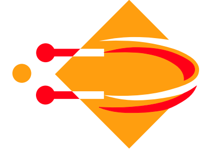

On the Formal Semantics of VisualSTATE Statecharts

Andrzej Wąsowski
Peter Sestoft

IT University Technical Report Series

TR-2002-19

ISSN 1600–6100

September 2002

Copyright © 2002, Andrzej Wąsowski
Peter Sestoft

IT University of Copenhagen
All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

ISSN 1600–6100

ISBN 87-7949-026-3

Copies may be obtained by contacting:

IT University of Copenhagen
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone: +45 38 16 88 88
Telefax: +45 38 16 88 99
Web www.it-c.dk

On the Formal Semantics of VisualSTATE Statecharts

Andrzej Wąsowski and Peter Sestoft

IT-University of Copenhagen
{wasowski,sestoft}@it-c.dk

Abstract. This paper presents a formal semantics of statecharts – a visual language successfully employed in design of control algorithms. Our formalization is implementation oriented, with efficiency in the focus. It has been used as a specification in development of SCOPE, an experimental code generator for embedded systems.
The version of statecharts we describe is that implemented in commercial development tool IAR visualSTATE . IAR visualSTATE statecharts are similar to Harel's original statecharts, with several additions and some restrictions. They mostly agree with UML state diagrams on syntax and semantics. A small survey is appended comparing visualSTATE statecharts terminology and concepts with those of D. Harel's original statecharts and UML statechart diagrams. The paper may be perceived as a formal equivalent to the official Concept Guide delivered with the visualSTATE software package from IAR.

1 Introduction

Statecharts are widely recognized as a suitable formalism for design of sophisticated control algorithms, specific for embedded software. Statecharts are precise enough to be formalized and thus used not only in design but also in verification and automatic synthesis of programs. This approach is advocated by vendors of CASE tools for embedded development (see [20, 9, 10] among others).

One argument for using abstract modeling with statecharts is the common availability of model-checking software. Reliable and efficient automatic code synthesis should be another convincing argument. Surprisingly, research on theoretically well-founded and efficient compilers for statecharts has not been very intensive. This weakens the impact of verification technologies and discourages developers from using high level description techniques. Mainstream imperative languages (C/C++ variants) and assemblers still lead on the market of embedded software development.

We believe that current code generation schemes can be significantly improved. The present paper is a first step in our long-term work on development of a correct and robust code generator. We start with giving an extensive, implementation-oriented semantics for a variant of statecharts. Our formalization is intended to be used as a specification for a compiler implementation.

The variant of statecharts we consider has been implemented in IAR visualSTATE, an environment specifically devoted to development of embedded software[1]. It covers all development stages, including verification and software synthesis. The statecharts of visualSTATE resemble Harel's original statecharts[5] and only slightly differ from UML statechart diagrams[19]. Most notably there is no explicit support for different synchronization states. Fork transitions are generalized and local events are syntactically distinguished from external events (there is no scoping for events though). The *run-to-completion* semantics is also somewhat different in UML.

The most important difference is that UML statecharts are defined in an object-oriented setting while visualSTATE charts remain at the event-processing reactive level. One can expect however that IAR visualSTATE will evolve in the object-oriented direction in the future.[2]

We give an exhaustive semantics definition, which means that we aim to cover all parts of the language, not only the parts which are relevant and feasible for verification (though some elements are factored out from the main presentation to an appendix for clarity). We discuss also features that are often glossed over, such as history, deep history, external function calls, shared variables, etc. We precisely describe the C language interface. One cannot

[1] We use IAR visualSTATE ver. 4.3
[2] The newest release include interfaces to object-oriented languages.

avoid such details in an implementation-oriented approach, because all parts of the model have some impact on the generated code.

We also focus on making the semantics complete in another sense: it should describe the lifetime of the system from initialization till its death (omitting initialization and finalization sections is a common practice).

Our semantics is hierarchical and syntax-driven, which means that we directly give meaning to syntax elements of a hierarchical system. This is a common thing, yet still contrasting with alternatives based on flattening. It is possible to give semantics and implementation of hierarchical model in terms of flattening to a set of plain Mealy machines having a well-known semantics (see [17, 1, 3, 4] and others). Hierarchical semantics can be used both to create hierarchy aware implementations of statecharts and to prove correctness of flattening algorithms (and thus support flattening based implementations).

Last but not least, the semantics given here is global, which means that it lacks compositionality. The behavior is only defined in terms of the whole system. There is no notion of subprogram and no notion of equivalence for subprograms. This may be considered a drawback, but we decided to follow closely the semantics of the actual implementation, which is not compositional. IAR visualSTATE does not currently support code reuse, exchangeable modules and libraries, so there is no need for firmly defined equivalence for such systems from a user point of view. Still some restricted equivalence definition would be useful for formalization of various compiler optimizations. It is questionable, though, whether full compositionality in the spirit typical for textual languages is needed.

The paper proceeds as follows. Section 2 summarizes some related work on visualSTATE and on the semantics of statecharts in general. Then we continue with defining the syntax of systems (sections 3–4), state of execution (section 5) and operational semantics (section 6). Section 7 gives a brief argument that our semantics is deterministic. Some suggestions for future work on semantics of statecharts are given in the conclusion.

Appendix A presents an example in concrete syntax. Appendix B discusses some elements excluded from the main text, while appendix C contains a small dictionary of terms that are used in conflicting ways in different papers and implementations.

2 Related Work

Statecharts were introduced by Harel [5] in 1987. Soon after that they were formally described in [7]. The present paper owes much to this early attempt, generally following its structure. The main difference is that our definition is deterministic, which has been imposed by two goals: it has to be feasible for both implementation and verification. Harel's semantics is nondeterministic and as such describes many possible implementations, making verification a much harder problem.

A more detailed rigorous description of Harel's statecharts appeared in [6], partly formalized in [18]. Numerous attempts have been made since then. Already in 1994 Beeck[21] described as many as 20 statechart variants. It should be emphasized that verification-oriented specifications dominated this list. Even more proposals of formalization emerged after statecharts had been incorporated in the family of UML languages [19].

IAR visualSTATE lacks a full, formal and implementation-oriented semantics. The existing descriptions are either informal [13, 11, 12] or incomplete. The informal descriptions are aimed at end users of statecharts (developers). A slightly outdated developer-oriented description of static structure of visualSTATE models can be found in unpublished internal IAR documentation [14].

The formal descriptions are often devised for language subsets with model-checking application in mind. The most complete verification-oriented semantics has been included in Lind Nielsen's thesis [15]. He however omits use of external C functions in actions and conditions, leaving out the subtle problems of side effects. We follow his approach and parameterize the semantics with a priority function for states in order to avoid non-determinism.

Argos[17] is a statechart-like language with fully formal semantics. An implementation oriented semantics for Argos has been given in [16] in terms of boolean flow equations, which is an entirely different approach than presented below.

3 Abstract syntax

A statechart system is built of basic elements which may be divided into four primitive groups: states, identifiers (variables and named values), events, and outputs (also called action functions or functions). These basic entities are then arranged together in various constructs like state hierarchy, state rules, history markings and transitions.

We define four mutually disjoint finite sets to represent primitive elements of a given system: *State*, *Id*, *Event* and *Output*.

3.1 List Notation

Various elements of the system are described using finite lists of simpler entities. We will write X^* for set of lists of elements of X. The list may be easily modeled as union of sets of tuples with finite number of components. We will use angle brackets to write list values with $\langle\rangle$ symbol denoting the empty list.

3.2 State Hierarchy

States are structured in a hierarchy as follows.

Definition 1 (Hierarchy Relation). *The* hierarchy relation $\sqsubset$ *is an irreflexive binary relation on states such that* $s \sqsubset s'$ *if* s *is a child of* s'*, i.e.:*

1. *There is a unique root state having no parents:* $\exists! root \in State.\ \forall s \in State.\ \neg(root \sqsubset s)$
2. *Each non-root state has a single parent:* $\forall s \in State.\ (s \neq root \Rightarrow \exists! s' \in State.\ s \sqsubset s')$
3. *Each state is reachable from root:* $\forall s \in State.\ s \sqsubset^* root$

We will further refer to irreflexive and reflexive transitive closures of hierarchy relation with symbols $\sqsubset^+$ and $\sqsubset^*$ respectively. We will also write $s \not\sqsubset s'$ as shorthand for $\neg(s \sqsubset s')$ and similarly for negations of closure relations.

Observation 2. *Following properties hold for hierarchy relation* $\sqsubset$ *on set of states State:*

1. *Relation* $\sqsubset$ *is irreflexive:* $\forall s \in State.\ s \not\sqsubset s$
2. *All chains are acyclic:* $\forall s \in State.\ s \not\sqsubset^+ s$
3. *Relation* $\sqsubset$ *is strongly not symmetric:* $\forall s, s' \in State.\ s \sqsubset s' \Rightarrow s' \not\sqsubset s$
4. *Relation* $\sqsubset$ *defines a rooted tree structure on Stateset.*

Proof. The properties follow directly from definition 1. □

Definition 3 (Parent, Children). *If* s, s' *are states and* $s \sqsubset s'$ *then* s' *is the unique* parent *of* s*. We will write* $s' = parent(s)$ *to denote it. Similarly we will say that* s *is a* child *of* s' *and write children*(s) *for set of children of* s'*.*

$$parent : State \hookrightarrow State$$
$$parent(s) = s' \Leftrightarrow s \sqsubset s'$$

$$children : State \rightarrow \mathcal{P}(State)$$
$$children(s) = \{s' \in State \mid s' \sqsubset s\}$$

Note that *parent* is a partial function and

$$dom(parent) = State \setminus \{root\},$$

which follows from the assumption that only non-root states have parents.

Above we defined the reflexive and irreflexive closures of the hierarchy relation. Following the same scheme, we define the corresponding closures of the *parent* and *children* relations:

Definition 4 (Ancestors, Descendants). *If* $s' \sqsubset^* s$ *then* s' *is a* descendant *of* s*, and descend*$^*(s)$ *denotes the set of all descendants of* s*, including* s*.*

Similarly, if $s \sqsubset^* s'$ *then* s' *is an* ancestor *of* s *and we write ancest*$^*(s)$ *for the set of all ancestors of* s*, including* s*.*

$$ancest^* : State \rightarrow \mathcal{P}(State)$$
$$ancest^*(s) = \{s' \in State \mid s \sqsubset^* s'\}$$

$$descend^* : State \rightarrow \mathcal{P}(State)$$
$$descend^*(s) = \{s' \in State \mid s' \sqsubset^* s\}$$

Note that for any given state

$$children(s) \subseteq descend^*(s)$$

and for any non-root state

$$parent(s) \in ancest^*(s).$$

We define sets of proper descendants and proper ancestors to be

$$ancest^+(s) = ancest^*(s) \setminus \{s\}$$
$$descend^+(s) = descend^*(s) \setminus \{s\}$$

Definition 5 (Siblings). *If two distinct states have the same parent we say that they are proper siblings.*

$$siblings(s) = \{s' \mid s' \neq s \wedge (\exists p.\ s \sqsubset p \wedge s' \sqsubset p)\}$$

This is easily relaxed to inclusive function:

$$siblings^*(s) = siblings(s) \cup \{s\}$$

A statecharts hierarchy exhibits two kinds of encapsulation: AND-decomposition and XOR-decomposition. The former describes concurrent activities, the latter sequential control flow. At this point we introduce the syntactic difference between the two kinds of states and impose some requirements on how they can be combined. These requirements are relatively rigid in visualSTATE compared to other versions of statecharts.

Definition 6 (State Typing). *A total function* $\Gamma_S : State \to \{\mathsf{and}, \mathsf{or}\}$ *is a* state typing *for state set and hierarchy relation iff*

1. $\Gamma_S(root) = \mathsf{and}$
2. $\forall s \in State.\ \Gamma_S(s) = \mathsf{or} \Rightarrow children(s) \neq \emptyset$
3. $\forall s, s' \in State.\ \Gamma_S(s) = \mathsf{or}\ \wedge\ s' \sqsubset s\ \Rightarrow$ $\Rightarrow \Gamma_S(s') = \mathsf{and}$
4. $\forall s, s' \in State.\ \Gamma_S(s) = \mathsf{and} \wedge s' \sqsubset s\ \Rightarrow$ $\Rightarrow \Gamma_S(s') = \mathsf{or}$

Such typing is not possible for arbitrary state sets and hierarchy relations. From now on we will consider only well-structured state sets, i.e. state sets with a hierarchy relation and state typing as defined above. Clearly, such well-structured sets exist.

In a well-structured state set a *root* element must be of type and. So must all minimal elements. The types of states alternate along any chain defined by the relation $\sqsubset$.

Thus a well-structured state set is a rooted tree whose root is the state *root* of type and . All leaves are also of type and and each branch in the tree connects two nodes of different types: types alternate on each path in the tree. The depth of the tree is odd: each path from *root* to a leaf state contains an odd number of nodes. For each node, all children have the same type.

