社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

ステートチャートのシンボリックな検証方式について

Jan Philipps　　米田 友洋

東京工業大学 計算工学専攻
〒 152 東京都目黒区大岡山 2-12-1

あらまし　ステートチャートはグラフィカルなモデル化手法の一つで，商用のものも含めいろいろなツールが開発されている．しかし，その多くのは非常に簡単な意味モデルを用いているため，ステートチャートの記述が複雑になったり，状態遷移条件に否定イベントを含むステートチャートの振る舞いが曖昧になるという問題が生じている．そこで本稿では，まずステートチャートのより厳密な意味モデルを提案する．この意味モデルでは，一つの外部イベントに対するステートチャートの一動作 (マクロステップ) をいくつかの状態遷移 (マイクロステップ) の列で定義し，論理プログラミングにおける negation as failure の手法を用いて否定イベントを取り扱う，次に，その意味モデルを 2 分決定グラフ (BDD) を用いてシンボリックに解析する方法を示し，簡単な例において実験結果を示す．

和文キーワード　　ステートチャート，検証，2 分決定グラフ，negation as failure

Symbolic Verification of Statecharts

Jan Philipps　　Tomohiro Yoneda

Department of Computer Science, Tokyo Institute of Technology
2–12–1 Ookayama Meguro-ku Tokyo 152, Japan

Abstract　We present an approach for the verification of Statechart specifications. In contrast to other work on Statecharts verification, our approach satisfies the synchrony assumption and handles trigger expressions with negations in a satisfactory way. Our semantic model uses two transition relations. A macrostep relation describes the observable behaviour of the statechart. It is defined through a microstep relation, which describes the chainreactions of transitions. By employing the negation as failure model from logic programming for negated events in trigger expressions, we ensure that our model is globally consistent. We built a prototype tool based on BDDs, and tested it on a small example.

英文 key words　Statecharts, Verification, Binary Decision Diagrams, Negation as Failure

1 Introduction

The Statecharts language [4] is a visual formalism for the specification of reactive systems. It was introduced in the mid 1980s for the specification of avionic systems. The original proposal introduced mainly the graphical constructs of the language, and presented only an informal semantics. There have been several suggestions for formal semantics [3, 9], but they turned out to be quite complicated. Maybe for this reason the currently most popular commercial CASE tool for Statecharts, STATEMATE [2], as well as the existing model checking tools [1, 6] use a very simple operational model that differs from the ideas outlined in [4].

In this work, we demonstrate how to use a semantic model that is more faithful to [4] for efficient verification. Our approach is twofold. We model the chainreactions of transitions causing other transitions with an embedded transition relation. The rechable, terminal states of this transition relation form the outer transition relation that describes the observable behaviour of a statechart. To properly handle negated events in the trigger of a transition arrow, the embedded transition relation branches according to guesses whether the events will be generated in later in the transition sequence. We show how to implement these ideas using BDDs for symbolic verification.

In the next section, we define the Statechart variant that our model can handle. In Section 3, we informally describe the operational semantics of a statechart and give an overview over some previous formal models. In Section 4 we describe our approach and in Secction 5 we show how it can be used for BDD based verification. Section 6 gives an example and Setion 7 concludes.

2 Statecharts

This section defines the Statecharts variant of our model. The definitions are based on the formal definition in [9]. As our running example, we take the binary stopwatch from the same paper (Figure 1).

A Statechart is a tuple $\langle B, S, h, t, \Delta, A\rangle$, where

- B is a finite set of basic events.
- S is a finite set of state names.
- $h \in S \to 2^S$ is the *hierarchy function*. It defines for each state the set of children of that state. For example, in Figure 1 we have $h(StopWatch) = \{On, Off\}$.

  The hierarchy function describes a tree:

  - There is exactly one state $r \in S$ which has no parent: $\forall s \in S : r \notin h(s)$. This is the root state of the statechart.
  - If $s \neq s'$ then $h(s) \cap h(s') = \emptyset$.
- $t \in S \to \{PRIM, AND, OR\}$ is the *type function*. It gives for each state its type: primitive state, AND-state, or OR-state.
  - *Primitive states* have no children: $t(s) = PRIM$ iff $h(s) = \emptyset$. In the example, $t(Off) = PRIM$.
  - *OR-states* contain a finite state machine with one or more substates (each of which may again be either primitive or complex); the substates are connected by transitions arrows. In the example, $t(Stopwatch) = OR$.
  - *AND-states* consist of two or more substates graphically separated by dashed lines. They introduce concurrency. In the example, $t(On) = AND$.
- $\Delta \in S \to 2^S$ is the *default state function*. It gives the default substates of a state. The function Δ satisfies the following requirements for all states s:
  - $\Delta(s) \subseteq h(s)$.
  - If $t(s) = AND$, then $\Delta(s) = h(s)$.
  - If $t(s) = OR$, then $\Delta(s) = \{s'\}$, where s' is graphically marked with a small arrow pointing to the state.

