may be of all the classes *UserStatement*, *Call*, *Parallel*, *Assignment*, *Skip*, *Select*, and *Case*. In order to model the generated conditions, we extend the class of conditions by the *Equality* as in Fig. 8. Generated conditions may be of the classes *UserCondition*, *Negation*, *Conjunction*, *Disjunction*, and *Equality*, but not of the class *Test*.

5 Translation

Assuming that the statechart is normalized and legal, code is generated for the variables, for the initialization, and for the operations in sequence. We write $x :=$ **new** C for creating a new object of class C. We assume that all classes are subtypes of class *Object*. If C is in addition a subtype of classes $C_1, \ldots, C_n$, then $x :=$ **new** C is defined as $x :\notin Object$; $Object := Object \cup \{x\}$; $C_1 := C_1 \cup \{x\}$; $\ldots$; $C_n := C_n \cup \{x\}$. Hence, we are identifying a class with the set of all objects of that class. For example, $c :=$ **new** *Code* is equivalent to $c :\notin Object$; $Object := Object \cup \{c\}$; $Code := Code \cup \{c\}$.

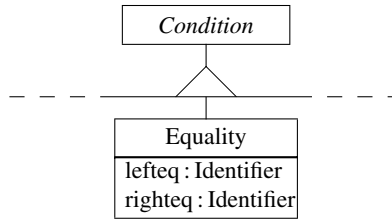
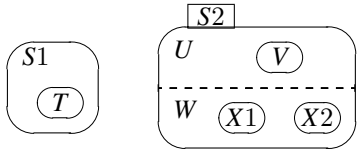


Fig. 8. Condition added to the class hierarchy for the purpose of code generation.

procedure *generate* : *Code*
  var *c* : *Code*
  begin *c* := **new** *Code* ;
    generateVariables(*c*) ;
    c.initialization := *generateInitialization*(*root*) ;
    generateOperations(*c*) ;
    return *c*
  end

For each composed state a variable is created. The values of this variable are the names of its children; we assume that in the preprocessing steps all states have been given a name. As the children may themselves be composed, new names have to be introduced. For example we could have:



$root \in \{S1, S2\}$
$s1 \in \{T\}$
$u \in \{V\}$
$w \in \{X1, X2\}$

The variable declarations are generated by traversing all XOR states and generating a corresponding variable. If *a* is an attribute of an object *x* of class *C*, we write *x.a* instead of $a(x)$.

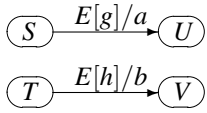
procedure *generateVariables*(*c* : *Code*)
  begin *c.variables* := $\emptyset$;
    for $s \in XOR$ **do**
      $c.variables(lc(s.name)) := UC((children\ ;\ name)[\{s\}])$
  end

The functions $lc(s)$ and $uc(s)$ convert an identifier represented as a string to lower case and upper case, respectively. For our purposes it suffices to assume that $lc(s) \neq uc(s)$ for any identifier *s*. The function $UC(ss)$ converts all strings of the set *ss* to upper case.

For each event in the statechart an operation in the code is generated:

procedure *generateOperations*$(c : Code)$
  begin $c.operations := \emptyset$;
    for $eid \in ran(event)$ **do**
      $c.operations(eid) := generateTransition(eid, root)$
  end

There can be several transitions on the same event, including transitions leaving the same state, leading possibly to nondeterminism. All transitions on the same level are translated to a select statement. For example, assuming that r is the variable for the enclosing XOR state, we generate:

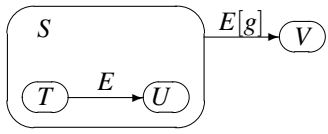


$E \mathrel{\widehat{=}}$ **SELECT** $r = S \wedge g$ **THEN** $r := U \parallel a$
WHEN $r = R \wedge h$ **THEN** $r := V \parallel b$
ELSE *skip*
END

Transitions on outer levels have priority over transitions on the same event in inner levels. The procedure *generateTransitions*(eid, s) first generates the code for transition tr with $scope(tr) = s$ and $event(tr) = eid$ and then recursively code for the children of s.

