# A Translation of Statecharts and Activitycharts into Signal Equations

Jean-René Beauvais, Roland Houdebine, Paul Le Guernic, Eric Rutten, Thierry Gautier

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *A translation of Statecharts and Activitycharts into Signal equations*

J-R Beauvais, R. Houdebine, P. Le Guernic, E. Rutten, T. Gautier

**N ° 3397**

Avril 1998

THÈME 1

Rapport de recherche

ISSN 0249-6399

# A translation of Statecharts and Activitycharts into Signal equations *

J-R Beauvais, R. Houdebine, P. Le Guernic, E. Rutten, T. Gautier†

Thème 1 — Réseaux et systèmes
Projet Ep-Atr

Rapport de recherche no 3397 — Avril 1998 — 56 pages

**Abstract:** The languages for modeling reactive systems can be divided in two styles: the imperative, state-based ones and the declarative, data-flow ones. Each of them is best adapted to a given application domain. This paper, through the example of the languages Statecharts and Signal, shows a way to translate an imperative specification (Statecharts) to a declarative, equational one (Signal). This translation makes multi-formalism specification possible, and provides a support for the interoperability of the languages. It gives access from a Statecharts specification to the DC+ exchange format between the tools implementing the synchronous technology, using e.g. the clock calculus available in Signal. Statecharts specifications can thereby be applied functionalities of verification, validation, compilation, optimization, efficient and compact code generation, distributed and execution architecture-dependent code generation. The results presented here cover the essential features of StateCharts as well as of another language of Statemate: Activitycharts.

**Key-words:** Signal, Statecharts, Activitycharts, DC+, reactive & real-time systems, synchronous languages, interoperability, code generation

*(Résumé : tsvp)*

* The work described in this paper is partly funded by the CEC as Esprit Project EP 20897 SACRES (SAfety CRitical Embedded Systems: from requirements to system architecture).

† email: {Beauvais|Thierry.Gautier|Roland.Houdebine|Paul.LeGuernic|Eric.Rutten}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

# Une traduction de Statecharts et Activitycharts en équations Signal

**Résumé :** Les langages de modélisation des systèmes réactifs peuvent être divisés en deux styles : les langages impératifs ou à états et les langages déclaratifs ou à flots de données. Chacun est plus adapté à un domaine d'application donné. Ce rapport, au travers des langages Statecharts et Signal, montre une méthode de traduction d'une spécification impérative en une spécification déclarative (équationelle). Cette traduction rend possible la spécification multi-formalisme, et fournit un support à l'interopérabilité des langages. Elle donne accès depuis une spécification en Statecharts au format DC+ d'échange entre les outils mettant en œuvre la technologie synchrone, et utilisant le calcul d'horloges de Signal. Des spécifications en Statecharts peuvent ainsi se voir appliquer des fonctionnalités de vérification, validation, compilation, optimisation, génération de code efficace et compact, génération de code distribué et dépendant de l'architecture d'exécution. Les resultats présentés ici couvrent les aspects essentiels de Statecharts, ainsi que d'un autre langage de StateMate: Activitycharts.

**Mots-clé :** Signal, Statecharts, Activitycharts, DC+, systèmes réactifs & temps réels, langages synchrones, interopérabilité, génération de code

# 1 Introduction

## 1.1 Context and objective

Different languages exist for the design of reactive systems: the languages Lustre [8] and Signal [6],[3] are declarative and equational data flow languages, while Esterel [5], Statecharts [9] and Argos [15] are imperative sequencing languages. The choice between the declarative and the imperative approach has an influence upon facility to handle a given application area. For instance, declarative languages easily handle signal processing while imperative formalisms are often used for sequential control systems. The need for a control mechanism such as task management for example appears in application domains involving the control of physical processes. For complex systems involving the two aspects, a multi-formalism specification can be useful.
This paper is a proposal to give a translation from the essential features of Statecharts and Activitycharts to the equational language Signal. Among the different semantics of Statecharts [13], the translation proposed here follows the Statemate one [11].

## 1.2 Motivations

Signal being a representative of the class of declarative synchronous languages, this translation:

- provides a way to merge imperative and declarative synchronous languages by simply composing equations (composition of Signal processes),
- fulfills the lack of imperative features of Signal,
- gives a compositional definition of the considered Statecharts semantics,
- opens a direct connection from a Statecharts design to the synchronous technology tools of the Signal environment, but also of those compatible with the DC+ format: compilers, simulators, verification systems,
- one of these tools is a code generator that can produce efficient and compact code from a Statecharts specification, using the clock calculus available in Signal. It can also generate distributed and architecture-dependent code.

We believe that the graphical readability of Statecharts makes it a good candidate for the design of an imperative specification. The Signal language, using an elaborate clock calculus makes it a good choice to extract clock properties from a specification in order to get efficiency in code generation and in verification. Moreover, keeping the structural information through the translation and not simply coding is a key issue for understanding interactions between components from different sources. Taking care of the traceability between the initial specification and the generated code allows to route information extracted from the verification tools back to the specification for user feedback (diagnostic, counter-example). The other way, from the specification to the generated code, traceability may be used to add, at specification time, directives for thee partitioning into tasks or distributed processes.

A concrete motivation and application of this work takes place in the context of the Sacres programming environment [7]. The purpose of the Esprit project SACRES (*SAfety CRitical Embedded Systems: from requirements to system architecture*) is to integrate into a unified and complete environment a variety of tools for specification, verification, code generation and validation of the code produced. Among the application domains targeted are avionics and process control. The question of certification and validation is integrated into the environment. Member partners of the SACRES project are: British Aerospace (UK), aircraft builder; i-Logix (UK), who develop and distribute STATEMATE, the environment for designing in STATECHARTS; INRIA (France), a research institute where new technologies are defined and developped around the synchronous language SIGNAL [6]; OFFIS (Germany), research institute bringing verification technology; Siemens (Germany), where controllers for industrial processes are developped; SNECMA (France), builder of aircraft engines; TNI (France), who develop and distribute the SILDEX tool and the SIGNAL language; the Weizmann Institute (Israel), as regards semantic aspects and the validation of code. Figure 1 illustrates the architecture of the SACRES toolset. It shows information flows between the elements of the toolset, and the central position of the format between the tools of the environment. Translators to and from DC+ are developed in the framework of the project, and enable the connection of all the representations specific to the different tools, using the common format. The translation from Statecharts to DC+ is one of them.

DC+ is an exchange format which supports the representation of Signal; as such, they are quite in nature: both are defined in terms of systems of equations over flows or signals. Signal being a programming language, it is prefereable to use it for readability purposes, hence this paper presents the translation in terms of Signal rather that DC+.

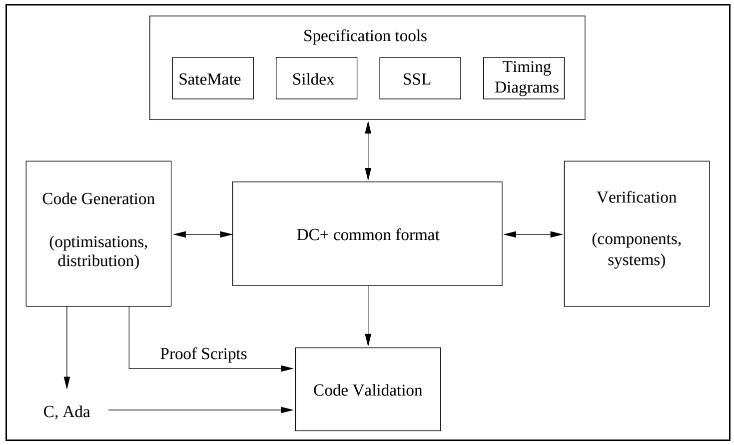


Figure 1: Global architecture of the SACRES environment.

### 1.3 Related work

Differents attempts have been made to mix imperative and declarative synchronous languages. In [15], the authors present a way to compile Argos (a hierarchical concurrent automata language, which can be considered one of the Statecharts variants) into a Mealy machine implicitly represented by a set of boolean equations in the declarative code DC [20]. Each state of the hierarchical automaton is associated with a boolean signal being *true* when the state is active and *false* otherwise. This boolean signal is updated when, in the Argos hierarchy, the state to which it belongs is activated. The configuration of a Statechart (the list of its active states) is hence represented as a tree of booleans. Mixing these equations with DC equations generated from other languages (e.g. Lustre), provides a way to mix imperative/declarative formalisms. The translation covers some basic features of the Statecharts: hierarchical parallel automata with event sending for actions. Then, using the semantics of both languages, the authors prove that the translation preserves the behavior from the point of view of trace. However, the semantics adopted is the Argos one which is a kind of purely synchronous semantics of Statecharts different from the Statemate one [11]. Also, a lot of features of the languages of Statemate are absent from Argos. In [4], the author gives a semantics of the ESTEREL synchronous language in terms of electric circuits. First, the substatements of a ESTEREL statement are individually translated into circuits, then, the obtained circuits are combined using appropriate auxiliary gates and wiring. Some aspects of the translation reveal to be close to the

one presented here: particularly the subcircuit interface which is close to the one of Section 4.1 and the wiring of the controls signals between the subcircuits.
A tool for the integration of different synchronous languages is being developed in the SYNCHRONIE project [14]. SYNCHRONIE is a workbench for synchronous programming. It provides compilation, simulation, testing and verification tools for various dialects of the synchronous programming paradigm. In the first instance ESTEREL, ARGOS and LUSTRE compilers are being developed and integrated. The integration is made through a common semantical representation: synchronous automata which are essentially Mealy machines. The equational translations of StateCharts into a synchronous model presented here might be supported also by SYNCHRONIE.
An attempt using the declarative Signal language has already been done in [16]. This is an extension of the Signal language called Signal *GTi* with constructs for hierarchical task preemption. In the declarative synchronous language Signal, a process defines a behavior of an unbounded series of instants. However, there are no explicit language constructs handling the termination, interruption or sequencing of processes, that is to say the limitation of behaviors to a slice of this series of instants. In Signal *GTi*, tasks are defined as the association of a data-flow process with a time interval on which it is executed. Both data-flow and tasking paradigms are available within the same language-level framework. An implementation of Signal *GTi* as a preprocessor to the Signal environment [17] consisted in the generation of equations for activity management, using additional control signal similar to the reactive box model of Section 4.1.

## 1.4 Organization of the paper

This paper extends a shorter presentation [2] with a wider coverage of Statemate actions, as well as a management of activities. After a description of the Signal and Statecharts formalisms in Sections 2 and 3, it presents the translation of the major constructs of the Statecharts in Section 4. Then a translation is given in Section 5, with some examples to illustrate it. It covers the essential features of a Statecharts and Activitycharts; it concentrates on the behavioral aspects, in the framework of the step semantics; Other aspects like elaborate data-types, the superstep semantics, some particular aspects of actions, ..., are part of the perspectives. Section 6 describes the translation schemes for Activitycharts. In Section 7 describes how Signal can be used to model nondeterminism, and indicates how the translation can be modified accordingly.