  In the example, $\Delta(Stopwatch) = \{Off\}$.
- A is a finite set of *arrows*. An arrow a is a tuple

  $$\langle Source, Dest, Trigger, Action\rangle,$$

  where
  - $Source \subseteq S$ is a nonempty orthogonal set of source states. Informally, this means that the statechart can be in all of the source states at the same time. We will define orthogonality below.
  - $Dest \subseteq S$ is a nonempty, orthogonal set of destination states.
  - $Trigger$ is an expression built from the events in E using negation, conjunction and disjunction. The trigger serves as a condition when the arrow may be taken.
  - $Action \subseteq B$ is a possibly empty set of events generated when this transition arrow is taken.

We define the event set E to contain all basic events from B, and for each state an event that will be generated when the state is entered or left.

$$\begin{aligned} E \;=\; & B \cup \{\, enter(s) \mid s \in S \,\} \\ & \cup \{\, leave(s) \mid s \in S \,\} \end{aligned}$$

We write h^+ for the transitive (and h^* for the reflexive-transitive) closure of h. For two states $s, s' \in S$ we say that

- s is an *ancestor* of s' (s' is a *descendant* of s), iff $s' \in h^*(s)$. Using h^+ instead of h^*, we get the notion of a *strict* ancestor (descendant).
- s is *ancestrally related* to s', if one is an ancestor of the other.

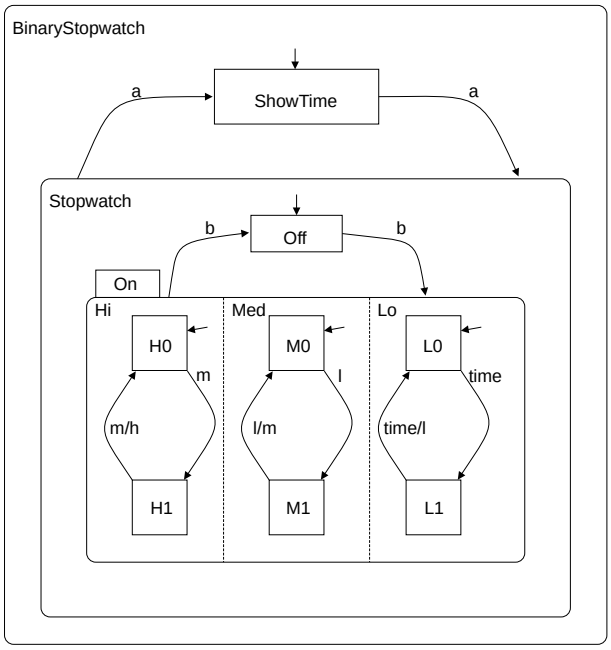

Figure 1: Binary Stopwatch

We define $anc(s) = \{\ t \mid s \in h^*(t)\ \}$ as the set of ancestors of s.

For a set $T \subseteq S$ of states, the *least common ancestor* $lca(T)$ of T is the unique state s such that:

- $T \subseteq h^*(s)$,
- for all $s' \in S$: if $T \subseteq h^*(s')$ then $s \in h^*(s')$.

The *scope* of an arrow

$$a = \langle Source, Dest, Trigger, Action \rangle$$

is defined as

$$scope(a) = lca(Source \cup Dest).$$

Arrows may not connect substates of an AND-state: we require that $t(scope(a)) = OR$ for all arrows $a \in A$.

If $t \in h^*(s)$, the *child of s containing t*, written $cont(s,t)$ is the unique state u such that $u \in h(s)$ and $t \in h^*(u)$.

Two states s, s' are *orthogonal* (written $s \perp s'$), if they are not ancestrally related, and their least common ancestor is an AND-state. When two states are orthogonal, the statechart can be in both of them at the same time. A set T of states is orthogonal, if for every $s, s' \in T$ either $s \perp s'$ or $s = s'$.