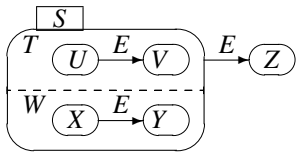
procedure *generateTransitions*$(eid : Identifier, s : State) : Statement$
  var $sel : Select$;
  begin $sel :=$ **new** $Select$; $sel.when := \emptyset$;
    for $tr \in scope^{-1}[\{s\}] \cap event^{-1}[\{eid\}]$ **do**
      $sel.when := sel.when \cup \{(generateGuard(tr, s), generateAction(tr, s))\}$;
    $sel.else := generateChildTransitions(eid, s)$;
    return sel
  end

For generating the code for transitions in children, we have to distinguish the type of the children. If a child is a Basic state, then it cannot contain transitions and no code needs to be generated. If the child is an XOR state, a case analysis needs to be generated. For example, assuming that r is the variable for the enclosing XOR state, we generate:



$E \mathrel{\widehat{=}}$ **SELECT** $r = S \wedge g$ **THEN** $r := V$
ELSE
  CASE r **OF**
    EITHER S **THEN**
      SELECT $s = T$ **THEN** $s := U$
      ELSE *skip*
      END
    OR V **THEN** *skip*
    END
  END
END

Simplifications of the generated code are discussed in Sec. 6. If the child is an AND state, then a parallel composition is generated. In the following example state S is an AND state with two XOR children. Assuming that r is the variable for the enclosing XOR state, we generate:

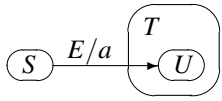



```
E ≙ SELECT r = S THEN r := Z
    ELSE
      CASE t OF
        EITHER U THEN t := V
        OR V THEN skip
        END
      END
      ||
      CASE w OF
        EITHER X THEN w := X
        OR Y THEN skip
        END
      END
    END
```

The procedure *generateChildTransitions*(eid, s) generates code for transitions on event *eid* where the scope of the transition is a child of s.

```
procedure generateChildTransitions(eid : Identifier, s : State) : Statement
  if s ∈ Basic then return new Skip
  else if s ∈ XOR then
    var ca : Case ;
    begin ca := new Case ; ca.tag := name(s) ; ca.cases := ∅ ;
      for cs ∈ children[{s}] do
        ca.cases := ca.cases ∪ {(name(cs), generateTransitions(eid, cs))} ;
      return ca
    end
  else
    var pa : Parallel ;
    begin pa := new Parallel ; pa.parstat := ∅ ;
      for cs ∈ children[{s}] do
        pa.parstat := pa.parstat ∪ {generateTransitions(eid, cs)} ;
      return pa
    end
```

Making a step to a new state requires updating as many variables as XOR and Basic states are entered. In addition, the action associated with the transitions is executed. For example, assuming that r is the variable for the enclosing XOR state, we generate:



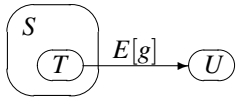

```
E ≙ SELECT r = S THEN
      r := T || t := U || a
    ELSE skip
    END
```

Procedure *generateAction*(tr, s) generates the parallel composition of assignment statements needed to move from the state s to the targets of the transition tr. Using parallel composition ensures that the new state is sensed only in the next step. If an XOR state is a target, the initialization of it is generated recursively. If the transition has an associated action, that action is added to the parallel composition without translation: broadcasting an event in the statechart is interpreted as calling the operation of that event in AMN.