## 2 Signal: a declarative synchronous language

Signal is a synchronous real-time language, declarative, data flow oriented and built around a minimal kernel of operators [6, 3]. It manipulates signals, which are unbounded series of typed values (e.g., `integer`, `logical`). They have an associated clock defined as the set of instants where values are present. Given a signal `X`, its clock is `CX` obtained by `CX := event X`, giving the event present simultaneously with `X`. The constructs of the language can be used to specify, in an equational style, relations or constraints of clock inclusion or clock equality between signals, and functions of values. Systems of equations on signals are built using composition. This composition is strictly synchronous, meaning that it constructs the system of equations on signals, describing the relation between all of them at the same logical instant. In particular, this involves that inputs and outputs of an equation are present at the same instant, and that composed equations share signals within that same instant. The kernel of Signal comprises the following *primitive processes*:

**Functions** `Y := f( X1, X2, ... , Xn )` e.g., boolean negation: `Y := not E`.

**Delay** `ZX := X$1 init V0` gives the past value of $X$ (with initial value $V_0$).

**Selection** `Y := X when C` according to a boolean condition `C`.

**Deterministic merge** `Z := X default Y` (with priority to `X` when both are present).

**Parallel composition** `(|` $P_1$ `|` $P_2$ `|)` union of the systems of equations.

The following table illustrates each of the primitives with a trace:

| n | 4 | 3 | 2 | 1 | 0 | 4 | 3 | 2 | 1 | 0 | 4 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `zn := n$1 init 0` | 0 | 4 | 3 | 2 | 1 | 0 | 4 | 3 | 2 | 1 | 0 | ... |
| `p := zn-1` | -1 | 3 | 2 | 1 | 0 | -1 | 3 | 2 | 1 | 0 | -1 | ... |
| `fill := true when zn=0` | t | | | | | t | | | | | t | ... |
| `empty := true when (n=0) default (not fill)` | f | | | | t | f | | | | t | f | ... |

The rest of the language is built upon this kernel. Derived operators have been defined from the primitive operators, providing programming comfort. E.g., `X ^= Y` constrains the signals `X` and `Y` to be synchronous, i.e. their clocks to be equal. A structuring mechanism is proposed in the form of `process` schemes. The process `CB := when B` gives the clock `CB` of occurrences of the logical signal `B` at the value `true`. The `^0` signal is the null clock ie. a signal that is never present.

The Signal compiler performs the analysis of the consistency of the system of equations (absence of causal cycles), and determines whether the synchronization constraints between the signals are verified or not. The compiler synthesizes control through a clock hierarchy based on instant presence inclusion. A clock calculus using algorithms on BDDs may reorder the clock hierarchy [1]. In the course of development, a program can also be checked for real-time properties through timing analysis [12]. Eventually, executable code can be produced automatically (in C, FORTRAN or ADA). The compiler is being re-designed as a virtual machine for the transformation of a hierarchical conditioned-dependencies-graph representation of programs, for which an external exchange format is DC+ [18]. The SACRES project, mentioned earlier, is the context of an application of this design.

```
process tank = {integer capacity;}
( ? event fill;
  ! boolean empty;)
(| when(zn=0) ^= fill
 | zn := n $1 init 0
 | p := zn - 1
 | n := (capacity when fill) default p
 | empty := when (n=0) default (not fill)
 |)
where integer n, zn, p;
end;
```

Figure 2: *A refillable tank*

Figure 2 shows an example of a Signal program describing a refillable tank. This process named "`tank`" has a constant parameter `capacity`, an input signal: `fill`

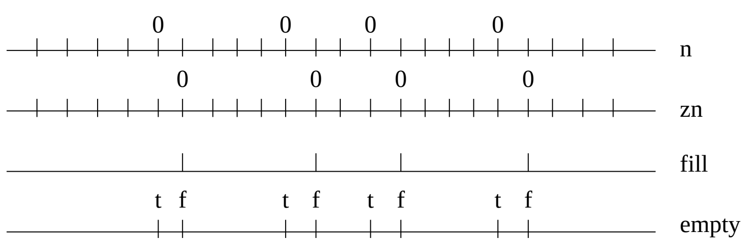


Figure 3: *The clocks of the tank process*

(pure event), an output signal `empty` (boolean) and a list of equations defining the body of the process. The behavior described in this process is the following: whenever the tank is filled, with input signal `fill` set, the level of water in the tank starts to decrease (`n`) until the level reaches 0. At this time, the output `empty` signal is set to true. Then, the next `fill` can refill the tank and set the `empty` signal to false.
The presence instants (clocks) of the signals in this program are illustrated in figure 3. One can notice that clocks of local (internal) signals is faster (i.e., includes) than that of inputs and outputs: in that sense it is possible to specify oversamplings in Signal, i.e. processes which are not necessarily strictly reactive to their inputs.

## 3 Statemate: Statecharts and Activitycharts

**Overall organization.** The Statecharts formalism has been introduced by Harel [9]. It is a graphical language based on automata. It is integrated in the StateMate environment, along with another language called Activitycharts, which is block-diagram oriented. It is implemented in the tool Magnum, designed by i-Logix. The specification of a model in Statemate is composed of charts. To each chart is associated the declaration of data-items (i.e. variables with a given type) and events, hence defining their scope: these are known inside the chart. Other data-items and/or events can be exchanged with the environment. The chart is further defined by either an Activitychart or a Statechart, which can be itself decomposed hierarchically into sub-charts. The entry point for a model is an Activitychart, which describes a structural decomposition by being divided into sub-activities, recursively. Some sub-activities, called control activities, can be defined by a Statechart.

**Hierarchical parallel automata.** A Statecharts design essentially consists of states and transitions like a finite automaton. In order to model depth, a state can be refined and contain sub-states and internal transitions. Two such refinements are available: **and** and **or** states, that give a state hierarchy. At the bottom of the hierarchy, **Basic**-states are not further refined. If the system specified by a Statechart resides in an **or** state, then it also resides in exactly one of its direct sub-states. Staying in an **and** state implies staying in all of its direct sub-states and models concurrency. When a state is left, each sub-state is also left, thereby modeling preemption. Sub-states of an **and** state may contain transitions which can be executed simultaneously. The configuration of a Statechart is defined by the hierarchy of states and sub-states in which it stays. The different **and** parts of a state may communicate by internal events which are broadcasted all over the scope

of the events. For instance, the emission of an event on a transition may be sensed somewhere else in the design and trigger a new transition.
In the Statechart example of figure 4, the basic components are states and transitions, some states clustered in **or** composition (`Sub_Running_Up` is an **or** state containing `S1` and `S2`) while some other groups in **and** composition (`Running` is an **and** state containing `Sub_Running_Up` and `Sub_Running_Down`).
When entering a state containing sub-states, different behaviors are possible: when entering a state by a transition pointing to the boundary of the state, the state targeted at by the **default** connector (a transition without origin) is activated. When reentering a state through a history connector (**H**), the sub-state activated is the one that was active when the state was left. When entering for the first time a state containing a history connector, the transition leaving this history connector is used to find the sub-state to activate. Finally, deep-history (**H***) is a connector that acts similarly to the history connector but applies to all the sub-states in the hierarchy. Note that the same state can have all the three ways of being entered, hence the corresponding mechanisme is applied, according to the transition through which it is entered.

**Transitions and actions in a step.** The transitions between states are labeled by reactions of the form: `e[C]/a`, where `e` is an event that possibly triggers the transition, `C` is a boolean guard condition that has to be true to pass the transition. The previous event and the boolean together give the trigger part of the transition while the right part of the "/" (`a`) contains the actions that are carried out if and

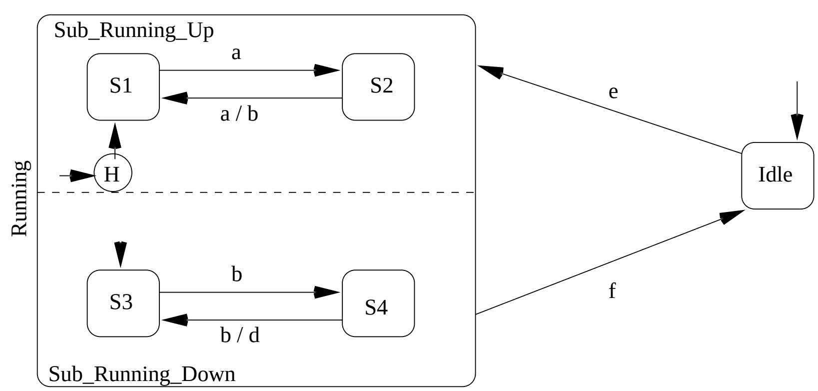


Figure 4: *A Statechart example*

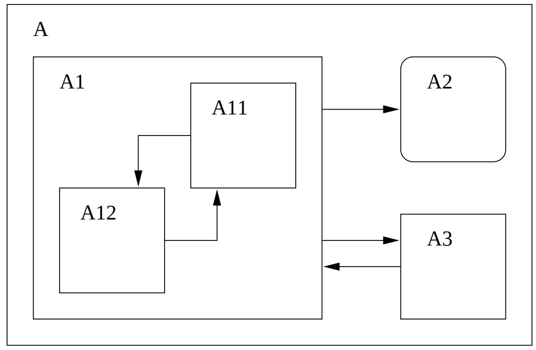


Figure 5: *An activities hierarchy.*

when the transition is fired. As a special kind of transitions, Statemate offers the possibility to associate such labels to a state. Whenever this state is active, the trigger part of the transition is evaluated and possibly the action is carried out. Such transitions are called static reactions.

The basic evolution of a Statechart consists in a step, where given the events currently present and the current values of variables, triggers and conditions are evaluated, and actions are carried out. In Statemate the effects (event generation, variable modifications) of the actions carried out in one step are sensed only at the following step. This makes a difference with other semantics [13], e.g. strictly synchronous as in ARGOS [15].

The step semantics is the interpretation of a StateMate specification where inputs from the environment are considered at each step, taking part in the current events and variables. Another semantics is the superstep intepretation: here, inputs take part only in the first of a series of steps, called a superstep. There, the following steps take in consideration only the effects of the previous one (i.e. locally emitted events and changed values), until there is no transition to take anymore, i.e. no step to make. This situation, called stable, is the end of the superstep, and inputs are acquired from the environment anew.

For actions consisting in assigning values to variables, the same variable can be referenced in assignments associated with different transitions: each provides with a contributed value, and the variable takes its values from the action contributing in the current step.

**Activities.** Besides the Statecharts, another language of the StateMate environment is Activitycharts [10]: it provides the designer with a notion of multi-level data-flow diagrams, as illustrated in Figure 5. Each of the blocks in the hierarchy represents an activity. The activities can be used to construct a structure decomposed hierarchically.
At each level, one of the activities, designated in the graphical syntax by a rounded-cornered box, can be a control activity (e.g. activiy $A2$ in Fig.5). It is associated with a Statechart defining its behavior. The latter can start, stop, suspend and resume the activity, as well as sense its current status.
Actions with a trigger can be associated with an activity, they are called mini-specs, and have the same form as labels seen associated with transitions or static reactions with states in Statecharts.
Links between activities represent the data or control exchanges between activities. They can be augmented with elements called data-stores representing the way data is kept along steps. This aspect of the language is however based on the fact that variables and events are known all over a chart (the scope of their definition). Therefore it will not be handled explicitely in the translation. The representation of data flow links is but an explicitation of communications existing anyway.
Finally, a concept of ModuleCharts also exists in StateMate, handling the association of a specification with an execution architecture. This point is not covered by the present work.

# 4 Translation principles

First, we introduce a reactive box model, then, in order to simplify and to structure the translation from Statecharts to Signal, some predefined processes useful for the translation are given. They correspond to the basic features of the Statecharts.

## 4.1 The reactive box

As a common framework for reactive specification, we define a model of reactive box with a normalized interface. Each part of the design to translate will have this interface scheme represented as a process in Signal. In particular, and-nodes and or-nodes will each be translated into a process with this structure, hence insuring the hierarchical propagation of control signals such as clocks and resets.
In the Signal language, an interesting property is that the behavior of the composition of two processes is the intersection of the behaviors of the constituent processes. This

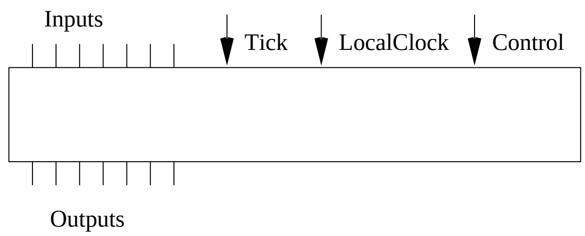


Figure 6: *The reactive box model*

is similar to the solutions of an equational system. Hence, this reactive box model is compositional in the same sense and gives a compositional semantics for Statecharts that may be used to do modular proofs.
A reactive box is a box containing input and output signals. Some of these interface signals are common to every box:

- **Tick** is the clock of the whole design. It has been added because in Statemate, the events generated by a step are sensed only at the beginning of the following step (generated events are shifted). This signal is the reference clock used for the purpose of shifting these events and value changes by one instant of the global clock[1].
- **LocalClock** is the clock of the box. This clock is present whenever the corresponding Statecharts component is active.
- **Control** is an enumerated-typed signal ranging in **Start, Stop, Resume, LocalResume**. This type is called `tcontrol`. It is used inside the box to know when and how the box is (re)entered. If its value is **Start**, all the (sub)levels of the box need to be reset. If its value is **LocalResume** (e.g. the corresponding state is activated through a history connector) the box needs to be reset for all the levels but the level where the **LocalResume** connector belongs. **Resume** means that the box is activated but no reset has to be performed. The **Resume** value is useful because for instance in Statemate, a state may contain **entering** in the trigger part of a static reaction that enables the static reaction when the state is entered. Similarly with **exiting**, the value **Stop** is used when leaving a state in Statemate.