It should be noted that the typing is not strictly needed while describing the structure of our statecharts[3]. The type of each state can be uniquely determined by its placement in the hierarchy.

Instead of introducing state typing and restricting ourselves to well-structured models, we could have imposed the same structure in definition of hierarchy relation and, if needed, infer the types based on locations of states. This would however result in a more cluttered description.

We say that a state s is an and-state if $\Gamma_S(s) = \mathsf{and}$. Similarly s is an or-state if $\Gamma_S(s) = \mathsf{or}$. We will write $State_{\mathrm{or}}$ and $State_{\mathrm{and}}$ for the respective disjoint subsets of *State*. An and-state with no children will be called a *basic state*.

Note that visualSTATE user documentation uses the term *region* for what we call an or-state and the term *state* for what we call and-state (while state is a more general notion in our definition).

3.3 Basic properties of the hierarchy

All dynamic changes in the system are local. In the first approximation the locality of a transitions (the scope of the transition) is determined by the distance between its source and target. The parts of the system outside the areas containing a transition are not affected by its execution. As transitions may cross different levels of hierarchy and explicitly refer to other parts of the system, the notion of transition scope becomes more elaborate.

Below we present basic properties of locality and dependence of states in statechart language. This will allow us to speak about distance between states and the scopes of a transition.

Definition 7 (Nearest Common Ancestor). *The* Nearest Common Ancestor *of a set of states* $X \subseteq State$ *is a state* y *such that:*

1. $\forall x \in X.\ x \sqsubset^+ y$
2. $\forall s \in State.\ \forall x \in X.\ x \sqsubset^+ s \Rightarrow y \sqsubset^* s$

We will denote it writing

$$y = NCA(X).$$

For any non-empty set of states $X \subseteq State$ not containing *root*, $NCA(X)$ exists and is uniquely determined.

Definition 8 (Orthogonality). *Two states x and y are orthogonal if one is not an ancestor of the other (they are not ancestrally related) and their NCA is an* and*-state.*

$$\forall x, y \in State.\ x \perp y \Leftrightarrow (x \not\sqsubset^* y\ \wedge\ y \not\sqsubset^* x\ \wedge\ \Gamma_S(NCA\{x, y\}) = \mathsf{and})$$

We will occasionally write $a \not\perp b$ for $\neg(a \perp b)$.

Definition 9 (Orthogonal Set). *A set of states is orthogonal if (it is a singleton or) any two distinct elements are mutually orthogonal.*

Definition 10 (Relative Orthogonal Set). *A set X of states is an* orthogonal set relative to state *s iff X is an orthogonal set of descendants of s.*

Definition 11 (Maximal Orthogonal Set). *A set of states X is a* maximal orthogonal set of states relative to state *s iff adding any new descendant of s would create a non-orthogonal set, i.e.*

$$\forall y \in State \setminus X.\ y \sqsubset^* s \Rightarrow X \cup \{y\} \text{ is not orthogonal.}$$

Definition 12 (Configuration). *A maximal orthogonal set of states relative to state s is said to be a* state configuration of *s iff all its elements are basic states.*

[3] As opposed to original statecharts described by D. Harel[5], which allowed a bit less uniform structure.

Note that a state configuration of *root* fully describes one possible global state of the hierarchical state machine. We will often write *state configuration* when we actually mean a *state configuration of the root state.*

Observation 13 (Properties of Orthogonality). *The following simple properties hold for the orthogonality relation in a well structured statechart.*

1. $\forall a \in State.\ a \not\perp a$ *(irreflexive)*
2. $\forall a, b \in State.\ a \perp b \Leftrightarrow b \perp a$ *(symmetric)*
3. $\exists a, b, c \in State.\ a \perp b \ \wedge\ b \perp c \ \wedge\ a \not\perp c$ *(not transitive)*
4. $\forall a, b, c \in State.\ a \perp b \ \wedge\ c \sqsubset^* a \Rightarrow NCA\{a, b\} = NCA\{c, b\}$
5. $\forall a, b, c \in State.\ a \perp b \ \wedge\ c \sqsubset^* a \Rightarrow c \perp b$
6. $\forall a, b, c \in State.\ a \perp b \wedge a \sqsubset^* c \wedge c \sqsubset^+ NCA(a, b) \Rightarrow c \perp b$
7. $\forall a, b, c \in State.\ a \not\perp b \ \wedge\ c \sqsubset^* a \Rightarrow c \not\perp b$

Proof. (sketch) Symmetry follows from definition of orthogonality. The counterexample for non-transitivity is $a \perp b$ and $c = b$. Property (4) follows from definition of orthogonality and it implies properties (5)–(7). □

Observation 14 (Properties of Orthogonal Sets). *The following property holds for orthogonal sets: any subset of an orthogonal set is itself orthogonal. The contrapositive can be used to prove nonorthogonality: any set containing a nonorthogonal subset is itself nonorthogonal.*

Following three theorems will help us to define semantics of transition firing recursively on subtrees of decomposition tree.

Theorem 15 (Configuration Union). *Let s be an* and*-state, $s_1, ..., s_k$ children of s and $\sigma_1, ..., \sigma_k$ state configurations of $s_1, ..., s_k$ respectively. Then $\sigma = \bigcup_{i=1}^{k} \sigma_i$ is a state configuration of s.*

Proof. (Sketch) We need to prove that elements of σ are basic (trivial), that σ is an orthogonal set, and that σ is a maximal orthogonal set. The state configuration σ is orthogonal because any two of states belonging to it either belong to the same σ_i (and are orthogonal by assumption) or belong to two different subtrees and are orthogonal by definition (their NCA is state s, which is an and-state). The set is maximal because adding any fresh basic state would make one of σ_i nonorthogonal (and further make σ nonorthogonal by observation 14). □

Theorem 16. *Let σ be a state configuration of state s and s' a child of state s ($s' \sqsubset s$). Then $\sigma' = \sigma \cap descend^*(s')$ is itself a state configuration of s'.*

Proof. (Sketch) Orthogonality is easily proved using observation 14. As far as maximality is concerned assume that $s'' \notin \sigma'$ is a basic descendant of s'. Then $\sigma \cup \{s''\}$ is not orthogonal. But any state s''' in σ but outside σ' would automatically have $NCA(s'', s''') = s$, which is an and-state, so $\sigma'' \perp \sigma'''$. So only states in σ' may violate orthogonality. We showed that for any newly added state there exists a state within σ' which is not orthogonal to it. It means that σ' is a maximal orthogonal set. As all its members are trivially basic σ' is a state configuration of s'. □

Note that the above theorem can be easily generalized for arbitrarily nested configuration subsets:

Theorem 17 (Configuration Subset). *Let σ be a state configuration of state s and let s' a be descendant of s ($s' \sqsubset^* s$). Then $\sigma' = \sigma \cap descend^*(s')$ is itself a state configuration of s'.*

Proof. By induction on distance between s and s' using theorem 16. □

Theorem 18. *Let σ_0 be a state configuration of* and*-state s and state s' a child of state s. If σ' is a state configuration of s' then $\sigma_1 = \sigma_0 \setminus descend^*(s') \cup \sigma'$ is itself a state configuration of s in which a subconfiguration for s' has been substituted with σ'.*

Proof. (sketch) Divide σ in configurations for children of s using theorem 16. Then exchange one of them for σ' and use theorem 15 to prove that σ_1 is a state configuration. □

Again the above theorem can be inductively generalized for substitution of arbitrarily nested subconfiguration:

Theorem 19 (Configuration Substitution). *Let σ_0 be a state configuration of* and*-state s and state s' a descendant of s. If σ' is a state configuration of s' then $\sigma_1 = \sigma_0 \setminus descend^*(s') \cup \sigma'$ is itself a state configuration of s in which a subconfiguration for s' has been substituted with σ'.*

3.4 Initial and History States

Some or-states enclose a visual mark indicating which of its children should be entered when no target is explicitly indicated on the transition. The indicator is called an *initial marking*. Each or-state must have either an initial marking or a history marking.

Definition 20 (Initial Marking). *An initial marking on the set* $State_{\mathrm{or}}$ *is a partial function* $initial : State_{\mathrm{or}} \hookrightarrow State_{\mathrm{and}}$ *such that* $\forall s \in dom(initial).\ initial(s) \sqsubset s$.

State s' is called an *initial state* of or-state s iff $s' = initial(s)$. In visualSTATE initial marks are visually attached to and-states instead of or-states. If a mark is attached to state s in a visualSTATE system then $initial(parent(s)) = s$ in our formal model.

A history marking resembles an initial marking. These two are strongly connected.

Definition 21 (History). *A history marking* η *is defined to be a total function of type:*

$$\eta : (State_{\mathrm{or}} \setminus dom(initial)) \to State_{\mathrm{and}}$$

such that

$$\forall s \in dom(\eta).\ \eta(s) \sqsubset s.$$

States belonging to $dom(\eta)$ *are called* history states.

Initial markings and history markings are complementary with respect to domain and similar in many other aspects. Beware, though, that there is only one fixed initial marking for the project, while the history marking is changing dynamically throughout the execution (its domain is fixed though). Thus history marking is not a purely syntactic notion, although the very first history marking just after resetting the system is expressed syntactically.

Note that, contrary to Harel's statecharts, only one kind of default entry is allowed for each or-state here. In particular it is not possible that the same entry is entered via the history path by some transitions and via the initial path by others.

3.5 Types

Entities such as variables, functions and expressions are typed. Their types resemble data types of typical programming languages and they should not be confused with state types in the hierarchy.

Types are only moderately significant for visualSTATE code generators and interpreters since all typed entities are forwarded to the underlying C compiler which performs its standard type checking algorithm. This is why we only present a restricted type system, and we do not give typing rules even for that. Occasionally type-correctness checks have been embedded in operational rules. This does not mean that the language is dynamically typed, but reflects our decision not to give a full static semantics here.

We only distinguish two simple arithmetic types:

$$SimpleType = \{\mathsf{int}, \mathsf{real}\}\ .$$

Simple types have domains $D_{\mathrm{int}} = \mathbb{Z}$ and $D_{\mathrm{real}} = \mathbb{R}$ respectively ($D_{\mathrm{int}} \subset D_{\mathrm{real}}$). We also consider aggregations of simple types in one-dimensional vectors:

$$\begin{aligned} Type = {} & SimpleType\ \cup \\ & \cup\ \{\mathsf{int}[n] \mid n > 0\ \wedge\ n \in D_{\mathrm{int}}\}\ \cup \\ & \cup\ \{\mathsf{real}[n] \mid n > 0\ \wedge\ n \in D_{\mathrm{int}}\} \end{aligned}$$

We will write $D(t)$ for the domain of type t, where $D(t)$ is defined to return D_{int} or D_{real} for simple types and a set of fixed-length vectors for aggregated types.[4]

Occasionally we will need to write conditions about types of various non trivial elements (such as expressions). We will use a type oracle function $\tau[\![\cdot]\!]$ when expressing these conditions, without giving typing rules. It is assumed that the type oracle returns a proper typing for the expression which agrees with the static semantics of C.

3.6 Identifiers and Variables

The set of global variables *Var* is a subset of the set of identifiers *Id*. Identifiers may also be used for other purposes than variables (for instance binding event parameters to names – see later). There is no support for local variables in visualSTATE. All variables are global.

Variables are typed as in C.

Definition 22 (Variable typing). *A variable typing* Γ_V *is a total function* $\Gamma_V : Var \to Type$.

[4] Strictly speaking visualSTATE disallows single cell arrays. We are slightly more general here following the definition of ANSI C and allowing arrays of length one.

3.7 Variable Accesses

The grammar for variable accesses is a subset of the C grammar for lvalues. There are no pointers, and therefore no pointer dereferencing. Similarly the array variable cannot be accessed as a whole. Only simple variables and array cells are syntactically legal:

$$Access ::= x \mid x[Aexp] \ ,$$

where *Aexp* denotes arithmetic expression (defined later) and x ranges over variable names.[5]

3.8 Expressions

We distinguish two kinds of expressions: expressions belonging to actions, and guard expressions. They roughly correspond to arithmetic expressions and boolean expressions in programming languages. They follow the same syntax and similar evaluation rules, so we will treat them jointly in many cases.

Our expressions are restricted C arithmetic expressions. They are generated by the following grammar:

$$Exp \ ::= \ v \mid a \mid \ unop\ e_1 \mid e_1\ binop\ e_2 \mid o(e_1, ..., e_k),$$

where v is a constant (integer or real), a is a variable access, $e_1, ..., e_k$ are expressions, *unop* is a unary C operator, *binop* is any binary C operator except for the assignment operators, and $o(e_1, ..., e_k)$ is an output instance (defined below).

