HAL
open science

Modelling Statecharts and Activitycharts as Signal equations

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

To cite this version:

Jean-René Beauvais, Eric Rutten, Thierry Gautier, Roland Houdebine, Paul Le Guernic, et al.. Modelling Statecharts and Activitycharts as Signal equations. ACM Transactions on Software Engineering and Methodology, 2001, 10 (4), pp.397-451. hal-00546080

HAL Id: hal-00546080

https://hal.science/hal-00546080v1

Submitted on 13 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Modelling Statecharts and Activitycharts as Signal equations

J.-R. Beauvais, E. Rutten, T. Gautier, R. Houdebine, P. Le Guernic and Y.-M. Tang
INRIA

The languages for modelling reactive systems are of different styles, like the imperative, state-based ones and the declarative, data-flow ones. They are adapted to different application domains. This paper, through the example of the languages Statecharts and Signal, shows a way to give a model of an imperative specification (Statecharts) in a declarative, equational one (Signal). This model constitutes a formal model of the Statemate semantics of Statecharts, upon which formal analysis techniques can be applied. Being a transformation from an imperative to a declarative structure, it involves the definition of generic models for the explicit management of state (in the case of control as well as of data). In order to obtain a structural construction of the model, a hierarchical and modular organization is proposed, including proper management and propagation of control along the hierarchy. The results presented here cover the essential features of Statecharts as well as of another language of Statemate: Activitycharts. As a translation, it makes multi-formalism specification possible, and provides support for the integrated operation of the languages. The motivation lies also in the perspective of gaining access to the various formal analysis and implementation tools of the synchronous technology, using the DC+ exchange format, as in the Sacres programming environment.

Categories and Subject Descriptors: D.2 [**Software**]: Software Engineering; F.3.2 [**Logics and Meanings of Programs**]: Semantics of programming Languages

General Terms: Languages

Additional Key Words and Phrases: Behavioral modelling, reactive systems, statechart, STATEMATE, synchronous languages, SIGNAL

1. INTRODUCTION

1.1 Context and objective

Different languages exist for the design of reactive systems: the languages Lustre [Halbwachs et al. 1991] and Signal [Le Guernic et al. 1991] are declarative and equational data flow languages, while others are imperative sequencing languages:

This work was partially supported by the Esprit Project EP 20897 Sacres.
Authors' addresses: EP-ATR project, IRISA/INRIA-Rennes, 35052 RENNES cedex, France; E. Rutten, BIP project, INRIA Rhône-Alpes, 655 avenue de l'Europe, F-38330 MONTBONNOT SAINT MARTIN, France
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or permissions@acm.org.

textual like Esterel [Berry and Gonthier 1992] and ELECTRE [Perraud et al. 1992], or graphical like Statecharts [Harel 1987] and Argos [Maraninchi and Halbwachs 1996]. The choice between the declarative and the imperative approach has an influence upon the facility with which a given application area can be handled. For instance, declarative languages easily handle applications like 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 presents a translation from the essential features of Statecharts and Activitycharts to the equational language Signal. Among the different semantics of Statecharts [M. von der Beeck 1994], the translation presented here follows the Statemate one [Harel and Naamad 1996]. This scheme yields a formal model of the Statemate semantics of Statecharts, upon which formal analysis techniques can be applied. Being a transformation from an imperative to a declarative structure, it involves the definition of generic techniques for the explicit management of state (in the case of control as well as of data). In order to obtain a structural construction of the model, a hierarchical and modular organization is proposed, including proper management and propagation of control along the hierarchy. The results presented here cover the essential features of Statecharts as well as of another language of Statemate: Activitycharts. This work was implemented in a prototype translator, covering only the most substantial parts of the Statemate languages (the control part of Activitycharts and Statecharts, and the simple actions). This prototype was used to treat examples of small to medium size (i.e., several activities, associated with Statecharts of several hierarchical levels), either made up or inspired by sample specifications from partners in the ESPRIT project Sacres.

1.2 Motivation

The primary motivation for this work is to make more widely available a corpus of results in the synchronous approach to reactive and real-time systems. 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),

—remedies the lack of imperative features of Signal,

—gives a compositional definition of Statecharts semantics,

—opens a direct connection from a Statecharts design to the synchronous technology tools of the Signal environment, and moreover to the tools compatible with the DC+ format [SACRES Consortium 1997a]: compilers, simulators, verification systems,

—in particular, it gives direct access to the Signal code generator, which 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 [Benveniste et al. 1998].

We believe that, on the one hand, the graphical readability of Statecharts makes it good for the designing an imperative specification. On the other hand, the Signal compiler uses an elaborate clock calculus, which makes it a good choice for

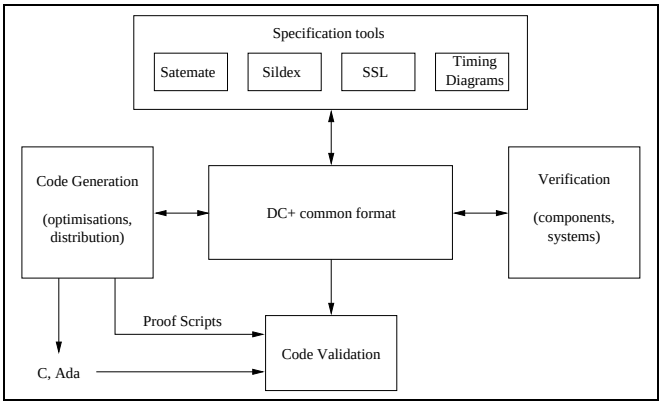

Fig. 1. *Global architecture of the Sacres environment*

extracting clock properties from a specification, in order to get efficiency in code generation and in verification. Hence, keeping the structural information through the translation, and not simply coding, is a key issue for understanding interactions between components from different sources. The traceable links between the initial specification and the generated code potentially allow us to route information extracted from the verification tools back to the specification, for user feedback (diagnostic, counter-example). In the other direction, from the specification to the generated code, traceability may be used to add, at specification time, directives for the partitioning into tasks or distributed processes.

This work constitutes part of the Sacres programming environment [Grazebrook 1997]. 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[1]. Among the application domains targeted are avionics and process control. The question of certification and validation is integrated into the environment. 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 that supports the representation of Signal; as such, they are quite similar: 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

[1]Member partners of the Sacres project are: British Aerospace (UK), aircraft builder; i-Logix (UK), developing and distributing StateMate, the environment for designing in Statecharts; INRIA (France), a research institute where new technologies are defined and developed around the synchronous language Signal [Gautier et al. 1994]; OFFIS (Germany), research institute bringing verification technology; Siemens (Germany), where controllers for industrial processes are developed; SNECMA (France), builder of aircraft engines; TNI (France), developing and distributing the SILDEX tool and the Signal language; the Weizmann Institute (Israel), as regards semantic aspects and the validation of code.

readability purposes, hence this paper presents the translation in terms of Signal rather that DC+.

1.3 Organization of the paper

Sections 2 and 3 describe the Signal and Statecharts formalisms. Section 4 describes the model on which the translation is based. Section 5 describes how each of the major constructs of Statecharts are translated, and gives illustrative examples. It covers the essential features of 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, and some particular aspects of actions, are part of the perspectives. Section 6 describes the translation schemes for Activitycharts. Section 7 describes how the relational aspects of Signal can be used to model execution schemes and 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 [Le Guernic et al. 1991].

The idea of the synchronous model is very simple: it assumes that a program, embedded in some environment, interacts with this environment through a finite set of communication links. At any particular instant, more than one event (reception or emission of messages, calculations) may occur, in which case they are simultaneous and they have the same temporal index. The evolution of the computation results from the sequences of these communications, but also from explicit changes expressed in the program (for example, for an equation $y_t = f(x_{t-1})$, the output y will be simultaneous with each one of the *next* occurrences of the input x). Without such an explicit change, the outputs are considered simultaneous with the inputs that have been used in their computation, like in Esterel, Lustre or Signal, or they are produced at the next instant, like in Statemate for instance.

