CASL-CHART: Syntax and Semantics

CoFI Document: CASL/Extension	
Version: 0.2	2 February 2000

G. Reggio - L. Repetto *E-mail address for comments: reggio@disi.unige.it*

CoFI: The Common Framework Initiative
http://www.brics.dk/Projects/CoFI

This document is available on WWW[*] *and by FTP*[†]

Abstract

CASL the basic language developed within CoFI, the Common Framework Initiative for algebraic specification and development, cannot be easily used for specifying reactive systems. CASL-CHART is an extension to overcome this limit, which combines CASL with a variant of the statechart, similar to that supported by STATEMATE.

This document gives a detailed summary of the syntax and intended semantics of CASL-CHART, see [RR00] for an introduction to CASL-CHART on an example, and it is intended for readers that are already familiar with CASL ([The99]).

[*] ...
[†] .....

Chapter 3

CASL-CHART

3.1 Visual and Textual Syntax of CASL-CHART

CASL-CHART is a visual specification language, but it has also a corresponding textual presentation, that we need to simply define its static and dynamic semantics. In this section we present such syntax, in the same way of [The99], together with the corresponding visual presentation.

Specification

```
CC-SPEC    ::=    cc-spec  SPEC  EVENT-DECS  VAR-DECS  CHART
```

The visual presentation of a CASL-CHART consists of the visual presentations of its four components as detailed in the following.

Casl specification

SPEC ::= abstract syntax of a CASL specification, see [The99]

The visual (but in this case is just formatted text) presentation of a CASL specification is given by the display concrete syntax of [The99].

Event declarations

```
EVENT-DECS         ::=  event-decs  EVENT-DEC*
EVENT-DEC          ::=  event-dec  EV-NAME  SORT*  MODE
EV-NAME            ::=  SIMPLE-ID
MODE               ::=  input  |  local  |  output
SIMPLE-ID, SORT    ::=   see CASL syntax in [The99]
```

The visual presentation of an event declaration depends on its mode:

local `event-dec` $e\ s_1\ \ldots s_n$ `local` is represented by $e(s_1, \ldots, s_n)$; .

All local event declarations, preceded by the keyword **events**, will be enclosed in a note box ( ), together with the local variable declarations, that will be put below the chart icon.

input `event-dec` $e\ s_1\ \ldots s_n$ `input` is represented by $\xrightarrow{e(s_1,\ldots,s_n)}$.

This representation will be attached (by the arrowhead) to the chart icon.

We may put together the representations of several input event declarations by drawing only one arrow and putting all inscriptions above it.

output `event-dec` $e\ s_1\ \ldots s_n$ `output` is represented by $\xleftarrow{e(s_1,\ldots,s_n)}$.

This representation will be attached (by the tail) to the chart icon.

We may put together the representations of several output event declarations by drawing only one arrow and putting all inscriptions above it.

Variable declarations

```
VAR-DECS        ::=  var-decs  VAR-DEC*
VAR-DEC         ::=  var-dec  VAR  SORT  MODE
VAR             ::=  SIMPLE-ID
```

The visual presentation of a variable declaration depends on its mode:

local `var-dec` $x\ s$ `local` is represented by x: s; .

All local variable declarations, preceded by the keyword **vars**, will be enclosed in a note box ( ), together with the local event declarations, that will be put below the chart icon.

input `var-dec` $x\ s$ `input` is represented by $\xRightarrow{x : s}$.

This representation will be attached (by the arrowhead) to the chart icon.

We may put together the representations of several input variable declarations by drawing only one arrow and putting all inscriptions above it.

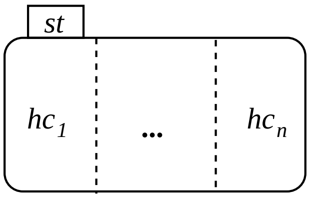
output `var-dec` $x\ s$ `output` is represented by $\xLeftarrow{x : s}$.

This representation will be attached (by the tail) to the chart icon.

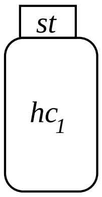
We may put together the representations of several output variable declarations by drawing only one arrow and putting all inscriptions above it.

Chart

```
CHART          ::=  chart  STATE  HIERARCHICAL-CHART+
STATE          ::=  SIMPLE-ID
```

If $n > 1$, then `chart` $st\ hc_1\ \ldots hc_n$ is represented by 

,

while `chart` $st\ hc_1$ is represented by 

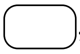
.

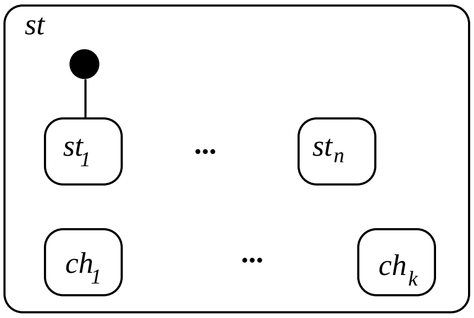
Hierarchical chart

```
HIERARCHICAL-CHART       ::=  hierarchical-chart  STATE  STATE+  STATE  TRANS  DECOMPS
DECOMPS                  ::=  decomps  CHART*
```

The visual presentation of a hierarchical chart

`hierarchical-chart` $st\ st_1\ \ldots st_n\ st_1\ trs$ `decomps` $ch_1 \ldots ch_k$

is the graph, enclosed by the icon , whose nodes are

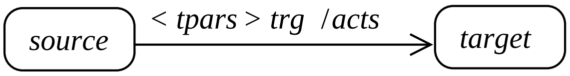


and whose arcs are determined the transitions *trs*, as explained below.

Transitions

```
TRANS          ::=  trans  TRAN*
TRAN           ::=  tran  STATE  TRAN-PARAMS  TRIGGER  ACTIONS  STATE
```

The visual presentation of a transition `tran` *source tpars trg acts target* is the following arc between two states of the owning hierarchical chart



If any of the three parts composing the inscription (transition parameters, trigger and actions) is null (precisely no parameters, constant true trigger, no actions), then such part with its separators is omitted in the drawing.

Transition parameters

```
TRAN-PARAMS        ::=  tran-params  TRAN-PARAM*  TRIGGER
TRAN-PARAM         ::=  tran-param  PARAM  SORT
PARAM              ::=  SIMPLE-ID
```

The visual presentation of a group of transition parameters

`tran-params` `tran-param` p_1 s_1 ...`tran-param` p_n s_n trg

is $p_1 \colon s_1 ; \ldots ; p_n \colon s_n \bullet trg$.

If the condition part (trg) is the constant true, then it is omitted together with the preceding $\bullet$.

Actions

```
ACTIONS         ::=  actions  ACTION*
```

The visual presentation of a group of actions `actions` $act_1 \ldots act_n$ is $act_1 ; \ldots ; act_n ;$.

Action

```
ACTION         ::=  assign  VAR  EXPR  |  gen-ev  EV-NAME  EXPR*
```

The visual presentation of an assignment `assign` x exp is $x := exp$; while that of an event generation statement `gen-ev` e $exp_1 \ldots exp_n$ is $e(exp_1, \ldots, exp_n)$.

Trigger

```
TRIGGER         ::=    true  |  happens  EV-EXPR  |  in-state  STATE  |
                       pred  PRED-NAME  EXPR*  |  equal   EXPR  EXPR  |
                       and  TRIGGER  TRIGGER  |  or  TRIGGER  TRIGGER  |
                       not  TRIGGER  |
                       quant-trigger  QUANT-OP  Q-VAR  SORT  TRIGGER  |
                       timed-trigger  TIMED-OP  EXPR  TRIGGER
Q-VAR           ::=  SIMPLE-ID
QUANT-OP        ::=  exists  |  texistsunique-unique  |  forall
TIMED-OP        ::=  timeout  |  since  |  before
PRED-NAME       ::=   see CASL syntax in [The99]
```

The visual presentation of the triggers is given following their inductive structure:

`true`	*true*
`happens` *e_exp*	*e_exp*
`in-state` *st*	*in st*
`pred` *pr* $exp_1 \ldots exp_n$	$pr(exp_1, \ldots, exp_n)$
`equal` exp_1 exp_2	$exp_1 = exp_2$
`and` trg_1 trg_2	$trg_1 \wedge trg_2$
`or` trg_1 trg_2	$trg_1 \vee trg_2$
`not` *trg*	$\neg$ *trg*
`quant-trigger exists` *q s trg*	$\exists$ *q*: *s* $\bullet$ *trg*
`quant-trigger texistsunique-unique` *q s trg*	$\exists\,!$ *q*: *s* $\bullet$ *trg*
`quant-trigger forall` *q s trg*	$\forall$ *q*: *s* $\bullet$ *trg*
`timed-trigger timeout` *exp trg*	*trg at exp*
`timed-trigger since` *exp trg*	*trg since exp*
`timed-trigger before` *exp trg*	*trg before exp*

Event expression

```
EV-EXPR          ::=  ev  EV-NAME  EXPR*  |  entered  STATE  |  exited  STATE
```

The visual presentation of the event expressions is given by cases:

`ev` *e* $exp_1 \ldots exp_n$	$e(exp_1,\ldots,exp_n)$
`entered` *st*	*entered st*
`exited` *st*	*exited st*

Expression

```
EXPR          ::=  VAR  |  PARAM  |  Q-VAR  |  appl  OP-NAME  EXPR*
OP-NAME       ::=  see CASL syntax in [The99]
```

The visual presentation of the expressions is given by induction on their structure as follows:

x (chart variable)	x
p (trnasition parameter)	p
q (quantified variable)	q
`appl` *op* $exp_1 \ldots exp_n$	$op(exp_1,\ldots,exp_n)$

Presentation options

If the Casl specification is structured, we may split its visual presentation on different sheets, following such structure. also intermixing it with the chart presentation.