[1]Note that, however, memorization associated to the shifting process will be managed with optimizations w.r.t inactive sub-statecharts

In Signal, basic objects are signals which always have a clock while in Statemate, only events are clocked. Variables in Statemate are of two kind: events or data-items. Data-items are valued and always present while events are only present or absent. In order to reach compositionality, during the translation, we need to associate a clock with each Statemate variable. The clocks of the data-items will be computed from the signals **Tick** and **LocalClock**.

## 4.2 Testing absence

Whereas in Statecharts conditions are always available, in Signal they have their own clock. This is why in the translation we mix the event and the condition guard of the transition in the `trigger` signal.

| `e` | Statechart `not e` | Signal `not e` |
|---|---|---|
| t | f | f |
| Absent | t | Absent |

Figure 7: *Differences between Statemate not and Signal not.* In Statechart, not e means: e did not occur, while in Signal it is a conservative extension of the not on the booleans

Statecharts offers the possibility to check whether an event is present or not because it is single clocked while in Signal (multi clocking), asking for absence is with regard to a reference clock. The `not` of the two languages have a different behavior (see figure 7). For a Statechart design using the `not` feature a Signal process is used:

```
process not_event =
( ? event e1, ref_clock;
  ! logical e2;)
(| e2 := not(e1) default ref_clock |)
end ;
```

This process takes an event `e1` and a clock `ref_clock` and returns a boolean `true` when `ref_clock` is present and not `e1`. Otherwise, the process returns `false`.

**Note on synchrony:** `and` is the Signal synchronous operator. Because `e1` and `not(e1)` are synchronous, this process does not generate any clock constraints.

**Note on and/or on events:** The Statecharts event `e1 and e2` occurs when both `e1` and `e2` occurred simultaneously. It is translated into Signal: `e1 when e2`. The

Statecharts event `e1 or e2` occurs when either `e1` or `e2` occurred. It is translated into Signal: `e1 default e2`.

**Note on conditions:** Statecharts uses `and, or, not` also for conditions. These ones are translated using the primitives `and, or, not` of Signal.

### 4.3 Transition

To check if a transition is triggered, a specific process is designed that is instantiated for each transition.

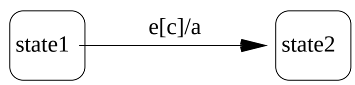


Figure 8: *A transition in Statecharts*

**Signal encoding:**

```
process transition = {state1,state2}
( ? state origin; event trigger;
  ! state target;)
(| target := state2 when trigger when (origin=state1)
 |)
end;
```

**Interface:** `state1` is the initial state of the transition (a value of an enumerated type called `state`), `state2` is the target state of the transition (a value of the same enumerated type), `origin` is a signal containing the current active state of the level where `state1` belongs (see the section 4.4 about configuration signal), `trigger` is the signal which triggers the transition, `target` gives the new state chosen when the transition is fired.

**Behavior:** To use this process, one needs to instantiate `{state1,state2}` with the initial and the target state of the actual transition. It outputs a new state value whenever the transition it describes is fired. *Presence* of an output means that the transition is enabled, its *value* shows the new state (which is `state2`) that will be reached if the transition is actually *taken*.

This output signal for each transition is then used to compute the new configuration. The choice between possibly several enabled transitions handles solving priority between conflicting transitions (see section 5.1 and, for the case of non-determinism, section 7). The result is fed in the new configuration, managed by the predefined process described next.

### 4.4 State

The configuration of a Statechart is the list of its active states at a given point in time. For a n-states flat automaton, the configuration (i.e., which state is active) is handled by a signal ranging on an enumerated type (`state`) of n different values: one value for one sub-state.
For figure 4, if `s, t, u` are respectively the configuration variables of the *top level*, and of sub-states `Sub_Running_Up` and `Sub_Running_Down`, then legal configurations are:

| s | t | u |
|---|---|---|
| Idle | *absent* | *absent* |
| Running | S1 | S3 |
| Running | S1 | S4 |
| Running | S2 | S3 |
| Running | S2 | S4 |

This encoding ensures a basic Statecharts property: A configuration cannot be simultaneously in two different sub-states of an **or**-state. This is ensured by the fact that configuration at each level of the hierarchy is stored in one signal having at most one value at each instant.
Because a state could be refined also into sub-states, a new configuration signal (ranging in a new enumerated type) will be associated with each sub-state. The states of a Statechart form a tree, hence we have a signal tree. The clock calculus of the Signal compiler uses this information to produce optimized code: whenever a state is not active, the signal associated with it is not present and hence all the sub-states in the tree will **not** be calculated. The clock hierarchy maps the state encapsulation.
We aim to get a simple translation from Statecharts to Signal where a Statechart is just encoded via a Signal process. In particular, the structure of the original Statechart should be visible in the translated process in order to have traceability, to use the Signal clock calculus and avoid the computation for any non-active state.

For every level of the Statecharts hierarchy, an instance of the following process is used to update the configuration variable.

**Signal encoding:**

```
process nextstate = {initial_state}
( ? event localclock; state new;  tcontrol control;
  ! zconfiguration, configuration;)
(| configuration := new default (initial_state when control=Start)
                    default zconfiguration
 | zconfiguration := configuration $1 init initial_state
 | configuration ^= localclock
 |)
where integer configuration, zconfiguration;
end;
```

**Interface:** `localclock` is the clock at which the state is active, as defined in section 5.1.3; it is local to each configuration variable. `new` is the new value of the configuration variable computed with processes `transition`. `control` is a signal of type `tcontrol` as defined in setcion 4.1, and used to reinitialize the configuration when its value is `Start`.

**Behavior:** This process is used to memorize the current configuration of the Statechart when no transition occurs (or a transition occurs somewhere else in the design). The parameter `initial_state` gives the default state of the **or**-state. When this process is used, three situations can occur:

- if `control=Start`, the current state takes the initial value given as the default parameter `initial_state`,
- if a new value occurs (`new` is present), `configuration` takes it as a new value,
- if `localclock` occurs alone, `configuration` remains unchanged (copied from its previous value).

**Extensions:** Associated with each state $S$ are some control events, which can be featured in triggers (see section 5.2.2). They are referring to the current state, and hence are not shifted to the next step, differently from other events (see section 4.5). Their definition can be added to the process `nextstate` and its profile as follows:

- in_$S$ is emitted when the current configuration is in state $S$: the process `nextstate` needs to have an output `in_s` (in the instanciation for state $S$, it is linked with in_$S$). The additional equation is, quite simply:

```
in_s := localclock
```

- en_$S$ is emitted when the state $S$ is currently being entered: the process `nextstate` needs to have an output `en_s` (linked with en_$S$). The additional equation is:

```
en_s :=  when control=Start
         default when control=Resume
         default when control=LocalResume
```

- ex_$S$ is emitted when the state $S$ is currently being exited: the `nextstate` process needs to have an output `ex_s` (linked with ex_$S$). The additional equation is:

```
ex_s := when control=Stop
```

## 4.5 Shift

The Statemate semantics of Statecharts [11] states that "calculation in one step is based on the situation at the beginning of the step" and "Reactions to external and internal events, and changes that occur in a step, can be sensed only after completion of the step". We hence need to postpone the result of the current calculation (generated events for instance) to the next "step". The process `shift` aims at that.

**Signal encoding:**

```
process shift =
( ? x;
    event tick;
  ! y;)
(| instant_x := event x default not tick
 | shift_instant_x := instant_x $1 init false
 | value_x := x default shift_value_x
 | shift_value_x := value_x $1
 | value_x ^= event x default tick
 | y := shift_value_x when shift_instant_x
```

```
 |)
where boolean instant_x, shift_instant_x;
      shift_value_x, value_x;
end;
```

The initialisation of `shift_value_x` is irrelevant since it will not be output, because of the filtering out by `shift_instant_x`, initially `false`.

**Interface:** `x` is the signal to be shifted, `y` the shifted signal and `tick` the clock of the StateMate step.

**Behavior:** A trace example with integer values for `x` is shown figure 9.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Tick | • | • | • | • | • | • | • | • | • | • | • |
| x | 1 | | 2 | | | | 3 | 4 | 5 | | |
| instant_x | t | f | t | f | f | f | t | t | t | f | f |
| shift_instant_x | f | t | f | t | f | f | f | t | t | t | f |
| value_x | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 4 | 5 | 5 | 5 |
| shift_value_x | | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 4 | 5 | 5 |
| y | | 1 | | 2 | | | | 3 | 4 | 5 | |

Figure 9: *Trace of the shift process (Integer values for x)*

Given a signal and a clock (usually the fastest clock), it shifts the signal to the next "tick" of the clock. This involves to encode the clock of the signal to be shifted into a Boolean `instant_x`, which is shifted in `shift_instant_x`. The value is also shifted, and output at the shifted clock.

All variables (except configuration variables) are encoded in signals at the fastest clock so that their value is always available. Hence `shift` is with respect to the fastest clock.

If one wants to have a *perfect synchrony hypothesis* [13], the shift should be removed and then input and corresponding output would occur at the same time. Solving causality cycles would be left (when possible) to the Signal compiler and this would correspond to a synchronous semantics of the Statecharts.

**Extensions:**

**Shifting events:** For efficiency reasons (code generation, verification...), if `shift` is used with events where there is no value to memorize, a specific optimized version is used instead, called `shift_event`, where `x` and `y` have no value, and corresponding indtructions have been removed:

```
process shift_event =
( ? event x, tick;
  ! event y;)
(| instant_x := event x default not tick
 | shift_instant_x := instant_x $1 init false
 | y := when shift_instant_x
 |)
where boolean instant_x, shift_instant_x;
end;
```

Here, the shifted event `y` is present at the shifted clock `shift_instant_x`.

**Additional control events:** Associated with each variable $X$ are some control events, signalling e.g. changes of values, which can be featured in triggers (see section 5.2.2). For each of them $e$, its definition can be added to the process `shift` and its profile as signal `e_`$e$; the `shift` process already performs a memorization of value, this is why it is the appropriate place to define these additional events. However, given that effects of actions can be sensed only at the following step, these events `e_`$e$ have to be shifted, using `shift_event`. In the particular case of control events for an event $E$, things are simpler: value changes have no meaning, and `rd_`$E$ and `wr_`$E$ can be defined directly as the `shift_event` of respectively `read_data_`$E$ and `written_data_`$E$; therefore the interface of `shift_event` does not need to be changed. Hence, for a variable $X$ and control event $e$, we will have:

```
    (X_current, e_e) := shift(X_next, tick)
 |  e := shift_event(e_e, tick)
```

which defines the value of $X$ in the current step ($X_{current}$), while accepting the value to be shifted to the next step ($X_{next}$); this also defines each control event $e$ by shifting the intermediary one produced unde the name `e_`$e$ by the `shift` of the variable. Definitions of the various control events are as follows:

- accesses to variables:
  - `rd_`$X$ is emitted when $X$ is read by action `read_data(`$X$`)`: the process `shift` needs to have an input `read_data_x` (in the instanciation for variable $X$, it is linked with signal `read_data_`$X$) and an output `e_rd_x`

(linked with `e_rd_`$X$, the `shift_event` of which will define `rd_`$X$). The additional equation is:

```
e_rd_x := read_data_x
```

- `wr_`$X$ is emitted when $X$ is written by action `write_data(`$X$`)` or by an assignment (i.e. union of clocks of contributed values or presence of a new value `x`): the process `shift` needs to have an input `written_data_x` (linked with `written_data_`$X$) and an output `e_wr_x` (linked with `e_wr_`$X$, the `shift_event` of which will define `wr_`$X$). The additional equation is:

```
e_wr_x := written_data_x default event x
```

- `ch_x` is emitted when `x` changes value: the process `shift` needs to have an output `e_ch_x` (linked with `e_ch_`$X$, the `shift_event` of which will define `ch_`$X$). The additional equation is:

```
  e_ch_x := when not (shift_value_x = value_x)
 when instant_x when ( (event x)$1 init false )
```

  Regarding the case of the first value received by $X$, a choice has to be made w.r.t. whether it constitutes a change or not; given that there is no initialisation of variables on Statecharts, it seems preferrable to consider that it is not a change. Hence, the event of the first occurrence of `x` must be ignored. This is the motivation for the last under-sampling appearing in the equation, as only the first occurrence of the delayed event will carry value `false`.