The composition of user defined action and broadcasts in statecharts is interpreted as their parallel composition in AMN.

procedure *generateAction*$(tr : Transition, s : State) : Statement$
  var $pa : Parallel$;
  begin $pa :=$ **new** $Parallel$; $pa.parstat := \emptyset$;
    for $t \in entered(tr) - children[AND]$ **do**
      var $as : Assignment$;
      begin $as :=$ **new** $Assignment$;
        $as.left := lc(name(parent(t)))$; $as.right := uc(name(t))$;
        $pa.parstat := pa.parstat \cup \{as\}$;
        if $t \in XOR \wedge t \in to(tr)$ **then**
          $pa.parstat := pa.parstat \cup \{generateInitialization(t)\}$
      end ;
    if $tr \in dom(action)$ **then**
      $pa.parstat := pa.parstat \cup \{action(tr)\}$;
    return pa
  end

Making a step out of a state requires testing as many variables as XOR and Basic states are exited. In addition, the guard associated with the transition needs to be tested. For example, assuming that r is the variable for the enclosing XOR state, we generate:



$E \mathrel{\widehat{=}}$ **SELECT** $r = S \wedge s = T \wedge g$ **THEN**
  $r := U$
ELSE *skip*
END

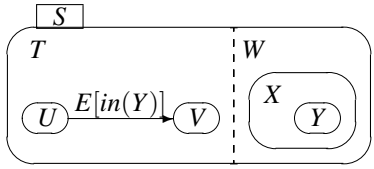
Procedure $generateGuard(tr, s)$ generates the conjunction of state tests needed for transition tr to leave state s. If the transition has an associated guard, that is added to the conjunction.

procedure *generateGuard*$(tr : Transition, s : State) : Condition$
  var $co : Conjunction$;
  begin $co :=$ **new** $Conjunction$; $co.terms := \emptyset$;
    for $t \in exited(tr) - children[AND]$ **do**
      $co.terms := co.terms \cup \{generateTest(t)\}$
    if $tr \in dom(guard)$ **then**
      $co.terms := co.terms \cup \{generateCondition(tr, guard(tr))\}$;
    return co
  end

Procedure $generateTest(s)$ generates the code for testing whether the parent of s is in s.

procedure *generateTest*$(s : State) : Condition$
  var $eq : Equality$;
  begin $eq :=$ *newEquality* ;
    $eq.lefteq := lc(name(parent(s)))$;
    $eq.righteq := uc(name(s))$;
    return eq
  end

A state test requires testing a concurrent state. If that is nested, its parents have to be tested as well, up to (but excluding) the AND state that is the closest common ancestor of the source states of the transition with the state test and the state being tested. However, tests are only necessary for XOR states that are not children of AND states. For example, assuming that r is the variable for the enclosing XOR state, we generate after some simplifications:



$E \mathrel{\widehat{=}}$ **IF** $r = S \wedge t = U \wedge w = X \wedge x = Y$ **THEN**
$t := V$
END

Procedure $generateCondition(tr, c)$ transforms conditions as they appear in statecharts to conditions in AMN. It does so by recursively traversing the condition and only replacing state test with a conjunction of equalities.

procedure $generateCondition(tr : Transition, c : Condition) : Condition$
  if $c \in UserCondition$ **then return** c
  else if $c \in Test$ **then**
    var $co : Conjunction, s : State$;
    begin $co :=$ **new** $Conjunction$; $co.terms := \emptyset$;
      $s := cca(from(tr) \cup \{c.test\})$;
      for $t \in path(s, \{c.test\}) - children[AND]$ **do**
        $co.terms := co.terms \cup \{generateTest(t)\}$;
      return co
    end
  else if $c \in Negation$ **then**
    var $ne : Negation$;
    begin $ne :=$ **new** $Negation$; $ne.term := generateCondition(tr, c.term)$;
      return ne
    end
  else
    var $in : Infix$;
    begin $in :=$ **new** $Infix$; $in.terms := \emptyset$;
      for $d \in c.terms$ **do**
        $in.terms := in.terms \cup \{generateCondition(tr, d)\}$;
      return in
    end

Finally, procedure $generateInitialization(s)$ generates the initialization of an XOR state by first determining the single transition leaving its *init* state.

procedure $generateInitialization(s : XOR) : Statement$
  return $generateAction(outgoing(init(s)), s)$

The restriction to only a single *init* transition can be lifted if instead a nondeterministic assignment is generated.