The Signal language is also a data flow language, where a program is a system of equations. Unlike in Lustre (which is also data flow), the system of equations of a Signal program describes a *relation* between the inputs and outputs of the system (not necessarily a function): this makes it possible to describe, in Signal, partial specifications or non deterministic behaviors.

A Signal program specifies a real-time system in terms of a collection of equations defining *signals*. The collections of equations can be organized hierarchically in subsystems (or *processes*). A signal is a sequence of values of a given type, with which is associated an implicit *clock* which defines the discrete set of instants at which these values are present. The different clocks are not necessarily linked together by fixed sampling frequencies: they can have occurrences depending on local data or external events (like interruptions for example). Among the types of the signals (which are the types of their values), there are classical types—Booleans, integers, etc.—and a special type called `event`, which is a subtype of the Boolean type with the single value *true*: it allows the representation of *pure signals,* and it is used in particular to represent in the language the clocks of the signals.

The Signal language is built on a small number of primitives, the semantics of which is described informally below. In order to give this semantics, we add a

distinguished symbol, denoted $\perp$, to the considered domain of values of the signals: this symbol represents the absence of value (note that it is not manipulated in the language). The semantics of a program is described by the set of acceptable sequences of valuations of the variables of the program in the domain of values completed by $\perp$. The notation X_t represents the value of the variable (or signal) X at the *instant* t in some sequence of valuations of some subset of variables (we call such a sequence a *flow*).

The kernel of Signal is composed of the following primitives, which define *elementary processes:*

—Functions or relations extended to sequences:

$$\texttt{Y := f(X1,...,Xn)} : \begin{cases} \forall t, \texttt{Y}_t = \perp \Rightarrow \texttt{X1}_t = ... = \texttt{Xn}_t = \perp \\ \forall t, \texttt{Y}_t \neq \perp \Rightarrow \texttt{Y}_t = \texttt{f}(\texttt{X1}_t, ..., \texttt{Xn}_t) \end{cases}$$

Examples of such operators are classical arithmetic operators (`+`, `*`...), relations (`=`, `<`...), Boolean operators (`not`, `or`...).

—Delay:

$$\texttt{Y := X \$1 init v0} : \begin{cases} \forall t, \texttt{Y}_t = \perp \Rightarrow \texttt{X}_t = \perp \\ \texttt{Y}_0 \neq \perp \Rightarrow \texttt{Y}_0 = \texttt{v0} \\ \forall t > 0, \texttt{Y}_t \neq \perp \Rightarrow \texttt{Y}_t = \texttt{X}_{t-1} \end{cases}$$

This operator expresses an explicit consumption of time (dynamic equation).

—Extraction on Boolean condition:

$$\texttt{Y := X when B} : \forall t, \begin{cases} \texttt{B}_t \neq true \Rightarrow \texttt{Y}_t = \perp \\ \texttt{B}_t = true \Rightarrow \texttt{Y}_t = \texttt{X}_t \end{cases}$$

—Merge with priority:

$$\texttt{Y := U default V} : \forall t, \begin{cases} \texttt{U}_t \neq \perp \Rightarrow \texttt{Y}_t = \texttt{U}_t \\ \texttt{U}_t = \perp \Rightarrow \texttt{Y}_t = \texttt{V}_t \end{cases}$$

The composition of two processes ($|P|Q|$) is defined by the set of all the flows respecting, in particular on common variables, the set of constraints respectively imposed by P and Q (the composition is commutative and associative).

Finally, the restriction of visibility of a signal `X`, denoted P `where X`, is the projection of the set of flows associated with P on the set of variables of P minus `X`.

As it can be seen on the above description of the primitives, each signal has its own temporal reference (its *clock*, or set of instants at which it is different from $\perp$). For example, the first two primitives are *single-clocked:* they constrain all the implied signals so that they have the same clock. On the other hand, the third and fourth ones are *multiple-clocked*, the signals may have different clocks.

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. We describe now those we use in this paper.

If `X` is a signal of any type, `Y := ^X` defines the `event` type signal which is present (with the value *true*) whenever `X` is present: it is used to represent the clock of the signal `X`.

A special case of an `event` type signal is the signal denoted `^0`, which is, by definition, never present.

The following special case of the extraction, `Y := when B`, where `B` is a Boolean signal, defines also an event type signal, which is present whenever `B` has the value *true*.

For any two signals `U` and `V`, the intersection and the union of their clocks, represented by event type signals, can be obtained respectively by the equations `Y := U ^* V` and `Y := U ^+ V` (these equations are more specific notations for respectively `Y := ^U when ^V` and `Y := ^U default ^V`).

Finally, constraints between clocks of signals can be specified by using clock equations such as `X1 ^= ... ^= Xn`, which asserts that the signals `X1`, ..., `Xn` have the same clock (i.e., are present at the same instants).

The Signal compiler mainly consists of a formal system which is able to reason about clocks, logics and dependence graphs. In particular, the *clock calculus* and the dependence calculus [Amagbegnon et al. 1995] provide a synthesis of the global synchronization of the program from the specification of the local synchronizations (given by the Signal equations), and a synthesis of the global scheduling of the specified calculations. Contradictions and inconsistencies can be detected by these calculi. The clock calculus using BDD techniques, synthesizes a *hierarchy* of clocks (based on inclusion of presence instants), which constitutes, together with the dependence graph, the basis on which a number of tools can be applied. These tools can perform automatic transformations in order to reorganize the program or to get specific optimizations, they can generate inputs for verification tools such as model checkers, they can generate sequential code or distribute the application on some architecture, they can evaluate the time consumption of the program on a given architecture, etc.

```
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;
```

Fig. 2. *A refillable tank*

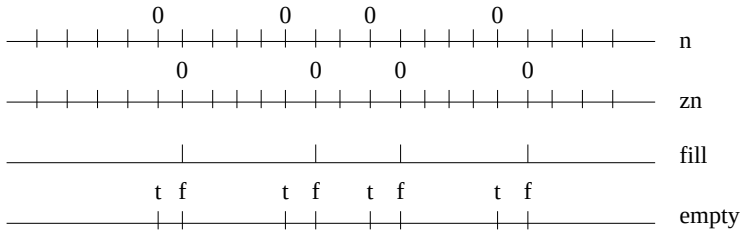

Fig. 3. *The clocks of the tank process*

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` (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 present instants (clocks) of the signals in this program are illustrated in Figure 3. One can notice that the clock of local (internal) signals is faster than (i.e., includes) 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. This particularity will be explored in the modelling of Statemate, particularly in Section 7.