- value changes for Booleans:

  - `tr_`$C$ is emitted when the condition becomes `true`: the process `shift` needs to have an output `e_tr_x` (linked with `e_tr_`$C$, the `shift_event` of which will define `tr_`$X$). The additional equation is:

```
  e_tr_x := when value_x when not shift_value_x
 when instant_x when ( (event x)$1 init false )
```

    with the same under-sampling as before, regarding the first occurrence of $C$.

  - `fs_`$C$ is emitted when the condition becomes `false`: the process `shift` needs to have an output `e_fs_x` (linked with `e_fs_`$C$, the `shift_event` of which will define `fs_`$X$). The additional equation is:

```
  fs_x := when (not value_x) when shift_value_x
 when instant_x when ( (event x)$1 init false )
```

with the same under-sampling as before, regarding the first occurrence of $C$.

# 5 Translation from Statecharts to Signal

This section introduces the general translation of the main Statecharts features into Signal, by illustrating them with an example before giving the translation scheme.

## 5.1 Or-states and And-states

### 5.1.1 Example

**Or-states.** For each Statecharts component, a box process as defined above is created. The structural hierarchy of the Statecharts design is preserved through the hierarchy of the Signal processes. For the example of figure 4 we define a configuration signal giving the next configuration (`nc`) of the Statechart:

```
  t1 := transition {Idle, Running} (c, e)
| t2 := transition {Running, Idle} (c, f)
| control ^= ^0
| localclock := tick
| (c,nc) := nextstate {Idle} (localclock, t1 default t2, control)
```

This process, corresponding to the top level, will compute its internal configuration (values `Running` or `Idle`) at the local clock, which is, at the top level, the clock `tick` of the StateMate step.
The signal `t1` corresponds to the transition from `Idle` to `Running` and `t2` to the transition from `Running` to `Idle`. They are of enumerated type and get the value of the target of the transition when the corresponding transition is enabled. In case several transitions are enabled, the `default` between them will make a deterministic choice. The handling of non-determinism is described in section 7, where Signal is used to build an explicit representation of the possible cases. The transition taken is then fed into the process `nextstate`.
The `control` input to `nextstate` is the null clock `^0` because there is no explicit entering or exiting of the top-level. The parameter `Idle` of the subprocess `nextstate` is given because `Idle` is the default entrance state of the whole Statechart. To summarize, at the clock of the step, `t1` and `t2` are computed from the current value of the configuration signal `c` and the result is used in `nextstate` to compute the next configuration `nc`.

**History and deep history.** After the description of the top level of the design illustrated in figure 4, we refine the state `Running` as decomposed into sub-states `sub_running_up` and `sub_running_down`. Both are **or**-states but the main difference between them is the way they are entered: through deep-history for `sub_running_up` and through the default connector for `sub_running_down`. When the event `f` is generated it preempts the sub-automata of `running` and the last active state of `sub_running_up` will be reestablished whenever event `e` occurs. In Signal, the `$` operator is related to the last present value of a signal. Therefore, keeping the value of the last active state in this way deals with deep-history in the translation. Indeed, the suspension of the sub-process is achieved by the absence of the configuration signal for `sub_running_up` between `f` and `e`. When re-entering `Running`, the clock of the configuration signal is present again, and the delayed signal encoding it takes its values from where it was suspended.
The situation for the default entrance behavior, e.g. `sub_running_down`, is more complicated, because the configuration has to be reinitialized to `S3` when the transition from `Idle` to `Running` is taken (`t1`). In order to achieve this, the input signal `control` of the process encoding `running` is set to `Start when event t1`.
The state `Running` of the top level is now refined as being the process `running`. Its interface is built according to the reactive box scheme introduced in section 4.1. Its single input variable is `a` since `e` and `f` are not used inside. Its outputs are `b` and `d`. The clock of the configuration variables refining `running` is defined as follows:

```
localclock_running := when (nc=Running)
```

It is defined by the instants when the next configuration is in the state `Running`. This way, at the instant of entering a state, its sub-state configuration variable is present, which is needed in case it has to be re-initialized. On the other hand, the sub-state variable is not present at the instant when the state is exited. This down-sampling of the clock of the configuration variable `nc` into subclocks according to its value is the way the clock hierarchy of the configuration signals is built. The clock `localclock` of the sub-states is less frequent than the clock of the local `localclock`.
The `subcontrol_running` signal is used to reinitialize the subprocess to its default configuration at the instants of entrance. In the case of the example, the sub-node `Running` starts again when transition `t1` is taken, hence:

```
subcontrol_running := Start when event t1
```

We choose to reset an **or**-state to its default configuration at the instants of entrance and not at the instants of exit because the semantics offers the possibility to execute

actions upon entering. Resetting when exiting like in [15] would execute actions at the wrong instants according to the Statemate semantics.
Putting all these informations together gives the parameters of the box `running`:

```
d := running(tick, localclock_running, subcontrol_running, a, b)
```

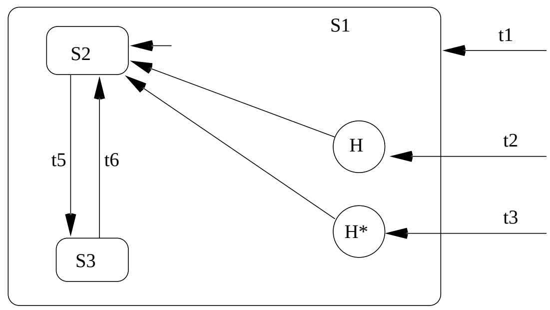


Figure 10: *Three ways to enter a state*

More generally, three ways are used to enter a state (see figure 10): Normal (t1), History (t2), Deep-History (t3). Depending on the entry chosen, the configuration signal of state `S1` must be reset and possibly the configuration signals of the sub-states `S2, S3`. The table shown on figure 11 gives the configuration variables that have to be reset and the value of the enumerated signal `control`.

| | | Reset S1 ? | Reset S2, S3 ? | Control Signal |
|---|---|---|---|---|
| t1 | Normal | yes | yes | Start |
| t2 | H | no | yes | LocalResume |
| t3 | H* | no | no | Resume |

Figure 11: *Which configuration variables to reset ?*

**And-states.** From the top level of the automaton we would like to refine the `Running` state into the **and** composition of two **or**-states (`sub_running_up` and `sub_running_down`). The processes `sub_running_up` and `sub_running_down` detailed further can be used to describe the `Running` state of figure 4 by just using the composition of Signal:

```
process running =
( ? event tick, localclock; tcontrol control; event a;
  ! event d; )
(| b := sub_running_up   (tick, localclock, control, a)
 | d := sub_running_down (tick, localclock, control, b)
 |)
where event b;
end;
```

**Sub-states.** We obtain the signal equations for the translation of `sub_running_up` using the **or**-state translation scheme except for a few difference with the top-level example given above. The difference is in the control of sub-nodes and their transitions. The clock of sub-state variables is defined in order to have an instant where re-initialization can be performed. However, in StateMate, transitions can not be taken at the instants when entering or exiting the corresponding or-node. The latter is given by the signal `control`, as seen earlier. Hence, the configuration input conditioning them has to be restricted to instants excluding the presence of `control`. This is done by defining `c_t`, which is given as the correct under-sampling of the configuration for transitions.
In fact, the same could be applied for the top-level, with the specificity that at that level `control^=^0`: i.e. there are no instants to exclude; therefore, the presentation could simplified by not mentioning the question.
The process `sub_running_up` encoding the corresponding state is as follows:

```
 process  sub_running_up =
       (  ?  event tick, localclock; tcontrol control; event a;
          !  event b; )
     (|   t3 := transition {S1, S2} (c_t, a)
      |   t4 := transition {S2, S1} (c_t, a)
      |   c_t := c when ( (not event control) default localclock)
      |   (c,nc) := nextstate {S1} (localclock, t3 default t4, ^0)
      |   b := ... (see section 5.2.3)
      |)
 where state t3, t4;
    end;
```

Equations for state `sub_running_down` are looking very similar.

### 5.1.2 Instantaneous States

An instantaneous state may be simultaneously entered and exited (in the same instant). Figure 12 provides an example where states `n1` and `n2` are instantaneous. Some semantics call these states: **condition, selection, junction, joint, fork** connectors, depending on the number of transitions entering and leaving the connector and if they apply to **and** states or not.

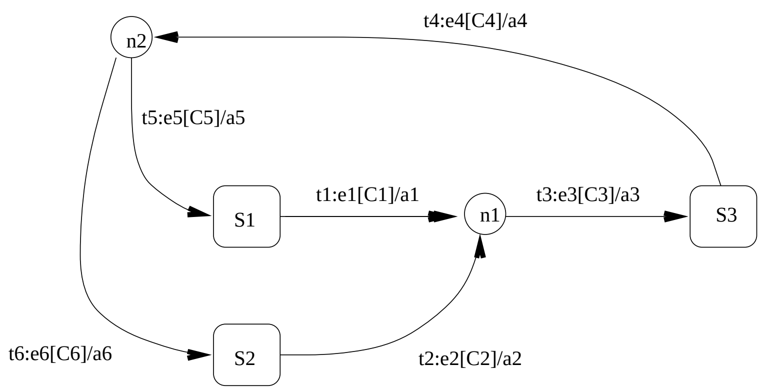


Figure 12: *A Statechart containing instantaneous states n1 and n2*

In some Statecharts semantics (Statemate for instance [11]), instantaneous states exist ,and we could handle them in the translation as shown below for the example of figure 12:

```
  t1 := transition {S1,n1} (c_t, C1 when e1)
| t2 := transition {S2,n1} (c_t, C2 when e2)
| t3 := transition {n1,S3} (t1 default t2, C3 when e3)
| t4 := transition {S3,n2} (c_t, C4 when e4)
| t5 := transition {n2,S1} (t4, C5 when e5)
| t6 := transition {n2,S2} (t4, C6 when e6)
| c_t := c when ( (not event control) default localclock)
| (c,nc) := nextstate {Sinit} (localclock, t3 default t5 default t6,
                                           control)
```

We use here the same process `transition` as the one used between ordinary states. The difference is in the configuration signal used in the `transition` process. When the origin of a transition is an instantaneous state (like the transition `t3`), ins-

tead of checking on the value of the configuration variable `c`, we use the transitions whose target is the considered instantaneous state. On the exemple figure 12: `t1 default t2`. Lastly, in the call of the process `nextstate`, only references to transitions whose target is a non-instantaneous state are given. In the exemple, `t3 default t5 default t6` gives the value of the `nextstate`.
It occurs however that the StateMate environment does perform an expansion of transitions going through instantaneous state into a set of transitions going between non-instantaneous states, combining triggers and actions accordingly. Hence the translation of this feature need not be studied specifically.