6 Further Processing

The generated code can in many cases be further simplified. The simplifications include:

- Intra-level transitions may generate code of the form **SELECT ELSE** *S* **END** which is simplified to *S*.
- If there is only a single transition on a level for an event, the generated code of the form **SELECT** *C* **THEN** *A* **ELSE** *B* **END** can be simplified to **IF** *C* **THEN** *A* **ELSE** *B* **END**.
- Case statements can be simplified by leaving out all alternatives with body *skip* and adding **ELSE** *skip* instead.
- Case statements with a single alternative after above simplification can be rewritten as if statements.
- An if statement of the form **IF** *C* **THEN** *A* **ELSE** *skip* **END** can be simplified to **IF** *C* **THEN** *A* **END**.

Currently all the simplifications are done during code generation. Here is the original and simplified code of the example in Sec. 5. It also illustrates how the testing of guards is moved:

```
E ≙ SELECT r = S ∧ g THEN r := V
    ELSE
      CASE r OF
        EITHER S THEN
          SELECT s = T THEN s := U
          ELSE skip
          END
        OR V THEN skip
        END
      END
    END
```

```
E ≙ IF r = S THEN
      IF g THEN r := V
      ELSE
        IF s = T THEN s := U
        END
      END
    END
```

Another example of simplified code is given in [14]. Further optimizations like merging nested if and select statements are left as future work.

AMN does not allow calls of operations within the same machine. As broadcasting of events is translated to calling of operations, auxiliary definitions are generated for the called operations and these auxiliary definitions are "called" instead. For this, the call dependency is analyzed and the operations are first topologically sorted. In case of circular dependencies an error is reported.

Transitions without an event are *spontaneous* as they can be taken without that an event is generated, but may have a guard and action associated. Our code generator implements a run-to-completion of transitions: if a state is reached that has an outgoing spontaneous transition that can be taken it is taken. This is repeated for further spontaneous transitions. Circularities in spontaneous transitions are detected and reported as an error.

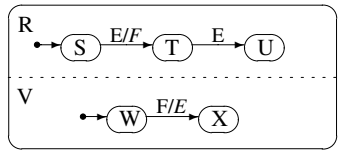
7 Discussion

A number of tools support code generation from statecharts, including xjCharts, with-Class, and Rhapsody. Compared to translation schemes used by other tools, ours can be characterized as *event-centric* rather than *state-centric*, as the main structure of the

code is that of events, rather than classes for states. Our scheme is suitable for those kind of reactive systems where events are processed quickly enough so that no queuing of events is necessary and where blocking of events is undesirable. To our experience so far, the resulting code is not only comprehensible, but compact and efficient as well.

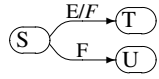
Our semantics of statecharts comes close to that of Mikk et al. [10], with the main difference being that we do not support sets of simultaneous events. Also, we currently do not support event parameters, enter and exit actions, histories, timed events, overlapping states, and sync states. These remain the subject of ongoing work.

An interesting consequence of our use of parallel composition and the restriction to operations that are not mutually recursive is that many problematic situations for which statecharts are known are ruled out. Here is an example with two concurrent states. On event E, one may argue whether state S ends up in T or U. To the right we give the code to which that would correspond. As the code would contain mutual recursion, the statechart is rejected as illegal:



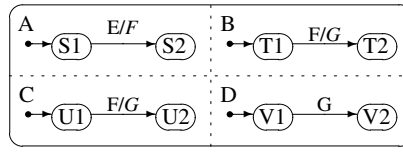

```
E ≙ IF r = noname0 THEN
      CASE r OF
        EITHER U THEN
          skip
        OR T THEN
          r := U
        OR S THEN
          F || r := T
        END
      END
    END
F ≙ IF r = noname0 THEN
      IF v = W THEN
        E || v := X
      END
    END
```

In the following statechart one can argue whether on event E a transition from S to T or to U is made. The translation would contain two assignments to variable r in parallel, which is not allowed.