Any element of a parallel decomposition of a chart (i.e., a hierarchical chart) may be represented on a separate sheet, in such case the name of its upper level state is written boldface in place of the hierarchical cart itself. Similarly, any chart component of a hierarchical chart may be represented on a separate sheet, in such case the name of its upper level state is written boldface in place of the cart itself.

Local variables and events used only in a subchart drawn on a separate sheet, may be reported in a note box only on such sheet.

The name of the main state of a chart is put inside the chart icon, if the chart is not decomposed in parallel, instead of in the external small rectangular box.

3.2 Static Semantics

In the following for each nonterminal symbol `NT` of the grammar of CASL-CHART (summarized in Appendix .2), we shall denote with `NT` also the language generated by such grammar assuming `NT` as start symbol.

We define the static semantics of CASL-CHART by means a deductive system with meta-rules of the form

$$\frac{\mathit{premises}}{\mathit{consequence}}\ \textbf{if}\ \mathit{condition}$$

where the premises and the consequence are made of judgments of the static correctness of a CASL-CHART specification or of one of its components.

Such judgments have the form $\vdash_{NT} n$, meaning that the element n of type `NT` is correct.

Usually the correctness of a construct of CASL-CHART depends on the context in which it appears, thus the judgments may have also the form $\rho \vdash_{NT} n$, meaning that the element n of type `NT` is correct in the context characterized by the information ρ.

A construct may contribute to the information on the context, thus we have also judgments of the form $\rho \vdash_{NT} n \mapsto \rho'$, meaning that the element n of type `NT` is correct in the context characterized by the information ρ, and that it changes such information to ρ'; together the modified context information it is possible to return also other additional informations.

The mathematical notations used in this section and in the following ones are reported in Appendix .1.

The information on the context of the CASL-CHART constructs are called *environment* and consist of

- the signature of the basic data types, specified by the CASL specification,
- the events used by the chart with their modes and the types of their arguments,
- the variables used by the chart with their modes and their types,
- the transition parameters with their types,
- the variables used by the logical quantifiers appearing in the triggers with their types,
- the states of the hierarchical chart currently considered.

We denote the set of all environments by *Env*, and we will use the following functions to access them. The detailed definition of *Env* is trivial, and it is omitted in this report.

- Pr_Fun : $\mathit{Env} \times$ `PRED-NAME` $\rightarrow$ `SORT*`

  returns the functionality of a predicate of the basic data types

- Op_Fun : $\mathit{Env} \times$ `OP-NAME` $\rightarrow$ (`SORT*` $\times$ `SORT`)

  returns the functionality of an operation of the basic data types

- Sorts : $Env \rightarrow \mathcal{P}_{fin}(\texttt{SORT})$

  returns the sorts of the signature of the basic data types

- Vars : $Env \rightarrow \mathcal{P}_{fin}(\texttt{VAR})$

  returns the names of the variables in the current scope (the chart variables, the transition parameters, and the variables used by the logical quantifiers in the triggers)

- Var_Type : $Env \times \texttt{VAR} \rightarrow \texttt{SORT}$

  returns the type of a variable

- Var_Mode : $Env \times \texttt{VAR} \rightarrow \texttt{MODE}$

  returns the mode of a variable

- Events : $Env \rightarrow \mathcal{P}_{fin}(\texttt{EV-NAME})$

  returns the names of the events of the chart

- Ev_Type : $Env \times \texttt{EV-NAME} \rightarrow \texttt{SORT*}$

  returns the types of the arguments of an event

- Ev_Mode : $Env \times \texttt{EV-NAME} \rightarrow \texttt{MODE}$

  returns the mode of an event

- States : $Env \rightarrow \mathcal{P}_{fin}(\texttt{STATE})$

  returns the set of the states of the current hierarchical chart

- Add_Var : $Env \times \texttt{VAR} \times \texttt{MODE} \times \texttt{SORT} \rightarrow Env$

  adds a variable with its mode and its type

- Add_QVar : $Env \times \texttt{Q-VAR} \times \texttt{SORT} \rightarrow Env$

  adds a logical quantifier variable with its type

- Add_Par : $Env \times \texttt{PARAM} \times \texttt{SORT} \rightarrow Env$

  adds a transition parameter with its type

- Add_Event : $Env \times \texttt{EV-NAME} \times \texttt{MODE} \times \texttt{SORT*} \rightarrow Env$

  adds an event with its mode and the types of its arguments

- Set_States : $Env \times \mathcal{P}_{fin}(\texttt{STATE}) \rightarrow Env$

  sets the set of the states of the current hierarchical chart

- Init : $Sig \rightarrow Env$

  returns the initial environment, where the signature of the basic data types is given

Specification

$$\vdash_{CC} \_ \subseteq \texttt{CC-SPEC}$$

$$\frac{\begin{array}{l}\vdash_{CASL} spec \mapsto \Sigma \\ \mathsf{Init}(\Sigma) \vdash_{EVS} evs \mapsto \rho \\ \rho \vdash_{VS} vars \mapsto \rho' \\ \rho' \vdash_{C} ch \mapsto ALLS, REFS\end{array}}{\vdash_{CC} \texttt{cc-spec}\ spec\ evs\ vars\ ch} \quad \textbf{if } REFS \subseteq ALLS$$

ALLS is the set of all the states of *ch*, and *REFS* is the set of the states referred in some trigger in *ch*; therefore, the side-condition asserts that all the referred states have to be states of the chart too.

Casl specification

$$\vdash_{CASL} \_ \mapsto \_ \subseteq \texttt{SPEC} \times Sig$$

where *Sig* denotes the class of (many-sorted, first-order) signatures (with predicates).

$\vdash_{CASL} spec \mapsto \Sigma$ means that

- *spec* is correct as CASL specification, **and** Σ is its signature (see [oS99]);
- $Sig(\textsc{Integer}) \subseteq \Sigma$ ($Sig(\textsc{Integer})$ is the signature of the standard specification of integers INTEGER, provided by the standard CASL libraries, see [RM99]);
- if two operations or two predicates in Σ have the same name, then their respective arities (i.e., sequences of argument sorts) are different[1];
- the CASL semantics of *spec* is a nonempty isomorphism class of structures (the models of *spec*) such that, if D is an element of this class, then $D_{|Sig(\textsc{Integer})}$ is (isomorphic to) the usual model of integers.

Event declarations

$$\_ \vdash_{EVS} \_ \mapsto \_ \subseteq Env \times \texttt{EVENT-DECS} \times Env$$

$$\frac{\rho_i \vdash_{EV} e_i \mapsto \rho_{i+1} \qquad i = 1, \dots, n}{\rho_1 \vdash_{EVS} \texttt{event-decs}\ e_1\ \dots\ e_n \mapsto \rho_{n+1}}$$

Event declaration

$$\_ \vdash_{EV} \_ \mapsto \_ \subseteq Env \times \texttt{EVENT-DEC} \times Env$$

[1] Otherwise, we have to extend our language with *qualified names* in expressions and triggers, as well as in CASL terms and formulæ.

$$\frac{}{\rho \vdash_{EVS} \texttt{event-dec}\ e\ s_1\ \ldots\ s_n\ m \mapsto \mathsf{Add\_Event}(\rho,e,m,s_1\ \ldots\ s_n)}$$

if $e \notin \mathsf{Events}(\rho)$ **and** $s_i \in \mathsf{Sorts}(\rho)$ for $i = 1, \ldots, n$

Variable declarations

$$\_ \vdash_{VS} \_ \mapsto \_ \subseteq Env \times \texttt{VAR-DECS} \times Env$$

$$\frac{\rho_i \vdash_V vdec_i \mapsto \rho_{i+1} \qquad i = 1, \ldots, n}{\rho_1 \vdash_{VS} \texttt{var-decs}\ vdec_1\ \ldots\ vdec_n \mapsto \rho_{n+1}}$$

Variable declaration

$$\_ \vdash_V \_ \mapsto \_ \subseteq Env \times \texttt{VAR-DEC} \times Env$$

$$\frac{}{\rho \vdash_V \texttt{var-dec}\ x\ s\ m \mapsto \mathsf{Add\_Var}(\rho,x,m,s)} \quad \textbf{if } x \notin \mathsf{Vars}(\rho) \textbf{ and } s \in \mathsf{Sorts}(\rho)$$

Chart

$$\_ \vdash_C \_ \mapsto \_ \subseteq Env \times \texttt{CHART} \times (\mathcal{P}_{fin}(\texttt{STATE}) \times \mathcal{P}_{fin}(\texttt{STATE}))$$

$\rho \vdash_C ch \mapsto ALLS, REFS$ means that ch is correct w.r.t. ρ, and is associated with $ALLS$ (the set of all its states), and $REFS$ (the set of the states referred in some of its triggers).

$$\frac{\rho \vdash_{HC} hc_i \mapsto ALLS_i, REFS_i \qquad i = 1, \ldots, n}{\rho \vdash_C \texttt{chart}\ st\ hc_1\ \ldots\ hc_n \mapsto \{st\} \cup_{i=1}^{n} ALLS_i, \cup_{i=1}^{n} REFS_i}$$

if $ALLS_1, \ldots, ALLS_n$ are pairwise disjoint **and** do not contain st

A correct chart consists of a nonempty list of hierarchical charts that do not share states.