3. STATEMATE: STATECHARTS AND ACTIVITYCHARTS

The Statecharts formalism was introduced by Harel [Harel 1987]. It is a graphical language based on automata. It is integrated into 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

modelling 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 broadcast 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 possibilities are available for specifying the behavior: when entering a state by a transition pointing to the boundary of the state, the state targeted 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 mechanism 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* for the transition to fire. 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 when the transition is fired. As a special kind of transition, 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 Statecharts proceeds in steps, where given the events currently present and the current values of variables, triggers and conditions are

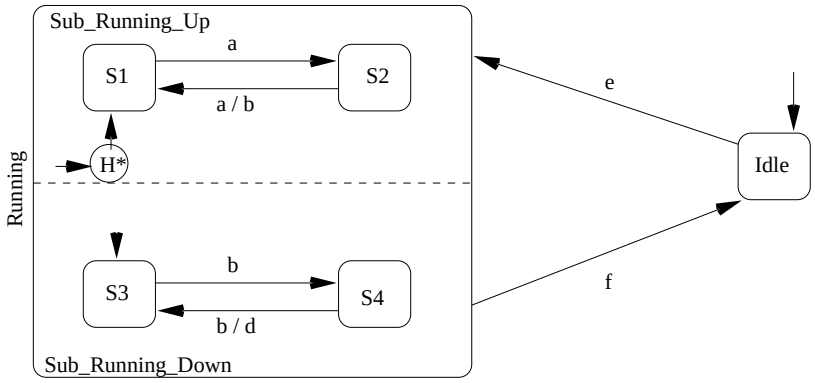

Fig. 4. *A Statechart example*

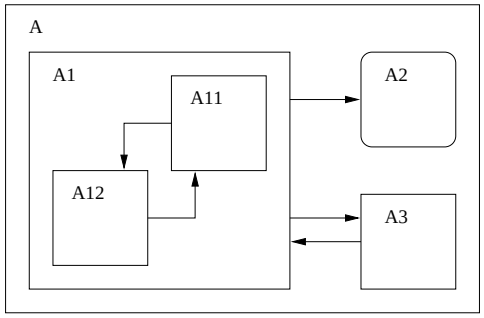

Fig. 5. *An activities hierarchy*

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 distinguishes Statemate from other semantics of Statecharts-like languages [M. von der Beeck 1994], e.g., the strictly synchronous one as in Argos [Maraninchi and Halbwachs 1996], where effects are sensed in the very same reaction: in the example, the reception of the fourth occurrence of `a` would result in emitting `b` and `d` in the same reaction.

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. This mechanism is close to the semantics of Sequential Function Charts *with search for stability* [IEC 1993; Marcé and Le Parc 1993].

There is a textual language, of a classical imperative form, *à la* Pascal, for constructing actions. For actions which consist of assigning values to variables, the same variable can be referenced in assignments associated with different transitions: each provides a contributed value, and the variable takes its values from the action contributing in the current step. The actions can feature a form of variable called *context* variables, local to the action, which can take several values within one step. Iteration constructs include the unbounded *while* loop, thereby incurring the risk of non-termination. Also, a *timeout* and a *schedule* action are available.

Activities. Besides the Statecharts, another language of the Statemate environment is Activitycharts [Harel and Naamad 1991]: 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 round-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. Activities can be associated

with states in Statecharts, in one of two ways: *within* and *throughout*, depending on the way they can be started (see Section 6).

Actions with a trigger can be associated with an activity, they are called mini-specs, and have the same form as the labels seen associated with transitions or static reactions with states in Statecharts. Another form of actions of activities is called *combinational assignments*: the difference with mini-specs is that the values they compute are available within the same step.

Links between activities represent the data or control exchanges between activities. This aspect of the language is, however, based on the fact that variables and events are known globally in a chart (the scope of their definition). Therefore it will not be handled explicitly in the translation. The data flow links are an optional explicit representation of communications that exist 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.

The languages of Statemate form a set of languages with a full spectrum of features, in different styles, and elaborate actions and data types. They allow for the specification and simulation of designs with such an expressiveness that even designs presenting potential risks can be written. Especially in the framework of safety-critical embedded systems, it is considered very risk-prone to use unbounded instructions or structures like, e.g., the *while* loop in actions, or queue data structures. Also, some features do not follow the overall structured quality of the language, like the possibility of drawing transitions crossing several levels of **and**- and **or**-nodes.

Proposals concerning the safe use of Statemate have been described, called Safechart [Armstrong 1996], which consisted first in directives and advices in using Statemate, rather than strict interdictions. They advise for example to avoid cross-level transitions, avoid using the negation on events, etc. Later on, a formally defined syntactic sub-language was identified, and it was formalized by a function mapping it to an axiomatization in Real Time Logic [Armstrong 1998].

In the following, we will restrict ourselves to a sub-language of Statecharts that we model in Signal, where all aspects covered are bounded.

4. TRANSLATION PRINCIPLES

This section presents the overall approach to the structural translation, and some useful basic elements. First, we introduce a reactive box model, and the way it is featured in the hierarchical 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 a 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. The hierarchical propagation of control signals such as clocks and resets will follow these interfaces.

An interesting property of the Signal language, is that the behavior of the com-

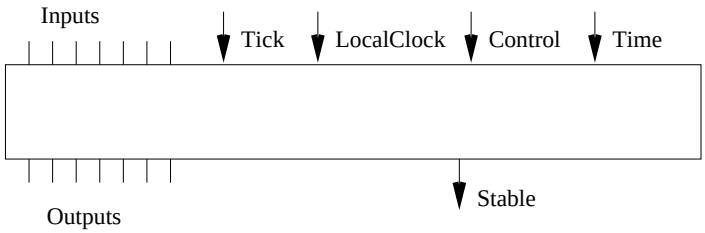

Fig. 6. *The reactive box model*

position of two processes is the intersection of the behaviors of the constituent processes. This is similar to the solutions of an equational system. This reactive box model is compositional in the same sense and gives a compositional semantics for Statecharts.

A reactive box is a box with 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[2].

—**LocalClock** is the clock of the box. This clock is present whenever the corresponding Statecharts component is active.

—**Control** is an enumerated-typed signal with values 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 apart from 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.

—**Time** is a signal recording the passing of the outside time. It can be given by impulses of the event type, or by values of dates.

—**Stable** is a Boolean signal that is *true* when the inner box has completed its job for the current reaction to inputs, and is ready to synchronize with its environment in order to accept new input. The clock of **Stable** is the clock of the box. **Stable** is used to cope with superstep semantics.

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

4.2 Hierarchical model

The reactive box is used in a hierarchical manner to build the model, which has consequences in the compilation.

4.2.1 *Structural translation.* As was presented in Section 3, a Statemate specification is a hierarchy of Statemate entities (Activitycharts ot Statecharts), each with associated actions, and sub-components (sub-activities of an activity, possibly a control activity, and subcharts of a Statecharts And- or Or-state).

The structural translation proposed here follows the top-down approach where, at each level of this Statemate hierarchy, we have a reactive box. At the highest level, its interface is connected to the environment (see Section 7.1). Such a box contains representations of the local behaviour, of sub-components and their interactions:

—for each variable declared at that level or scope, a model of a memory (see Section 4.6),

—a model of the local behaviour, which can involve state and transitions (see Sections 4.4 and 4.5),

—for each sub-component, a reactive box, with the appropriate input-output profile, and for each of them:

  —the global tick and time are transmitted as such,

  —a local clock is computed from that of the current level component, and can depend on its state and actions,

  —a control signal is computed from that received from the upper level by the current one, and can depend on its state, actions, and transitions.

—the stability signals from all sub-components models are collected and compute a stability value for the current level.

The aspects specific to hierarchies in Statecharts and Activitycharts are further described respectively in Sections 5.1.2 and 6.1.

4.2.2 *Hierarchy of clocks.* In Signal, basic objects are signals which always have a clock, while in Statemate, only events are clocked. Variables in Statemate are of two kinds: 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**.

It is important to note that this is where the hierarchy of the Statemate structure is reflected in a hierarchy of clocks in Signal: the local clock passed to the lower levels of the hierarchy is a sub-sampling of the local clock of the current level, according to the state of activity of that lower level. Besides this hierarchical aspect, a formal analysis of the clock system is performed during the compilation of Signal. It consists in arranging clocks in a hierarchical structure depending on the definition of the ones as expressions of the others, especially as extractions defining sub-clocks. This might improve the global clock hierarchy by establishing relations between clocks not coming from this structural aspect.