### 5.1.3 General translation scheme

The previous example introduced the general translation scheme given here.
Let $\mathtt{default}(\mathtt{a_1}, ..., \mathtt{a_n})$ defined as: $a_1$ `default` $\mathtt{a_2}$ `default` ... `default` $\mathtt{a_n}$.
Signals `tick`, `localclock` and `control` are considered to be contained in the inputs of the process encoding the state under translation, according to the reactive box structure described in Section 4.1.
Given a state:

- named *statename*,
- where OR(*statename*)=true if it is an or-state (otherwise, it is an and-state),
- with *nbss* sub-states named $Sub_i, i = 1..nbss$,
- with *nbtr* transitions between these sub-states, $i = 1..nbtr$, each from state $origin_i$ to state $target_i$ with $label_i$,
- with sub-state $subdefault$ as default entrance state (i.e. initial state),
- where H(*statename*)=true if the default arrow has the $H$ connector for target,
- where H*(*statename*)=true if the default arrow has the $H*$ connector for target,
- where $e_i1$,...,$e_ip$ are the indexes of the transitions with target (i.e., entering) $Sub_i$.
- where $x_i1$,...,$x_iq$ are the indexes of the transitions with origin (i.e., exiting) $Sub_i$.

- where $h_i1,...,h_ir$ are the indexes of the transitions with target the H connector in $Sub_i$.
- where $k_i1,...,k_is$ are the indexes of the transitions with target the H* connector in $Sub_i$,
- $input_i, output_i$ are the inputs and outputs of the process $Sub_i$ as defined in section 5.3,

The translation of this state in Signal is a process of the same name, made of the composition of the following equations:

- for each transition $t_i, i = 1..nbtr$:

  ```
  t_i := transition{origin_i, target_i}(c_t,α(label_i))
  ```

  where $\alpha(label_i)$ is the translation of the trigger and condition of the label of the transition, as described further (see Section 5.2.2).

- concerning state, in all cases:

  ```
  c_t := c when ( (not event control) default localclock)
  ```

  if H*(*statename*) or H(*statename*) then:

  ```
  (c,nc):=nextstate{subdefault}(localclock,
                      default(t_1,...,t_nbtr), ^0)
  ```

  else:

  ```
  (c,nc):=nextstate{subdefault}(localclock,
                default(t_1,...,t_nbtr), control)
  ```

- if H*(*statename*) then $\forall i = 1..nbss$:

  ```
  subcontrol_i := ^0
  ```

  else $\forall i = 1..nbss$:

  ```
  subcontrol_i :=  Start when (control=LocalResume)
         default  Start when (event default(t_ei1,...,t_eip))
         default  Stop when (event default(t_xi1,...,t_xiq))
         default  LocalResume when (event default(t_hi1,...,t_hir))
         default  Resume when (event default(t_ki1,...,t_kis))
         default  control
  ```

- if OR(*statename*) then $\forall i = 1..nbss$:

  `output_i := Sub_i(tick, when (nc=Sub_i), subcontrol_i, input_i)`

  else (if it is an And-node) $\forall i = 1..nbss$:

  `output_i := Sub_i(tick, localclock, control, input_i)`

## 5.2 Transition labels: triggers and actions

The general syntax of the label on a transition in Statechart is as follows:

$\langle label \rangle \rightarrow \langle Trigger \rangle \ / \ \langle Action \rangle$

The trigger as well as the action on a transition are making reference to the value of variables and events. The way they are handled in the Signal translation is presented next, before treating triggers, and then actions.

### 5.2.1 Variables

They are declared at the level of a state *statename*. They have to be managed in such a way that they comply with their definition:

- they are assigned their new value (if any)
- their value is carried to the next step coming from different possible actions

The scope of the Statechart variables is the chart where they are defined, as mentioned in section 3. In our translation, each chart is translated into a Signal process, itself decomposed into sub-processes. All the signals representing variables are given as inputs to all the sub-processes in order to obtain a broadcasting.
Given a state named *statename*, as before:

- where variables $a_1, ..., a_{nbvar}$ are declared locally,
- where variable $a_i$ has $a_{i_1}, ...a_{i_{nbcv}}$ contributed values,

The translation of this state in Signal features the following equations concerning variables:
$\forall i = 1..nbvar$:

`a_i := shift(default(a_i1, ...a_i_nbcv), tick)`

The variables is translated into an invocation of the process `shift`, the input of which is the merge of all contributed values; If we want to represent explicitely the

possibility of racing conditions, i.e., presence of two contributed values at the same instant, it would be possible to apply the techniques described in section 7. The process `shift` carries the value to the next step, which is given by the clock `tick`. Actually, a less frequent clock might be used, if the chart in question is sometimes deactivated.
In the translation, the signals carrying the contributed values have names derived from the variable name $a_i$ by adding a suffix $j$ to it: $a_{i_j}$. These names are used in the translation of the actions producing these values for $a_i$, which is described further. For this, we use $current(a)$ and $next(a)$, two functions delivering integers associated to variable $a$. These are used to have a counter associated to variable $a$. $current(a)$ gives the current value of the counter associated to $a$ and $next(a)$ the incremented value of this counter. The counters associated to each variable start at 1. These functions are used for Zeidt effect purposes[2].
Concerning the extension with control events mentioned in section 4.5, we follow the same scheme, i.e. for a variable $a_i$ (with $n1$ the number of contributing sources for `read_data_`$a_i$, and $n2$ the same for `written_data_`$a_i$):

```
read_data_ai := default(read_data_ai1, ...  read_data_ain1)
```

```
written_data_ai := default(written_data_ai1, ...  written_data_ain2)
```

#### 5.2.2 Triggers

**Syntax of triggers.** Triggers of transition label can be of the following form:

$$
\begin{array}{lll}
\langle Condition \rangle & \rightarrow & \langle Expression1 \rangle \langle Rel \rangle \langle Expression2 \rangle \\
& | & not \langle Condition \rangle \\
& | & \langle Condition1 \rangle and \langle Condition2 \rangle \\
& | & \langle Condition1 \rangle or \langle Condition2 \rangle \\
& | & \langle Variable \rangle
\end{array}
$$

$$\langle Rel \rangle \rightarrow \; = | <> | < | > | \leq | \geq$$

$$
\begin{array}{lll}
\langle Expression \rangle & \rightarrow & \langle Expression \rangle \langle Op \rangle \langle Expression \rangle \\
& | & \langle Variable \rangle \\
& | & \langle Number \rangle
\end{array}
$$

$$\langle Op \rangle \rightarrow + | - | * | /$$

[2] also called “effet de Bohr” in French.

$$
\begin{array}{rcl}
\langle Trigger \rangle & \rightarrow & \epsilon \\
& | & \langle EventName \rangle \\
& | & \langle Trigger \rangle [\langle Condition \rangle] \\
& | & not \langle Trigger \rangle \\
& | & \langle Trigger1 \rangle and \langle Trigger2 \rangle \\
& | & \langle Trigger1 \rangle or \langle Trigger2 \rangle \\
& | & in(\langle State \rangle) \\
& | & entered(\langle State \rangle) \\
& | & exited(\langle State \rangle) \\
& | & true(\langle Condition \rangle) \\
& | & false(\langle Condition \rangle) \\
& | & read(\langle Variable \rangle) \\
& | & written(\langle Variable \rangle) \\
& | & changed(\langle Variable \rangle)
\end{array}
$$

**General translation scheme.** Translating them in Signal amounts to evaluating the trigger event and condition parts, and feeding them as input to the `transition` process defined earlier. The translation function $\alpha$ delivers the Signal expression (of type `event`) translating the trigger of the transition label. It is defined as follows:

- in the reactions $\alpha$ handles the translation of the trigger only (see further function $\beta$ for actions):

  $\alpha(\langle Trigger \rangle / \langle Action \rangle) = \boxed{\alpha(\langle Trigger \rangle)}$

- expressions on triggers:

  - the empty trigger is satisfied at the global clock:
    $\alpha(\epsilon) = \boxed{\texttt{tick}}$
  - presence of an event $\langle EventName \rangle$:
    $\alpha(\langle EventName \rangle) = \boxed{\langle EventName \rangle}$
  - combined event and condition trigger:
    $\alpha(\langle Trigger \rangle \langle Condition \rangle) = \boxed{\alpha(\langle Trigger \rangle) \texttt{ when } \alpha(\langle Condition \rangle)}$
  - logical expressions on triggers:
    $\alpha(\text{not } \langle Trigger \rangle) = \boxed{\texttt{when not\_event(}\alpha(\langle Trigger \rangle)\texttt{, tick)}}$

    $\alpha(\langle Trigger1 \rangle \text{ and } \langle Trigger2 \rangle) = \boxed{\alpha(\langle Trigger1 \rangle) \texttt{ when } \alpha(\langle Trigger2 \rangle)}$

$$\alpha(\langle Trigger1\rangle \text{ or } \langle Trigger2\rangle) = \boxed{\alpha(\langle Trigger1\rangle) \texttt{ default } \alpha(\langle Trigger2\rangle)}$$

- dynamic triggers:
  - on variables: for a $\langle Variable\rangle$ named $X$:
    $\alpha(\text{read}(X)) = \boxed{\texttt{rd\_}X}$
    $\alpha(\text{written}(X)) = \boxed{\texttt{wr\_}X}$
    $\alpha(\text{changed}(X)) = \boxed{\texttt{ch\_}X}$
    where these events are produced in relation with the management of the variable $X$ (see section 4.5).
  - on conditions: for a $\langle Condition\rangle$ (which can be an expression) computed in a variable $C$ (which can be an intermediate variable for computing the expression):
    $\alpha(\text{true}(C)) = \boxed{\texttt{tr\_}C}$
    $\alpha(\text{false}(C)) = \boxed{\texttt{fs\_}C}$
    where these events are produced in relation with the management of the boolean variable $C$ (see section 4.5).
  - on states:
    $\alpha(\text{in}(S)) = \boxed{\texttt{in\_}S}$
    $\alpha(\text{entered}(S)) = \boxed{\texttt{en\_}S}$
    $\alpha(\text{exited}(S) = \boxed{\texttt{ex\_}S}$
    where these events are produced in relation with the management of the state $S$ (see section 4.4).

- expressions on conditions:

$$\alpha(\langle Expression1\rangle\langle Rel\rangle\langle Expression2\rangle) = \boxed{\langle Expression1\rangle\ \langle Rel\rangle\ \langle Expression2\rangle}$$

$$\alpha(not\langle Condition\rangle) = \boxed{\texttt{not }\ \alpha(\langle Condition\rangle)}$$

$$\alpha(\langle Condition1\rangle and\langle Condition2\rangle) = \boxed{\alpha(\langle Condition1\rangle) \texttt{ and } \alpha(\langle Condition2\rangle)}$$

$$\alpha(\langle Condition1\rangle or\langle Condition2\rangle) = \boxed{\alpha(\langle Condition1\rangle) \texttt{ or } \alpha(\langle Condition2\rangle)}$$

$$\alpha(\langle Expression\rangle\langle Op\rangle\langle Expression\rangle) = \boxed{\langle Expression\rangle\ \langle Op\rangle\ \langle Expression\rangle}$$

$$\alpha(\langle Variable \rangle) = \boxed{\langle Variable \rangle}$$

$$\alpha(\langle Number \rangle) = \boxed{\langle Number \rangle}$$

### 5.2.3 Actions

We present the translation scheme for a sub-set of the actions language of StateMate. Not covered yet are the notions of context variables (which can take several values within a step), loops (`for` or `while` loops) which would involve the definition of a microstep, ...