Hierarchical chart

$$\_ \vdash_{HC} \_ \mapsto \_ \subseteq Env \times \texttt{HIERARCHICAL-CHART} \times (\mathcal{P}_{fin}(\texttt{STATE}) \times \mathcal{P}_{fin}(\texttt{STATE}))$$

$\rho \vdash_{HC} hc \mapsto ALLS, REFS$ means that hc is correct w.r.t. ρ, and is associated with $ALLS$ (the set of all its states) and $REFS$ (the set of all the states referred in some of its triggers).

$$\frac{\rho' \vdash_{TRS} trs \mapsto \mathit{REFS'} \qquad \rho' \vdash_{DCS} dcs \mapsto \mathit{INNS}, \mathit{REFS''}}{\rho \vdash_{HC} \texttt{hierarchical-chart}\ st\ stl\ start\ trs\ dcs \mapsto S', \mathit{REFS'} \cup \mathit{REFS''}}$$

$$\textbf{if} \begin{cases} S \text{ is the set of the states appearing in } stl, \\ S' = \{st\} \cup S \cup \mathit{INNS}, \\ start \in S, \\ st \notin S \\ \rho' = \mathsf{Set\_States}(\rho, S'), \\ (\{st\} \cup S) \ \cap \mathit{INNS} = \emptyset \end{cases}$$

The states of a hierarchical chart (S') are the upper level state (st), those on its first level not decomposed (S), and those on the first level decomposed together their inner states (INNS).

Decompositions

$$\_ \vdash_{DCS} \_ \mapsto \_ \subseteq \mathit{Env} \times \texttt{DECOMPS} \times (\mathcal{P}_{fin}(\texttt{STATE}) \times \mathcal{P}_{fin}(\texttt{STATE}))$$

$\rho \vdash_{DCS} dcs \mapsto \mathit{INNS}, \mathit{REFS}$ means that dcs is correct w.r.t. ρ, and is associated with INNS (the set of all its states), and REFS (the set of the states referred in the triggers of the transitions of its components).

$$\frac{\rho \vdash_C ch_i \mapsto \mathit{INNS}_i, \mathit{REFS}_i \qquad i = 1, \ldots, n}{\rho \vdash_{DCS} \texttt{decomps}\ ch_1\ \ldots\ ch_n \mapsto \cup_{i=1}^n \mathit{INNS}_i, \cup_{i=1}^n \mathit{REFS}_i}$$

if $\mathit{INNS}_1, \ldots, \mathit{INNS}_n$ are pairwise disjoint

Transitions

$$\_ \vdash_{TRS} \_ \mapsto \_ \subseteq \mathit{Env} \times \texttt{TRANS} \times \mathcal{P}_{fin}(\texttt{STATE})$$

$\rho \vdash_{TRS} trs \mapsto \mathit{REFS}$ means that trs is correct w.r.t. ρ, and is associated with REFS (the set of the states referred in its triggers).

$$\frac{\rho \vdash_{TR} tr_i \mapsto \mathit{REFS}_i \quad i = 1, \ldots, n}{\rho \vdash_{TRS} \texttt{trans}\ tr_1\ \ldots\ tr_n \mapsto \cup_{i=1}^n \mathit{REFS}_i}$$

Transition

$$\_ \vdash_{TR} \_ \mapsto \_ \subseteq \mathit{Env} \times \texttt{TRAN} \times \mathcal{P}_{fin}(\texttt{STATE})$$

$\rho \vdash_{TR} tr \mapsto \mathit{REFS}$ means that tr is correct w.r.t. ρ, and is associated with REFS (the set of the states referred in its trigger).

$$\frac{\begin{array}{l}\rho \vdash_{TPS} tpars \mapsto \rho' \\ \rho' \vdash_{TG} trg \mapsto REFS \\ \rho' \vdash_{AS} acts\end{array}}{\rho \vdash_{TR} \texttt{tran}\ source\ tpars\ trg\ acts\ target \mapsto REFS}\ \textbf{if}\ source,\ target \in \mathsf{States}(\rho)$$

Notice that both the trigger *trg* and the actions *acts* are judged w.r.t. the environment ρ' enriched by the declarations of the transition parameters *tpars*.

Transition parameters

$$\_ \vdash_{TPS} \_ \mapsto \_ \subseteq Env \times \texttt{TRAN-PARAMS} \times Env$$

$\rho \vdash_{TPS} tpars \mapsto \rho'$ means that *tpars* is correct w.r.t. ρ and updates the environment returning ρ' by adding the transition parameters.

$$\frac{\begin{array}{l}\rho_i \vdash_{TP} tp_i \mapsto \rho_{i+1} \quad i = 1, \ldots, n \\ \rho_{n+1} \vdash_{TG} tr \mapsto \emptyset\end{array}}{\rho_1 \vdash_{TPS} \texttt{tran-params}\ tp_1\ \ldots\ tp_n\ tr \mapsto \rho_{n+1}}$$

if neither `happens`, nor `in-state`, nor `timed-trigger` occur in *tr*

Transition parameter

$$\_ \vdash_{TP} \_ \mapsto \_ \subseteq Env \times \texttt{TRAN-PARAM} \times Env$$

$$\frac{}{\rho \vdash_{TP} \texttt{tran-param}\ p\ s \mapsto \mathsf{Add\_Par}(\rho,p,s)}\ p \notin \mathsf{Vars}(\rho)\ \ \textbf{and}\ s \in \mathsf{Sorts}(\rho)$$

Actions

$$\_ \vdash_{AS} \_ \subseteq Env \times \texttt{ACTIONS}$$

$$\frac{\rho \vdash_{A} act_i \quad i = 1, \ldots, n}{\rho \vdash_{AS} \texttt{actions}\ act_1\ \ldots\ act_n}$$

Action

$$\_ \vdash_{A} \_ \subseteq Env \times \texttt{ACTION}$$

$$\frac{\rho \vdash_{EX} exp \mapsto s}{\rho \vdash_{A} \texttt{assign}\ x\ exp}\ \textbf{if}\ \mathsf{Var\_Type}(\rho,x) = s\ \ \textbf{and}\ \mathsf{Var\_Mode}(\rho,x) \neq \texttt{input}$$

$$\frac{\rho \vdash_{EX} exp_i \mapsto s_i \quad i = 1, \ldots, n}{\rho \vdash_{A} \texttt{gen-ev}\ e\ exp_1\ \ldots\ exp_n}\ \textbf{if}\ \mathsf{Ev\_Type}(\rho,e) = s_1\ \ldots\ s_n\ \ \textbf{and}\ \mathsf{Ev\_Mode}(\rho,e) \neq \texttt{input}$$

Trigger

$$\_ \vdash_{TG} \_ \mapsto \_ \subseteq Env \times \texttt{TRIGGER} \times \mathcal{P}_{fin}(\texttt{STATE})$$

$\rho \vdash_{TG} trg \mapsto REFS$ means that trg is correct w.r.t. ρ, and is associated with $REFS$ (the set of the states referred in itself).

$$\frac{}{\rho \vdash_{TG} \texttt{true} \mapsto \emptyset} \qquad \frac{\rho \vdash_{EE} e\_exp \mapsto REFS}{\rho \vdash_{TG} \texttt{happens}\ e\_exp \mapsto REFS}$$

$$\frac{}{\rho \vdash_{TG} \texttt{in-state}\ s \mapsto \{s\}} \qquad \frac{\rho \vdash_{EX} exp_i \mapsto s_i \quad i = 1, \ldots, n}{\rho \vdash_{TG} \texttt{pred}\ pr\ exp_1\ \ldots\ exp_n \mapsto \emptyset}\ \textbf{if}\ \mathsf{Pr\_Fun}(\rho, pr) = s_1\ \ldots\ s_n$$

$$\frac{\begin{array}{l}\rho \vdash_{EX} exp_1 \mapsto s\\ \rho \vdash_{EX} exp_2 \mapsto s\end{array}}{\rho \vdash_{TG} \texttt{equal}\ exp_1\ exp_2 \mapsto \emptyset} \qquad \frac{\begin{array}{l}\rho \vdash_{TG} trg_1 \mapsto REFS_1\\ \rho \vdash_{TG} trg_2 \mapsto REFS_2\end{array}}{\rho \vdash_{TG} \texttt{and}\ trg_1\ trg_2 \mapsto REFS_1 \cup REFS_2}$$

$$\frac{\begin{array}{l}\rho \vdash_{TG} trg_1 \mapsto REFS_1\\ \rho \vdash_{TG} trg_2 \mapsto REFS_2\end{array}}{\rho \vdash_{TG} \texttt{or}\ trg_1\ trg_2 \mapsto REFS_1 \cup REFS_2} \qquad \frac{\rho \vdash_{TG} trg \mapsto REFS}{\rho \vdash_{TG} \texttt{not}\ trg \mapsto REFS}$$

$$\frac{\mathsf{Add\_QVar}(\rho, q, s) \vdash_{TG} trg \mapsto REFS}{\rho \vdash_{TG} \texttt{quant-trigger}\ qop\ q\ s\ trg \mapsto REFS}\ \textbf{if}\ s \in \mathsf{Sorts}(\rho)$$

$$\frac{\begin{array}{l}\rho \vdash_{EX} ie \mapsto Integer\\ \rho \vdash_{TG} trg \mapsto REFS\end{array}}{\rho \vdash_{TG} \texttt{timed-trigger}\ top\ ie\ trg \mapsto REFS}\ \textbf{if}\ \texttt{timed-trigger}\ \text{does not occur in}\ trg$$

Event expression

$$\_ \vdash_{EE} \_ \mapsto \_ \subseteq Env \times \texttt{EV-EXPR} \times \mathcal{P}_{fin}(\texttt{STATE})$$

$\rho \vdash_{EE} e\_exp \mapsto REFS$ means that e_exp is correct w.r.t. ρ, and is associated with $REFS$ (the set of the states, at most one, referred in itself).