Expression syntax applies both to *boolean* and *arithmetic expressions*. Boolean expressions are used in guards – conditions evaluated in order to check if given transition is enabled. Arithmetic expressions are used in assignments – executable parts of transitions. We will write *Bexp* for the set of all boolean expressions and *Aexp* for set of all arithmetic expressions. They are both subsets of *Exp*, but we make distinct semantic assumptions for them. Boolean expressions are assumed to be pure while arithmetic expressions may not be pure in general. The difference will be formalized while giving dynamic semantics of expression evaluation.

3.9 Outputs

Action functions (also called outputs) represent external C functions which will be called by the control kernel to query or change the environment. We avoid giving a full formal semantics for execution of action functions as this would demand giving a complete dynamic semantics for the C programming language. Instead we model only function prototypes and define their behavior informally.

Definition 23 (Output Typing). *An output typing is a total function:*

$$\Gamma_O : Output \rightarrow SimpleType \times Type^*.$$

The first result component represents the return type, and the second component represents the parameter types. Note that only simple types may be returned from action functions.

Definition 24 (Output Instance). *Output instance is a term $o(v_1, ..., v_k)$, where $v_1, ..., v_k$ are constants:*

$$\begin{aligned} Oinst = \{ o\,(v_1, ..., v_k) \ \mid k = |\pi_2(\Gamma_O(o))| \\ \wedge \ \forall i \in \{1..k\}.\ v_i \in D(\pi_i(\pi_2(\Gamma_O(o)))) \\ \wedge \ o \in Output \ \} \end{aligned}$$

Definition 25 (Output Expression). *An output expression is a term $o(e_1, ..., e_k)$, where $e_1, ..., e_k$ are properly typed expressions:*

$$\begin{aligned} Oexpr = \{ o\,(e_1, ..., e_k) \ \mid o \in Output \\ \wedge \ k = |\pi_2(\Gamma_O(o))| \ \wedge \\ \wedge \ \forall i \in \{1..k\}.\ e_i \in Exp \ \wedge \\ \wedge \ (\tau [\![e_1]\!], ..., \tau [\![e_k]\!]) = \pi_2(\Gamma_O(o)) \ \} \end{aligned}$$

We shall use the terms *function* and *action function* instead of output, depending on the context (when the output is a part of an assignment or expression).

3.10 Assignments

The executable parts of the system are called actions. Actions may be executed while firing a transition or when entering/leaving a state. There are three possible kinds of actions: assignments, output expressions and signaling events. We will define them formally now.

Definition 26 (Assignments). *The set of all possible assignments Assgn is defined to be*

$$\begin{aligned} Assgn = \{ (a, e) \mid a \in Access \ \wedge \ e \in Aexp \ \wedge \\ \wedge \ \tau [\![e]\!] = \tau [\![a]\!] \} \ . \end{aligned}$$

[5] In reality the visualSTATE grammar is even more limited. General expressions are not allowed in accesses. The actual grammar is: $Access ::= x_1 \mid x_2[n] \mid x_2[x_1]$, where x_1 runs over simple type variables, x_2 over arrays and n over integer constants. Expressions cannot be freely used in external function calls, outputs, and signal expressions. We abandon these restrictions to simplify both the presentation and our tools, and to increase generality.

3.11 Events

Events in the *Event* set correspond to both external and internal stimuli. Some events may carry parameters (event fields). Consequently events are typed in a manner similar to C function prototypes.

Definition 27 (Event Typing). *Event typing is defined to be a total function*

$$\Gamma_E : \mathit{Event} \to \mathit{SimpleType}^*,$$

For a given event e, event type value $\Gamma_E(e)$ describes types of subsequent parameters. An *event binding* assigns local names to these parameters.

Definition 28 (Event Binding). *A name binding for event e is a term $e(p_1, ..., p_k)$, where e is an event and $p_1, ..., p_k$ are identifiers, i.e.*

$$\begin{aligned}\mathit{Ebind} = \{e(p_1, ..., p_k) \mid k = |\Gamma_E(e)| \ \wedge \\ \wedge\ \forall i \in \{1..k\}.\ p_i \in \mathit{Id}\}\end{aligned}$$

The identifiers $p_1, ..., p_k$ will usually be fresh in the context. If a variable name is used for p_i then the parameter will hide the variable in the scope of the event binding (which extends over single transition).

A related concept is *event instance*, which assigns concrete runtime values to event parameters. Name bindings are used statically, like formal parameters in C, and instances are used dynamically to represent actual values of event parameters.

Definition 29 (Event Instance). *An instance of an event e is a term $e(v_1, ..., v_k)$, where $v_1, ..., v_k$ are values consistent with event typing, i.e.*

$$\begin{aligned}\mathit{Einst} = \{e(v_1, ..., v_k) \mid k = |\Gamma_E(e)| \ \wedge \\ \wedge\ \forall i \in \{1..k\}.\ v_i \in D(\pi_i(\Gamma_E(e)))\} \ .\end{aligned}$$

Event instances *Einst* convey information about events as present at runtime. They also emerge as results of evaluation of signal expressions:

Definition 30 (Signal Expression). *A signal expression describes the computation of an event instance to be triggered (signaled), i.e.*

$$\begin{aligned}\mathit{Sexpr} = \{e(e_1, ..., e_k) \mid e \in \mathit{Event} \ \wedge \\ \wedge\ \forall i \in \{1..k\}.\ e_i \in \mathit{Exp} \ \wedge \\ \wedge\ (\tau [\![e_1]\!], ..., \tau [\![e_k]\!]) = \pi_2(\Gamma_E(e))\ \}\end{aligned}$$

3.12 Actions

Definition 31 (Action). *An action is a triple of type $\mathit{Action} = \mathit{Assgn}^* \times \mathit{Oexpr}^* \times \mathit{Sexpr}^*$, containing a sequential list of assignments, outputs, and events to be signaled. Action components are supposed to be executed from left to right and may have side effects.*

Each and-state may have assigned two actions: one for entry and one for exit.

$$\begin{aligned}\mathit{entry} : \mathit{State}_{\text{and}} \to \mathit{Action} \\ \mathit{exit} : \mathit{State}_{\text{and}} \to \mathit{Action}\end{aligned}$$

Note that these mappings are total. States that do not have actions assigned in visual syntax, are assigned the empty action $(\langle\rangle, \langle\rangle, \langle\rangle)$ in our semantics.

3.13 Transitions

Finally we can give a formal definition of transition.

Definition 32 (Multitarget Transition). *Each system has a final set of transitions Trans such that:*

$$\begin{aligned}\mathit{Trans} \ \subseteq\ & \mathit{Ebind} \times \mathit{State}_{\text{or}} \times \mathcal{P}(\mathit{State}_{\text{and}}) \times \\ & \times \mathcal{P}(\mathit{State}_{\text{and}}) \times \mathit{Bexp} \times \\ & \times \mathit{Action} \times \mathcal{P}(\mathit{State}_{\text{and}})\end{aligned}$$

Each transition has seven components. We define a family of functions to access components of transitions. The *event* function determines the triggering event for the transition.

$$\mathit{event}(t) = e \text{ if } \pi_1(t) = e(p_1, ..., p_k) \in \mathit{Ebind}$$

In order to access local identifiers set we define the predicate *params*:

$$\mathit{params}(t) = \{p_1, ..., p_k\} \text{ if } \pi_1(t) = e(p_1, ..., p_k)$$

The third and fourth components represent negative and positive state conditions. Negative and positive conditions are constraints put on transition. It may only be fired if all states in the positive condition set are active and none of those in the negative condition set is.

$$\mathit{pos}(t) = \pi_3(t) \qquad \mathit{neg}(t) = \pi_4(t)$$

Each transition is attached to a pair of states: target and source (possibly the same state, for self-loops). We will call these two states the *explicit source* and the *explicit target*. The explicit source and target states determine the *explicit scope* of the transition.

The explicit scope of a transition is always an or-state: the arrow will never cross the boundary

between or-states (regions). In visualSTATE, the explicit scope of a transition is defined to be the nearest common ancestor of the explicit source and explicit target. The transition's scope obtained this way from the concrete syntax is represented in the abstract syntax by the transition's second component:

$$scope(t) = \pi_2(t)$$

It is noteworthy that the explicit transition scope has only a limited impact in the semantics of transition firing. The actual scope of a state change depends solely on the current state configuration and the target state. We will formalize this later, using the notion of *generalized implicit scope*.[6] The explicit scope of a transition (and hence the visual arrow in concrete syntax) simply indicates the priority of the transition, used in conflict resolution. This is discussed under the operational semantics rules.

As we said, a visualSTATE transition must have at least one target – the explicit one. Other targets may be added as so-called *force state actions*, a special kind of actions which force states to become active. They generalize UML fork transitions (which can only target inactive states). Force state actions can change the state of a currently operating component. Both force state actions and explicit targets are modeled together in one set:

$$targets(t) = \pi_7(t)$$

You may think of transitions as changes from one state configuration pattern to another. The initial state configuration pattern is described by positive and negative conditions while the target pattern is given by an orthogonal set of targets.

We use the name *multitarget transition* when we want to distinguish from UML transitions. Multitarget transitions are more general than UML fork transitions, since they can target states orthogonal to the source state. They are a bit more limited than UML join (or synchronization) transitions, because there is no direct support for synch pseudostates. However, the corresponding behavior may be easily described using a regular state instead of a UML pseudostate.

Force state actions complicate designs and semantics, especially correctness conditions, and conflict detection and resolution. On the other hand they may be very efficient if implemented carefully.

The remaining transition components are a guard and an action. The guard is a boolean expression, evaluated to determine whether the transition is active. The action is executed when an enabled transition fires. Both are optional. An omitted action is modeled by an empty action triple. An omitted guard is modeled by the constant true expression (integer constant 1).

$$guard(t) = \pi_5(t) \qquad action(t) = \pi_6(t)$$

3.14 Closed Formulæ

Below we give exact specification of what are free variables for different elements, thus extending the notion of closedness for various objects. We will write $fvars(X)$ for the set of free variables of element X.

$$fvars : Access \to \mathcal{P}(Id)$$

$$fvars(a) = \begin{cases} \{x\} & \text{if } a = x \\ \{x\} \cup fvars(e) & \text{if } a = x[e] \end{cases}$$

$$fvars : Oexpr \to \mathcal{P}(Id)$$

$$fvars(o(e_1, ..., e_k)) = \bigcup_{i=1}^{k} fvars(e_i)$$

$$fvars : Sexpr \to \mathcal{P}(Id)$$

$$fvars(e(e_1, ..., e_k)) = \bigcup_{i=1}^{k} fvars(e_i)$$

$$fvars : Exp \to \mathcal{P}(Id)$$

$$fvars(e) = \begin{cases} \emptyset & \text{if } e = v \in D_{\text{int}} \cup D_{\text{real}} \\ fvars(a) & \text{if } e = a \in Access \\ fvars(e_1) & \text{if } e = unop\ e_1 \\ fvars(e_1) \cup fvars(e_2) & \\ & \text{if } e = e_1\ binop\ e_2 \\ fvars(o) & \text{if } e = o \in Oexpr \end{cases}$$

$$fvars : Assgn \to \mathcal{P}(Id)$$

$$fvars(a = e) = fvars(a) \cup fvars(e)$$

$$fvars : Action \to \mathcal{P}(Id)$$

$$fvars(\langle a_1, ..., a_k \rangle, \langle o_1, ..., o_m \rangle, \langle s_1, ..., s_n \rangle) = \bigcup_{i=1}^{k} fvars(a_i) \cup \bigcup_{i=1}^{m} fvars(o_i) \cup \bigcup_{i=1}^{n} fvars(s_i)$$

$$fvars : Trans \to \mathcal{P}(Id)$$

$$fvars(t) = (fvars(guard(t)) \cup \cup\ fvars(action(t)))\ \setminus\ params(t)$$

[6] The explicit target has slightly different scoping semantics in the actual visualSTATE implementation. We consider this a design mistake in the tool and propose a cleaner semantics. See transition firing later on.

Definition 33 (Closedness). *A model element X (transition, action, etc.) is said to be closed iff $\mathit{fvars}(X) \subseteq \mathit{Var}$. We will write $\mathit{closed}(X)$ meaning that element X is closed.*

4 Static Correctness

Definition 34 (Consistent entry/exit Actions). *The entry and exit actions of the system are consistent if all of them are closed:*

$$\forall s \in \mathit{State}_{\mathrm{and}}.\ \mathit{closed}(\mathit{entry}(s)) \land \mathit{closed}(\mathit{exit}(s))$$

Definition 35 (Consistent Transitions). *Transition t is consistent if it fulfills all of the following conditions:*

1. *$\mathit{closed}(t)$*
2. *$\mathit{root} \notin \mathit{pos}(t) \cup \mathit{neg}(t) \cup \mathit{targets}(t)$*
3. *$\mathit{targets}(t)$ is an orthogonal set.*
4. *$\mathit{targets}(t) \neq \emptyset$*
5. *$\mathit{pos}(t) \neq \emptyset$*
6. *$\mathit{guard}(t)$ and $\mathit{action}(t)$ contain only well-typed expressions and function calls.*

We will only give dynamic semantics for consistent systems as defined below.