**The clock of actions.** Actions are activated when the transition is actually taken; this activation condition defines the clock of the actions.
Given a state named *statename*, as before, with *nbac* actions on transitions: $i = 1..nbac$, uniquely identified by function $a(i)$ (this is to make a difference with static reactions actions $i = 1..nbsr$ etc, see further), and $tr(a(i))$ giving the index in $1..nbtr$ of the transition $\mathtt{t}_{tr(a(i))}$ of which it is a label. For each action, we defined its clock by the following equation, with:

- event $\mathrm{t}_{tr(a(i))}$ is the event that the or-state is neither entering or exiting, and that the trigger event and condition of the transition are satisfied, and that the current state is the origin of the transition
- (nc=$target_{tr(a(i))}$) tell us that the transition is actually taken as the next state is its target; this is necessary in order to insure that this transition is the one that was actually chosen in case several were enabled (see section 5.1 and, for the handling of non-determinism: section 7)

i.e., $\forall i = 1..nbac$ :

$$\boxed{\texttt{clockaction}_{a(i)} \texttt{ := event t}_{tr(a(i))} \texttt{ when (nc=}target_{tr(a(i))}\texttt{)}}$$

**Syntax of actions.** Transition label actions can be of the following form:

$$
\begin{array}{lll}
\langle Action \rangle & \rightarrow & \epsilon \\
 & | & \langle EventName \rangle \\
 & | & \langle Variable \rangle := \langle Expression \rangle \\
 & | & read\_data(X) \\
 & | & write\_data(X) \\
 & | & make\_true(C) \\
 & | & make\_false(C) \\
 & | & when \langle Event \rangle then \langle Action1 \rangle [else \langle Action2 \rangle] endwhen \\
 & | & if \langle Condition \rangle then \langle Action1 \rangle [else \langle Action2 \rangle] endif \\
 & & \langle Action1 \rangle ; \langle Action2 \rangle
\end{array}
$$

**General translation scheme.** The translation of actions amounts to generating equations for each action, $\forall i = 1..nbac$:

$\boxed{\beta(\langle Action \rangle, \mathtt{clockaction}_{a(i)})}$

where:

- in a reaction, $\beta$ handles the translation of the actions only:
  $\beta(\langle Trigger \rangle / \langle Action \rangle, Clk) = \boxed{\beta(\langle Action \rangle, Clk)}$
- basic actions:
  - empty action: $\beta(\epsilon, Clk)$ is void.
  - event emission: if the $\langle EventName \rangle$ is $a$:
    $\beta(\langle EventName \rangle, Clk) = \boxed{\mathsf{a}_{next(a)} \ \mathtt{:=}\ Clk}$
    where $next(a)$ is the function introduced in section 5.2.1 for the purpose of naming signals carrying contributing values for $a$.
  - variable assignment:
    $\beta(\langle Variable \rangle := \langle Expression \rangle, Clk) =$
    $\boxed{\mathsf{a}_{next(a)} \ \mathtt{:=}\ \alpha(\langle Expression \rangle)\ \mathtt{when}\ Clk}$
    where $\mathsf{a}$ is the name of the variable, and $next(a)$ is used as explained in section 5.2.1 in order to manage names of signals carrying contributed values.
  - variable access:
    $\beta(\mathsf{read\_data}(X), Clk) = \boxed{\mathtt{read\_data\_}X_{next(\mathtt{read\_data\_}X)} \ \mathtt{:=}\ Clk}$
    $\beta(\mathsf{write\_data}(X), Clk) = \boxed{\mathtt{written\_data\_}X_{next(\mathtt{written\_data\_}X)} \ \mathtt{:=}\ Clk}$

- action expressions:
  - $\beta$(when $\langle Event\rangle$ then $\langle Action1\rangle$ [else $\langle Action2\rangle$] end when,$Clk$) =

```
     β(⟨Action1⟩, Clk when ⟨Event⟩)
[|   β(⟨Action2⟩, Clk when not_event(⟨Event⟩,tick))]
```

  - $\beta$(if $\langle Condition\rangle$ then $\langle Action1\rangle$ [else $\langle Action2\rangle$] end if,$Clk$) =

```
     β(⟨Action1⟩, Clk when α(⟨Condition⟩))
[|   β(⟨Action2⟩, Clk when not α(⟨Condition⟩))]
```

  - $\beta(\langle Action1\rangle;\langle Action2\rangle, Clk)$ =

```
(| β(⟨Action1⟩, Clk) | β(⟨Action2⟩, Clk) |)
```

**Static reactions** The labels attached to a state are called static reactions. They have the same syntax as labels associated with transitions. The general static reaction construct makes it possible to define the reaction of the system to a trigger when a particular state is active. As long as the state is active, except when entering or exiting, the trigger part of the static reaction is evaluated and the action part possibly carried out. The fact that the state is active can be constructed from the clock `localclock` and the signal `control`, both featured as an input in the interface of the Signal process encoding the state in question, as described in section refreactivebox. In particular, in the case of an empty trigger (i.e., the left part of the "/" is empty), actions are to be carried out at each step when the system is in the state in question. Performing the action is done whenever the trigger part of the static reaction is enabled and the state associated with the static reaction is active.
For static reactions $SR_i, i = 1..nbsr$:

```
    clockaction_sr(i) :=
        α(SR_i) when ( (not event control) default localclock)
|   β(SR_i,clockaction_sr(i))
```

where $sr(i)$ uniquely identifies static reaction $i$.
The possibility exists in Statemate to carry out actions upon entering or exiting a particular state. This is done by associating special static reactions with the state $S$, triggered by `en_`$S$ and `ex_`$S$ events. Firing these special static reactions in the translation is done using signal `control` from the interface of the state being translated (as described in section 4.4).

### 5.3 Profile

Given a state:

- named $N$,
- with *declvar* the set of variables declared in this node,
- with sub-states $N_i, i = 1..nbsn$,
- with *actionvar* the set of variables modified in an action (whether associated with a transition label or a static reaction) of this node. If an action contains `read_data(x)` or `write_data(x)` then `rd_X` or `wr_X` is added to *actionvar*.
- with *transitionvar* the set of variables used (all except *actionvar*) in a trigger or an action on a transition label or a static-reaction. of this state.

$local(N) = declvar$

$input(N) =$
$\{x/x \in transitionvar \backslash local(N)\} \cup \{x/\exists i \in (1..nbsn), x \in input(N_i) \backslash local(N)\}$

$outputsc(N) =$
$\{x/x \in actionvar \backslash local(N)\} \cup \{x/\exists i \in (1..nbsn), x \in outputsc(N_i) \backslash local(N)\}$

$output(N) = \bigcup_{i \in (1..nbsn)} local(N_i) \cup outputsc(N)$

Standard inputs for every node are, as described in the reactive box of section 4.1 (and in this order in the interface): `tick, localclock, control`.

# 6 Translation from Activitycharts to Signal

## 6.1 Status of an activity

The hierarchical structure of activities is translated by following the hierarchical structure and generating one Signal process for each Activity, following the reactive box principle described in section 4.1, with control signals `tick, localclock, control` in the inputs of the interface.

Each activity can be controlled (started, stopped, suspended, resumed, and sensed for status) in response to events emitted by a control activity, itself defined by a Statechart. The status of an activity follows a behavior illustrated by a Statechart in Figure 13, which shows how an activity $A$ commutes between the states `active`, `hanging` and `inactive`, according to events `st_`$A$, `sp_`$A$, `sd_`$A$, `rs_`$A$. The suspension and resuming can occur only from the `active` status; if stopped by `st_`$A$ while in the `hanging` status, an activity goes in status `inactive`.

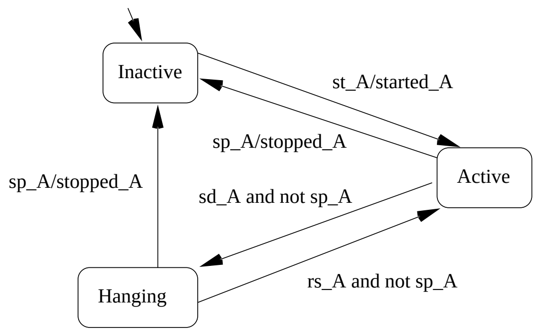


Figure 13: States of an activity.

The translation of each activity hence involves the generation of a Signal process encoding this behavior: only in its state `active` will the activation clock be transmitted to its actions and/or subactivities, thereby implementing the control of activities. Given an Activity:

- named $A$,
- with sub-activities $Subact_i, i = 1..nbac$
- with mini-specs $MS_i, i = 1..nbms$

the translation follows a scheme similar to that for an or-state (see section 5.1); one difference is that activation of sub-activities can occur at the same instant as the activation of their parent activity: hence it is `c_`$A$ which is given as input to the `transition` process instances. The management of the status is as follows:

```
   t_1:=transition{inactive,active}(c_A,st_A)
|  t_2:=transition{active,inactive}(c_A,sp_A)
|  t_3:=transition{hanging,active}(c_A,
                         rs_A when not_event(sp_A,tick))
|  t_4:=transition{active,hanging}(c_A,
                         sd_A when not_event(sp_A,tick))
|  t_5:=transition{hanging,inactive}(c_A,sp_A)
|  (c_A,nc_A) := nextstate{inactive}(localclock,
       default(t_1,t_2,t_3,t_4,t_5),control)

|  started_A := shift(event t_1,tick) when (nc_A=active)
|  stopped_A := shift(event default(t_2,t_5), tick)
                                    when (nc_A=inactive)
```

where events `st_A, sp_A, rs_A, sd_A` can be received from a Statecharts defining control activities, and events `started_A` and `stopped_A` are produced for them.
For sub-activities, the translation is as follows, with $input_i$ and $output_i$ representing respectively the lists of inputs and outputs of the sub-activity $i$: $\forall i = 1..nbac$:

```
   localclock_i := when nc_A=active
    subcontrol_i :=  Start when st_A
|          default  Stop when sp_A
           default  Resume when rs_A
|  output_i := SubA_i(tick, localclock_i, subcontrol_i, input_i)
```

where they are transmitted a clock which is a sub-sampling, in the `active` status, of the local clock of activity $A$.
For mini-specs associated with that activity, we have, for each $MS_i$ of them, $\forall i = 1..nbms$:

```
   clockaction_ms(i) :=
        α(MS_i) when ( (not event control) default localclock)
|  β(MS_i,clockaction_ms(i))
```

where $ms(i)$ is an absolute identification of mini-spec number $i$, and function $\alpha$ defined for transition labels is reused.

## 6.2 Triggers and actions related to activities

A number of triggers and actions related to activities are featured in the language. Therefore we extend the definitions of functions $\alpha$ and $\beta$ in order to encompass them.

### 6.2.1 Triggers on activivites

For each of these dynamic triggers of event or logical type, we give its definition:

- active($A$), ac($A$): activity A is in the active state.

  $\alpha(\text{active}(A)) = \boxed{\texttt{ac\_}A}$

- hanging($A$), hg($A$): activity A is in the suspended state.

  $\alpha(\text{hanging}(A)) = \boxed{\texttt{hg\_}A}$

- started(A), st(A): activity A is started. This event is issued as a result of action `start(`$A$`)`.

  $\alpha(\text{started}(A)) = \boxed{\texttt{st\_}A}$

- stopped(A), sp(A): activity A is stopped. It is issued as a result of action `stop(`$A$`)`.

  $\alpha(\text{stopped}(A)) = \boxed{\texttt{sp\_}A}$

where these events are produced in relation with the definition of the status of activity $A$:

$\boxed{\texttt{ac\_}A\ \texttt{:= when (nc\_}A\ \texttt{= active)}}$

$\boxed{\texttt{hg\_}A\ \texttt{:= when (nc\_}A\ \texttt{= hanging)}}$

### 6.2.2 Actions on activities

The actions related to activities concern the control of their status, i.e. starting, stopping, suspending (putting in `hanging` status) or resuming an activity:

- start($A$) puts an activity $A$ in status `active`:

  $\beta(\text{start}(A), Clk) = \boxed{\texttt{st\_}A_{next(\texttt{st\_}A)}\ \texttt{:=}\ Clk}$

  where $next(\texttt{st\_}A)$ updates the counter of contributing values to variable $\texttt{st\_}A$.

- stop($A$) puts an activity $A$ in status `inactive`:

  $\beta(\text{stop}(A), Clk) = \boxed{\texttt{sp\_}A_{next(\texttt{sp\_}A)}\ \texttt{:=}\ Clk}$

- suspend($A$) puts an activity $A$ in status `hanging`:

  $\beta(\text{suspend}(A), Clk) = \boxed{\texttt{sd\_}A_{next(\texttt{sd\_}A)}\ \texttt{:=}\ Clk}$

- resume($A$) puts an activity $A$ in status `active`:

  $\beta(\text{resume}(A), Clk) =$ $\boxed{\texttt{rs\_}A_{next(\texttt{rs\_}A)} \texttt{ := } Clk}$

### 6.3 Profile

The signals related to activities also contribute to the computation of profiles: w.r.t. the one described in section 5.3, if an action on a transition, in a static reaction or in a mini-spec contains start($A$), stop($A$), suspend($A$) or resume($A$) then st_$A$, sp_$A$, rs_$A$ or sd_$A$ is added to *actionvar*. Also, variables used in a mini-spec (except if already in *actionvar*) are to be added to *transitionvar*. Events ac_$A$ and hg_$A$ are featured in the outputs of the activity $A$.