$$\frac{\rho \vdash_{EX} exp_i \mapsto s_i \quad i = 1, \ldots, n}{\rho \vdash_{EE} \text{ev}\ e\ exp_1\ \ldots\ exp_n \mapsto \emptyset}\ \textbf{if}\ \mathsf{Ev\_Type}(\rho, e) = s_1\ \ldots\ s_n,\ \textbf{and}\ \mathsf{Ev\_Mode}(\text{e}) \neq \text{output}$$

$$\frac{}{\rho \vdash_{EE} \texttt{entered}\ s \mapsto \{s\}} \qquad \frac{}{\rho \vdash_{EE} \texttt{exited}\ s \mapsto \{s\}}$$

Expression

$$\_ \vdash_{EX} \_ \mapsto \_ \subseteq Env \times \texttt{EXPR} \times \texttt{SORT}$$

$\rho \vdash_{EX} exp \mapsto s$ means that exp is correct w.r.t. ρ, and has type s.

$$\frac{}{\rho \vdash_{EX} x \mapsto \mathsf{Var\_Type}(x)} \ \textbf{if } x \in \mathsf{Vars}(\rho)$$

$$\frac{}{\rho \vdash_{EX} p \mapsto \mathsf{Var\_Type}(p)} \ \textbf{if } p \in \mathsf{Vars}(\rho) \qquad \frac{}{\rho \vdash_{EX} q \mapsto \mathsf{Var\_Type}(q)} \ \textbf{if } q \in \mathsf{Vars}(\rho)$$

$$\frac{\rho \vdash_{EX} exp_i \mapsto s_i \quad i = 1, \ldots, n}{\rho \vdash_{EX} \texttt{appl}\ op\ exp_1\ \ldots\ exp_n \mapsto s} \ \textbf{if } \mathsf{Op\_Fun}(\rho, op) = (s_1\ \ldots\ s_n, s)$$

3.3 Synchronous Time Semantic (Single Step)

In this section, and in the next one too, we consider only those CASL-CHART specifications that are statically correct, according to Sect. 3.2; all semantics functions are partial and by default are undefined on all the uncorrect specifications.

We give a dynamic semantics to a correct CASL-CHART specification in two steps:

- first, we give a semantics to its CASL specification component, and use the resulting model (an algebra) to define the semantic domains;
- then, we shall give the semantics for its chart component using such domains.

Casl specification

- CASL: $\mathtt{SPEC} \to_{\mathrm{p}} Alg$

where Alg is the class of the many-sorted first-order structures (or algebras).

$\mathsf{CASL}(spec)$ is defined iff *spec* is statically correct, and in such cases

$$\mathsf{CASL}(spec) = D$$

where the Σ-algebra D is an element of the CASL semantics of *spec*.

This is a good definition, because the static correctness of *spec* ensures that the CASL semantics of *spec* is a class of isomorphic algebras. Moreover we have that $D_{|Sig(\mathrm{INTEGER})}$ is isomorphic to the usual algebra of the integers.

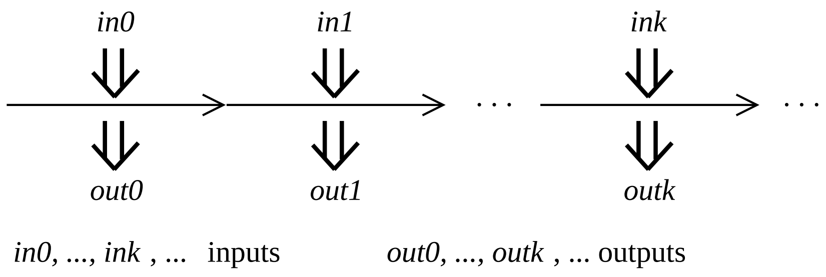
CASL-CHART Specification

Assume to consider a statically correct CASL-CHART specification,

$$cc\text{-}spec = \mathtt{cc\text{-}spec}\ spec\ evs\ vars\ ch,$$

s.t. $\mathsf{CASL}(spec) = D$, where D is a Σ-algebra.

The semantics of *cc-spec* should represent all possible executions of the system modelled by such specification. We represent a system execution by saying at each time unit (step) what is the input and what is the output of the system: in this case the input is given by the values updating the input variables and by the happened input events, while the output is given by the values updating the output variables and by the happened output events. Thus, a generic execution may be graphically presented as below, where a simple arrow ($\to$) represents the system life during a time unit (a step).



A time-unit corresponds to a single step in the "synchronous time model", whereas it corresponds to a macro-step in the "asynchronous time model" [HN96], see Sect. 3.4.

The synchronous time model assumes that the system executes one (single) step per time-unit, reacting to the external changes represented by the inputs. The internal changes (i.e., updating of variables, and generating events), performed by the system during a step execution, and the input received during such step, will be perceived only in the next step, and so on. Hence, assuming that the computation of the involved conditions and the execution of the involved actions terminate, a step execution always terminates.

Formally, we define the system executions as a partial function from streams[2] of inputs into sets of streams of outputs. We cannot simply use functions from input streams to output streams, because the behaviour of the modelled system may be nondeterministic, i.e., many outputs may be the result of the same input. The inputs and the outputs are pairs, whose components are the updates of the the input (output) variables and the received input (generated output) events.

The index D below denotes the Σ-algebra modelling the data used by the system.

- CC: $\texttt{CC-SPEC} \rightarrow_{\mathrm{p}} \bigcup_{D \in Alg} Executions_D$

$\mathsf{CC}(\texttt{cc-spec}\ spec\ evs\ vars\ ch)=$

if $\texttt{cc-spec}\ spec\ evs\ vars\ ch$ is correct, **then** $\mathsf{CC'}_{\mathsf{SP}(spec)}(evs,\ vars,\ ch)$ **else** undefined.

- $Executions_D = [Input_D^{\infty} \rightarrow_{\mathrm{p}} \mathcal{P}(Output_D^{\infty})]$
- $Input_D = Output_D = Updates_D \times Events_D$
- $Updates_D = \mathcal{P}_{fin}(\texttt{VAR} \times Value_D)$
- $Value_D = \bigcup_{s \in Sorts(\Sigma)} D_s$ (therefore, $\mathbf{Z} \approx D_{Integer} \subseteq Value$)
- $Events_D = \mathcal{P}_{fin}((\texttt{EV-NAME} \times Value_D^{*}) \cup \{(\texttt{entered}, st), (\texttt{exited}, st) \mid st \in \texttt{STATE}\})$
- $\mathsf{CC'}_D$: $\texttt{EVENT-DECS} \times \texttt{VAR-DECS} \times \texttt{CHART} \rightarrow Executions_D$

To define $\mathsf{CC'}$ we have to consider that, in general, the effects produced by an acceptable input stream over the system modelled by a CASL-CHART specification in a particular execution, starting from a given initial configuration, can be suitably represented by a

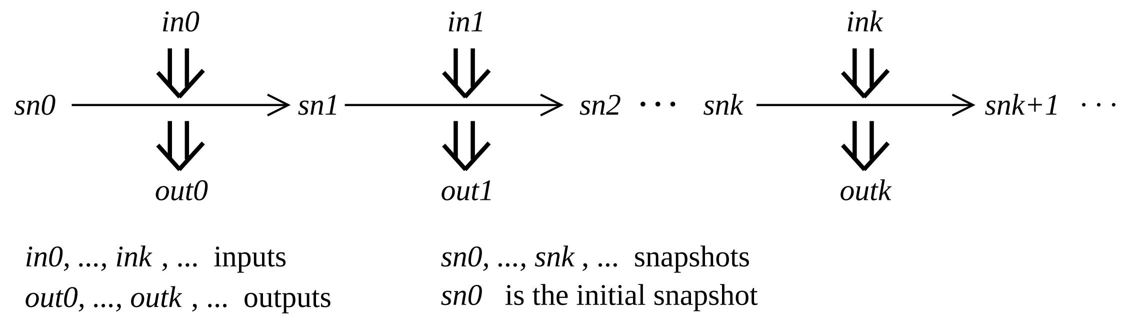
[2] A stream is an infinite sequence.

stream of "snapshots". Intuitively, a snapshot represents the situation of the system *at the end* of a step; it contains:

- the set (finite) of all the current active states (i.e., the states of the chart in which the system happens to be) at the end of that step;
- the current store, that associates with variables their current values of appropriate sorts (the values of the input variables are those given by the received input);
- the set (finite) of the events that have been generated in this step, including those events that signal an exiting from or entering into a state, and the input events (received from outside); such events will be sensed in the following step.

The initial snapshot is characterized by the start states of the whole chart, the empty store, and the events that represent the entering into the start states.

Graphically:



Formally

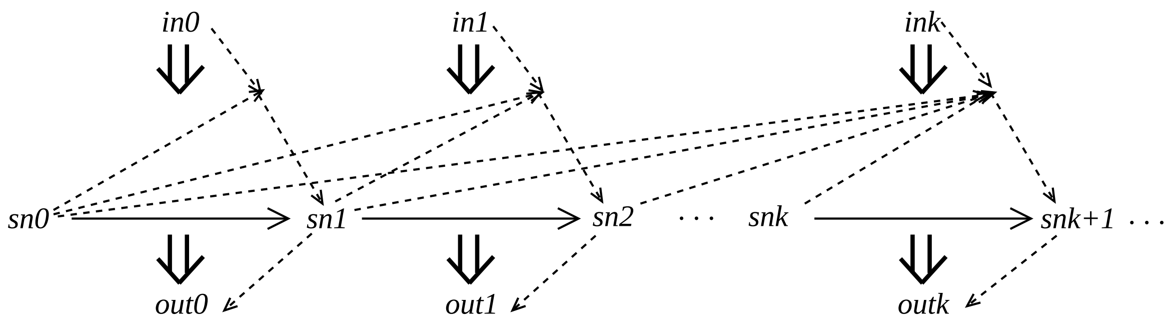
- $Snapshot_D = States \times Store_D \times Events_D$
- $States = \mathcal{P}_{fin}(\mathtt{STATE})$
- $Store_D = [\mathtt{VAR} \to_{\mathrm{p}} Value_D]$

From now on we will drop the index D referring to the data part from the various domains and functions.