A set $T \subseteq S$ of states is *consistent*, if for all $s, s' \in T$ either s and s' are ancestrally related, or $s \perp s'$.

T is *maximally consistent*, if for every state $s \in S \setminus T$, $T \cup \{s\}$ is not consistent.

A *configuration* is a maximally consistent set of states. A configuration contains for each state s all states above s in the hierarchy. If $t(s) = OR$, then it contains exactly one child of s, if $t(s) = AND$, it contains all children of s.

The *default completion* of a consistent set T, $comp(T)$, is the unique configuration T' with $T \subseteq T'$ such that for all $s \in T'$ with $t(s) = OR$, if $T \cap h(s) = \emptyset$, then $\Delta(s) \in T'$. This means that T is extended in such a way that for each OR-state s in the configuration T' its default substate is also in T', unless T already contains some other child of s.

The *initial configuration* of a statechart with root state r is $comp(\{r\})$. In the example, we have the initial configuration $\{BinaryStopwatch, ShowTime\}$.

3 Execution model

A run of a statechart is a sequence

$$c_0 \xrightarrow{e_0/g_0} c_1 \xrightarrow{e_1/g_1} c_2 \xrightarrow{e_2/g_2} \dots$$

of configurations c_i, sets $e_i \subseteq B$ of input events, and sets $g_i \subseteq B$ of generated events, where:

- c_0 is the initial configuration of the statechart.
- Each e_i contains events provided by some external environment of the specified system.
- Each g_i contains the events from e_i as well as any events from B generated by the arrows that were taken in the step from c_i to c_{i+1}.

We do not make any restrictions on the environment, and allow all subsets of B as input. It is possible to model a more restricted environment with temporal logic formulas, another statechart, or by simply giving a formula that characterizes possible subsets of B.

Statecharts semantics differ in how they define the relationship between c_i, e_i, g_i and c_{i+1}.

The CASE tool STATEMATE [2] uses a very simple model: at each step $c_i \xrightarrow{e_i/g_i} c_{i+1}$, a maximal set of enabled, nonconflicting transition arrows is followed. The

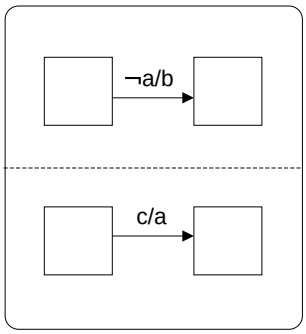

Figure 2: Global consistency problem

events generated are treated as part of the input of the next step, i.e. $g_i \subseteq e_{i+1}$. The problem with this approach can be seen in the example in Figure 1: Assume the current configuration is { H1, M1, L1, ... }, and the external input consists of just the event *time*. We would expect the next configuration to be of { H0, M0, L0, ... }, but the STATEMATE semantics yields the configuration { H1, M1, L0, ... }, and generates l. Assuming that the external input is $\emptyset$ from now on, $M0$ would be entered and m generated, and finally, in the third step, $H0$ would be entered. Therefore, instead of jumping from 111 to 000, the semantics leads to the sequence 111, 110, 100, 000. Because of its simplicity and its commercial importance, this is also the model used in the model checkers in [1, 6]. To get the proper behaviour for the stopwatch example, one would have to add more states and synchronizing events.

A more sophisticated approach, and closer to the ideas of [4], is to regard each step $c_i \xrightarrow{e_i/g_i} c_{i+1}$ in the execution of a statechart as a *macrostep* that consists of a sequence of *microsteps* [3]

$$c_i \xrightarrow{e_i^0/g_i^0} \bullet \xrightarrow{e_i^1/g_i^1} \bullet \ldots \bullet \xrightarrow{e_i^k/g_i^k} c_{i+1},$$

where $e_i^0 = e_i$, $e_i^j \subseteq g_i^j$, $e_i^{j+1} = g_i^j$, and $g_i = g_i^k$. Each microstep corresponds to taking a transition arrow. An arrow can be taken if the trigger expression is satisfied by the input set e_i^j, and the source of the arrow is in c_i. The arrow then generates the events of its action set. They are added to the input set, yielding g_i^j. This is also the input set e_i^{j+1} for the next microstep, which can trigger other arrows. When there are no more arrows that can be taken, the new configuration is computed according to the destination states of all arrows that were taken, and g_i is equal the union of e_i and all generated events. Conceptually it is assumed that this sequence takes no time to execute: this is called the *synchrony assumption*. Of course, in a implementation of a statechart, the sequence will not execute in zero time. For the assumption to hold, it is only necessary that no new input from the environment arrives during the sequence.