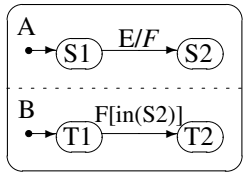

```
E ≙ IF r = S THEN
      r := T || F
    END
F ≙ IF r = S THEN
      r := U
    END
```

In the next example the event G is broadcast twice when the transition on event E is taken. The translation would contain $G \| G$, which is illegal.




```
E ≙ IF r = noname0 THEN
      IF a = S1 THEN
        F || a := S2
      END
    END
F ≙ IF r = noname0 THEN
      IF c = U1 THEN
        G || c := U2
      END
      ||
      IF b = T1 THEN
        G || b := T2
      END
    END
G ≙ IF r = noname0 THEN
      IF d = V1 THEN
        d := V2
      END
    END
```

However, if G would be broadcast only once, the statechart would be valid. Finally, here is an example that is legal and illustrates that the parallel composition leads to state changes being sensed only after the transition. Hence, the transition on event F is not taken.



$$
\begin{array}{lcl}
E & \widehat{=} & \textbf{IF } r = noname0 \textbf{ THEN} \\
 & & \quad \textbf{IF } a = S1 \textbf{ THEN} \\
 & & \quad\quad F \parallel a := S2 \\
 & & \quad \textbf{END} \\
 & & \textbf{END} \\
F & \widehat{=} & \textbf{IF } r = noname0 \textbf{ THEN} \\
 & & \quad \textbf{IF } b = T1 \wedge a = S2 \textbf{ THEN} \\
 & & \quad\quad b := T2 \\
 & & \quad \textbf{END} \\
 & & \textbf{END}
\end{array}
$$

Acknowledgement

We are grateful to the reviewers for their comments.

References

1. J.-R. Abrial. *The B Book: Assigning Programs to Meaning*. Cambridge University Press, 1996.
2. D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1996.
3. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(5):293–333, 1996.
4. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
5. R. Laleau and A. Mammar. An overview of a method and its support tool for generating B specifications from UMl notations. In *15th IEEE International Conference on Automated Software Engineering, ASE 2000*, Grenoble, France, 2000. IEEE Computer Socitey Press.
6. K. Lano, K. Androutsopoulos, and P. Kan. Structuring reactive systems in B AMN. In *3rd IEEE International Conference on Formal Engineering Methods*, York, England, 2000. IEEE Computer Socitey Press.
7. J. Lilius and I. P. Paltor. Formalising UML state machines for model checking. In R. France and B. Rumpe, editors, *UML'99 – The Unified Modeling Language Beyond the Standard*, Lecture Notes in Computer Science 1723, pages 430–445, Fort Collins, Colorado, 1999. Springer-Verlag.
8. J. Lilius and I. Paltor. vUML: a tool for verifying UML models. In *14th IEEE International Conference on Automated Software Engineering, ASE'99*, Cocoa Beach, Florida, 1999. IEEE Computer Socitey Press.
9. E. Mikk, Y. Lakhnech, C. Petersohn, and M. Siegel. On the formal semantics of statecharts as supported by statemate. In *BCS-FACS 2nd Northern Formal Methods Workshop*, Ilkley, 1997. Springer-Verlag.
10. E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing statecharts in Promela / Spin. In *Second IEEE Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, 1998. IEEE Computer Society Press.
11. H. P. Nguyen. *Dérivation De Spécifications Formelles B à Partir De Spécifications Semi-Formelles*. Doctoral thesis, Centre d'Études et de Recherche en Informatique du CNAM, 1998.

12. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddi, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
13. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
14. E. Sekerinski and R. Zurob. iState: A statechart translator. In M. Gogolla and C. Kobryn, editors, *UML 2001 – The Unified Modeling Language, 4th International Conference*, Lecture Notes in Computer Science 2185, Toronto, Canada, 2001. Springer-Verlag.
15. E. Sekerinski. Graphical design of reactive systems. In D. Bert, editor, *2nd International B Conference*, Lecture Notes in Computer Science 1393, Montpellier, France, 1998. Springer-Verlag.