We define CC' by means of an auxiliary function, **step**, describing the change of the configuration (snapshot) of the system modelled by a chart due to a given input. Because the trigger of a transition may concern the past history of the chart (i.e., the sequence of the past snapshots), **step** will take a history (a finite sequence of snapshots) as an argument instead of just a snapshot.

The output produced at each step is be determined by a function **output** by looking at the snapshot at the end of the step.

Graphically:



sn0 = init_snapshot()show grestore /Times-Italic findf*out0* = output()show grestore /Times-Italic fi
sn1 = step()show grestore /Times-Italic findfont 14 sc*out1* = output()show grestore /Times-Italic fi
sn2 = step()show grestore /Times-Italic findfont 14 sca...lefont setfont gsave 69 259 translate 1 -1 sca
... *outk* = output()show grestore /Times-Italic fi
snk+1 = step()show grestore /Times-Italic findfont 14 ...scalefont setfont gsave 84 289 translate 1 -1
...

CC'$(evs,\ vars,\ ch) = exec$, where

$exec(is)$ is defined iff is is an admissible input stream for the chart, i.e., the updates in is provide values only to input variables and the events contained in is are input only; formally, for all $j \geq 0$, if $is{\downarrow}_j = (ups,\ ES)$:

- if $(x,\ val) \in ups$, then `var-dec` x s `input` appears in $vars$ and $val \in D_s$
- if $(e,\ val_1\ \ldots\ val_n) \in ES$, then `event-dec` e s_1 $\ldots$ s_n `input` appears in evs, and $val_i \in D_{s_i}$ for $i = 1, \ldots, n$.

Whenever $exec(is)$ is defined, we have that:

$exec(is) =$
$\{$ outputs$(ss,\ evs,\ vars) \mid \exists ss \in Snapshot^\infty$ **s.t.** $ss{\downarrow}_0 =$ init_snapshot(ch) **and**
for all $j \geq 0$: $ss{\downarrow}_{j+1} \in$ step$(ch, ss\rfloor_j, is{\downarrow}_j)$ $\}$

Output projection

- outputs: $Snapshot^\infty \times$ `EVENT-DECS` $\times$ `VAR-DECS` $\rightarrow$ $Output^\infty$

outputs$(sn_0\ \&\ sn_1\ \&\ sn_2\ \&\ \ldots,\ evs,\ vars) =$

output$(sn_1,\ evs,\ vars)$ & output$(sn_2,\ evs,\ vars)$ & $\ldots$

- output: $Snapshot \times$ `EVENT-DECS` $\times$ `VAR-DECS` $\rightarrow$ $Output$

output$(\langle AS,\ \sigma,\ ES\rangle,\ evs,\ vars) = \langle\ \{\ (x, \sigma(x)) \mid x \in OUTVS\},\ ES \cap OUTES\ \rangle$

$OUTVS$ is the set of the output variables, i.e.,

$\{x \mid$ `var-dec` x s `output` appears in $vars\}$,

and $OUTES$ is the set of the output events, i.e.,

$\{e|$ `event-dec` $e\ s_1\ \ldots\ s_n$ `output` appears in $evs\}$.

Initial snapshot

- init_snapshot: `CHART` $\rightarrow$ *Snapshot*

init_snapshot$(ch) = \langle AS,\ Empty_{\mathrm{F}},\ \{($`entered`$,\ st) \mid st \in AS\}\rangle$
where $AS =$ start_states(ch).

- start_states: `CHART` $\cup$ `HIERARCHICAL-CHART` $\rightarrow$ *States*

start_states(ch) returns the states that are entered whenever ch is activated.

- start_states(`chart` $st\ hc_1\ \ldots\ hc_n) = \{st\ \} \cup_{i=1}^{n}$ start_states(hc_i)
- start_states(`hierarchical-chart` $st\ stl\ start\ trs\ dcs) = start$

A step

- step: `CHART` $\times$ *History* $\times$ *Input* $\rightarrow \mathcal{P}(Snapshot)$
- *History* $= Snapshot^{+}$ (histories, i.e., finite sequences of snapshots)

This is the basic function that "executes" a (single) step. Given a chart and a history till the beginning of a certain step, an input, then a possible result of a step consists of a new configuration of the system, i.e., a snapshot.

Notice that we might have a numerable infinity [3] of different results, due to the transition parameters; and that there is at least one result, due to the possibility of the "idle move", whenever no other move is allowable. By *move*, we mean the execution of a transition.

In general, nondeterminism occurs, due to both the choice from the allowable sets of transitions and the effect of their respective actions; actually, the same set of assignment actions might lead to different stores, it being understood that all the expressions on the right side evaluate according to the same store – the one at the beginning of the current step – and the same transition parameter evaluation, because *write-write racing conditions* [HN96] might occur on some local or output variables. As well as in STATEMATE, during a step execution, a maximal subset of the enabled transitions [4] is taken, in which no pair of transitions is in conflict and no transition having higher priority is left out. For our purposes, we point out that:

- a transition is enabled if the system is in its source state (i.e., its source state belongs to the set of the current active states) and its trigger holds (i.e., its trigger evaluates to *true*, w.r.t. a possible transition parameter evaluation and the current history);

[3] It is numerable, if we assume D term generated.

[4] Obviously, the transitions we consider are enabled w.r.t. the snapshot at the beginning of the step, considering the new input.

– two transitions are in conflict if there exists a same state from which the system exits if either is taken;

– a transition has higher priority than another, if the former's source state "contains" the latter's one as inner state.

$\mathsf{step}(ch,\ h\ \&\ \langle AS,\ \sigma,\ ES\rangle,\ (ups_I,\ ES_I)) =$
$\{\ \ (AS \setminus EXIT)\ \cup ENTER,\ \sigma',\ INT \cup ES_I)\ |$
$\quad (EXIT,\ ENTER,\ ups,\ INT) \in \mathsf{C}(ch,\ h\ \&\ \langle AS,\ \sigma,\ ES\rangle)\ \ \textbf{and}\ \sigma' \in \mathsf{apply}(ups \cup ups_I,\ \sigma)$
$\}$

Apply updates

- apply: $\mathit{Updates} \times \mathit{Store}\ \rightarrow\ \mathcal{P}_{\mathit{fin}}(\mathit{Store})$

$\mathsf{apply}(ups,\ \sigma)$ returns the set of all the stores that might come out owing to the application of all the updates in ups to σ in all the possible orders. Recall that an update is a pair consisting of a variable and of an assigned value.

$\mathsf{apply}(ups,\ \sigma) =$
$\quad \textbf{if}\ ups = \emptyset\ \ \textbf{then}\ \{\sigma\}$
$\quad \textbf{else}\ \{\sigma'[x \mapsto val]\ |\ (x,\ val) \in ups\ \ \textbf{and}\ \sigma' \in \mathsf{apply}(ups \setminus \{(x,\ val)\},\ \sigma)\ \}$

Single-step basic function

- C: $\texttt{CHART} \times \mathit{History}\ \rightarrow\ \mathcal{P}(\mathit{Local\_Result})$
- $\mathit{Local\_Result} = \mathit{States}\ \times \mathit{States}\ \times \mathit{Updates} \times \mathit{Events}$

C is the basic function defining the semantics of a (single) step. Rather than a set of snapshots, as **step** does, C returns a more disaggregate information, named *local results*, precisely a set of quadruples:

– the exited states;

– the entered states;

– the variable updates performed by the taken transitions, it is necessary to keep this information, instead of the new stores, because of possible write-write racing conditions;

– the events whose generation is associated with the taken transitions, including those that signal an exiting from or entering into a state (note that this set does not include input events, because them cannot be generated by the chart).

$\mathsf{idle} = (\emptyset,\ \emptyset,\ \emptyset,\ \emptyset) \in \mathit{Local\_Result}$ is the result of the "idle move" (i.e., when the input values and sensed events produced no effect), so no change in the active states, no produced updates, and no generated events.

Obviously, several (even infinite) quadruples might be produced. This may happen only if different sets of transitions can be taken in the current snapshot, considering the given history.