The problem with this approach can be seen in Figure 2. If the external input contains c but not a, should the reaction be that both arrows are taken (the upper arrow was followed first), yielding event set $\{a, b, c\}$, or only the second one (the lower arrow was followed first), yielding event set $\{a, c\}$? This semantics leads to very nondeterministic behaviour.

Finally, in [9] a *globally consistent* model was suggested. Whereas in [3] in each microstep the trigger conditions are evaluated with respect to the set of input events and the events generated in previous microsteps (g_i^j), here they are evaluated with respect to the events present in the complete macrostep (g_i). If there are no negated trigger events, the models from [3] and [9] coincide. However, the model in [9] is rather complicated, and it is not obvious how to apply it for verification tools.

Our work is related to [9], but simpler: in particular, we do not need to build sets of enabled arrows, but can talk directly about configurations and events. Another difference is that in [9] $enter(s)$ and $leave(s)$ events are generated only after the last microstep, and global consistency does not hold for them. In our work, these events are implicitly generated when following a transition arrow, and in trigger expressions they are treated in the same way as the basic events. In particular, this ensures that an arrow $\langle \{s\}, \{t\}, \{e \wedge \neg enter(t)\}, \emptyset \rangle$ can never be taken. Finally, our approach is aimed at symbolic verification.

4 Proposed Semantic Model

We separate the development of our semantics into two parts. First we show how the microsteps can be implemented using an embedded transition relation, then we describe how to handle negated trigger events in a globally consistent way.

For simplicity, we make the following assumptions. We assume that the source and destination sets of each arrow are disjunct. This restriction will be removed later. As a further simplification, we assume that the trigger is just a set of events $e \in E$, or negated events $\neg e, e \in E$; we regard the elements of this set to be implicitly conjoined. Since a trigger expression can be written in disjunctive normal form, and an arrow with more than one disjunct in the trigger can be replaced by a set of arrows with a single disjunct and identical source, destination, and action set, this is no real restriction.

Microsteps. The microstep sequence defining the macrostep $c_i \xrightarrow{e_i/g_i} c_{i+1}$ can be defined by an embedded transition relation. We use intermediate configurations c_i^j for the points in the sequence, and sets e_i^j for the set of input events and generated events. A microstep is then defined by a predicate $R_{MIC}(c_i, e_i, c_i^j, e_i^j, c_i^{j+1}, e_i^{j+1})$, where $e_i^j \subseteq e_i^{j+1}$.

In each microstep an arrow is taken. Consider an arrow

$$a = \langle Source_a, Dest_a, Trigger_a, Action_a \rangle,$$

where $Trigger_a$ is a set of events (negated events are introduced below).

The arrow can be taken, if the current configuration c_i^j contains the source states $Source_a$, and the trigger is enabled. In detail, we require that:

- $Source_a \subseteq c_i^j$;
- $Trigger_a \cap B \subseteq e_i^j$ (the basic events are in the current event set);
- for all events $enter(s) \in Trigger_a : s \in c_i^j \wedge s \notin c_i$ (the state s has been entered in a previous microstep of the sequence);
- for all events $leave(s) \in Trigger_a : s \in c_i \wedge s \notin c_i^j$ (the state s has been left in a previous microstep).

This is not quite enough: consider the state Lo in Figure 1. If we start in $L0$ with event *time*, we could take the arrow to $L1$, and in the next microstep take the second arrow back to $L0$. This behaviour is not intended, and we therefore need to add the requirement

- $Source_a \subseteq c_i$.

If the arrow a is taken, it will have the following effects. The scope of an arrow is the lowest OR-state in the hierarchy that is not entered or left by taking a. The arrow causes the child of the scope containing the source states and its descendants to be left. The child of the scope containing the destination states will be entered. Here we have to make sure the proper default states among its descendants are entered, if the destination does not already prescribe which substate of an OR-state to enter.

Therefore, we obtain the next configuration and event set as follows:

- Let $C_s = cont(scope(a), lca(Source_a))$ be the child of the scope with the source states, and $C_d = cont(scope(a), lca(Dest_a))$ the child containing the destination states. Then the states in $h^*(C_s)$ are left, and the states in $h^*(C_d) \cap comp(\{C_d\} \cup Dest_a)$ are entered. This gives the next configuration c_i^{j+1}.
- The actions in $Action_a$ are added to e_i^j. This becomes the new event set e_i^{j+1}.