4.3 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`	Statecharts `not e`	Signal `not e`
t	f	f
Absent	t	Absent

Fig. 7. *Differences between Statemate not and Signal not.* In Statecharts, `not e` means: `e` did not occur, while in Signal it is a conservative extension of the `not` operator 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;
  ! boolean e2;)
(| e2 := not(e1) default ref_clock |)
;
```

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`.

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`.

Statecharts use `and, or, not` also for conditions. These ones are translated using the Boolean primitives `and, or, not` of Signal.

4.4 Transition

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

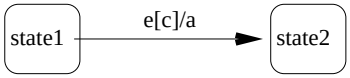

Fig. 8. *A transition in Statecharts*

4.4.1 *Signal encoding:*

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

;

where the constant parameters do not need to be typed, the Signal compiler being provided with a type inference mechanism.

4.4.2 *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 4.5 about configuration signal), `trigger` is the signal which triggers the transition, `target` gives the new state chosen when the transition is fired.

4.4.3 *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 7.2). The result is fed into the new configuration, managed by the predefined process described next.

4.5 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 the 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.

In this paper, we define a structural translation from Statecharts to Signal where we follow the hierarchy for a double purpose. On the one hand, it favours readabilty and traceability in the produced Signal mode. On the other hand, this hierarchical structure involves the control of the activity of sub-charts, and this can be encoded in a hierarchy of clocks, hence enabling the use of the clock calculus.

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

4.5.1 *Signal encoding:*

```
process nextstate = {state initial_state;}
( ? event tick, localclock; tcontrol Control; event Time; state new;
  ! boolean Stable; state zconfiguration, configuration;)
(| configuration := (initial_state when control=Start)
                    default new default zconfiguration
 | zconfiguration := configuration $1 init initial_state
 | configuration ^= localclock ^+ Control
 | Stable := (configuration = zconfiguration)
 |)
;
```

4.5.2 *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`. `Stable` is a Boolean indicating whether stability is reached or not

In order to conform with the reactive box structure, inputs `Time` and `Tick` are added, though unused.

4.5.3 *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` (this re-initialization is of greater priority with regard to possibly enabled transitions which would produce a value of `new`); else:

—if a new value occurs (`new` is present), `configuration` takes it as a new value; else:

—if `localclock` occurs alone, `configuration` remains unchanged (copied from its previous value).

The output `Stable` is *true* when the configuration remains unchanged. The clock of `Stable` is that of the configuration.

4.6 Shift

The Statemate semantics of Statecharts [Harel and Naamad 1996] 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.

4.6.1 *Signal encoding:*

```
process shift =
( ? x;
    event tick;
  ! y; boolean Stable; )
(| value_x := x default y
 | value_x ^= ^x default tick
 | y := value_x $1
 | Stable := not ^x default tick
 |)
where value_x;
end;
```

4.6.2 *Interface:.* `x` is the signal to be shifted, `y` the shifted signal (their types will be inferred for each instantiation), `tick` the clock of the Statemate step, and `Stable` the stability Boolean.

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

Tick	t	t	t	t	t	t	t	t	t
x	1		2				3	4	5
value_x	1	1	2	2	2	2	3	4	5
y		1	1	2	2	2	2	3	4

Fig. 9. *A trace of the* `shift` *process with* `x` *integer*

Given a signal and a clock (usually the fastest clock), it shifts the new values of the signal to the next 'tick' of the clock, while memorizing the last new value.

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* [M. von der Beeck 1994], the shift would have to be 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. The case of events is detailed in Section A.1.

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). The process `shift` is the right place to define them, as it is there that memorization of values takes place: hence it can be extended to manage these features, as presented in Section A.2.

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, and outlining an overview of the hierarchical structure of the model, before giving the translation scheme.

5.1 Or-states and And-states

5.1.1 *Example.* As we said in Section 4.2.1, 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. We will follow the structure of the example in Figure 4, introducing some important features as we go. An Or-state is at the top-level, with two states and two transitions. One of its states is refined into a subchart, which is an And-state. This latter is also refined into two Or-states, with different history connectors.

5.1.1.1 *Or-states..* For the top-level, we define a configuration signal giving the next configuration (`nc`) of the Statechart, in function of its current configuration `c`. 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`. 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.

```
| t1 := transition {Idle, Running} (c_t, e)
| t2 := transition {Running, Idle} (c_t, f)
| Control ^= ^0
| c_t := c when ( (not ^Control) default LocalClock)
| (Stable,c,nc) := nextstate {Idle}
        (Tick, LocalClock, Control, Time, t1 default t2)
```

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 7.2, 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`.

5.1.1.2 *State refinement..* 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

	Tick	t	t	t	t	t	t	t	t	t	...
Toplevel	LocalClock	t	t	t	t	t	t	t	t	t	...
(future)	nc	I	R	R	R	R	R	R	I	I	...
(present)	c	I	I	R	R	R	R	R	R	I	...
	c_t	I	I	R	R	R	R	R	R	I	...
	e		t								...
	f								t		...
	a				t	t					...
	b						t				...
Sub_	LocalClock			t	t	t	t	t	t		...
Running_	Control		Start						Stop		...
Up	nc		S1	S1	S2	S1	S1	S1	S1		...
	c		S1	S1	S1	S2	S1	S1	S1		...
	c_t			S1	S1	S2	S1	S1			...
Sub_	LocalClock			t	t	t	t	t	t		...
Running_	Control		Start						Stop		...
Down	nc		S3	S3	S3	S3	S4	S4	S4		...
	c		S3	S3	S3	S3	S3	S4	S4		...
	c_t			S3	S3	S3	S3	S4			...

Fig. 10. *Execution trace for the example in Figure 4*

not used inside. Its outputs are `b` and `d`. The clock of the configuration variables refining `Running` is defined as follows:

```
localclock_running:= when (c=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 ^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 executing actions upon entering. Resetting when exiting like in [Maraninchi and Halbwachs 1996] would execute actions at the wrong instants according to the Statemate semantics.

Putting all this information together gives the parameters of the box `running`:

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

Figure 10 illustrates the different clocks in presence, where I is for state Idle, and R for Running.

5.1.1.3 *And-states..* The `Running` state is the **and** composition of two **or**-states, called: `sub_running_up` and `sub_running_down`. It can be modeled as the syn-

chronous composition of the processes `sub_running_up` and `sub_running_down`, detailed further, which model the two sub-states:

```
process running =
( ? event Tick, LocalClock; tcontrol Control; event Time;
    event a, b;
  ! boolean Stable; event b, d;)
(| (Stable_up,b) :=
              sub_running_up(Tick, LocalClock, Control, Time, a)
 | (Stable_dn,d) :=
              sub_running_down (Tick, LocalClock, Control, Time, b)
 | Stable := Stable_up and Stable_dn
 |)
;
```

The clocks transmitted to the sub-processes are the same as for the **and**-state.

5.1.1.4 *Sub-states..* Using the **or**-state translation scheme we obtain the signal equations for the translation of `sub_running_up` except for a few differences with the top-level example given above. The clock of sub-state variables is defined in order to have an instant where re-initialization can be performed.

The process `sub_running_up` encoding the corresponding state is as follows:

```
process sub_running_up =
{ ? event Tick, LocalClock; tcontrol Control; event Time;
    event a;
  ! boolean Stable; event b;}
(| t3 := transition {S1, S2} (c_t, a)
 | t4 := transition {S2, S1} (c_t, a)
 | c_t := c when ((not ^Control) default LocalClock)
 | (Stable,c,nc) := nextstate {S1}
        (Tick, LocalClock, ^0, Time, t3 default t4)
 | b := ... (see translation of actions)
 |)
;
```

Equations for state `sub_running_down` are looking very similar. The difference is in the control of reinitialization. `sub_running_up` has a history connector, hence should not be reinitialized when `Running` is re-entered. Therefore the `control` input is set to the null clock `^0`. In a different way, `sub_running_up` has no history connector. therefore its control input must be adequately controlled, as explained below.