$\mathsf{C}(\texttt{chart}\ st\ hc_1\ \ldots\ hc_n,\ h) =$
$\{\ (\cup_{i=1}^{n} EXIT_i,\ \cup_{i=1}^{n} ENTER_i,\ \cup_{i=1}^{n} ups_i,\ \cup_{i=1}^{n} EVS_i)\ |$
for $i = 1, \ldots, n\ (EXIT_i,\ ENTER_i,\ ups_i,\ EVS_i) \in \mathsf{HC}(hc_i,\ h)\ \}$

Hierarchical Cahrt

- HC: `HIERARCHICAL-CHART` $\times$ *History* $\rightarrow \mathcal{P}(Local\_Result)$

$\mathsf{HC}(\texttt{hierarchical-chart}\ st\ stl\ start\ trs\ dcs,\ h) =$
let $AS = \pi_1(hd(h))$;
hd(h) is the snapshot at the beginning of the current step; its first component, AS, is the set of the active states in the whole chart, not only in this hierarchical chart
$S = \mathsf{active\_states}(stl,\ dcs,\ AS)$;
S is the set of active states in this hierarchical chart, at most it contains one element
in
if $S = \emptyset$ **then**
$\{\ \mathsf{idle}\ \}$
else
let $\{st'\} = S$, *recall that S has at most one element*
$EFFS = \mathsf{TRS}(trs,\ st',\ h)$
EFFS is the set of the effects due to the execution of any possible habilitated transition in trs starting from st′
in if $EFFS = \emptyset$ **then** *i.e., if the first level of this hierarchical chart cannot move*
if $\mathsf{is\_decomposed}(st',\ dcs) = true$ **then**
$\mathsf{C}(\mathsf{decomp}(st',\ dcs),\ h)$
whenever possible, the inner chart, in which st′ is decomposed, will move
else $\{\ \mathsf{idle}\}$
this hierarchical chart does not move, so the only result is the "idle move"
else *this is the case when* $EFFS \neq \emptyset$
let $EXIT = AS \cap \mathsf{substates}(st',\ dcs)$; *states exited by any move in EFFS*
in
$\{\ (\ EXIT,\ ENTER,\ ups,$
$GEN \cup \{(\texttt{exited},\ st'')\ |\ st'' \in EXIT\} \cup \{(\texttt{entered},\ st'')\ |\ st'' \in ENTER\}\ )\ |$
$(target,\ ups,\ GEN) \in EFFS$ **and**
$ENTER = \{target\} \cup \mathsf{start\_states}(\mathsf{decomp}(target,\ dcs)$
$\}$

- is_decomposed: `STATE` $\times$ `DECOMPS` $\rightarrow \{true, false\}$

  $\mathsf{is\_decomposed}(st,\ \texttt{decomps}\ ch_1\ \ldots\ ch_n) =$

  $st = \mathsf{upper\_state}(ch_1)$ **or** $\ldots$ **or** $st = \mathsf{upper\_state}(ch_n)$

- upper_state: CHART $\rightarrow$ STATE+

  upper_state(chart $st\ hc_1\ \ldots\ hc_n$) $= st$

- decomp: STATE $\times$ DECOMPS $\rightarrow_{\mathrm{p}}$ CHART

  - decomp(st, decomps Λ) is undefined
  - decomp(st, decomps $ch_1\ \ldots\ ch_n$) $=$
    if $st =$ upper_state(ch_1) **then** ch_1 **else** decomp(st, decomps $ch_2\ \ldots\ ch_n$)

  Note that is_decomposed(st, dcs) $= true$ iff decomp(st, dcs) is defined.

- active_states: STATE+ $\times$ $DECOMPS$ $\times$ $States$ $\rightarrow$ $States$

  active_states(stl, $decs$, AS) returns the active states of the hierarchical chart made of stl and $decs$ w.r.t. AS; it returns either a singleton or the empty set.

  active_states($st_1\ \ldots\ st_n$, decomps $ch_1\ \ldots\ ch_k$, AS) $=$

  $\{st_1, \ldots, st_n,$ upper_state(ch_1), $\ldots$, upper_state(ch_k) $\}$ $\cap AS$

- substates: STATE $\times$ DECOMPS $\rightarrow$ $States$

  substates(st, dcs) is the set of the states, whose elements are st, and all the states of the possible chart, among those listed in dcs, that forms the decomposition of st.

  - substates(st, dcs) $=$ all_states(decomp(st, dcs))

- all_states: CHART $\cup$ CHART $\rightarrow$ $States$

  - all_states(chart $st\ hc_1\ \ldots\ hc_n$) $= \{st\} \cup_{i=1}^{n}$ all_states(hc_i)
  - all_states(hierarchical-chart $st\ st_1\ \ldots st_n\ start\ trs$ decomps $ch_1\ \ldots\ ch_k$) $=$
    $\{st, st_1, \ldots, st_n\} \cup_{i=1}^{k}$ all_states(ch_i)

Transitions

- TRS: TRANS $\times$ STATE $\times$ $History$ $\rightarrow$ $Effects$
- $Effects = \mathcal{P}$(STATE $\times$ $Updates \times Events$)

TRS(trs, st, h) returns the set of all the possible "effects" associated with those transitions listed in trs that are enabled w.r.t. the active state st and the history h, i.e., whose source state is st and whose trigger evaluates to $true$ w.r.t. a permitted transition parameter evaluation and the history h. Note that we might have a (numerable) infinity of different effects, due to transition parameters, or even no effect, whenever no transition is enabled.

Each effect, caused by taking an enabled transition, is a triple that consists of:

- a target state,
- a finite set of updates, due to the assignment actions of the transitions,

– a finite set of local or output events, due to the event generation actions of the transitions.

$\mathsf{TRS}(\texttt{trans}\ tr_1\ \ldots\ tr_n,\ st,\ h) = \mathsf{TR}(tr_1,\ st,\ h)\ \cup \ldots \cup \mathsf{TR}(tr_n,\ st,\ h)$

Transition

- TR: $\texttt{TRAN} \times \texttt{STATE} \times \mathit{History} \ \to\ \mathit{Effects}$

TR defines the possible effects of a transition w.r.t. a current active state and a history.

First of all, we check whether the current state is the source state of the transition. If the answer is no, we get the empty set of effects. Otherwise, the possible transition parameters are processed, taking into account any permitted instantiation of them that satisfies the given condition, considering the current store too. This might lead to a (numerable) infinity of effects, as a final result.

Then, the transition trigger is evaluated w.r.t. each permitted instantiation of the transition parameters, and the same history; whenever it evaluates to *true* – i.e., the transition is enabled – a further part of the effect is given by the evaluation of the actions (assignments and event generations, separately).

$\mathsf{TR}(\texttt{tran}\ \mathit{source}\ \mathit{tpars}\ \mathit{trg}\ \mathit{acts}\ \mathit{target},\ st,\ h) =$
if $\mathit{source} = st$ **then**
  let $\mathcal{I} = \mathsf{TPS}(\mathit{tpars},\ h)$
  in $\bigcup_{\pi \in \mathcal{I}} \mathsf{TR'}(\mathit{trg},\ \mathit{acts},\ \mathit{target},\ h,\ \pi)$
else $\emptyset$

- $\mathsf{TR'}$: $\texttt{TRIGGER} \times \texttt{ACTIONS} \times \texttt{STATE} \times \mathit{History} \times \mathit{PEval} \ \to\ \mathit{Effects}$
- $\mathit{PEval} = [\texttt{PARAM} \to_{\mathrm{p}} \mathit{Value}]$ (transition parameter instantiations)

If the transition trigger evaluates to *true* w.r.t. the given history and transition parameter evaluation, then a single effect can be calculated.

$\mathsf{TR'}(\mathit{trg},\ \mathit{acts},\ \mathit{target},\ h,\ \pi) =$
if $\mathsf{TG}(\mathit{trg},\ h,\ \pi,\ \mathit{Empty}_{\mathrm{F}}) = \mathit{true}$ **then**
  let $\sigma = \pi_2(hd(h))$ *the current store*
    $(\mathit{ups},\ \mathit{ES}) = \mathsf{AS}(\mathit{acts},\ \sigma,\ \pi)$
  in $\{\ (\mathit{target},\ \mathit{ups},\ \mathit{ES})\ \}$
else $\emptyset$

Transition Parameters

- TPS: $\texttt{TRAN-PARAMS} \times \mathit{History} \ \to\ \mathcal{P}(\mathit{PEval})$

define the possible instantiations of the transition parameters.

$\mathsf{TPS}(\mathtt{tran\text{-}params}\ tpl\ trg,\ h) =$

$\{\pi \in PEval \mid \mathsf{ok}(tpl,\ \pi) = true\ \ \mathbf{and}\ \mathsf{TG}(trg,\ h,\ \pi,\ Empty_{\mathrm{F}}) = true\ \}$

- ok: $\mathtt{TRAN\text{-}PARAM*} \times PEval\ \rightarrow\ \{true, false\}$

$\mathsf{ok}(\Lambda,\ Empty_{\mathrm{F}}) = true$

$\mathsf{ok}(\mathtt{tran\text{-}param}\ p\ s\ tpl,\ \pi) = \mathsf{ok}(tpl, \pi_{|\mathrm{Dom}(\pi)\setminus\{p\}})\ \ \mathbf{and}\ \pi(p) \in D_s$

Actions

- AS: $\mathtt{ACTIONS} \times Store \times PEval \rightarrow_{\mathrm{p}} (Updates \times Events)$

$\mathsf{AS}(\mathtt{actions}\ act_1\ \ldots\ act_n,\ \sigma,\ \pi) =$

$\mathbf{let}\ (ups_1,\ EVS_1) = \mathsf{A}(act_1,\ \sigma,\ \pi),\ \ldots,\ (ups_n,\ EVS_n) = \mathsf{A}(act_n,\ \sigma,\ \pi)$

$\mathbf{in}\ (\cup_{i=1}^{n} ups_i,\ \cup_{i=1}^{n} EVS_i)$

Action

- A: $\mathtt{ACTION} \times Store \times PEval \rightarrow_{\mathrm{p}} Updates \times Events$

$\mathsf{A}(\mathtt{assign}\ x\ exp,\ \sigma,\ \pi) = (\{\ (x,\ \mathsf{EX}(exp,\ \sigma,\ \pi,\ Empty_{\mathrm{F}}))\}, \emptyset)$

$\mathsf{A}(\mathtt{gen\text{-}ev}\ e\ exp_1\ \ldots\ exp_n,\ \sigma,\ \pi) =$

$\quad (\emptyset,\ \{(e,\ \mathsf{EX}(exp_1,\ \sigma,\ \pi,\ Empty_{\mathrm{F}})\ \ldots\ \mathsf{EX}(exp_n,\ \sigma,\ \pi,\ Empty_{\mathrm{F}}))\ \})$

Trigger

- TG: $\mathtt{TRIGGER} \times History \times PEval \times QEval\ \rightarrow\ \{true, false\}$
- $QEval = [\mathtt{Q\text{-}VAR} \rightarrow_{\mathrm{p}}\ Value]$ (logical variables instantiations)

To evaluate a trigger we need, besides the transition parameters evaluation, the current history as well (see timed triggers).