Given the microstep relation R_{MIC} we can define the macrostep relation R_{MAC}. A step $c_i \xrightarrow{e_i/g_i} c_{i+1}$ is in R_{MAC}, iff there is a sequence

$$c_i^0 \xrightarrow{e_i^0/e_i^1} c_i^1 \xrightarrow{e_i^1/e_i^2} c_i^2 \ldots \xrightarrow{e_i^k/e_i^{k+1}} c_i^{k+1},$$

such that

- $c_i^0 = c_i$ and $c_i^{k+1} = c_{i+1}$
- $e_i^0 = e_i$ and $g_i = e_i^{k+1}$
- for all $0 \leq j \leq k$: $R_{MIC}(c_i, e_i, c_i^j, e_i^j, c_i^{j+1}, e_i^{j+1})$
- there are no c', e' such that: $R_{MIC}(c_i, e_i, c_i^{k+1}, e_i^{k+1}, c', e')$

The first two conditions state that the macrostep is consistent with the endpoints of the sequence, the third condition states that the sequence indeed consists of microsteps, and the fourth conditions requires that the sequence is maximal.

Global consistenccy. Global consistency leads to nonmonotonicity: if an arrow is taken in a microstep, the events it generates can violate the trigger expression of an arrow taken in a previous microstep.

A similar situation occurs in the *negation as failure* model in logic programming, where an atom is considered to be false, if it can not be proven by the clauses in the program. In the theoreom prover MGTP [7, 5] this model is implemented through *case splitting* on atoms that occur negatively in clauses. In one case, the atom is assumed to be true, in the other to be false. Solutions for the program have to fulfill not only the program's clauses, but also some additional constraints: an atom that is assumed to be false may neither be assumed to be true through another case splitting later in the search, nor may it finally be proven to be true. In addition, all atoms that are assumed to be true have to be later proven to be true.

We incorporate this idea in our microstep relation. The trigger of the arrow

$$a = \langle Source_a, Dest_a, Trigger_a, Action_a \rangle$$

can be partitioned into positive events $P = Trigger_a \cap E$ and negative events $N = Trigger_a \setminus P$. When at an intermediate configuration c_i^j of the microstep sequence the arrow

$$\langle Source_a, Dest_a, Pos, Action_a \rangle$$

would be enabled, the microstep relation will branch according to the following cases:

- no event e with $\neg e \in N$ will be generated in this microstep; neither is it a member of the input events. Then the arrow can be taken.
- at least one event e with $\neg e \in N$ will be generated or is a member of the input events. Then the arrow will not be taken.

We then allow only those microstep sequences, that are consistent (an event may not be predicted to be absent, while being present or predicted to be present) and self-fulfilling (events predicted to be generated have to be generated, if they are not already in the input event set).

5 Symbolic Verification

In this section, we show how to implemement our model using BDDs. First we discuss the variable encoding, then we build the microstep relation. Finally, we show how the macrostep relation is constructed out of the microstep relation.

5.1 Variable encoding

Events. We encode each event $e \in B$ with a variable expressing the event's presence or absence. For simplicity, we call this variable e as well. The enter and leave events in $E \setminus B$ are handled implicitly, and need no extra variables, as we will see below.

States and configurations. We need to be able to express the following predicates for each state $s \in S$ and arrow $a \in A$:

- $inside(s)$: the state s is a member of the configuration.
- $outside(s)$: the state s is not in the current configuration. This must imply that none of the children of s is in the configuration.
- $entering(a)$: characterizes the set of states entered when the arrow a is taken.
- $leaving(a)$: characterizes the set of states left when arrow a is taken.

We encode a configuration using a variable for each state $s \in S$. Again, we just write s for this variable. Note that this encoding is not particularly compact: it would be sufficient to use only the primitive states. An even better encoding is presented in [6].