Definition 36 (System). *A consistent system is a tuple $(\mathit{State}, \sqsubset, \Gamma_S, \mathit{initial}, \eta_0, \mathit{exit}, \mathit{entry}, \mathit{Var}, \mathit{Event}, \mathit{Trans})$, such that State is a well-structured state set with hierarchy relation $\sqsubset$ and typing Γ_S, $\mathit{initial}$ and η_0 are respective markings, entry and exit are mappings of rules, and Trans is a set of consistent transitions.*

5 Global State

The language of statecharts is imperative. The state of execution is built from four main components: explicit state of hierarchical state machine, variable store, history marking, and signal queue. We defined history marking above; now we we describe the remaining components.

The explicit state describes activity/inactivity properties of states at given point in time.

Definition 37 (Explicit State). *The state configuration σ of the root state describes the* explicit state *of execution.*

A variable store gives the values of system variables at a given point of execution.

Definition 38 (Store). *A store ϱ is a total mapping of type*

$$\varrho \in \mathit{Store} \equiv \mathit{Var} \to \bigcup \mathit{Type}$$

such that $\forall x \in \mathit{Var}.\ \varrho(x) \in D(\Gamma_V(x))$.

The signal queue is a list of pending local events, i.e. events that have been signaled as result of some actions and have not yet been processed.

Definition 39 (Signal Queue). *Signal queue ω is defined to be an ordered list of event instances: $\omega \in \mathit{Einst}^*$. We will use the caret symbol ($\hat{}$) for expressing concatenation and pattern matching elements in the queue.*

The signal queue is a first-in-first-out queue. The notational convention is that new elements are concatenated at the end of the list (suffixed). In this sense $\hat{}$ is a queue constructor. Concatenation in front is used to express internal structure of the list (prefixed event instance). It may be understood as pattern matching.

The signal queue is always empty at the beginning of event processing. First, an external event is processed. As a result some internal events may be signaled, that is, stored in the signal queue. These internal events (signals) are then processed consecutively, until the signal queue is empty. At that point the system is ready for the next external event.

The signal queue is not used to buffer externally detected events. There is no queue for external events, though it can be implemented in user C code. The semantics of this external implementation is beyond the scope of this paper.

6 Operational Dynamic Semantics

We define the dynamic semantics of a visualSTATE system by describing how expressions are evaluated, rules executed, states (scopes) exited and entered, and transitions fired. Then we describe the processing of a single internal event in a so-called microstep, and specify how a sequence of microsteps makes up a macrostep, processing a single external event. Last but not least we also say how a system is initialized.

Note that visualSTATE systems are reactive synchronous systems. Reactive means that the system responds to incoming events, and synchronous means that the response is considered to be infinitely fast [2, 8]. In practice this means that the system must be much faster than the environment. Full synchrony in the sense of Berry [2] is not guaranteed, though. This is because of the explicit microsteps

and the use of queues (instead of boolean signals mimicking the behavior of hardware output wires).

Another strong assumption, which is different from Harel's original statecharts, is that only one external event may be present at a time. These three assumptions together eliminate the need for maintaining a queue of pending external events.

We start the description with semantics of basic elements (expressions, functions, variable accesses). Those simple definitions mostly follow the general semantics of the C programming language. They have been included here to understand and describe the subtleties of evaluation, mainly purity constraints on some parts. Also this elements cannot be simply executed under C semantics, as technically some variables are not visible in the C interface (see appendix B.3).

After behavior of basic elements is defined we proceed with specific statechart features in section 6.8.

6.1 Access Evaluation

Accesses to variables and array cells are evaluated to values of variables and contents of array cells. The evaluation of an access may yield a new store as non-pure expressions might be present in array indices.

Definition is given only for closed variable accesses. It is mutually recursive with arithmetic expression evaluation. We say that access a evaluates in store ϱ_1 to value v and store ϱ_2 and write:

$$\langle\, a, \varrho_1 \,\rangle \xrightarrow[\mathit{access}]{} \langle\, v, \varrho_2 \,\rangle \;\; .$$

The relation is given by two rules:

$$\frac{x \in \mathit{Var} \qquad \Gamma_V(x) \in \mathit{SimpleType}}{\langle\, x, \varrho \,\rangle \xrightarrow[\text{access}]{} \langle\, \varrho(x), \varrho \,\rangle}$$

$$\frac{\begin{array}{c} x \in \mathit{Var} \qquad \Gamma_V(x) \notin \mathit{SimpleType} \qquad e \in \mathit{Aexp} \\ \langle\, e, \varrho_1 \,\rangle \xrightarrow[\text{Aeval}]{} \langle\, n, \varrho_2 \,\rangle \;\wedge\; n \in D_{\text{int}} \end{array}}{\langle\, x[e], \varrho_1 \,\rangle \xrightarrow[\text{access}]{} \langle\, \pi_n(\varrho_2(x)), \varrho_2 \,\rangle}$$

6.2 Output Expression Evaluation

Output expressions evaluate to output instances which then evaluate according to C semantics. There is a single evaluation rule given for closed output expressions:

$$\frac{\begin{array}{c} o \in \mathit{Output} \qquad \forall i.\ \langle\, e_i, \varrho_{i-1} \,\rangle \xrightarrow[\text{Aeval}]{} \langle\, v_i, \varrho_i \,\rangle \\ \langle\, o(v_1, ..., v_k), \varrho_k \,\rangle \xrightarrow[\text{C-sem}]{} \langle\, v, \varrho_{k+1} \,\rangle \end{array}}{\langle\, o(e_1, ..., e_k), \varrho_0 \,\rangle \xrightarrow[\text{output}]{} \langle\, v, \varrho_{k+1} \,\rangle} \;\; .$$

The definition is mutually recursive with evaluation of arithmetic expressions defined below. The *C-sem* relation denotes execution of an external function according to the usual C semantics.

6.3 Boolean Expression Evaluation

Boolean expressions in visualSTATE are assumed to be side-effect free. They can rely on external state (like registers of some device), but their computation is assumed to be infinitely fast, so the value of external physical properties may not change throughout execution of all guards in the system. The two assumptions support a deterministic semantics of guards.

The definition is given by structural induction and only for closed expressions. We say that expression b evaluates to value v in store ϱ and write:

$$\langle\, b, \varrho \,\rangle \xrightarrow[\mathit{Beval}]{} v$$

where $b \in \mathit{Bexp}$, $\varrho \in \mathit{Store}$ and $v \in D_{\text{real}} \cup D_{\text{int}}$. The relation is given by rules:

$$\frac{v \in D_{\text{int}} \cup D_{\text{real}}}{\langle\, v, \varrho \,\rangle \xrightarrow[\text{Beval}]{} v}$$

$$\frac{a \in \mathit{Access} \qquad \langle\, a, \varrho_0 \,\rangle \xrightarrow[\text{access}]{} \langle\, v, \varrho_1 \,\rangle}{\langle\, a, \varrho_0 \,\rangle \xrightarrow[\text{Beval}]{} v}$$

$$\frac{o \in \mathit{Oexpr} \qquad \langle\, o, \varrho_0 \,\rangle \xrightarrow[\text{output}]{} \langle\, v, \varrho_1 \,\rangle}{\langle\, o, \varrho_0 \,\rangle \xrightarrow[\text{Beval}]{} v}$$

$$\frac{\langle\, e_0, \varrho \,\rangle \xrightarrow[\text{Beval}]{} v_0 \qquad \langle\, e_1, \varrho \,\rangle \xrightarrow[\text{Beval}]{} v_1}{\langle\, e_0 \ \mathit{binop}\ e_1, \varrho \,\rangle \xrightarrow[\text{Beval}]{} v_0 \ \mathit{binop}\ v_1}$$

$$\frac{\langle\, e, \varrho \,\rangle \xrightarrow[\text{Beval}]{} v}{\langle\, \mathit{uop}\ e, \varrho \,\rangle \xrightarrow[\text{Beval}]{} \mathit{unop}\ v}$$

In contrast to standard C evaluation, the above expressions are pure: there is no assignment operator in our grammar. Note that the new store is discarded in both the second and the third rule. This is implementationally feasible because we assume that boolean expressions are pure, so $\varrho_1 = \varrho_2$ in both cases.

6.4 Arithmetic Expression Evaluation

In contrast to boolean guards, arithmetic expressions may be impure and have a more C-like semantics. The evaluation relation is given by structural induction for closed expressions. We say that expression e is evaluated in store ϱ_0 to value v and finishes in store ϱ_1, written:

$$\langle\ a, \varrho_0\ \rangle \xrightarrow[Aeval]{} \langle\ v, \varrho_1\ \rangle$$

where $a \in Aexp$, $\varrho_0, \varrho_1 \in Store$ and $v \in D_{\text{real}} \cup D_{\text{int}}$. The rules are very similar to those for boolean expressions. Assignments are not allowed in expressions. The only source of potential impurity are external function calls. This time the resulting store is propagated after modifications made by the C function.

$$\frac{v \in D_{\text{int}} \cup D_{\text{real}}}{\langle\ v, \varrho\ \rangle \xrightarrow[\text{Aeval}]{} \langle\ v, \varrho\ \rangle}$$

$$\frac{a \in Access \qquad \langle\ a, \varrho_0\ \rangle \xrightarrow[\text{access}]{} \langle\ v, \varrho_1\ \rangle}{\langle\ a, \varrho_0\ \rangle \xrightarrow[\text{Aeval}]{} \langle\ v, \varrho_1\ \rangle}$$

$$\frac{o \in Oexpr \qquad \langle\ o, \varrho_0\ \rangle \xrightarrow[\text{output}]{} \langle\ v, \varrho_1\ \rangle}{\langle\ o, \varrho_0\ \rangle \xrightarrow[\text{Aeval}]{} \langle\ v, \varrho_1\ \rangle}$$

$$\frac{\langle\ e_0, \varrho_0\ \rangle \xrightarrow[\text{Aeval}]{} \langle\ v_0, \varrho_1\ \rangle \quad \langle\ e_1, \varrho_1\ \rangle \xrightarrow[\text{Aeval}]{} \langle\ v_1, \varrho_2\ \rangle}{\langle\ e_0\ binop\ e_1, \varrho_0\ \rangle \xrightarrow[\text{Aeval}]{} \langle\ v_0\ binop\ v_1, \varrho_2\ \rangle}$$

$$\frac{\langle\ e, \varrho_0\ \rangle \xrightarrow[\text{Aeval}]{} \langle\ v, \varrho_1\ \rangle}{\langle\ unop\ e, \varrho_0\ \rangle \xrightarrow[\text{Aeval}]{} \langle\ unop\ v, \varrho_1\ \rangle}$$

6.5 Variable Assignments

Assume that a is a variable access and e is an arithmetic expression. Then we say that ϱ is updated with value of expression e at location of lvalue a, writing:

$$\langle\ a = e, \varrho_0\ \rangle \xrightarrow[Asgn]{} \varrho_1$$

The relation is given by rules on figure 1.

6.6 Signal Expression Evaluation

Signal expressions evaluate to (internal) event instances, which are then queued and processed in subsequent steps. The rule for signal evaluation resembles the rule for output evaluation:

$$\frac{\begin{array}{c} e(e_1, ..., e_k) \in Sexpr \\ \forall i \in \{1..k\}.\ \langle\ e_i, \varrho_{i-1}\ \rangle \xrightarrow[\text{Aeval}]{} \langle\ v_i, \varrho_i\ \rangle \\ \forall i \in \{1..k\}.\ v_i \in D(\pi_i(\Gamma_E(e))) \end{array}}{\langle\ e(e_1, ..., e_k), \varrho_0, \omega\ \rangle \xrightarrow[\text{signal}]{} \langle\ \varrho_k, \omega\hat{}\langle e(v_1, ..., v_k)\rangle\ \rangle}$$

6.7 Action Execution

The execution relation for actions is defined in terms of assignments, output evaluation, and signal expression evaluation. Consider an action (as, os, ss), where as is a sequence of assignments, os a sequence of output expressions, and ss a sequence of signal expressions. The action is executed in a given store ϱ_0 and signal queue ω_0, possibly affecting both. We denote it by writing:

$$\langle\ (as, os, ss), \varrho_0, \omega_0\ \rangle \xrightarrow[exec]{} \langle\ \varrho_1, \omega_1\ \rangle$$

Figure 2 presents a single inference rule defining this relation.

6.8 Priority Function

Our semantics is fully deterministic. We follow Lind-Nielsen [15] in parameterizing the semantics with an arbitrary priority function to impose order on processing of otherwise unordered elements.

Definition 40. *Let δ be a natural function on State such that δ is a bijection:*

$$\delta : State \to \{1, ..., |State|\}$$

Then δ is a good priority function for semantics of statecharts. Priority function is a total ordering of elements of State. Priority value 1 is assigned to the state with highest priority.

This way we can describe many different approaches to conflict resolution (most of them completely useless though). The UML statecharts definition in [19] assigns higher priority to more nested transitions and lower to outer ones. In his classic paper [5], Harel proposes a converse approach: outermost transitions having the highest priority.