## 7 Modelling non-determinism

This section deals with the fact that it is possible to use Signal to model non-determinism, in the sense that it can be used to define processes with a set of possible behaviors. Hence, this can be applied to the modeling of non-determinism in Statecharts.

### 7.1 Conflicting transitions

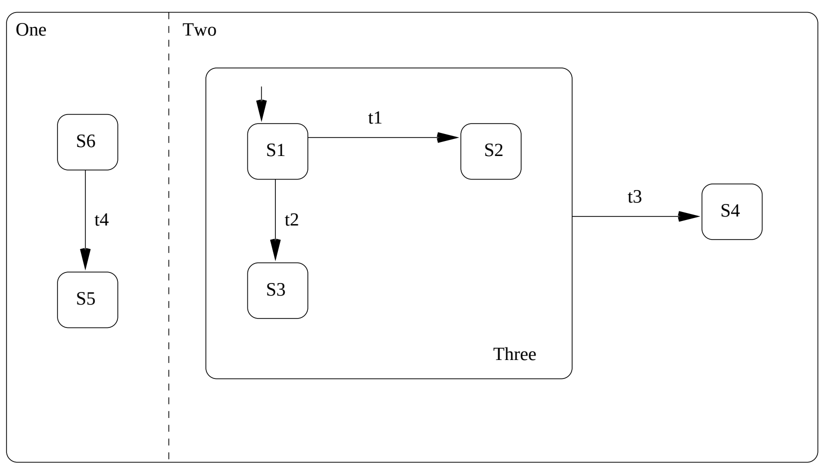


Figure 14: *conflict*

**Example.** For the Statechart of the figure 14, if the configuration is (`S1, S6`) and `t1`, `t2`, `t3` and `t4` are enabled transitions, the maximal non conflicting sets of transitions are:

- {t4, t1}
- {t4, t2}
- {t4, t3}

Here, `t3` has higher priority over `t1` and `t2` (Statemate semantics) and `t3` is chosen. This is preserved in the Signal translation since the clock of substates depends on the transitions of the higher state, where the triggers to the transition labels are computed before. `t3` is chosen as an enabled transition and then, the process `Three` will not sense any inputs because the active state is `S4`.
When `t1` and `t2` are enabled but not `t3`, we need to choose one to fire. There is a non-deterministic conflict and any one of the two transitions could be chosen. If it is preferred to encode an arbitrary choice, as the Magnum simulator can do, then in the example, to take `t1` preferably to `t2` in all the cases is what the translation of previous sections does:

```
| (c,nc) := nextstate {S1} (localclock, t1 default t2, ^0)
                                       %^^^^^^^^^^^^^%
```

This the translation scheme that was developped in this paper. In this section, we describe how it is possible to represent non-deterministic choices explicitely in Signal.

**Non-conflicting sets.** Here we want to deal with situations where more than one transition are enabled at the same instant. Following [11]:

- Two enabled transitions are in *conflict* if there is some common state that would be exited if any one of them were to be taken.
- A set of transitions is *non conflicting* if no two transitions in the set are in conflict.
- Being *maximal* for a non-conflicting set of transitions means that each enabled transition not included in the set is in conflict with at least one transition that is included in the set. Otherwise, this transition may be added to the set.

At each step, Statemate fires a *maximal non-conflicting set* of transitions. When there is more than one such a set enabled, a non-deterministic choice is performed. The maximal is reached in the Signal translation because whenever one transition is enabled in an **Or**-state, in the corresponding call to the `nextstate` process, the `default` on the transitions will choose one. Not taking a maximal non-conflicting set in the Signal translation would be to have an enabled transition with its associated `t`$i$ signal present. The `default` on the list of `t`$i$ signals is hence present and at least one transition is taken. The translation shown in the previous sections does a `default` on the $t_i$ signals; this way it chooses arbitrarily between the non-deterministic possibility.

**Representing non-determinism explicitely.** The non-determinism may be represented explicitely by adding a boolean `K` to the Signal equation choosing between the different enabled transitions:

```
| (c,nc) := nextstate {S1}(localclock, t, ^0)
| t := t1 when (K default true) default t2
| ^t1 when ^t2 ^= ^t1 when ^t2 when ^K)
```

The equation on clocks featuring a `^=` is used to avoid situations where `t1` and `t2` are both present and `K` is absent because these situations could lead to the choice of transition `t1` when no particular (explicit) decision has been made to remove the non determinism. This way, when `K` is true, `t1` is chosen and when `K` is false, `t2` is chosen. If the signal `K` is not present, the `t1 when (K default true) default t2` rewrites into `t1 default t2` that is the behavior of a deterministic process.
At this stage, we have an exact model of the non-deterministic behavior of this part of the specification. What it can then happen in order to handle the non-determinism is the following:

- this process may be composed with other processes that make `K` useless (e.g. composed with a process where `t1` and `t2` are exclusive),
- the signal `K` could be explicitly given as an input of the process and the environment may choose between `t1` and `t2`, hence moving the resolution of the non-determinism to the environment.

**Firing the right actions.** There are cases for which the above scheme is not sufficient to uniquely determine which actions are actually executed as was proposed in section 5.2.3. Figure 15 illustrates this with an example, where, if `e1[c1]` and `e2[c2]` are not exclusive, then it is possible that both transitions are enabled, with the same target state. Hence, the latter is not a sufficient discriminating criterion, notably when it has to be decided whether action `a1` or action `a2` is executed.

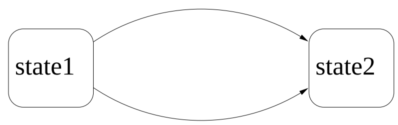


Figure 15: *Conflicting transitions in Statecharts*

This requires additional information identifying the transitions: if each of them is given a name or index $\texttt{i}_i$, an equation similar to the previous one has to produce the identity of the one actually chosen:

```
i := i1 when (K default true) default i2
```

The definition of the clock of the actions associated to a transition then becomes:
$\texttt{clockaction}_{a(i)}$ `:= when (i =` $\texttt{i}_i$`)`

### 7.2 Racing

An other kind of non-determinism is possible in Statecharts, through the variable assignment: two actions in two parallel components occurring at the same moment and giving different values to the same variable. Sometimes called "racing", this non-determinism could be handled the same way the non-determinism on transitions is handled: Introducing a boolean `K` choosing between the different values of the variable. The different ways to handle non-determinism shown above are also valid for racing.

## 8 Conclusion and perspectives

We have proposed here a way to translate the essential features of Statecharts and Activitycharts into Signal. This translation gives clocks to every part of a Statechart (states, transitions, actions). It keeps the structural and hierarchical informations through the translation to privilege the traceability from specification to the generated code. It is expected that this will have consequences on the compilation

process and the optimization algorithms offered by the Signal/DC+ environment, in the perspective of producing efficient code, for possibly distributed executaion architectures, from Statecharts specifications, using the clock calculus and the BDD's techniques of the Signal compiler. Non-determinism may be modeled and handled through boolean adjunction. Verification of the behavior is possible using the tools based on the synchronous technology. Real-time properties of a Statechart could be checked through timing analysis of a Signal program. The main contribution of this work is to get an access to the already existing Signal tools from a Statechart design. This translation provides a support for co-execution of co-simulation of Signal and the languages of StateMate. Interoperability between Signal and Statecharts is possible by composing the resulting Signal process with any Signal context. The interaction between the two parts is then managed by the synchronous composition.
An implementation of such a translation is being done in C++ in the context of the SAfety CRitical Embedded Systems (SACRES) European Project [21]. The translation is done in a variant of Signal called DC+ [18] which is a common format of the synchronous languages. The Statemate tool from *i-Logix* is used to draw the Statechart, then the automatic translator uses an API of the Statemate tools in order to extract the needed informations of the Statechart design and generate the DC+.
Perspectives presently worked upon concern other features of the Statemate languages. For instance, variables can have different data-types and scopes. In the actions, mechanisms for timeouts and scheduled events could be encoded as counters on the number of steps. Context variables are special variables that can take several different values within one step: they are used in connection with loops in the actions.
The proof that the translation is correct from a behavioral point of view is now needed in the context of safety critical systems. Such a proof may be in terms of equality of the traces of the initial Statechart and the target Signal program, based on the semantics of the languages [19]. This semantics defines Signal, and its derived format DC+, as well as the languages of StateMate, in terms of fair Synchronous Transition Systems (fSTS). This provides a common basis for comparing the translation to the source, and establishing the correctness.

## References

[1] T. Pascalin Amagbegnon, Loïc Besnard, and Paul Le Guernic. Implementation of the data-flow synchronous language SIGNAL. In *Proceedings of the ACM Symp.*

*on Programming Languages Design and Implementation, PLILP'95*, pages 163–173. ACM, 1995.

[2] J.-R. Beauvais, R. Houdebine, P. Le Guernic, E. Rutten, and T. Gautier. A translation of STATECHARTS into SIGNAL. In *Proceedings of the International Conference on Application of Concurrency to System Design (CSD'98), Aizu-Wakamatsu, Japan*, March 1998.

[3] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[4] G. Berry. The constructive semantics of pure ESTEREL. Book in preparation, current version 2.0, http://zenon.inria.fr/meije/esterel.

[5] Gerard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[6] Thierry Gautier, Paul Le Guernic, and Olivier Maffeïs. For a new real-time methodology. Research Report 2364, INRIA, October 1994. `http://www.inria.fr/RRRT/RR-2364.html`.

[7] Alvery Grazebrook. Sacres - formalism for real projects. In F. Redmill and T. Anderson, editors, *Safer Systems*, London, 1997. Springer-Verlag.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.

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

[10] David Harel and Amnon Naamad. The languages of Statemate. *i-Logix Inc*, January 1991.

[11] David Harel and Amnon Naamad. The Statemate semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[12] A. Kountouris and P. Le Guernic. Profiling of Signal programs and its application in the timing evaluation of design implementations. In *Proceed. of the IEE on HW-SW Cosynthesis for Reconfigurable Systems*, pages 6/1–6/9, HP Labs Bristol UK, Feb. 1996.

[13] M. von der Beeck. A Comparison of Statecharts Variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148, Lübeck, Germany, September 1994. Springer-Verlag.

[14] Olivier Maffeïs and Axel Poigné. Synchronous automata for reactive, real-time or embedded systems. Technical report, GMD, Jan. 1996. no 967.

[15] F. Maraninchi and N. Halbwachs. Compiling ARGOS into Boolean equations. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, Uppsala, Sweden*, volume 1135 of *Lecture Notes in Computer Science*, pages 72–90. Springer-Verlag, September 1996.

[16] Eric Rutten and Paul Le Guernic. Sequencing data flow tasks in signal. In *In Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems, Orlando, Florida*, June 1994. `http://www.cs.umd.edu/users/pugh/sigplan_realtime_workshop/lct-rts94/`.

[17] Eric Rutten and Florent Martinez. Signal GTi, implementing task preemption and time intervals in the synchronous data flow language Signal. In IEEE Computer Society Press, editor, *Seventh Euromicro Workshop on Real-Time Systems*, pages 176–183, June 1995.

[18] SACRES. The common format of synchronous languages - the declarative code DC+ version 1.4. Technical report, EP 20897 Project, November 1997.

[19] SACRES. The semantic foundations of SACRES. Technical report, EP 20897 Project, March 1997.

[20] SYNCHRON. The common format of synchronous languages - the declarative code DC version 1.0. Technical report, C2A-SYNCHRON project, October 1995.

[21] SACRES. Deliverable report I1.1.A: Statemate translation to DC+. Technical report, EP 20897 Project, 1998 (to appear).