Using this encoding, the predicates *inside* and *outside* can be defined as:

- $inside(s) \equiv anc(s)$,
- $outside(s) \equiv \bigwedge_{t \in h^*(s)} \neg t$,

Given an arrow

$$a = \langle Source_a, Dest_a, Trigger_a, Action_a \rangle,$$

let $C_s = cont(scope(a), lca(Source_a))$, and $C_d = cont(scope(a), lca(Dest_a))$ as defined in Section 4. Then we define the predicates *leaving* and *entering* as:

- $leaving(a) = h^*(C_s)$
- $entering(a) = h^*(C_d) \cap comp(\{C_d\} \cup Dest_a)$

Priming. As we have seen in the last section, the microsteps not only express a relation between consecutive intermediate points, but also depend on the first point in the sequence. In contrast to standard transition relations, which just have a pre-state and a post-state, we have three set of variables, and use the following convention. For each variable v, v^* denotes its value at the beginning of the microstep sequence, v its current value, v' its value at the next point in the microstep sequence. We use this convention also for predicates, and write e.g. $inside(s)^*$.

Implicit events. We can define the events corresponding to entering and leaving a state as follows:

$$\begin{aligned} enter(s) &\equiv inside(s) \wedge outside(s)^*, \\ leave(s) &\equiv inside(s)^* \wedge outside(s). \end{aligned}$$

In the sequel, whenever we refer to an event e, we mean either the variable e if e is a basic event, or otherwise one of the predicates above.

5.2 Microstep relation

The microstep relation has the form

$$\Phi_{MIC}(c^*, e^*, c, e, c', e'),$$

where c^*, e^* are the values of the state and event variables at the beginning of the microstep sequence, c, e are their current values, and c', e' will be their values at the next point in the microstep sequence.

Arrow translation. The microstep relation is built up from transitions $\Phi_a(c^*, e^*, c, e, c', e')$ for each arrow $a \in A$. From now on, we consider a given arrow

$$a = \langle Source_a, Dest_a, Trigger_a, Action_a \rangle.$$

Using the predicates from Section 5.1, we can encode the enabling condition and the effect of the arrow as

$$\begin{aligned} En_a(c^*, e^*, c, e, c', e') \equiv \\ \bigwedge_{s \in Source_a} (inside(s) \wedge inside(s)^*) \wedge \\ \bigwedge_{e \in Trigger_a} e, \end{aligned}$$

$$\begin{aligned} Ef_a(c^*, e^*, c, e, c', e') \equiv & \bigwedge_{e \in Action_a} e' \wedge \\ & \bigwedge_{s \in entering(a)} s' \wedge \\ & \bigwedge_{s \in leaving(a)} \neg s' \wedge \\ & \bigwedge_{v \in UC} v = v'. \end{aligned}$$

where UC is the set of all variables that don't appear primed in any of the other four conjuncts. The last conjunct states that all variables that are not restricted by one of the previous conjuncts remain unchanged.

The predicate Φ_a is then defined as

$$\begin{aligned} \Phi_a(c^*, e^*, c, e, c', e') \equiv \\ En_a(c^*, e^*, c, e, c', e') \wedge \\ Ef_a(c^*, e^*, c, e, c', e'). \end{aligned}$$

For example, the arrow from state $ShowTime$ to state $Stopwatch$ in Figure 1 is translated into the formula

$$\begin{aligned} \Phi \equiv & \; ShowTime \wedge BinaryStopwatch \wedge \\ & ShowTime^* \wedge BinaryStopwatch^* \wedge a \wedge \\ & \neg ShowTime' \wedge BinaryStopwatch' \wedge \\ & Stopwatch' \wedge Off' \wedge \\ & time = time' \wedge m = m' \wedge \ldots \end{aligned}$$

We abbreviated the last conjunct; it contains an equality for all variables that don't occur primed in the first four lines.

Finally, if there are no negated trigger events, the microstep relation is defined as the disjunction of the individual arrow relations:

$$\Phi_{MIC} \equiv \bigvee_{a \in A} \Phi_a.$$

Negated events. Let $Neg \subseteq E$ be the set of all events that occur negatively in the trigger of any arrow $a \in A$. We introduce two new variables $\mathsf{K}e$ and $\mathsf{N}e$ for each $e \in Neg$. Informally, $\mathsf{K}e$ means that e is believed to be produced at some later microstep, while $\mathsf{N}e$ means that e is believed to be absent during the complete macrostep.

Assume that the trigger expression includes k negated trigger events:

$$Trigger_a = \{\neg e_1, \ldots, \neg e_k\} \cup Pos,$$

where $\{e_1, \ldots, e_k\} \subseteq Neg$ and $Pos \subseteq E \setminus Neg$.