A priority function is also used to determine the order in which components of an and-state are processed. Reasonable priority function choice will assign consecutive values to children of an and-state providing left to write processing of components.

$$\frac{x \in \mathit{Var} \qquad \mathit{\Gamma}_V(x) \in \mathit{SimpleType} \qquad \langle\ a, \varrho_0\ \rangle \xrightarrow[\text{Aeval}]{} \langle\ v, \varrho_1\ \rangle \qquad v \in D(\mathit{\Gamma}_V(x))}{\langle\ x = e, \varrho_0\ \rangle \xrightarrow[\text{Asgn}]{} \varrho_1[v/x]}$$

$$\frac{\begin{array}{c} x \in \mathit{Var}\ \wedge\ \mathit{\Gamma}_V(x) \notin \mathit{SimpleType}\ \wedge\ \langle\ e_0, \varrho_0\ \rangle \xrightarrow[\text{Aeval}]{} \langle\ v_0, \varrho_1\ \rangle \\ v_0 \in D_{\text{int}}\ \wedge\ v_0 > 0\ \wedge\ \langle\ e_1, \varrho_1\ \rangle \xrightarrow[\text{Aeval}]{} \langle\ v_1, \varrho_2\ \rangle\ \wedge\ v = \varrho_2(x)_{[v_1/v_0]} \end{array}}{\langle\ x[e_0] = e_1, \varrho_0\ \rangle \xrightarrow[\text{Asgn}]{} \varrho_2[v/x]}$$

Fig. 1. The assignment execution relation. The last premise in the second rule means that v is equal to the vector assigned to x in ϱ_2, but with the v_0'th element changed to v_1.

$$\frac{\begin{array}{c} as = \langle a_1, ..., a_k\rangle\ \wedge\ \forall i \in \{1..k\}.\ \langle\ a_i, \varrho_{i-1}\ \rangle \xrightarrow[\text{Asgn}]{} \varrho_i \\ os = \langle o_1, ..., o_m\rangle\ \wedge\ \forall i \in \{1..m\}.\ \langle\ o_i, \varrho_{k+i-1}\ \rangle \xrightarrow[\text{output}]{} \langle\ v_i, \varrho_{k+i}\ \rangle \\ ss = \langle s_1, ..., s_n\rangle\ \wedge\ \forall i \in \{1..n\}.\ \langle\ s_i, \varrho_{k+m+i-1}, \omega_{i-1}\ \rangle \xrightarrow[\text{signal}]{} \langle\ \varrho_{k+m+i}, \omega_i\ \rangle \end{array}}{\langle\ (as, os, ss), \varrho_0, \omega_0\ \rangle \xrightarrow[\text{exec}]{} \langle\ \varrho_{k+m+n}, \omega_n\ \rangle}$$

Fig. 2. The action execution relation. Note that the return values of output calls are discarded. The only way to get the value of a function call is to use it in an expression.

$$\frac{\sigma = \{s\}}{\langle\ s, \sigma, \varrho, \eta, \omega\ \rangle \xrightarrow[\text{exit}]{} \langle\ \varrho, \eta, \omega\ \rangle}$$

$$\frac{\begin{array}{c} \mathit{\Gamma}_S(s) = \mathsf{or}\ \wedge\ s' \sqsubset s\ \wedge\ \sigma \subseteq \mathit{descend}^*(s')\ \wedge\ s \notin \mathit{dom}(\eta_0) \\ \langle\ s', \sigma, \varrho_0, \eta_0, \omega_0\ \rangle \xrightarrow[\text{exit}]{} \langle\ \varrho_1, \eta_1, \omega_1\ \rangle \qquad \langle\ \mathit{exit}(s'), \varrho_1, \omega_1\ \rangle \xrightarrow[\text{exec}]{} \langle\ \varrho_2, \omega_2\ \rangle \end{array}}{\langle\ s, \sigma, \varrho_0, \eta_0, \omega_0\ \rangle \xrightarrow[\text{exit}]{} \langle\ \varrho_2, \eta_1, \omega_2\ \rangle}$$

$$\frac{\begin{array}{c} \mathit{\Gamma}_S(s) = \mathsf{or}\ \wedge\ s' \sqsubset s\ \wedge\ \sigma \subseteq \mathit{descend}^*(s')\ \wedge\ s \in \mathit{dom}(\eta_0) \\ \langle\ s', \sigma, \varrho_0, \eta_0, \omega_0\ \rangle \xrightarrow[\text{exit}]{} \langle\ \varrho_1, \eta_1, \omega_1\ \rangle \qquad \langle\ \mathit{exit}(s'), \varrho_1, \omega_1\ \rangle \xrightarrow[\text{exec}]{} \langle\ \varrho_2, \omega_2\ \rangle \end{array}}{\langle\ s, \sigma, \varrho_0, \eta_0, \omega_0\ \rangle \xrightarrow[\text{exit}]{} \langle\ \varrho_2, \eta_1[s'/s], \omega_2\ \rangle}$$

$$\frac{\begin{array}{c} s \notin \sigma \wedge\ \mathit{\Gamma}_S(s) = \mathsf{and} \wedge \{s_1, ..., s_k\} = \mathit{children}(s) \wedge \forall i, j \in \{1..k\}.\ i < j \Rightarrow \delta(s_i) < \delta(s_j) \\ \forall i \in \{1..k\}.\ \langle\ s_i, \sigma \cap \mathit{descend}^*(s_i), \varrho_{i-1}, \eta_{i-1}, \omega_{i-1}\ \rangle \xrightarrow[\text{exit}]{} \langle\ \varrho_i, \eta_i, \omega_i\ \rangle \end{array}}{\langle\ s, \sigma, \varrho_0, \eta_0, \omega_0\ \rangle \xrightarrow[\text{exit}]{} \langle\ \varrho_k, \eta_k, \omega_k\ \rangle}$$

Fig. 3. The exit relation.

6.9 Exiting a State

As we said before, each state has an exit action assigned. The action should be executed before the state is exited. Also the actions of all descendant states should be executed in proper order (outwards). Occasionally we will also have to leave some levels up above our source state. This depends on where the target state is. Actually the scope to be left only depends on current state configuration and target state. For this reason we define the *exit* relation for current configuration and the scope which should be left. The scope itself is not left but all its active descendants are.

Assume that state s is the above mentioned scope, σ is a maximal orthogonal set relative to s, ϱ_0 is current store, η_0 is the current history marking and ω_0 is the current symbol queue. Then we write:

$$\langle\ s, \sigma, \varrho_0, \eta_0, \omega_0\ \rangle \xrightarrow[exit]{} \langle\ \varrho_1, \eta_1, \omega_1\ \rangle$$

meaning that all descendants of s have been exited and the corresponding exit rules executed in proper order, resulting in new store ϱ_1, new history marking η_1 and new signal queue ω_1. The relation is defined by rules (see figure 3).

Exit rules are executed in bottom-up order. First the most nested descendants are exited, then their parents and recursively until the direct children of s. Descendants of and-state components are exited in components priority ordering δ. For example if order is from left to right, the *exit* relation performs a postorder traversal of the statechart hierarchy.

Exit rules are only executed for and-states, as only those states may have rules assigned. Also note that history marking is updated directly after a state has been exited (third rule).

The well-formedness of the rule (that it is always called on a proper state configuration) follows from theorem 17, the Configuration Subset theorem.

6.10 Entering a State

Entering a state resembles exiting. The entry rules of a state and its descendants should be executed in proper order. Moreover, for each or-state it should be determined which of its children is the default state, determined by the current history marking or the initial marking.

States are normally entered after a certain scope has been exited. So we start not with a proper configuration, but with a maximal orthogonal set of *root* where some non-basic states are contained. Also apart from the default path (indicated by initial and history markings), there is a bunch of targets which take precedence before default values. This makes entering slightly more complex than exiting.

Definition 41 (Default state). *The default child of a given* or*-state s in the current history marking η is given by:*

$$default(s, \eta) = \begin{cases} initial(s) & \text{if } s \in dom(initial) \\ \eta(s) & \text{if } s \in dom(\eta) \end{cases}$$

Note that *default* is total and deterministic on $State_{\mathrm{or}}$ set, since $dom(\eta) = State_{\mathrm{or}} \setminus dom(initial)$.

When a transition is fired we are not only interested in entering its target state. Obviously the descendants of the target state should also be entered in the process (via the default path). Sometimes, when the transition goes down the hierarchy, we will also enter some ancestors of target, or even some states orthogonal to the target. The latter happens when control enters a component of an and-state from outside: then one must initialize all other components of the and-state as well.

Thus we give the enter relation definition for scope s and the set of target states T such that $T \subseteq descend^*(s)$ and write

$$\langle\ T, s, \varrho_0, \eta_0, \omega_0\ \rangle \xrightarrow[enter]{} \langle\ \sigma, \varrho_1, \omega_1\ \rangle,$$

meaning that descendants of s have been properly entered, resulting in σ, a state configuration of s. All actions have been executed in top-down order (priority order for elements on the same level of hierarchy). If priority function δ orders children of and-states from left to right then the enter relation performs preorder traversal of statechart hierarchy.

The history marking is not changed in entering phase. The relation is defined by rules given in figure 4.

6.11 Transition Firing

The scope of changes involved in switching to an arbitrary state s depends on the target state s itself and the current configuration σ. For a given target state s we need to find the closest relative in the current configuration (the state $s' \in \sigma$ minimizing $NCA(parent(s), parent(s'))$). The nearest common ancestor found is the implicit scope of the transition. The explicit and implicit scopes of a transition are identical for one of the transition's targets: the explicit target.

$$\frac{T = \emptyset \quad \Gamma_S(s) = \mathsf{and} \quad \mathit{children}(s) = \emptyset}{\langle\ T, s, \varrho, \eta, \omega\ \rangle \xrightarrow[\text{enter}]{} \langle\ \{s\}, \varrho, \eta\ \rangle}$$

$$\frac{\begin{array}{c} T \subseteq \mathit{descend}^*(s)\ \wedge\ \Gamma_S(s) = \mathsf{and}\ \wedge\ \{s_1, ..., s_k\}\ = \mathit{children}(s) \\ \forall i, j\in \{1..k\}.\ i < j \Rightarrow \delta(s_i) < \delta(s_j) \quad \forall i\in \{1..k\}.\ \langle\ T \cap \mathit{descend}^*(s_i), s_i, \varrho_{i-1}, \eta, \omega_{i-1}\ \rangle \xrightarrow[\text{enter}]{} \langle\ \sigma_i, \varrho_i, \omega_i\ \rangle \end{array}}{\langle\ T, s, \varrho_0, \eta, \omega_0\ \rangle \xrightarrow[\text{enter}]{} \left\langle\ \bigcup_{i=1}^k \sigma_i, \varrho_k, \omega_k\ \right\rangle}$$

$$\frac{\begin{array}{c} T \neq \emptyset\ \wedge\ T \subseteq \mathit{descend}^+(s)\ \wedge\ \Gamma_S(s) = \mathsf{or}\ \wedge\ \mathit{NCA}(T) \sqsubset^* s' \sqsubset s \\ \langle\ \mathit{entry}(s'), \varrho_0, \omega_0\ \rangle \xrightarrow[\text{exec}]{} \langle\ \varrho_1, \omega_1\ \rangle \quad \langle\ T \backslash \{s'\}, s', \varrho_1, \eta, \omega_1\ \rangle \xrightarrow[\text{enter}]{} \langle\ \sigma, \varrho_2, \omega_2\ \rangle \end{array}}{\langle\ T, s, \varrho_0, \eta, \omega_0\ \rangle \xrightarrow[\text{enter}]{} \langle\ \sigma, \varrho_2, \omega_2\ \rangle}$$

$$\frac{\begin{array}{c} T = \emptyset\ \wedge\ \Gamma_S(s) = \mathsf{or}\ \wedge\ s' = \mathit{default}(s, \eta) \\ \langle\ \mathit{entry}(s'), \varrho_0, \omega_0\ \rangle \xrightarrow[\text{exec}]{} \langle\ \varrho_1, \omega_1\ \rangle \quad \langle\ \emptyset, s', \varrho_1, \eta, \omega_1\ \rangle \xrightarrow[\text{enter}]{} \langle\ \sigma, \varrho_2, \omega_2\ \rangle \end{array}}{\langle\ T, s, \varrho_0, \eta, \omega_0\ \rangle \xrightarrow[\text{enter}]{} \langle\ \sigma, \varrho_2, \omega_2\ \rangle}$$

Fig. 4. The enter relation. Well-formedness of the configuration resulting in second rule is ensured by theorem 15, the configuration union theorem.

$$\frac{\begin{array}{c} \mathit{iscopes}(t, \sigma_0) = \{s_1, ..., s_k\}\ \wedge\ \forall i, j\in \{1..k\}.\ i < j \Rightarrow \delta(s_i) < \delta(s_j) \\ \forall i\in \{1..k\}.\ \langle\ s_i, \sigma_0 \cap \mathit{descend}^*(s_i), \varrho_{i-1}, \eta_{i-1}, \omega_{i-1}\ \rangle \xrightarrow[\text{exit}]{} \langle\ \varrho_i, \eta_i, \omega_i\ \rangle \\ \langle\ \mathit{action}(t), \varrho_k, \omega_k\ \rangle \xrightarrow[\text{exec}]{} \langle\ \varrho_{k+1}, \omega_{k+1}\ \rangle \\ \forall i\in \{1..k\}.\ \langle\ \mathit{targets}(t) \cap \mathit{descend}^*(s_i), s_i, \varrho_{k+i}, \eta_k, \omega_{k+i}\ \rangle \xrightarrow[\text{enter}]{} \langle\ \sigma_i, \varrho_{k+i+1}, \omega_{k+i+1}\ \rangle \\ \sigma_{k+1} = \sigma_0 \setminus \left(\bigcup_{j=1}^k \mathit{descend}^*(s_j)\right) \cup \left(\bigcup_{j=1}^k \sigma_j\right) \end{array}}{\langle\ t, \sigma_0, \varrho_0, \eta_0, \omega_0\ \rangle \xrightarrow[\text{fire}]{} \langle\ \sigma_{k+1}, \varrho_{2k+1}, \eta_k, \omega_{2k+1}\ \rangle}$$

Fig. 5. The single-transition firing relation. Configuration σ_{k+1} is guaranteed to be a well-formed state configuration by theorem 19, the substitution theorem for configurations.