5.1.1.5 *History and deep history..* When the event `f` is generated, it preempts the sub-automata of `Running` and in the sub-automaton `sub_running_up` the last active state will be re-established 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 ^t1`.

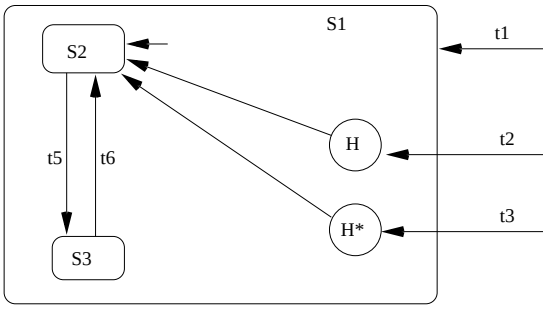

Fig. 11. *Three ways to enter a state*

More generally, three ways are used to enter a state (see Figure 11): 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 substates `S2, S3`. The table shown on Figure 12 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

Fig. 12. *Which configuration variables to reset?*

5.1.2 *Overview of the structure of the model.* In this Section we give an overview of the structure of the model, illustrating how the equations and processes examplified before, and explained further, are grouped into a global model.

Figure 13 illustrates the simplified structure of the model for an OR-node (not detailing all the interface of the reactive box). It has the structure of the example in Figure 4: it has two states, and two transitions. For more generality, we represent the model for two transitions each labeled with a trigger $trig_i$ and an action act_i, the latter computing values $contrib_i$ of a variable var.

—memorization of variables is handled by instances of the process `shift`, taking as inputs contributed values from the actions (see 5.2.1);

—they serve to evaluate triggers, as modelled according to the translation α of Section 5.2.2;

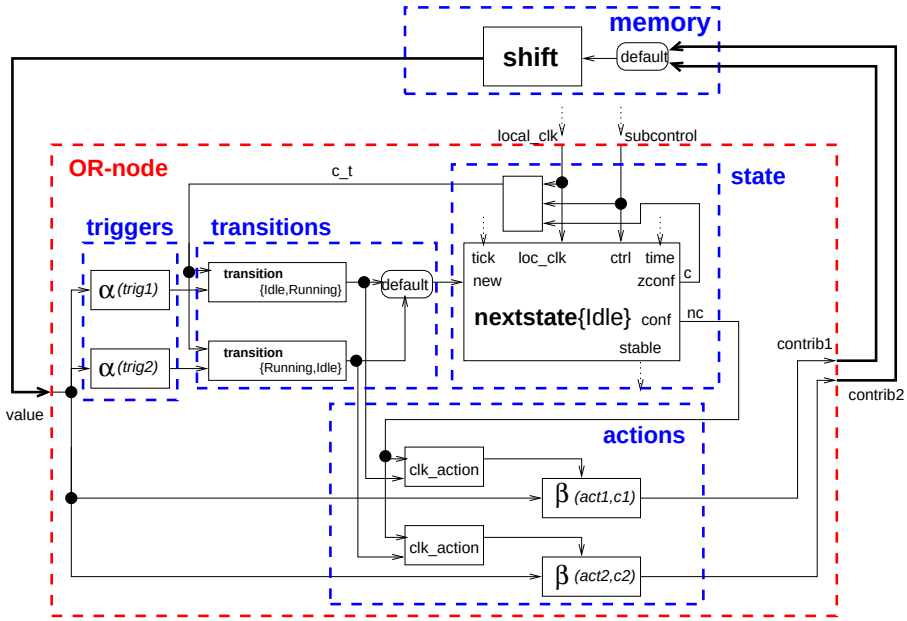

Fig. 13. *An overview of the model for an OR-node*

—the triggers, combined with configuration `c_t`, are used to determine whether each transition is fireable, using instances of the process `transition` of Section 4.4;

—an instance of the process `nextstate` manages the value of the configuration of Section 4.5;

—transitions and the next state serve to evaluate clocks of actions,

—the actions themselves are translated according to function β of Section 5.2.3, and produce values $contrib_i$.

Figure 14 illustrates the hierarchical structure, and particularly how at each level the previous OR-node pattern is re-used (here the outputs `s` and `t` correspond, respectively, to the state and to the transition signals involved in the computation of `subcontr`, detailed in the following), and how the control is propagated to sub-processes (modelling sub-nodes) following the hierarchy of OR-nodes. In the case of AND-nodes, inputs and control signals are forwarded as such to each of the models of sub-nodes, and contribution outputs are gathered. In the example of Figure 14, the OR-node has two states `S1` and `S2`, each of them refined by other OR-nodes. In Figure 14 it appears that for each sub-node:

—the local clock of the refined `S1` is the downsampling of the incoming clock at the condition `s = S1`;

—the local control $\texttt{ctrl}_i$ is computed as a function `subctrl` of the control signal received from upper layers, and the actual firing of the transition `t` which can cause reinitialisation of the sub-node;

—in the model of the AND-node, the two latter control signals are forwarded as such to each of the models of the sub-states;

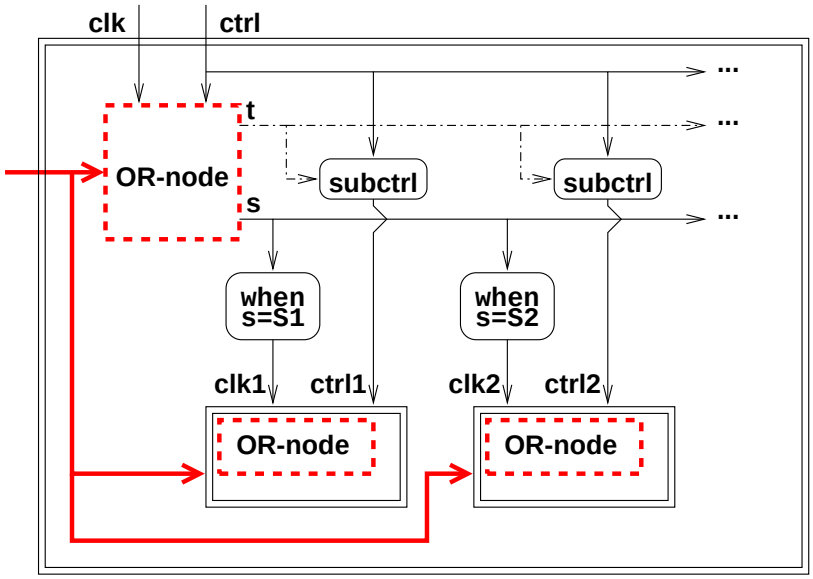

Fig. 14. *An overview of the hierarchical structure of the model*

—it has the structure of the model of an OR-node;

—inputs are forwarded to the or-node as well as to all its sub-nodes.

Not shown in the Figure, for clarity, is that the outputs towards the memory are grouping all contributions from all levels.

Instantaneous states are treated using the general framework, as explained in Section A.3.

5.1.3 *General translation scheme.* The previous example introduced the general translation scheme given here.

Let `default`($\mathtt{a}_1, ..., \mathtt{a}_n$) defined as: a_1 `default` $\mathtt{a}_2$ `default` ... `default` $\mathtt{a}_n$.

Signals `Tick`, `Localclock`, `Control` and `Time` 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 (Or-nodes), in all cases:

```
c_t := c when ( (not ^Control) default LocalClock)
```

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

```
(Stable_s,c,nc):=nextstate{subdefault}(Tick, LocalClock,
                                ^0, Time, default(t_1, ..., t_nbtr))
```

else:

```
(Stable_s,c,nc):=nextstate{subdefault}(Tick, LocalClock,
                        Control, Time, default(t_1, ..., t_nbtr))
```

—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 (^+(t_e_i1, ..., t_e_ip))
       default   Stop when (^+(t_x_i1, ..., t_x_iq))
       default   LocalResume when (^+(t_hi1, ..., t_hir))
       default   Resume when (^+(t_ki1, ..., t_kis))
       default   Control
```

—`in_S`, `en_S` and `ex_S` are special events signalling respectively activity, activation and desactivation of state `S`. Their occurrence instants are represented in Figure 22.