- $\mathsf{TG}(\mathtt{true},\ h,\ \pi, \varphi) = true$
- $\mathsf{TG}(\mathtt{happens}\ e\_exp,\ h\ \&\ \langle AS,\ \sigma,\ ES\rangle,\ \pi,\ \varphi) =$
  $\mathbf{if}\ \mathsf{EE}(e\_exp,\ \sigma,\ \pi, \varphi) \in ES\ \mathbf{then}\ true\ \mathbf{else}\ false$

  Intuitively, `happens` *e_exp* means that the event coming out from the evaluation of *e_exp* is happening (it is perceived just now), at the beginning of the current step.
- $\mathsf{TG}(\mathtt{in\text{-}state}\ st,\ h\ \&\ \langle AS,\ \sigma,\ ES\rangle,\ \pi,\ \varphi) = \mathbf{if}\ st \in AS\ \mathbf{then}\ true\ \mathbf{else}\ false$

  `in-state` *st* means that the system is in the state *st* at the beginning of the current step.

- $\mathsf{TG}(\texttt{pred}\ pr\ exp_1\ \dots\ exp_n,\ h\ \&\ \langle AS,\ \sigma,\ ES\rangle,\ \pi,\ \varphi) =$
  if $(\mathsf{EX}(exp_1,\ \sigma,\ \pi,\varphi),\ \dots,\ \mathsf{EX}(exp_n,\ \sigma,\ \pi,\varphi)) \in pr^D$ **then** *true* **else** *false*

- $\mathsf{TG}(\texttt{equal}\ exp_1\ exp_2,\ h\ \&\ \langle AS,\ \sigma,\ ES\rangle,\ \pi,\ \varphi) =$

  if $\mathsf{EX}(exp_1,\ \sigma,\ \pi,\varphi) = \mathsf{EX}(exp_2,\ \sigma,\ \pi,\varphi)$ **then** *true* **else** *false*

- $\mathsf{TG}(\texttt{and}\ trg_1\ trg_2,\ h,\ \pi,\varphi) =$
  if $\mathsf{TG}(trg_1,\ h,\ \pi,\varphi) = true$ **and** $\mathsf{TG}(trg_2,\ h,\ \pi,\varphi) = true$ **then** *true* **else** *false*

- $\mathsf{TG}(\texttt{or}\ trg_1\ trg_2,\ h,\ \pi,\varphi) =$
  if $\mathsf{TG}(trg_1,\ h,\ \pi,\varphi) = false$ **and** $\mathsf{TG}(trg_2,\ h,\ \pi,\varphi) = false$ **then** *false* **else** *true*

- $\mathsf{TG}(\texttt{not}\ trg,\ h,\ \pi,\varphi) =$ **if** $\mathsf{TG}(trg,\ h,\ \pi,\varphi) = true$ **then** *false* **else** *true*

- $\mathsf{TG}(\texttt{quant-trigger exists}\ q\ s\ trg,\ h,\ \pi,\varphi) =$
  if there exists $val \in D_s$ **s.t.** $\mathsf{TG}(trg,\ h,\ \pi,\varphi[q \mapsto val]) = true$ **then** *true* **else** *false*

- $\mathsf{TG}(\texttt{quant-trigger forall}\ q\ s\ trg,\ h,\ \pi,\varphi) =$
  if for all $val \in D_s$ $\mathsf{TG}(trg,\ h,\ \pi,\varphi[q \mapsto val]) = true$ **then** *true* **else** *false*

- $\mathsf{TG}(\texttt{timed-trigger timeout}\ ie\ trg,\ h,\ \pi,\varphi) =$

  let $\sigma = \pi_2(hd(h))$ *the current store* **in** $\mathsf{timeout}(\mathsf{EX}(ie,\ \sigma,\ \pi,\varphi),\ trg,\ h,\ \pi,\ \varphi)$

  where

  $\mathsf{timeout}(\mathrm{n},\ trg,\ h,\ \pi,\varphi) \stackrel{\mathrm{def}}{=}$
    if $\mathrm{n} \leq 0$ **then** *true* **else**
      if $\mathsf{TG}(trg,\ h,\ \pi,\varphi) = true$ **then** *false* **else**
        if $tl(h) = \Lambda$ **then** *true* **else** $\mathsf{timeout}(\mathrm{n} - 1,\ trg,\ tl(h),\ \pi,\ \varphi)$

  Intuitively, `timed-trigger timeout` *ie trg* means that at least *ie* time-units have elapsed (i.e., at least *ie* steps have been performed) since the last time *trg* was true; and this is regarded as true, if from the current history it turns out that *trg* was always false.

- $\mathsf{TG}(\texttt{timed-trigger since}\ ie\ trg,\ h,\ \pi,\varphi) =$

  let $\sigma = \pi_2(hd(h))$ **in** $\mathsf{since}(\mathsf{EX}(ie,\ \sigma,\ \pi,\varphi),\ trg,\ h,\ \pi,\ \varphi)$

  where

  $\mathsf{since}(\mathrm{n},\ trg,\ h,\ \pi,\varphi) \stackrel{\mathrm{def}}{=}$
    if $\mathrm{n} \leq 0$ **then** *true* **else**
      if $\mathsf{TG}(trg,\ h,\ \pi,\varphi) = false$ **then** *false* **else**
        if $tl(h) = \Lambda$ **then** *false* **else** $\mathsf{since}(\mathrm{n} - 1,\ trg,\ tl(h),\ \pi,\ \varphi)$

  Intuitively, `timed-trigger since` *ie trg* means that, now and at least through the last *ie* time-units (or steps), *trg* was true; and this is regarded as false, if less than *ie* steps have been performed till now.

– TG`timed-trigger before` $ie\ trg$, h, π,φ) =

  let $\sigma = \pi_2(hd(h))$ **in** $\mathsf{before}(\mathsf{EX}(ie, \sigma, \pi,\varphi), trg, h, \pi, \varphi)$

  where

  $\mathsf{before}(\mathrm{n}, trg, h, \pi,\varphi) \stackrel{\mathrm{def}}{=}$
  if $\mathrm{n} < 0$ **then** *false* **else**
  if $\mathrm{n} = 0$ **then** $\mathsf{TG}(trg, h, \pi,\varphi)$ **else**
  if $tl(h) = \Lambda$ **then** *false* **else** $\mathsf{before}(\mathrm{n} - 1, trg, tl(h), \pi, \varphi)$

  Intuitively, `timed-trigger before` *ie trg* means that *trg* was true exactly *ie* time-units (or steps) ago, and this is regarded as false, if less than *ie* steps have been performed till now.

Event expression

- EE: `EV-EXPR` $\times$ *Store* $\times$ *PEval* $\times$ *QEval* $\rightarrow_{\mathrm{p}}$ *Events*

– $\mathsf{EE}(\mathrm{ev}\ e\ exp_1 \ldots exp_n, \sigma, \pi,\varphi) = \{(e, \mathsf{EX}(exp_1, \sigma, \pi,\varphi) \ldots \mathsf{EX}(exp_n, \sigma, \pi,\varphi))\}$

– EE(`entered` st, σ, π,φ) = {(`entered`, st)}

– EE(`exited` st, σ, π,φ) = {(`exited`, st)}

Expression

- EX: `EXPR` $\times$ *Store* $\times$ *PEval* $\times$ *QEval* $\rightarrow_{\mathrm{p}}$ *Value*

– **if** $x \in$ `VAR`, **then** $\mathsf{EX}(x, \sigma, \pi,\varphi) = \sigma(x)$

– **if** $p \in$ `PARAM`, **then** $\mathsf{EX}(p, \sigma, \pi,\varphi) = \pi(p)$

– **if** $q \in$ `Q-VAR`, **then** $\mathsf{EX}(q, \sigma, \pi,\varphi) = \varphi(q)$

– EX(`appl` $op\ exp_1 \ldots exp_n, \sigma, \pi,\varphi) = op^D(\mathsf{EX}(exp_1, \sigma, \pi,\varphi), \ldots, \mathsf{EX}(exp_n, \sigma, \pi,\varphi))$

3.4 Asynchronous Time Model (Macro-steps) Semantics

The asynchronous time model allows several steps – that form a *macro-step, or super-step* – to take place within a time-unit: time does not advance inside a macro-step. In this model, the system is assumed to repeatedly execute a single step – here called *micro-step* – reacting to both the same input and the effects due to the previous micro-steps in the current macro-step, until the system is in a *stable configuration* – i.e., there is no internally generated event, even of the form (`exited`, *st*) or (`entered`, *st*). Only then the system becomes ready to react to the next input. Hence, it might result in an infinite loop.