Definition 42 (Implicit scope). *Let s be the target state and σ the current state configuration (global state). The implicit scope is defined recursively:*

$$iscope(\sigma, s) = \begin{cases} parent(s) \\ \quad \text{if } descend^*(parent(s)) \cap \sigma \neq \emptyset \\ iscope(\sigma, parent(s)) \ \text{otherwise} \end{cases}$$

The implicit scope determines the impact of an actual state change. All descendants of the scope will be exited when firing a transition, and some new states (depending on targets and markings) will be entered. Each transition has several implicit scopes: at most as many as there are target states. Two orthogonal targets may have the same implicit scope (for instance when simultaneously entering several components of the same and-state).

Definition 43 (Generalized Scope). *The generalized scope of transition t in configuration σ is the set of its implicit scopes:*

$$iscopes(\sigma, t) = \{iscope(\sigma, s) \mid s \in targets(t)\}$$

Observation 44 (Scope Properties).

1. *The implicit scope is always an* or*-state.*
2. $\forall \sigma, t. \ |iscopes(\sigma, t)| \leq |targets(t)|$.
3. *The generalized scope is an orthogonal set.*

Proof. (Sketch) The first property follows from the fact that only and-states are targets and from definition of implicit scope. The two other facts can be proved from static correctness conditions for transitions. □

The single-transition firing relation relates a transition, the current state, store, history marking and signal queue with a new state, store, history marking and signal queue. This means that while the transition is fired, a new state configuration may arise, some variables may be modified, the history marking may be updated, and some signals may be issued. The relation is defined by combined application of formerly specified exit, exec and enter relations. See figure 5.

The transition must be enabled and all local substitutions for event parameters should be done before the firing (see microstep definition, figure 6 for the context).

Our semantics of firing has a subtle difference compared to the visualSTATE implementation. All targets are treated in the same way: there is no special semantics for the explicit target. This is because we use the same *iscope* definition for all targets. As a result, a transition forcing an additional already active state behaves as a safe loop transition on the forced state: it is exited and entered again. In visualSTATE such transitions will not perform the forced state's exit and entry actions: they behave as if the those actions have been disabled.

We think that having different semantics for explicit and implicit targets is not necessary. It complicates both definition and implementation. If necessary, a model transformation could decorate transitions with extra conditions to guarantee strict visualSTATE behavior also within our framework. Our approach presents a minor difference from user point of view and allows cleaner semantics and slightly more efficient implementations.

6.12 Enabled Transitions Set

This section's main concern is the scheduling of transitions:

Definition 45 (Enabled Transition). *A transition t is enabled for event instance $e(v_1, ..., v_k)$ in state σ and store ϱ iff*

1. $e = event(t)$
2. $\forall s \in pos(t). \ \exists s' \in \sigma. \ s' \sqsubset^* s$
3. $\forall s \in neg(t). \ \forall s' \in \sigma. \ s' \not\sqsubset^* s$
4. *If* $\pi_1(t) = e(p_1, ..., p_k)$ *then* $\exists v \neq 0.$
   $\langle \ guard(t) \, [v_1/p_1, ..., v_k/p_k] \, , \varrho \ \rangle \xrightarrow[Beval]{} v$

Definition 46 (Enabled Transitions Set). *We will write $enabled(e(v_1, ..., v_k), \sigma, \varrho)$ for the set of all enabled transitions for current event instance $e(v_1, ..., v_k)$, state σ, and store ϱ.*

Note that boolean expression evaluation plays an important role in the definition of enabledness. Here the assumption of pure boolean expressions is important. We required boolean expressions to be pure (having no side effects). This guarantees that whatever order we take to iterate over transitions in the system, the same transitions will be considered enabled. Moreover, this would permit an optimized implementation to duplicate or skip the evaluation of a boolean expression, without affecting the global variable state.

Unfortunately, this is insufficient to guarantee determinism of the enabledness property. C functions called in boolean expressions may (and normally should) refer to external properties of devices, which in turn are dynamic in time. Only the full synchrony assumption [2], that all guards are computed infinitely fast, can achieve a deterministic enabled set computation.

For efficiency reasons we impose yet another assumption: actions executed in the current microstep should not influence the value of guard conditions in the same microstep. In this way we guarantee that the computation of action functions and guards may be interleaved at runtime, so there is no need for an expensive explicit computation of the set of enabled transitions. This assumption seems to be restrictive and unclear at the first sight, but actually it agrees with the informal understanding of statechart semantics. Two transitions in concurrent components are fired in parallel, so if determinism is to be ensured, there should be no interaction between their guards and actions.

6.13 Conflict Resolution

Informally two transitions are in conflict if they should both happen in the same scope. Formally:

Definition 47 (Conflicting Transitions). *Two distinct transitions t_1, t_2 are in conflict iff*

$$\exists s_1 \in \mathit{iscopes}(t_1), s_2 \in \mathit{iscopes}(t_2).\ s_1 \sqsubset^* s_2 \vee s_2 \sqsubset^* s_1$$

This means some activities of transitions overlap.

The current version of visualSTATE assumes that systems containing possible conflicts are illegal. Verification tools aid developers in detecting and eliminating potential (reachable) conflicts. Here we follow [5, 19] and distinguish resolvable and non-resolvable conflicts. This distinction is also likely to be implemented in future releases of visualSTATE.

Definition 48 (Non-resolvable conflict). *Two transitions are in non-resolvable conflict if they are conflicting and their explicit scopes are identical. This can be rephrased as priorities of transitions being the same (the priority function was defined to be a bijection).*

We assume that systems containing non-resolvable conflicts are illegal and we do not consider this kind of conflicts from now on.

The remaining conflicts are resolvable. For any pair of transitions in a resolvable conflict, UML discards the outermost one, whereas Harel's statecharts preserve the outermost one and discard the inner one. In our semantics we use the priority function δ to pick one of the conflicting transitions: if two transitions are in conflict the one with lower priority is discarded from the active set. The priority approach allows modeling of both the UML and the Harel styles.

Note that it may often happen that only some parts of transitions are in conflict; transitions having many scopes may only interfere on some of them. Nevertheless with this approach one of the transitions is discarded as a whole – not only the conflicting part.

Definition 49 (Resolved Set). *Let $\{t_1, ..., t_k\} = \mathit{enabled}(e(v_1, ..., v_m), \sigma, \varrho)$ be a set of enabled transitions, possibly containing some resolvable conflicts, i.e.*

$$\forall i, j \in \{1..k\}.\ i < j \Rightarrow \delta(\mathit{scope}(t_i)) < \delta(\mathit{scope}(t_j))$$

A resolved enabled set $\mathit{renabled}(e(v_1, ..., v_m), \sigma, \varrho)$ is a set such that:

1. *The set $\mathit{renabled}(e(v_1, ..., v_m), \sigma, \varrho)$ is a maximal subset of $\mathit{enabled}(e(v_1, ..., v_m), \sigma, \varrho)$ containing no conflicting transitions.*
2. *If for any $t', t'' \in \mathit{enabled}(e(v_1, ..., v_m), \sigma, \varrho)$ it holds that $\delta(t') < \delta(t'')$ and t is in conflict with t', then $t'' \notin \mathit{renabled}(e(v_1, ..., v_m), \sigma, \varrho)$*

Note that this definition is the only place where the actual arrow of a transition's concrete syntax is used, in the guise of the static *scope*() function. Remember that we only model the (explicit) scope of a visual arrow in the abstract syntax, and this scope is used (maximized) in condition (2) of the definition of resolved enabled set.

Dynamic conflict resolution is a well-known concept in history of statechart languages. Still we believe that this is an expensive concept to implement at runtime, and that it does not aid good modeling practices. It is doubtful whether it should be implemented in a code generator targeting embedded systems. The assumption that there are no conflicts requires a cleaner model and gives better opportunities for compile-time optimizations.

6.14 Microstep

A single microstep of execution constitutes of firing all transitions that are active, given a state, store, and event instance. We denote it

$$\langle\ e(v_1, ..., v_m), \sigma_0, \varrho_0, \eta_0, \omega_0\ \rangle \xrightarrow[\mathit{micro}]{} \langle\ \sigma_1, \varrho_1, \eta_1, \omega_1\ \rangle$$

and the semantics is to fire all enabled transitions ordered by priority. See figure 6.

Microstep execution also implies event parameter substitution in transition arrows. This is a textual substitution at the syntactical level. Note that substitution cannot be made on the variable store

$$\frac{\begin{array}{c} \mathit{renabled}(e(v_1,...,v_m),\sigma_0,\varrho_0) = \{t_1,...,t_k\} \ \wedge \ \forall i,j \in \{1..k\}.\ i<j \Rightarrow \delta(\mathit{scope}(t_i)) < \delta(\mathit{scope}(t_j)) \\ \forall i \in \{1..k\}.\ \exists! e(x_1^k,...,x_m^k) = \pi_1(t_k).\ \Big\langle\ t_i[v_1/x_1^k,...,v_m/x_m^k], \sigma_{i-1}, \varrho_{i-1}, \eta_{i-1}, \omega_{i-1}\ \Big\rangle \xrightarrow[\text{fire}]{} \langle\ \sigma_i, \varrho_i, \eta_i, \omega_i\ \rangle \end{array}}{\langle\ e(v_1,...,v_m), \sigma_0, \varrho_0, \eta_0, \omega_0\ \rangle \xrightarrow[\text{micro}]{} \langle\ \sigma_k, \varrho_k, \eta_k, \omega_k\ \rangle}$$

Fig. 6. The microstep relation.

$$\frac{}{\langle\ \sigma_0, \varrho_0, \eta_0, \langle\rangle\ \rangle \xrightarrow[\text{macro}]{} \langle\ \sigma_0, \varrho_0, \eta_0\ \rangle}$$

$$\frac{\langle\ e(v_1,...,v_m), \sigma_0, \varrho_0, \eta_0, \omega_0\ \rangle \xrightarrow[\text{micro}]{} \langle\ \sigma_1, \varrho_1, \eta_1, \omega_1\ \rangle \qquad \langle\ \sigma_1, \varrho_1, \eta_1, \omega_1\ \rangle \xrightarrow[\text{macro}]{} \langle\ \sigma_2, \varrho_2, \eta_2\ \rangle}{\langle\ \sigma_0, \varrho_0, \eta_0, \langle e(v_1,...,v_m)\rangle\hat{}\,\omega_0\ \rangle \xrightarrow[\text{macro}]{} \langle\ \sigma_2, \varrho_2, \eta_2\ \rangle}$$

Fig. 7. The macrostep relation.

instead, which is often equivalent in typical imperative languages. This would overwrite the values of global variables, affecting execution of external C functions (which are not supposed to access values of event parameters anyway).

6.15 Macrostep

A macrostep is a chain of microsteps initiated by single external event. After performing the microstep for the external event the microsteps are reiterated over internally signaled events until the system reaches stability (the signal queue is empty).

The macrostep relation is defined for an initial state configuration, store, history marking, and an external event instance (embedded as the first element of the signal queue).

$$\langle\ \sigma_0, \varrho_0, \eta_0, \langle e(v_1,....,v_m)\rangle\ \rangle \xrightarrow[\mathit{macro}]{} \langle\ \sigma_1, \varrho_1, \eta_1\ \rangle$$

See figure 7 for the rules. Not surprisingly the rules resemble while-loop execution rules for imperative languages: a macrostep is usually implemented as a while loop over microsteps.