```
(en_S,Stable1) :=  shift(when Control=Start default
                   when Control=Resume default
                   when Control=LocalResume,Tick)
(ex_S,Stable2) :=  shift(when Control=Stop, Tick)
```

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

```
in_Sub_i := when (c=Sub_i)
(Stable_i,output_i) :=
       Sub_i(tick, in_Sub_i, subcontrol_i, input_i)
```

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

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

(there are no sub-processes for nodes that are leaves in the hierarchy).

—Concerning the combination of `Stable` signals from sub-nodes and `shifts`, the part of the design is considered stable if its sub-nodes are stable, and locally memorized signals are; hence,
  —for an And-node a conjunction has to be made (using the synchronous operator `and` is possible, because all sub-nodes have the same clock; alternaltely `when` works too)
  —for an Or-node, only one sub-node at a time is active, hence signals $\texttt{Stable}_i$ have differnet clocks (even exclusive, actually), therefore, instead of conjunction, `default` must be used, giving to the local `Stable` the value of that of the currently active sub-node.

5.2 Transition labels: triggers and actions

The general syntax of the label on a transition in Statecharts 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, then the handling of triggers, and then that of actions.

The same form as for transition labels is also used in static reaction, associating labels to the presence in a state, or mini-specs, associating them to the active status of an Activity in Activitycharts: the translation presented here is valid also in those cases.

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 a Statechart variable is the chart where it is 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_inbcv), tick)
```

The variables is translated into an invocation of the process `shift`, the input of which is the merging of all contributed values; If we want to represent explicitly 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 7.2. 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, there is a set of intermediate signals carrying the contributed values; they have to be given different names, which are derived from the variable name a_i by adding a subscript 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. We use two functions in order to deliver integer indexes associated to variable name a_i: $current(a_i)$ gives the current value of the index associated to a_i and $next(a_i)$ the incremented value of this index. The indexes associated to each variable start at 1. This way we can have as much as necessary intermediate signals carrying intermediate contributions in a transition; the fact that a variables can only appear a bounded number of times in an actions insures that the indexes will remain bounded.

Concerning the extension with control events mentioned in Sections 4.6 and A.2, 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_a_i := default(read_data_a_i1, ...  read_data_a_in1)
```

```
written_data_a_i := default(written_data_a_i1, ...  written_data_a_in2)
```

5.2.2 *Triggers*

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

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

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

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

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

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

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

—in the reactions α handles the translation of the trigger only (see further function β 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:
$\alpha(\epsilon) = \boxed{\texttt{LocalClock}}$
—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:
$\alpha(\text{read}(X)) = \boxed{\texttt{read_X}}$
$\alpha(\text{written}(X)) = \boxed{\texttt{written_X}}$
$\alpha(\text{changed}(X)) = \boxed{\texttt{changed_X}}$
where these events are produced in relation with the management of the variable X (see Section 4.6).
—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):
$\alpha(\text{true}(C)) = \boxed{\texttt{true_}C}$
$\alpha(\text{false}(C)) = \boxed{\texttt{false_}C}$
where these events are produced in relation with the management of the Boolean variable C (see Section 4.6).
—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 4.5).

—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 integrated yet are, for example, the notions of context variables (which can take several values within a step) and loops (`for` or `while` loops) which would involve the definition of a microstep [Nebut 1998].

5.2.3.1 *The clock of actions..* Actions are activated when the transition is actually taken; this activation condition defines the clock of the actions. Special cases like actions on default transitions, on entering or exiting a state, or relative to time, are described in Section A.4.

Given a state named *statename*, as before, with $a(i)$ giving the index in $i = 1..nbac$ of the actions on the transitions (as a difference to static reactions actions $i = 1..nbsr$ etc, see further), and $tr(a(i))$ giving the index in $1..nbtr$ of the transition $\texttt{t}_{tr(a(i))}$ of which it is a label. For each action, we define its clock by the following equation, with:

—event $\texttt{t}_{tr(a(i))}$ 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

—$(\texttt{nc=}target_{tr(a(i))})$ 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 7.2)

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

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

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

$$
\begin{aligned}
\langle Action\rangle \rightarrow\ & \epsilon \\
| \ & \langle EventName\rangle \\
| \ & \langle Variable\rangle := \langle Expression\rangle \\
| \ & read_data(X) \\
| \ & write_data(X) \\
| \ & make_true(C) \\
| \ & make_false(C) \\
| \ & when\langle Event\rangle then\langle Action1\rangle[else\langle Action2\rangle]endwhen \\
| \ & if\langle Condition\rangle then\langle Action1\rangle[else\langle Action2\rangle]endif \\
& \langle Action1\rangle;\langle Action2\rangle
\end{aligned}
$$

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

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

where:

—in a reaction, β 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:

$\beta(\langle EventName\rangle, Clk) = \boxed{\texttt{a}_{next(a)}\ \texttt{:=}\ Clk}$

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:

$\beta(\langle Variable\rangle := \langle Expression\rangle, Clk) =$

$\boxed{\texttt{a}_{next(a)}\ \texttt{:=}\ \alpha(\langle Expression\rangle)\ \texttt{when}\ Clk}$

where 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:

$\beta(\mathsf{read_data}(X), Clk) = \boxed{\texttt{read_data_}X_{next(\texttt{read_data_}X)}\ \texttt{:=}\ Clk}$

$\beta(\mathsf{write_data}(X), Clk) = \boxed{\texttt{written_data_}X_{next(\texttt{written_data_}X)}\ \texttt{:=}\ Clk}$

—action expressions:

—$\beta(\mathsf{when}\ \langle Event\rangle\ \mathsf{then}\ \langle Action1\rangle\ [\mathsf{else}\ \langle Action2\rangle]\ \mathsf{end\ when}, Clk) =$

$$\boxed{\begin{array}{ll} & \beta(\langle Action1\rangle, Clk\ \texttt{when}\ \langle Event\rangle) \\ [| & \beta(\langle Action2\rangle, Clk\ \texttt{when not_event(}\langle Event\rangle\texttt{,tick)})] \end{array}}$$

—$\beta(\mathsf{if}\ \langle Condition\rangle\ \mathsf{then}\ \langle Action1\rangle\ [\mathsf{else}\ \langle Action2\rangle]\ \mathsf{end\ if}, Clk) =$

$$\boxed{\begin{array}{ll} & \beta(\langle Action1\rangle, Clk\ \texttt{when}\ \alpha(\langle Condition\rangle)) \\ [| & \beta(\langle Action2\rangle, Clk\ \texttt{when not}\ \alpha(\langle Condition\rangle))] \end{array}}$$

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

$$\boxed{\texttt{(|}\ \beta(\langle Action1\rangle, Clk)\ \texttt{|}\ \beta(\langle Action2\rangle, Clk)\ \texttt{|)}}$$

5.2.3.4 *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 ^c_t
  |   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 4.5).

5.3 Input-output profile

The structural translation of the Statemate hierarchy into a hierarchy of Signal processes involves defining at each level the input-output 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.

$local(N) = declvar$

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

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

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, Time, Stable`.

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 15, 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`.

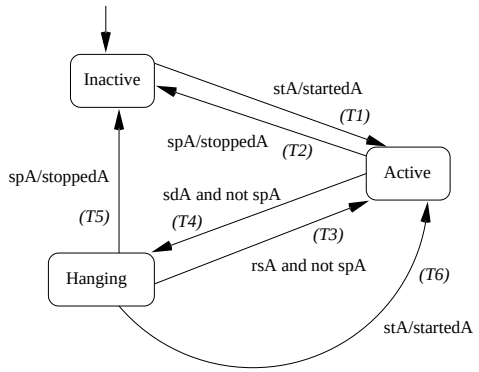

Fig. 15. *States of an activity*

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). The management of the status is as follows:

```
| T1:= transition{Inactive,Active}(c_t, st_A)
| T2:= transition{Active,Inactive}(c_t, sp_A)
| T3:= transition{Hanging,Active}(c_t, rs_A when
                                        Not_Event(sp_A,Tick))
| T4:= transition{Active,Hanging}(c_t, sd_A when
                                        Not_Event(sp_A,Tick))
```

```
| T5:= transition{Hanging,Inactive}(c_t, sp_A)
| T6:= transition{Hanging,Active}(c_t, st_A)
| c_t := c when ((not ^Control) default LocalClock)
| (Stable,c,nc):= nextstate{Inactive}
    (Tick,LocalClock,Control,Time, T1 default T2 default T3
                            default T4 default T5 default T6)
| Started_A := ^(T1 default T6) when (nc=Active)
| Stopped_A := ^(T2 default T5) when (nc=Inactive)
| SubControl:= (Start when (st_A default (Control=Resume))
               default (Stop when sp_A)
               default (Resume when rs_A)
               default Control) when Activated
| Activated:= when (nc=Active)
```

where events st_A, sp_A, rs_A, sd_A can be received from a Statechart defining a control activity. It can be noted that events started_A and stopped_A are produced instantaneously, contrary to the usual event emission mechanism (see Figure 16).

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

```
| (Stable_i,output_i) :=
        Subact_i(Tick, Activated, SubControl, Time, input_i)
```

where they are transmitted the 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 ^control) default localclock)
|   β(MS_i,clockaction_ms(i))
```

where $ms(i)$ is an absolute identification of mini-spec number i, and functions α and β defined for transition labels are reused. These functions have to be extended to a few triggers and actions specific to activities.

As shown above, the initial status of an activity is inactive; at the highest level of a design however, the top-level activity should be constructed according to the same scheme but be initially active.

Activities can have a controlled termination (meaning controlled by the action of an external Control Activity), or an automatic termination, meaning that their Control Activity features a termination connector T, which, when reached, causes the activity to be stopped. The presence of a termination connector in a control activity means that when reaching it the activity it controls must be stopped. This is treated by considering the termination connector like a state, and having the action stop(A) associated to all transitions leading to it.

Concerning the combination of Stable signals from sub-activities, unlike what happens with And-nodes and Or-nodes, the relation between clocks of sub-activities is neither equality nor exclusion, because each of them can be started and stopped independently. But the stability of an activity is defined by that of all its parts. So it is considered unstable (i.e., Stable is *false*) when one of those parts which

are active is unstable. Considering inactive parts as stable (i.e., contributing *true* when $\texttt{Stable}_i$ is absent) produces the right value when making the conjunction by `when` of, for each of them, $\texttt{Stable}_i$ `default true`.

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 α and β in order to encompass them.

6.2.1 *Triggers on activivites.* For each of these dynamic triggers of event or Boolean type, we give its definition:

—started(A), st(A): activity A is started. This event is issued as a result of action `start(`A`)`.
$\alpha(\text{started}(A)) = \boxed{\texttt{started_}A}$

—stopped(A), sp(A): activity A is stopped. It is issued as a result of action `stop(`A`)`.
$\alpha(\text{stopped}(A)) = \boxed{\texttt{stopped_}A}$

—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.
$\alpha(\text{hanging}(A)) = \boxed{\texttt{hg_}A}$

where the two first events are produced as seen above, and the two conditions are produced in relation with the definition of the status of activity A:

```
ac_A := (c = active)
hg_A := (c = 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`:
$\beta(\text{resume}(A), Clk) = \boxed{\texttt{rs_}A_{next(\texttt{rs_}A)} \texttt{ := } Clk}$

The way these events occur is illustrated in Figure 16.

Tick	t	t	t	t	t	t	t	t	...
start(A)	t			t					...
stop(A)			t				t		...
suspend(A)					t				...
resume(A)						t			...
started(A)	t			t					...
stopped(A)			t				t		...
active(A)	t	t		t	t	t			...
hanging(A)					t				...

Fig. 16. *Control of an activity*

6.3 Input-output profile

The signals related to activities also contribute to the computation of input-output 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.

6.4 Activities throughout or within a state

It is a possible in Statemate to associate an Activitychart A with one state S in a Statechart, according to two modes: "throughout" or "within". In the "throughout" mode, entering state S activates A and exiting S deactivates A. This can be encoded by adding the following static reactions to S

```
entering/st_A;;
exiting/sp_A
```

In the "within" mode, only the deactivation of A when exiting S occurs (the activation of A is left to other events). This can be encoded by adding the following static reactions to S

```
exiting/sp_A
```

However, a difference with usual events emission is that these special ones are simultaneously present.

6.5 Combinational assignments

To an activity can be associated combinational assignments, which are equations on the Activitycharts variables, evaluated instantaneously. Their general syntax is:

```
X := Y1 when C1 else
     Y2 when C2 else
     ...
     Yn
```

They are evaluated at the end of the step, taking into account the new values of variables. If a variable defined by a combinational assignment is modified, then other combinational assignments where it appears as an operand are re-evaluated. This can lead to unbounded loops, or even infinite ones (the Magnum environment puts arbitrary bounds in these cases).

In the framework for modelling in a synchronous formalism, unbounded features can not be supported. However for cases such as, for example, the infinite loop `X := X + 1`, or even less trivial ones involving conditionals with `when`, the dependency cycle detection constitutes an analysis and diagnostic of possibly infinite loops, and allows us to recognize them and possibly amend them. The translation of this in Signal is then:

```
X :=  α(Y1) when α(C1) default
      α(Y2) when α(C2) default
      ...
      α(Yn)
```

where the requirement of being instantaneous is taken care of by using simple Signal equations.

7. RELATIONAL ASPECTS IN THE TRANSLATION

As was mentioned in Section 2, one of the distinguishing features of Signal among synchronous languages is that it enables the specification of *relations* between signals, i.e., not only *functions* from inputs and internal state to outputs and a new internal state.

This Section gives indications as to how this particularity can be used to tackle, in the framework on a uniform model, a variety of time granularity levels in Statemate, as well as the representation of non-determinism.

7.1 Execution schemes: superstep, step, microstep

As illustrated in Section 2 with the example of process in Figure 2 and the trace illustrated in Figure 3, it is possible to construct programs where the internal clock is faster than that of inputs. In other words, Signal describes not only reactive systems, but also "pro-active" systems. This feature is homogeneous and can occur with arbitrary depth inside a Signal specification. It can be called up-sampling, or internal acceleration. When this is the case, and the internal state of the process is enough to decide on the synchronization with new inputs, the Signal compiler can also generate executable code for such programs. In other cases, e.g., when several internal accelerations are specified, where clocks can be derived from oneanother, (e.g., two internal clocks at different rates), there is no execution scheme readily available, but the specification can be submitted to the various analysis and verification techniques like clock calculus and model-checking.

In the framework of modelling Statemate in Signal, this feature of Signal is used to open perspectives towards the modelling of different time granularities in Statemate (superstep, step, and microstep inside the actions). It is involved at two different levels: the relations between superstep and step modes, and the relation between the step level and the sequence of instructions in an action (which can be called microsteps), inside a step, especially in case of a *while* loop. These two levels are nested in a uniform way, i.e., they both use the same mechanism.

7.1.1 *Step and superstep.* The superstep mode, as was said in Section 3, consists in the acquisition of inputs, followed by the launching of a series of steps, each of them reacting to the effects of the preceding step; this lasts until no transition can be triggered any more, i.e., until stability. At that point, the systems decides to

synchronize again with its environment in order to acquire the next input.

In previous Sections we have seen how the Boolean `Stable`, standard output in the reactive box model, is evaluated along the structure of the Signal model. It becomes *true* in the absence of internal change, meaning that no new transition is going to be triggered. We can define, by appropriately relating signals `Tick, Time, Stable`, the different execution schemes of Statemate, without changing the internals of the modelling proposed before.

—In the step semantics (sometimes also called GO), inputs (clock `Go`) are considered at each step(clock `Tick`), and time is incremented at each step (`Time`).