We construct an arrow $\bar{a}$ via

$$\bar{a} = \langle Source_a, Dest_a, Pos, \{\mathsf{N}e_1, \ldots, \mathsf{N}e_k\} \cup Action_a \rangle.$$

This arrow is translated to a predicate $\Phi_{\bar{a}}$ as shown above. The events $\mathsf{N}e_i$ state that the negative trigger events are assumed not to be generated in the microstep sequence. Alternatively, we could assume that any of them will be generated. Then the arrow can not be taken. We express this with the predicate

$$\Psi_i(c, e, c', e') \equiv (\mathsf{K}e_i)' \wedge \bigwedge_{v \in UC_i} v = v',$$

where UC_i is the set of all variables except $\mathsf{K}e_i$.

The relation corresponding to an arrow a with negated trigger events is then the disjunction of these $k+1$ cases:

$$\begin{aligned} \Phi_a(c^*, e^*, c, e, c', e') \quad \equiv \quad & \Phi_{\bar{a}}(c^*, e^*, c, e, c', e') \vee \\ & \bigvee_{1 \le i \le k} \Psi_i(c, e, c', e'). \end{aligned}$$

A macrostep is inconsistent if an event is believed to occur and believed to be absent at the same time, or when it has already been generated although it is believed to be absent:

$$\Phi_{CONS}(c, e) \equiv \bigwedge_{a \in Neg} \neg(\mathsf{K}a \wedge \mathsf{N}a) \wedge \neg(a \wedge \mathsf{N}a).$$

Again, the microstep relation is built out of the disjunction of the individual arrow relations, but we make sure that the next point in the microstep sequence is consistent:

$$\begin{aligned} \Phi_{MIC}(c^*, e^*, c, e, c', e') \quad \equiv \quad & \bigvee_{a \in A} \Phi_a(c^*, e^*, c, e, c', e') \wedge \\ & \Phi_{CONS}(c', e'). \end{aligned}$$

We must also make sure that the microstep sequence if *self-fulfilling*, i.e. that all events assumed to be true (by the $\mathsf{K}e$ events) are generated in the microstep sequence. This condition is included in the definition of the macrostep relation below.

Note that while the enter and leave events are defined implictly, we still need to introduce the belief variables $\mathsf{K}enter(s)$, $\mathsf{N}enter(s)$, $\mathsf{K}leave(s)$ and $\mathsf{N}leave(s)$, if they occur negatively in the trigger of an arrow.

5.3 Macrostep Relation

The macrostep relation can be expressed as a predicate

$$\Phi_{MAC}(c, e, c', e'),$$

where c and c' represent the current and next configuration, respectively, e is the set of input events, and e' contains the events from e and the events generated in this step.

Given the relation Φ_{MIC} from the last section, we construct Φ_{MAC} in the following way. First, we can define the set of initial states of a microstep sequence by

$$\begin{aligned} \Phi_{INIT}(c, e, c', e') \quad \equiv \quad & (c = c' \wedge e = e') \wedge \\ & \bigwedge_{e \in Neg} (\neg Ke \wedge \neg Ne \wedge \neg Ke' \wedge \neg Ne'). \end{aligned}$$

A microstep sequence ends, when no more arrows are enabled, and there is no successor state in the microstep relation:

$$\Phi_{TERM}(c, e, c', e') \equiv \neg \exists c'', e'' : \Phi_{MIC}(c, e, c', e', c'', e'').$$

We construct the set of intermediate points in the microstep sequence by the μ-calculus formula

$$\begin{aligned} \Phi_{INT}(c, e, c', e') \equiv \mu\Psi.[\Phi_{INIT}(c, e, c', e') \vee \\ \exists c'', e'' : \Psi(c, e, c'', e'') \wedge \Phi_{MIC}(c, e, c'', e'', c', e')]. \end{aligned}$$

As stated above, the macrostep has to be self-fulfilling:

$$\Phi_{SF}(c, e) \equiv \bigwedge_{a \in Neg} \mathsf{K}a \Rightarrow a$$

Of course, if there are no negated trigger events, Φ_{SF} is trivially true for all c and e.