6.16 Initialization Step

The step execution relations above are given for a system in operation. They demand a state configuration to begin with. However, the initial state configuration is not given in concrete syntax, but must be derived from the initial and history markings. In fact, the system must be *initialized*, that is, entry actions of `and`-states must be executed while building the initial state configuration. The process is given by execution of enter relation for *root* scope with empty explicit target set:

$$\langle\ \emptyset, \mathit{root}, \varrho_0, \eta_0, \langle\rangle\ \rangle \xrightarrow[\mathit{enter}]{} \langle\ \sigma_1, \varrho_1, \omega_1\ \rangle$$

where ϱ_0 is an initial variable environment (part of the system) and η_0 is an initial history marking. Before the execution may proceed with first external event instance, events signaled during the initialization step should be processed:

$$\langle\ \sigma_1, \varrho_1, \eta_0, \omega_0\ \rangle \xrightarrow[\mathit{macro}]{} \langle\ \sigma, \varrho, \eta\ \rangle$$

The state configuration σ, variable store ϱ and history marking η together with empty queue constitute the initial global state for processing of first external event.

7 Determinism

We claim that the semantics presented above is deterministic. The formal proof would be lengthy and not particularly inventive, so let us argue only informally. The semantics is deterministic because we ensured that every construction step of semantics can be taken deterministically. This was achieved by several means:

1. Whenever a relation is defined by rules we ensure that rules are exclusive.
2. All auxiliary functions are normal algebraic functions, so they are deterministic.
3. Strong assumptions have been made about absence of interdependencies between guards and actions, and about the purity of guards, to facilitate a deterministic result regardless of order of execution.

4. Whenever nondeterministic choice appeared naturally in the unordered set of elements we ordered it with priority function to ensure fixed control flow.
5. We used existential quantifications in relation rules generously. The context is always simple enough to prove that the choice of value is unique.

Note that the trick with parameterizing the semantics with priority functions allows to keep some advantages of nondeterminism, while determinism needed in the implementation is still guaranteed. We still allow various implementations, but all of them have to be deterministic (see for instance figure 4, third rule).

8 Conclusions and Future Work

We presented a global operational semantics for visualSTATE systems. The core of the semantics has been presented formally. The presentation was annotated with various comments on implementation issues. We also provide some hints on how other elements may be incorporated in this framework (see appendix B).

The present specification has been used in the implementation of SCOPE, an experimental code synthesizer for visualSTATE models maintained at the IT University of Copenhagen. SCOPE focuses on efficient, though formally motivated generation of embedded software.

Our semantics does not include a notion of equivalence of subprograms. We believe that a general notion of compositionality is not needed for statecharts. Still we hope to propose some restricted notions for equivalence of states and transitions so some model optimizations may be performed.

9 Acknowledgements

The authors would like to thank IAR Inc. for extensive support throughout this project, especially to Kent Ren Simonsen for quick and patient replies to numerous detailed questions.

References

[1] Behrmann, G., Larsen, K. G., Andersen, H. R., Hulgaard, H., and Lind Nielsen, J. Verification of hierarchical state/event systems using reusability and compositionality. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Mar. 1999), vol. 1579 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 163–177.

[2] Berry, G. The Esterel v5 language primer. version v5_91, JULY 2000.

[3] Björklund, D., Lilius, J., and Porres, I. Towards efficient code synthesis from statecharts. In *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. Workshop of the pUML-Group held together with the UML'2001* (Toronto,Canada, October 1st, 2001), A. Evans, R. France, and A. M. B. Rumpe, Eds., Lecture Notes in Informatics P-7, GI.

[4] Drusinsky, D., and Harel, D. Using statecharts for hardware description and synthesis. *IEEE Transactions on Computer Aided Design 8*, 7 (1989), 798–807.

[5] Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8* (1987), 231–274.

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

[7] Harel, D., Pnueli, A., J.P.Schmidt, and R.Sherman. On the formal semantics of statecharts. In *Proceedings of 2nd IEEE Symposium on Logic in Computer Science* (New York, 1988), IEEE Press, pp. 396–406.

[8] Huizing, C., and de Roever, W. Introduction to design choices in the semantics of Statecharts. *Information Processing Letters 37* (1991), 205–213.

[9] I-Logix Inc. Rhapsody® in MicroC. http://www.ilogix.com.

[10] IAR Inc. IAR visualSTATE®. http://www.iar.com/Products/VS/.

[11] IAR Inc. IAR visualSTATE® concept guide, 1999. Part of visualSTATE standard documentation.

[12] IAR Inc. IAR visualSTATE® reference guide, Dec. 2000. Part of visualSTATE standard documentation.

[13] IAR Inc. IAR visualSTATE® user guide, Oct. 2000. Part of visualSTATE standard documentation.

[14] Larsen, M. VisualSTATE rule base 4.0 specification (IAR Inc.). Internal Document of IAR Inc., 1998.

[15] Lind Nielsen, J. B. *Verification of Large/State Event Systems*. PhD thesis, Technical University of Denmark, April 2000.

[16] Maraninchi, F., and Halbwachs, N. Compiling ARGOS into boolean equations. In *Proc. 4th Int. School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT)* (Uppsala, Sweden, Sept. 1996), vol. 1135 of *Lecture Notes in Computer Science*, Springer-Verlag.

[17] Maraninchi, F., and Rémond, Y. Argos: an automaton-based synchronous language. *Computer Languages 27*, 1–3 (2001), 61–92.

[18] Mikk, E., Lakhnech, Y., Petersohn, C., and Siegel, M. On formal semantics of statecharts as supported by STATEMATE. In *2nd BCS-FACS Northern Formal Methods Workshop* (1997), Springer-Verlag.

[19] Object Management Group. OMG Unified Modelling Language specification, 1999. http://www.omg.org.

[20] Rational Software Corp. Rational Rose® Real Time (RoseRT). http://www.rational.com/products/rosert/.

[21] von der Beeck, M. A comparison of statecharts variants. In *ProCoS: Third International Symposium on Formal Techiniques in Real Time and Fault-Tolerant Systems.* (1994), vol. 863 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 128–148.

A Concrete Syntax Example

Figure A presents a system in concrete syntax. This is a simplified microwave oven controller. The main components are: standby button, door, microwave emitter, grill heater, mode switch, and safety unit. The initial state of the system is *OFF*, which can be changed by pressing the *Standby* button. The door is built in such a way that it can only be opened when the power is on. Moreover, the system cannot enter the *OFF* state if the door is opened. This is guaranteed by adding a positive condition *Closed* to the transition triggered when the *Standby* button is pressed.

The right region of the system represents the state of the user interface. Currently there is only one switch having three modes. It indicates the kind of heating which should be used in the system: microwave heat (H), electrical grill (G), or both (HG). Note that microwave heat is the default, as indicated by the initial marking placed on state *H*. The mode can be changed by consecutive presses to the *Mode* button, which generates *ModePressed* events.

The light inside the oven is switched on whenever the door is closed. It is switched back off as soon as the user opens the door. The first time the oven is turned on and the door is closed, both heaters do not operate (they are in the idle mode). The user needs to initiate heating using the *Start* button. Note that the proper heater (or both heaters) are started depending on the state of mode.

Whenever the door is opened (regardless of heaters states) all heating is switched off (see exit actions of *Closed* state). The corresponding C functions presumably perform the task of communicating with the actual hardware. The same happens whenever a *HeatAlarm* is issued by the safety unit. If the door is closed the heating units resume in the state in which they were interrupted (due to use of history marking). When the safety alarm is off the heaters are not restored to their original state; instead they enter the idle mode (to increase safety). Note that the transition from *Overheated* to *Closed* has two targets: one explicit (*Grill.idle*) and one implicit (*MicroIdle*).

B Omitted Language Features

We have left some language constructs and properties out of the description. The reason was to simplify the presentation, which is already full of details. We discuss them in this section, giving some advice how they can be incorporated into the above framework. This section is not as self-contained as the main text, but assumes some basic knowledge about statecharts. This can be obtained from informal descriptions since these features are rarely discussed in formal definitions.

B.1 Qualified State Names

The abstract syntax used in the above description operates on abstract objects such as states, ignoring their identifiers (except for variables and parameters). This does not reflect the full static semantics of state names needed to implement a compiler.

State names have local scope in visualSTATE, so they need not be unique. For uniqueness, qualified names may be used. The qualified name of state s includes dot-separated ancestor names. The name of s is fully qualified if it starts with the *root* state and ends with s. The fully qualified names are isomorphic to states in our presentation.

B.2 Name Spaces

State names have a name space on their own. Other elements, such as events, variables and outputs, share a space name because all of them exist together in the synthesized C program.

B.3 External and Internal Variables

In visualSTATE, variables come in two flavors: external and internal. External variables are visible in user space routines. Internal variables can only be referred and modified from visualSTATE system space (for instance in assignments put on transitions). The important semantical difference is that whenever a

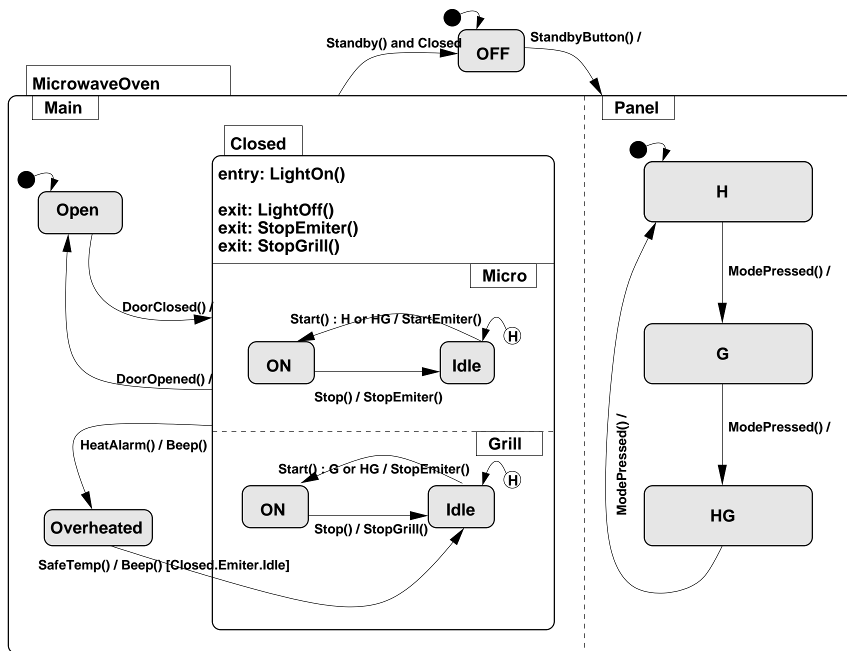

Fig. 8. Simple microwave system.

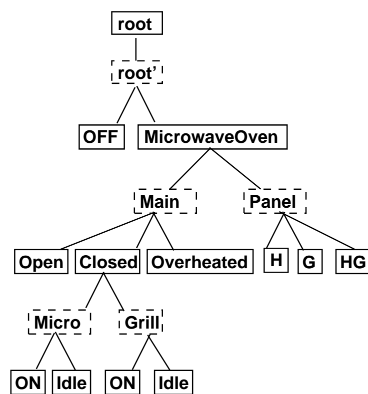

Fig. 9. Hierarchy tree (decomposition tree) for microwave oven system. Edges connect nodes representing related states. For instance $Panel \sqsubset MicrowaveOven$. Nodes with solid boundaries represent and-states, nodes with dash-line boundaries represent or-states.

C function is called, there is a guarantee that internal variables will remain unchanged. This mainly aids verification of visualSTATE models.

B.4 Named Constants

IAR visualSTATE supports named constants, which we have not modeled. They may be modeled as a mapping from subset of identifiers to values. One should add constants to rules using substitution. Static correctness conditions would have to be extended too. For instance name clashes between variables and constants should be avoided.

B.5 Internal and External Events

We have not distinguished between external and internal events in our definition. The semantics of both kinds of events is the same. The only difference is that external events (also called events) are detected in external environment while internal events (also called signals) are only generated internally (in actions). Events are then fed to the step with macrostep relation (section 6.15). The microstep relation (see section 6.14) starts with processing an external event and then continues processing signals (internal events) until the queue is empty. The next external event must be consumed by a new macrostep.

The distinction between (external) events and (internal) signals is mostly helpful in verification, to limit the number of potential input events in a step. It may also prove to be helpful in code generation, though. The mechanism for internal event broadcast and reception can sometimes be transformed to a multitarget transition, which may be implemented more efficiently.

B.6 Event Groups

Our transitions can only contain one triggering event. Contrary to other statechart variants, event conjunctions are not allowed. This is because only one event is active at a time in the system. Still the triggering conditions are slightly more flexible than what we presented. A disjunction of events may be used instead of single event. This may be modeled as syntactic sugar for several transitions. However, implementation-wise this may prove to be an expensive approach.