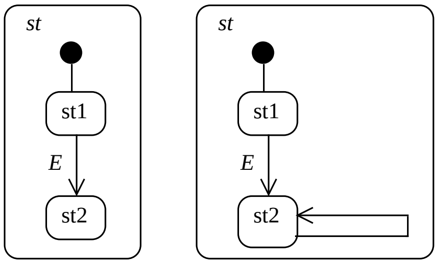


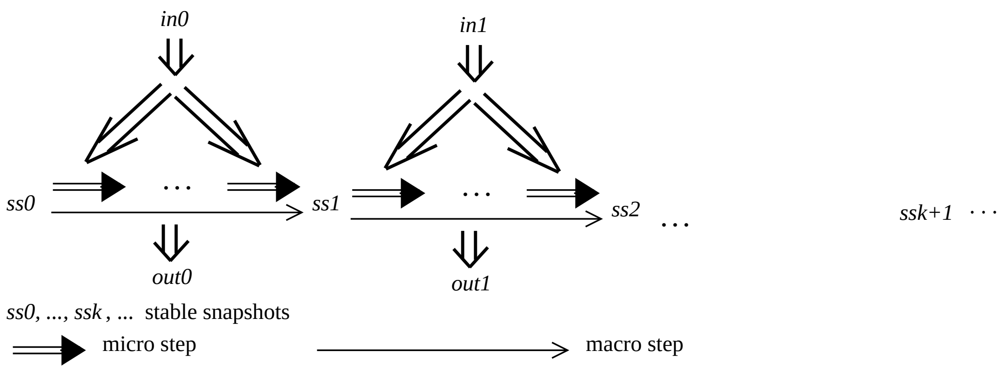
Figure 3.1: Two different charts.

In other words, the next input takes effect if and when the *whole* chart reaches a stable configuration, following a maximal chain reaction composed of a finite sequence of micro-steps.

For simplicity, according to one of the possibilities analyzed in [?], we assume that the lifetime of each external event in the input is the whole macro-step, whereas lifetime of each internally generated event is since its generation till the possible end of the macro-step. Just as before, the new values are assigned to the required input variables at the beginning of the (first micro-step in the) macro-step, whereas the appropriate local and output variables are updated at the end of each micro-step. Furthermore, a macro-step – provided that it terminates – takes one time-unit.

Let us be more explicit about what we mean by "stable configuration". Quite simply, a configuration is stable if no more moves are allowable, with the exception of the idle move. Note that the two charts in Figure 3.1 are different in their behaviour: in the former case, being in the state *st*, no move is possible (except the idle one); but, in the latter case, one move is always allowable, which generates the events (`exited`, *st*) and (`entered`, *st*): actually, in the latter case, a macro-step does not terminate. Therefore, as a general rule, we can assert that, because *no event* has been internally generated, it follows that the system has stopped: no transition has been taken, and so the status reached by the system is stable.

Graphically



In order to define this alternative dynamic semantics, it is sufficient to modify slightly the definition of CC', see 3.3: now, it applies $\mathsf{Macro}(ch,\ is{\downarrow_j},\ ss\rfloor_j)$, in place of $\mathsf{step}(\dots)$.

Hence, we define the function Macro in terms of repeated applications of another function, Micro, that "executes" one micro-step at a time; in turn, Micro is defined in terms of the same **step** as before.

- Macro: $\mathtt{CHART} \times \mathit{HistoryInput} \times \mathit{Input} \rightarrow_{\mathrm{p}} \mathcal{P}(\mathit{Snapshot})$
- Macro: $\mathtt{CHART} \times \mathit{History} \times \mathit{Input} \rightarrow_{\mathrm{p}} \mathcal{P}(\mathit{Snapshot})$

$\mathsf{Macro}(ch,h,\ in) = \mathsf{Micro}(\mathsf{step}(ch,h,in))$

$\mathsf{Micro}(ch,h,\ in) =$

if $\mathsf{step}(ch,h,\ in) = \{\ hd(h)\}$ **then**

the head of h consists of a stable configuration

$\{\ hd(h)\}$

else $\cup_{sn \in\ \mathsf{step}(ch,h,\ in)} \mathsf{Micro}(ch,h\ \&\ sn,\ in)$

.1 Notations

Let X and Y be two sets.

- $[X \to Y]$ denotes the class of the *total functions* from X to Y, whereas $[X \to_{\mathrm{p}} Y]$ denotes the class of the *partial functions*.
- If $f \in [X \to_{\mathrm{p}} Y]$, then $\mathrm{Dom}(f)$ denotes the *domain* of f, i.e., the set $\{x \mid x \in X \text{ and } f(x) \text{ is defined}\}$.
- $\mathit{Empty}_{\mathrm{F}}$ denotes the function in $[X \to_{\mathrm{p}} Y]$ s.t. $\mathrm{Dom}(\mathit{Empty}_{\mathrm{F}}) = \emptyset$.
- If $f \in [X \to_{\mathrm{p}} Y]$, $x' \in X$, and $y' \in Y$, then $f[x' \mapsto y']$ denotes the function s.t. $\mathrm{Dom}(f[x' \mapsto y']) = \mathrm{Dom}(f) \cup \{x'\}$, and $f[x' \mapsto y'](x) \stackrel{\text{def}}{=} \textbf{if } x = x' \textbf{ then } y' \textbf{ else } f(x)$.
- f^k, where $k \geq 1$, denotes k repeated applications of the function $f \in [X \to X]$.
- π_i denotes the i-th projection of a generic tuple $(x_1, \ldots, x_n)$, where $n \geq 1$: $\pi_i(x_1, \ldots, x_i, \ldots, x_n) = x_i$ for $i = 1, \ldots, n$.
- $\mathcal{P}_{\mathit{fin}}(X)$ [resp., $\mathcal{P}(X)$] denotes the set of the finite [resp., finite or infinite] subsets of X.
- X^* denotes the set of the possibly empty [resp., nonempty] finite sequences (or *lists*) of elements in X.
- X^∞ denotes the set of the infinite sequences of elements in some nonempty set X (or *streams*), that is to say total functions from naturals to X.
- We use the following functions on streams and lists
  - Λ denotes the empty list.
  - $\_\&\_$: $X \times X^* \to X^+$ adding an element to a list
  - hd: $X^* \to_{\mathrm{p}} X$
    - $hd(\Lambda)$ undefined
    - $hd(x \;\&\; l) = x$
  - tl: $X^* \to_{\mathrm{p}} X^*$
    - $th(\Lambda)$ undefined
    - $th(x \;\&\; l) = l$
  - $\_\downarrow\_$: $X^\infty \times NAT \to_{\mathrm{p}} X$

    $xs\downarrow_j$ returns the j-th element of xs
  - $\_\rfloor\_$: $X^\infty \times NAT \to_{\mathrm{p}} X^+$
    - $xs\rfloor_0 = (xs\downarrow_0) \;\&\; \Lambda$
    - $xs\rfloor_{j+1} = (xs\downarrow_{j+1}) \;\&\; (xs\rfloor_j)$

.2 Abstract Syntax

```
CC-SPEC             ::=  cc-spec  SPEC  EVENT-DECS  VAR-DECS  CHART

EVENT-DECS          ::=  event-decs  EVENT-DEC*
EVENT-DEC           ::=  event-dec  EV-NAME  SORT*  MODE
EV-NAME             ::=  SIMPLE-ID
MODE                ::=  input  |  local  |  output

VAR-DECS            ::=  var-decs  VAR-DEC*
VAR-DEC             ::=  var-dec  VAR  SORT  MODE
VAR                ::= SIMPLE-ID

CHART               ::=  chart  STATE  HIERARCHICAL-CHART+
HIERARCHICAL-CHART  ::=hierarchical-chart  STATE+  STATE  TRANS  DECOMPS

STATE               ::=  SIMPLE-ID

DECOMPS             ::=  decomps  CHART*

TRANS               ::=  trans  TRAN*
TRAN                ::=  tran  STATE  TRAN-PARAMS  TRIGGER  ACTIONS  STATE

TRAN-PARAMS         ::=  tran-params  TRAN-PARAM*  TRIGGER
TRAN-PARAM          ::=  tran-param  PARAM  SORT
PARAM               ::=  SIMPLE-ID

ACTIONS             ::=  actions  ACTION*
ACTION              ::=  assign  VAR  EXPR  |  gen-ev  EV-NAME  EXPR*

TRIGGER             ::=  true  |  happens  EV-EXPR  |  in-state  STATE  |
                         pred  PRED-NAME  EXPR*  |  equal  EXPR  EXPR  |
                         and  TRIGGER  TRIGGER  |  or  TRIGGER  TRIGGER  |  not  TRIGGER  |
                         quant-trigger  QUANT-OP  Q-VAR  SORT  TRIGGER  |
                         timed-trigger  TIMED-OP  EXPR  TRIGGER
Q-VAR                ::=  SIMPLE-ID
QUANT-OP             ::=  exists  |  forall
TIMED-OP             ::=  timeout  |  since  |  before

EV-EXPR              ::=  ev  EV-NAME  EXPR*  |  entered  STATE  |  exited  STATE

EXPR                 ::=  VAR  |  PARAM  |  Q-VAR  |  appl  OP-NAME  EXPR*
```

We refer to the CASL summary [oLD98] for what concerns the nonterminal symbols SPEC, SORT, OP-NAME, PRED-NAME, and SIMPLE-ID.

Bibliography

[HN96] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[oLD98] The CoFI Task Group on Language Design. CASL Summary. Version 1.0. Technical report, 1998. Available on `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.

[oS99] The CoFI Task Group on Semantics. textscCasl The Common Algebraic Specification Language: Semantics CoFI Note S-9. Technical report, 1999. `ftp://ftp.brics.dk/Projects/CoFI/Notes/S-9/`.

[RM99] M. Roggenbach and T. Mossakovski. Basic Data Types in CASL . CoFI Note L-12. Technical report, 1999. `http://www.brics.dk/Projects/CoFI/Notes/L-12/` .

[RR00] G. Reggio and L. Repetto. CASL-CHART: A Combination of Statecharts and of the Algebraic Specification Language CASL. Technical Report DISI-TR-00-2, DISI – Università di Genova, Italy, 2000. `ftp://ftp.disi.unige.it/person/ReggioG/ReggioRepetto00b.ps`.

[The99] The CoFI Task Group on Language Design. CASL The Common Algebraic Specification Language Summary. Version 1.0. Technical report, 1999. Available on `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.