Finally, we hide the belief atoms $\mathsf{K}e$ and $\mathsf{N}e$ to define the macrostep relation. If $Neg = v_1, \ldots, v_k$, then

$$\begin{aligned} \Phi_{MAC}(c, e, c', e') \quad \equiv \quad & \exists \mathsf{K}v_1, \mathsf{K}v_1', \mathsf{N}v_1, \mathsf{N}v_1', \ldots, \mathsf{N}v_k, \mathsf{N}v_k' : \\ & \Phi_{INT}(c, e, c', e') \wedge \\ & \Phi_{TERM}(c, e, c', e') \wedge \\ & \Phi_{SF}(c', e'). \end{aligned}$$

We can then use the macrostep relation to calculate the set of reachable configurations of a statechart with another μ-calculus formula. The initial configuration is encoded by a predicate $\Phi_{IC}(c)$ that is true iff $c = comp(\{r\})$, where r is the root state of the statechart. The set of reachable state is then defined as:

$$\begin{aligned} \Phi_R(c) \equiv \mu\Psi.[\Phi_{IC}(c) \vee \\ \exists c', e, e' : \Psi(c') \wedge \Phi_{MAC}(c', e', c, e)]. \end{aligned}$$

Here we just assume the existence of arbitrary set e and e' for the generated events and the input events respectively. The simplest way to include some assumptions on the environment of the statechart would be to conjoin some predicate $Env(e')$.

5.4 Extensions

Priorities. In the microstep relation, all arrows are assumed to have the same priority. Intuitively, however, it might seem reasonable to assume that for instance in Figure 1 the arrows label with a should have higher priority than the arrows in the states Hi, Med and Lo. We can introduce this priority by conjoining $\neg leave(On)$ to the trigger of the transition arrows. This can be done automatically, if we want it to be the standard behaviour of a statechart.

Loops. We required that the source and destination sets of each arrow are disjunct, because in the case of a loop it is not possible to determine that the arrow has been taken by comparing the *init* and *pre* variables. To allow loops from state s, we simply add a new variable $exit(s)$, and in the translation of the arrow, we conjoin $\neg exit(s)$ and $exit(s)'$ to the transition relation of the arrow. Note that the negation here is at the logical level; it does not undergo the transformation from section 5.2. The generation of the events $enter(s)$ and $leave(s)$ will depend on the variable $exit(s)$ as well.

6 Experimental results

We have implemented a prototype tool that translates a statechart description into input of the μ-calculus verifier by Janssen [8]. On the stopwatch example, we get the following statistics to calculate the macrostep relation Φ_{MAC} and the reachable configurations Φ_{R0}:

	Nodes	Time
Φ_{MIC}	490	3.0
Φ_{INIT}	60	3.0
Φ_{INT}	2379	116.6
Φ_{MAC}	966	8.3
Φ_{IC}	15	5.3
Φ_{R0}	28	5.3

All times are in seconds, measured on a Sun 20. Especially the time to calculate the fixpoint of the microstep relation Φ_{INT} (9 iterations) seem to be rather long. We assume that this is caused by the BDD package we employed, which introduces some overhead through the direct support of μ-calculus operations. The package also does automatic variable reordering to reduce the size of intermediate results.

7 Conclusion

We have presented a globally consistent semantic model for Statecharts, and shown how to implement efficient verification tools using symbolic techniques for this model. We believe that specifications in globally consistent semantics can be more concise than in the STATEMATE semantics of other verification tools.

Further work will include extending our Statecharts variant by variable assignments, history variables and timeout events, and of course in further optimizations. Our prototype can no doubt be made faster by finding a good static variable ordering, directly accessing BDD operations and – most effectively – by using a better encoding of the configurations, for example the encoding of [6].

References

[1] Nancy Day. A model checker for statecharts. Technical Report 93-35, University of British Columbia, 1993.

[2] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Enginieering*, 16(4):403–413, 1990.

[3] D. Harel, A. Pnueli, J. Pruzan-Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proc. Symposium on Logicin Computer Science*, pages 54–64, 1987.

[4] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[5] R. Hasegawa, H. Fujita, and M. Fujita. A parallel theorem prover in KL1 and its applications to program synthesis. Technical Report TR-588, ICOT Research Center, 1990.

[6] Johannes Helbig and Peter Kelb. An OBDD-representation of statecharts. In *European Design And Test Conference 1994*, pages 142–149, 1994.

[7] Katsumi Inoue, Miyuki Koshimura, and Ryuzo Hasegawa. Embedding negation as failure into a model generation theorem prover. In D. Kapur, editor, *Automated Deduction – CADE-11, Lecture Notes in Artificial Intelligence 607*, pages 400–415, 1992.

[8] Geert Janssen. ROBDD software. Eindhoven University of Technology, 1995.

[9] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A.R. Meyer, editors, *TACS 91, Lecture Notes in Computer Science 526*, pages 244–264, 1991.