One can instead propose a notion of triggers of union type (of event binding and event sets) and make them the first component of a transition, instead of the event binding. A special rule should be provided for firing event group transitions. This complicates the semantics on one hand, but explicitly represents a sharing between transitions on the other, which definitely is an interesting implementation issue.

Disjunctions of events are called *event groups* in visualSTATE. Transitions with event group triggers do not have access to parameters of events.

B.7 Type System

We have already mentioned that the type system presented above is very restricted. In reality visualSTATE tools support a handful of various arithmetic types with several precisions and void pointer types. As for aggregations, only one-dimensional vectors are supported. The most notable features of the actual type system is the support for void functions (to be used as outputs on transitions), void pointers and variable ranges.

Void pointer is the only pointer type currently supported. Moreover there is no support for typical pointer related operations such as dereferencing, address taking, and type casting. Consequently pointers behave like plain integers. They can only be reasonably used in external code. This is the reason why we have not devoted any special attention to them.

Another important verification-oriented feature of visualSTATE is ranges. A range is a restriction on the value set of an integer variable. Ranges may be put on variable types and on array types. In the latter case, it means that all variable cells are supposed to obey the range restriction. Ranges may significantly decrease the state space of the model and may thus make verification feasible.

Ranges may also become interesting in some code generation schemes, namely those targeting typical workstations, where there is enough environment and resources to implemented full runtime correctness checking. This is true for instance for simulators, animation tools, etc.

Ranges may be implemented by extending the type system and decorating evaluation rules with guards on assignments. It should be stressed, however, that there is no compile-time means available to analyze the correctness of assignments in the general case (because expressions may call external C functions). For this reason ranges, cannot be currently efficiently used in embedded systems oriented code synthesizers.

B.8 History

There are three types of markings available: initial marking, history marking and deep history marking. We only defined the former two: initial and history marking. The *deep history marking* may be understood as syntactic sugar for a history marking on an entire subtree of the hierarchy tree. If a deep history state is entered, not only is its default child activated from the last saved value, but the whole subconfiguration of its descendants is restored.

Again one could try to model deep history markings more efficiently than by using a syntactic sugar expansion, which may result in explosion. A deep history marking is a marking from or-states to state configurations. For each state s the value of a marking would be a state configuration of state s. In this case only a basic state assignment would be saved instead of an assignment for all internal nodes of the decomposition subtree.

B.9 Internal Reactions

Internal reactions are transitions with no targets. They behave like self loop transitions, except that no states are entered nor exited when the internal rule if fired. Introducing internal reactions to the above framework demands relaxing consistency condition for transitions. A transition should be allowed to have an empty target set (see section 4). Also another rule in firing relation specific for internal reactions should be introduced (see section 6.11). It would omit the exit and enter sections as internal reactions do not modify current configuration.

Internal reactions can also be conveniently described as syntactic sugar without excessive memory cost. In this case no modifications in the framework are need. Each internal reaction can be translated to a state machine containing a single state and a single transition, placed in a newly created or-state (region) in the state where it belongs. The transition will safely loop over a single state without calling any exit/entry rules and yet still perform the intended transition actions.

B.10 Do-Reactions

A *do-reaction* (or *do-invocation*) is a state machine belonging to an and-state s which is active while the parent state is active. It differs from a typical state machine belonging to the or-state child of an and-state by having a special *final state* and a *termination transition* going out of the parent state. Whenever the do-reaction enters a designated final state the termination transition fires instantly (in the same step).

Do-reactions can be modeled as an ordinary state machine within an and-state. The do-reaction should be placed in a region (or-state) of its own, preserving all its structure, transitions etc.

Assume that there is one termination transition and a number of transitions targeting a final state (called *finalizing transitions*). We add a fresh transition for each finalizing transition, but redirect its target to the target of the termination transition (so it becomes a cross level transition). The new transitions should have a firing condition which is a conjunction of firing conditions for finalizing transition and termination transition. Similarly, the action parts should be sequenced: first execute finalizing transition actions, then termination transition actions.

A problem arises if side effects of one action interfere with other actions. This may be overcome by building a small action language so we can easily interleave actions of different kinds (signals, outputs, assignments). In the current triple model, actions of different types cannot be interleaved.

To ensure determinism we conjoin guards of all existing finalizing transitions with the negation of the termination transition's firing condition.

Note that we will double the number of finalizing transitions for each termination transition outgoing from s. To avoid a size explosion, one could model do-reactions explicitly, introducing final states, termination transitions and special firing rules for transitions targeting final states. However this would complicate the execution model seriously.

B.11 Timers

There is some support for time in visualSTATE models. *Timer* is a special kind of action which starts a timer and issues an indicated event when the timer expires. The language provides this construct only syntactically. A timer action actually calls a user-provided function which should start a system timer and issue appropriate (external) time events. Because of the implementation's simplicity there is no difference between timer actions and other outputs, from a code synthesis point of view. In our framework we simply translate them to outputs.

B.12 Models versus Systems

The module language distinguishes systems and models. Systems are full visualSTATE specifications

which may be composed in parallel with other (perhaps identical) systems. Only one of the systems is operational at a time. The user code is supposed to call a provided API function, switching system activity. Systems are totally isolated. There may be no communication among them, except for a set of common variables.

We do not define multisystem models as this is not a statechart issue specifically. If one needs to describe a model like that, one should define it as tuple of systems, an activity indicator, and a global variable store. The store of each system is composed of the system's store and the global store. The switching/scheduling algorithm needs to be given in terms of C semantics.

B.13 Partly Defined Elements

The visualSTATE syntax is somewhat more flexible than the one presented above. It is syntactically legal to have states with no identifiers or without a default (history or initial) indicator. Similarly, other elements may be only partially specified. This is a common feature in commercial tools: the user should be allowed to save a (possibly incomplete) model at any stage of design.

As incomplete programs do not have reasonable semantics, we tightened the syntax conditions, only considering complete designs. Alternatively one could try to give semantics of incomplete design (by ignoring incomplete elements for instance) to facilitate compilation of programs at any design stage. This may be useful for tools like simulators. A system could be analyzed before the end of development process.

C Terminology Survey

Numerous kinds of statecharts have been described. We have already mentioned a survey by von Beeck[21] listing 20 variants, and many new versions of syntax and semantics have appeared since that survey was written, most notably UML statechart diagrams and their industrial implementations. The abundance of statechart variants discourages exchange of information and cooperation between various groups working on verification and implementation.

Table 1 contains a brief comparison of four statechart versions: present definition, visualSTATE manuals, UML statechart diagrams and Harel's original proposal. The aim is to make visualSTATE statecharts easier to understand for those readers who are familiar with statechart techniques. For this reason the survey is informal and incomplete. Also it is not self-contained in many aspects.

All native term variants are written in *italics*. Comments and descriptive names (when the term is not defined explicitly in given variant) are written in Roman font. If a feature is not supported we place a dash (–), arrows are sometimes used to refer to content of adjacent cell to indicate similarities.

Table 1: Comparison of statechart terminology and semantics.

This Paper	**visualstate** [11, 12]	**OMG UML Spec.**[19]	**D.Harel**[5]
Language, diagrams and modules			
[def. 36] ⟶	*system*	visualSTATE system roughly corresponds to a state diagram for a composite complex *object*	–
–	*model*	*model* is a very broad term referring to the whole object-oriented design, encompassing many diagrams of various types	–
*visual*STATE *statechart* or *statechart*	*statechart diagram* (*state diagram* in older releases of visualSTATE)	*statechart diagram*	*statechart*
States and Hierarchy			
[p. 3] *state*	*state* or *region*	*state* or *subregion*	*state* or *component*
[p. 4] *basic state* or childless and-*state*	*simple state*	*simple state*	A simple *state* not participating in any *abstraction* or *refinement*
[p. 4] and-*state* having children	*state*	*composite state*	result of and-decomposition of its children
[p. 4] or-*states*	*region*	*a subregion*, a special type of *composite state*	result of xor-decomposition of its children
[def. 8] *orthogonal states*	*concurrent states*	*concurrent states*	*orthogonal states* or components
[def. 1] *root*	*topstate*	a toplevel *state*	⟵
–	*final state* see *termination transition*	*final state* see *completion transition*	–
–	–	*synch pseudostate* is used with *concurrent transitions*	there is some notation for transitions with common target which contains a pseudostate in the middle, but the semantics is unclear
–	–	–	*overlapping states*

| **This Paper** | **visualstate** [11, 12] | **OMG UML Spec.**[19] | **D.Harel**[5] |

Table 1: Comparison of statechart terminology and semantics.

This Paper	**visualstate** [11, 12]	**OMG UML Spec.**[19]	**D.Harel**[5]
Default states indication			
[def. 20] The *initial marking* is not a state. No transitions incoming and none outgoing. May not coexist with a *history marking* but this can be overcome using explicit targets in transition.	an *initial state* is the state marked with some kind of initial indicator. It corresponds to the target of initial transition in UML. Initial state is a value of initial marking function from this paper.	an *initial pseudostate* may have many transitions incoming and one outgoing. The outgoing transition may have an action assigned	*default state* is what visualSTATE calls an initial state. The pseudostate is also present as in UML with similar semantics, but it is not named
[def. 21] *history marking* is not a state. No transitions incoming and none outgoing. May not coexist with *initial marking* in the same state, but this can be overcome by using explicit targets in transitions.	A *history state* is the state marked with a history indicator. It corresponds to the first value of the history state indicator in UML.	*history state indicator* can have many transitions incoming and one outgoing	*history* of the state has semantics which is even more general than UML. For instance it is possible to go back as many steps in history as needed. History may be reset to initial value by special kind of action.
[app. B.8] *deep history marking* (comments as above)	A *deep history state* is a state indicated with deep history sign. (comments as above)	*deep history indicator* see above for comments	deep *history* is available but has no special name (denoted by $\mathbf{H}^*$). Comments as above.
–	–	–	*Condition* entry to the state
–	–	–	*Selection* entry to the state
Transitions			
[def. 32] *multitarget transition*	*transition*	A *simple transition* or *fork transition.* Some visualSTATE transitions do not have related terms in UML (those activating targets in components orthogonal to transition source)	*transition* but more restricted than in visualSTATE. See UML column. ⟵
[def. 32] ⟶	A special case of *transition*	*simple transition*	⟵
[def. 32] ⟶	see below	*concurrent transition* is either a *join transition* or *fork transition* – see below	see below

This Paper | **visualstate** [11, 12] | **OMG UML Spec.**[19] | **D.Harel**[5]

Table 1: Comparison of statechart terminology and semantics.

This Paper	**visualstate** [11, 12]	**OMG UML Spec.**[19]	**D.Harel**[5]
⟶	No direct support for joins but they can be easily described as syntactic sugar	*synchronization transition* or *join transition* is a special case of *concurrent transition*	limited notational support for transitions with common target. The semantics is unclear though.
[def. 32]⟶	A special case of *transition*	*fork transition* is a special case of *concurrent transition*	*split transitions* or *splits*
–	the notion of *initial transition* exists but the transition has no labels	*initial transition* is the transition outgoing from *initial pseudostate*	the transition is present but has no special name
–	*termination transition* is an unlabeled transition firing when the activity enclosed in its source state reaches the *final state*	*completion transition* is an unlabeled transition firing when state machine enclosed in its source reaches final state	there seems to be no special construct and name for completion transition, but similar behavior may be achieved with conditions detecting end of activity and normal transitions
Events			
[p. 3, sec. 3.11] ⟶	*event*	*event*	Arbitrary condition may cause a transition to fire
[sec. 3.11] ⟶	*event* (also *external event*) is a simple tag with several parameters	Various kinds of *events* occurring when a condition is true or transition fires or state is entered etc. Still only one event can be processed at a time (as in visualSTATE)	⟵
[def. 30, app. B.5] ⟶	*signal* or *internal event* conveys a message from one part of the system to another (there are no objects)	*signal* conveys a message from one object to another object	*broadcast*
[app. B.6] *event disjunction*	*event group*	not precisely formulated but may be allowed depending on language used for expressing guards	arbitrary combinations of events are allowed in guard conditions

This Paper	**visualstate** [11, 12]	**OMG UML Spec.**[19]	**D.Harel**[5]

Table 1: Comparison of statechart terminology and semantics.

This Paper	**visualstate** [11, 12]	**OMG UML Spec.**[19]	**D.Harel**[5]
Actions			
[sec. 3.12] ⟶	*action*	*action* or *action expression*	*action*
[sec. 3.9] *output* or *action function*	*action function* (*output* in older releases of visualSTATE)	special case of *action* or *action expression*	special case of *action*
[app. B.10] ⟶	*do reactions*	*do activities*	*activities*
Steps			
[sec. 6.15] ⟶	*macrostep* executes until there are no more events in the *signal queue*	*run-to-completion* semantics is a different approach to event processing. It roughly corresponds to maintaining a stack of signals instead of FIFO queue	semantics is unclear; formalization has been attempted many times since then

| **This Paper** | **visualstate** [11, 12] | **OMG UML Spec.**[19] | **D.Harel**[5] |