# A An example of translation of Statemate into Signal

## A.1 An example of Statemate specification

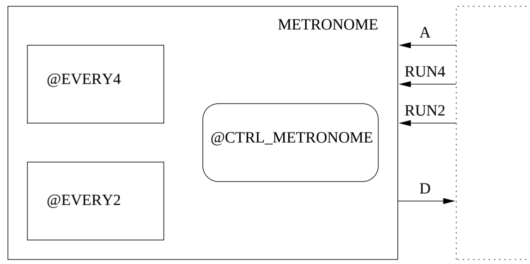


Figure 16: Example: the top-level activity.

The example is built as an activity called METRONOME, as shown in Figure 16. Its inputs are:

- A: an event to be counted,
- RUN2: an event commanding the counting modulo 2
- RUN4: an event commanding the counting modulo 4

The output is:

- D, an event occurring once every 2 or every 4 occurrences of A, depending on the mode the counter is in.

The activity METRONOME is decomposed into sub-activities EVERY2 and EVERY4, controlled by the statechart CTRL_METRONOME.

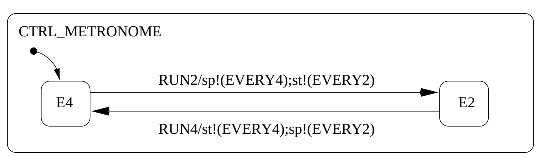


Figure 17: Example: the metronome controller.

The control statechart CTRL_METRONOME is shown in Figure 17. It alternates between the two states E4 and E2. The initial state is E4. There are two transitions:

- one from E4 to E2, which can be taken when event RUN2 is present. The action associated to the transition consists of stopping sub-activity EVERY4 and starting sub-activity EVERY2.
- the other one from E2 to E4, which can be taken when event RUN4 is present. The action associated to the transition consists of stopping sub-activity EVERY2 and starting sub-activity EVERY4.

Activity EVERY2 is defined by the statechart shown in Figure 18. It alternates between two states S3 and S4, upon reception of event A, and emits an event D every second A.

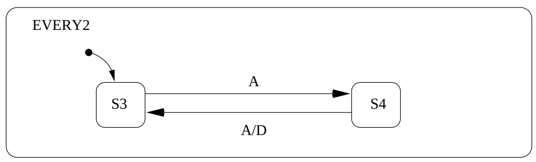


Figure 18: Example: the counter modulo 2.

Activity EVERY4 is defined by the statechart shown in Figure 19. It is an AND-node, composing two sub-statecharts similar to the one in EVERY2. They are linked by the event B, in such a way that the sub-statechart UP emits a B every two As, and the sub-statechart DOWN emits a D every two Bs. Hence EVERY4 emits a D every four As.

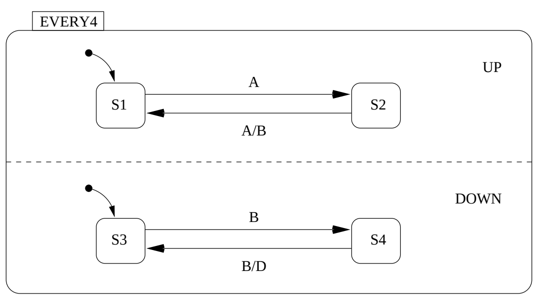


Figure 19: Example: the counter modulo 4.

## A.2 Its translation in Signal

Here follows the complete translation in Signal of the example. It shows the hierarchical structure of the encoding, following the structure of the original Statemate specification. This translation has been performed manually, and is simplified a bit in places for the sake of readability. The actual translator is implemented with the DC+ format as a target, directly, as mentioned in section 8.

```
process METRONOME=
     ( ? event A,RUN2,RUN4
       ! event D
     )
(| (D,ACTIVE2,ACTIVE4):= METRONOME_BODY(TICK,TICK,
                                        0 when NUL_K(TICK),
                                        GO(TICK),NUL_K(TICK),
                                        NUL_K(TICK),NUL_K(TICK),
                                        A,RUN2,RUN4)
 | TICK:= A default RUN2 default RUN4
 |)
where
process METRONOME_BODY=
     ( ? event TICK,LOCALCLOCK;
         integer CONTROL;
         event STM,SPM,RSM,SDM,A,RUN2,RUN4
       ! event D,ACTIVE2,ACTIVE4
     )
   (| (STARTED_M,STOPPED_M,ACTIVE,SUBCONTROL):=
                   ACTIVITYSTATE(TICK,LOCALCLOCK,CONTROL,STM,
                                        SPM,RSM,SDM)
    | (STE2,SPE2,RSE2,SDE2,STE4,SPE4,RSE4,SDE4):=
                         CTRL_METRONOME(TICK,ACTIVE,SUBCONTROL,
                                        RUN2,RUN4)
    | (D2,ACTIVE2):= EVERY2(TICK,ACTIVE,SUBCONTROL,STE2,SPE2,
                                        RSE2,SDE2,A)
    | (D1,ACTIVE4):= EVERY4(TICK,ACTIVE,SUBCONTROL,STE4,SPE4,
                                        RSE4,SDE4,A)
    | D:= SHIFT(D1 default D2,TICK)
```

```
    |);
process CTRL_METRONOME=
     ( ? event TICK,LOCALCLOCK;
         integer CONTROL;
         event RUN2,RUN4
       ! event STE2,SPE2,RSE2,SDE2,STE4,SPE4,RSE4,SDE4
     )
   (| T1:= TRANSITION{0,1}(C_T,RUN2)
    | T2:= TRANSITION{1,0}(C_T,RUN4)
    | C_T := C when (not (^CONTROL) default LOCALCLOCK)
    | (C,NC):= NEXTSTATE{0}(TICK,LOCALCLOCK,CONTROL,T1
                                        default T2)
    | STE2:= ^ T1
    | SPE4:= STE2
    | STE4:= ^ T2
    | SPE2:= STE4
    | RSE2:= NUL_K(TICK)
    | SDE2:= NUL_K(TICK)
    | RSE4:= NUL_K(TICK)
    | SDE4:= NUL_K(TICK)
    |);
process EVERY2=
     ( ? event TICK,LOCALCLOCK;
         integer CONTROL;
         event STE2,SPE2,RSE2,SDE2,A
       ! event D2,ACTIVE
     )
   (| (STARTED_E2,STOPPED_E2,ACTIVE,SUBCONTROL):=
                                        ACTIVITYSTATE(TICK,
                                          LOCALCLOCK,CONTROL,
                                          SSTE2,SPE2,RSE2,DE2)
    | D2:= EVERY_2(TICK,ACTIVE,SUBCONTROL,A)
    |);
process EVERY_2=
     ( ? event TICK,LOCALCLOCK;
         integer CONTROL;
         event A
```

```
       ! event D2
     )
   (| T1:= TRANSITION{0,1}(C_T,A)
    | T2:= TRANSITION{1,0}(C_T,A)
    | C_T := C when (not (^CONTROL) default LOCALCLOCK)
    | (C,NC):= NEXTSTATE{0}(TICK,LOCALCLOCK,CONTROL,
                                         T1 default T2)
    | D2:= ^ T2
    |);
process EVERY4=
     ( ? event TICK,LOCALCLOCK;
         integer CONTROL;
         event STE4,SPE4,RSE4,SDE4,A
       ! event D1,ACTIVE
     )
   (| (STARTED_E4,STOPPED_E4,ACTIVE,SUBCONTROL):=
                                         ACTIVITYSTATE(TICK,
                                           LOCALCLOCK,CONTROL,
                                           STE4,SPE4,RSE4,SDE4)
    | (D1,B1):= EVERY_4(TICK,ACTIVE,SUBCONTROL,A,B)
    | B:= SHIFT(B1,TICK)
    |);
process EVERY_4=
     ( ? event TICK,LOCALCLOCK;
         integer CONTROL;
         event A,B
       ! event D1,B1
     )
   (| B1:= UP(TICK,LOCALCLOCK,CONTROL,A)
    | D1:= DOWN(TICK,LOCALCLOCK,CONTROL,B)
    |);
process UP=
     ( ? event TICK,LOCALCLOCK;
         integer CONTROL;
         event A
       ! event B1
     )
```

```
   (| T1:= TRANSITION{0,1}(C_T,A)
    | T2:= TRANSITION{1,0}(C_T,A)
    | C_T := C when (not (^CONTROL) default LOCALCLOCK)
    | (C,NC):= NEXTSTATE{0}(TICK,LOCALCLOCK,CONTROL,
                                        T1 default T2)
    | B1:= ^ T2
    |);
process DOWN=
     ( ? event TICK,LOCALCLOCK;
         integer CONTROL;
         event B
       ! event D1
     )
   (| T1:= TRANSITION{0,1}(C_T,B)
    | T2:= TRANSITION{1,0}(C_T,B)
    | C_T := C when (not (^CONTROL) default LOCALCLOCK)
    | (C,NC):= NEXTSTATE{0}(TICK,LOCALCLOCK,CONTROL,
                                        T1 default T2)
    | D1:= ^ T2
    |);

% **************** Library ********************%

process ACTIVITYSTATE=
     ( ? event TICK,LOCALCLOCK;
         integer CONTROL;
         event STA,SPA,RSA,SDA
       ! event STARTEDA,STOPPEDA,ACTIVE;
         integer SUBCONTROL
     )
   (| T1:= TRANSITION{0,1}(C,STA)
    | T2:= TRANSITION{1,0}(C,SPA)
    | T3:= TRANSITION{2,1}(C,RSA)
    | T4:= TRANSITION{1,2}(C,SDA)
    | T5:= TRANSITION{2,0}(C,SPA)
    | (C,NC):= NEXTSTATE{0}(TICK,LOCALCLOCK,CONTROL,
                                        T1 default T2
```

```
                            default T3 default T4 default T5)
    | STARTEDA:= SHIFT((^ T1) when (NC=1),TICK)
    | STOPPEDA:= SHIFT((^ (T2 default T5)) when (NC=0),TICK)
    | SUBCONTROL:= (0 when (STA default (CONTROL=3))
                    default (1 when SPA) default (2 when RSA)
                    default CONTROL)  when ACTIVE
    | ACTIVE:= when (NC=1)
    |);
process TRANSITION=
     { STATE1,STATE2 ; }
     ( ? integer ORIGIN;
         event TRIGGER
       ! integer TARGET
     )
   (| TARGET:= STATE2 when TRIGGER when (ORIGIN=STATE1) |);
process NEXTSTATE=
     { integer INITIAL_STATE; }
     ( ? event TICK,LOCALCLOCK;
         integer CONTROL,NEW
       ! integer ZCONFIGURATION,CONFIGURATION
     )
   (| CONFIGURATION:= NEW default
                      (INITIAL_STATE when (CONTROL=0)) default
                      ZCONFIGURATION
    | ZCONFIGURATION:= CONFIGURATION $ 1
    | CONFIGURATION^=LOCALCLOCK
    |)
   where
      integer CONFIGURATION,ZCONFIGURATION init INITIAL_STATE;
   end %NEXTSTATE%;
process SHIFT=
     ( ? X;
         event TICK
       ! Y
     )
   (| INSTANT_X:= (^ X) default (not TICK)
    | SHIFT_INSTANT_X:= INSTANT_X $ 1
```

```
     | VALUE_X:= X default SHIFT_VALUE_X
     | SHIFT_VALUE_X:= VALUE_X $ 1
     | VALUE_X^=(^ X) default TICK
     | Y:= SHIFT_VALUE_X when SHIFT_INSTANT_X
     |)
   where
      boolean INSTANT_X,SHIFT_INSTANT_X init false;
      event SHIFT_VALUE_X,VALUE_X;
   end %SHIFT%;
process NUL_K=
     ( ? event SOMETHING
       ! event NOTHING
     )
   (| NOTHING:= when (not SOMETHING) |);
process GO=
     ( ? event TICK
       ! event OUT
     )
   (| OUT:= when ZFIRSTTICK
    | ZFIRSTTICK:= FIRSTTICK $ 1
    | FIRSTTICK:= false
    | FIRSTTICK^=TICK
    |)
   where
      boolean ZFIRSTTICK init true ;
   end %GO% ;
end %METRONOME%
```

# Contents

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399