```
Tick ^= Time ^= Go
```

—In superstep semantics (sometimes also called GO-REPEAT) stability is reached when no change has occurred at the previous step, implying that no new transition can be triggered:

```
  ZStable := Stable $1 init false
| when ZStable ^= Go
| Tick ^= Stable
```

Signals `Stable` and `Tick` (the latter designating the step) are up-samplings w.r.t. `Go`

—GO-ADVANCE is different from GO-REPEAT in that time is incremented:

```
  ZStable := Stable $1 init false
| when ZStable ^= Go
| Tick ^= ^Stable
| Time ^= Go
```

—GO-STEP is similar to GO without incrementing time:

```
Tick ^= Go
```

—GO-NEXT consists in incrementing time without making a step:

```
Time ^= Go
```

7.1.2 *Step and microstep.* In the action language, there is a sequence operator, noted classically ";". Regarding ordinary variables, it does not actually sequence assignments, as effects of actions are sensed only at the next step. Another kind of variable, called a context variable, is local to each action, and can take several values during one step, in the order specified by the sequence. In particular, in the case of unbounded conditional loops (*while*), context variables are used, notably in the loop condition.

The modelling of such actions, when considering only the bounded sub-language (i.e., involving *for* loops but no *while* loop), can be made as an instantaneous computation, using Signal composition, and encoding the sequence in the data dependencies between operations [Nebut 1998]. For a *while* loop, the idea is to have an up-sampled clock giving one *microstep* at each iteration, and resynchronizing with the sequence when the loop condition is *false*. In a step, there can be a number of transitions taken in parallel; synchronizing the microstep level with the step level involves determining the termination of all the corresponding actions.

This refinement of time granularity involves modifications of the modelling as presented above, in that the management of memorization has to be done differ-

ently. The process `shift` is meant to work at a step clock; for microsteps, updates, memorizing and presence have to dissociated.

This modelling of the sequencing of actions in an unbounded series of instants can be used in other contexts, e.g., the IEC 1131 international standard, where Sequential Function Charts can be evaluated with or without stability research, which is similar to the relation between superstep and step in Statemate, and the Structured Text language shows unbounded loops, involving microstep [IEC 1993].

7.2 Modelling non-determinism

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 possibilities of behavior. Hence, this can be applied to the modelling of non-determinism in Statecharts.

7.2.1 *Conflicting transitions*

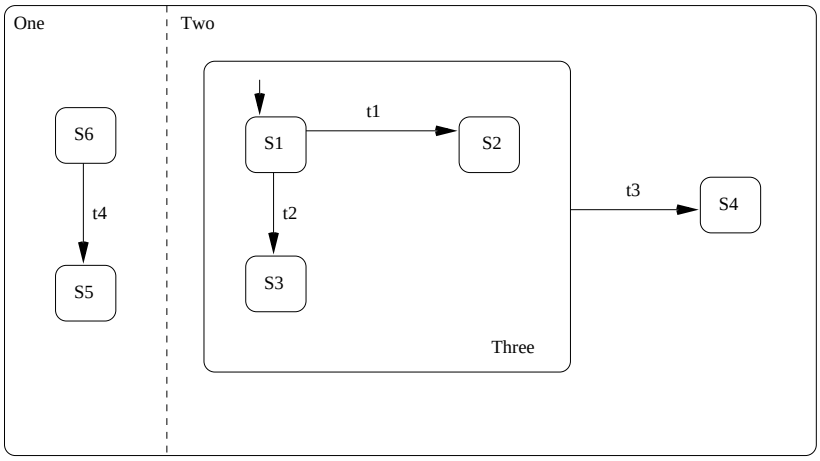

Fig. 17. *conflict*

7.2.1.1 *Example..* For the Statechart of the Figure 17, 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:

```
(Stable,c,nc) := nextstate {S1}
    (tick, localclock, control, time, t1 default t2, ^0)
```

This is the translation scheme that was developed in this paper. We describe how it is possible to represent non-deterministic choices explicitly in Signal. The motivation is to build an exact model of the set of behaviours, for analysis and verification purposes, and to make explicit a non-determinism that can then be solved by modification of the specification.

7.2.1.2 *Non-conflicting sets..* Here we want to deal with situations where more than one transition are enabled at the same instant. Following [Harel and Naamad 1996]:

—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. In the translation shown in the previous Sections, a `default` operator is applied on the $\mathtt{t}_i$ signals; this way it chooses arbitrarily between the non-deterministic possibility.

7.2.1.3 *Representing non-determinism explicitly..* The non-determinism may be represented explicitly by adding a Boolean `K`, that can be called an oracle, to the Signal equation, which chooses between the different enabled transitions:

```
  (c,nc) := nextstate {S1}(tick, localclock, ^0, time, t)
| t := t1 when (K default true) default t2
| t1 ^* t2 ^= t1 ^* t2 ^* 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: the clock calculus [Amagbegnon et al. 1995] will compute

the set of solutions to the system of constraint equations on Booleans and events, and it will consider oracles as free variables. Hence, in the process of analysing, e.g., dependency cycles, or of model-checking dynamic behavioural properties, non-deterministic specifications can be taken into account. When code is generated, no arbitrary execution scheme is chosen: this avoids making implicit choices without approval of the designer.

What can then be done in order to solve 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.

7.2.1.4 *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 18 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 to be executed.

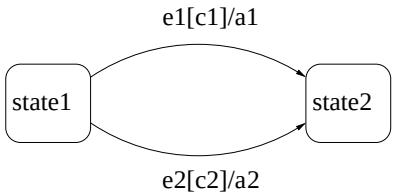

Fig. 18. *Conflicting transitions in Statecharts*

This requires additional information identifying the transitions: if each of them is given a name or index $\mathtt{i}_i$, 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:
$\mathtt{clockaction}_{a(i)}$ `:= when (i =` $\mathtt{i}_i$`)`

7.2.2 *Racing.* Another kind of non-determinism is possible in Statecharts, involving 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 mentioned above are also valid for racing.

This model of non-determinism can be related to external functions and simulated, for example, by the following process, where between two integer values `X`

and Y, the choice is made, when they are both present (X ^* Y), according to a Boolean NDCHOICE which is called for from the outside by the function ALEA_BOOL.

```
process MERGE_INT_EXTERNAL_CHOICE =
  ( ? integer X, Y;
    ! integer Z; )
(| NDCHOICE := ALEA_BOOL{}
 | NDCHOICE ^= (X ^* Y)
 | Z := (X when NDCHOICE) default (Y when not NDCHOICE)
        default X default Y
 |)
where
  boolean NDCHOICE;
  process ALEA_BOOL =
    ( ? ! boolean NDCHOICE; ) ;
end;
```

In summary, our approach allows to statically detect non determinism. If it is desired to simulate non determinism, either the compiler can make implicit choices, or the non determinism has to be kept and solved at the execution through oracles.

8. DISCUSSION

8.1 Related work

Related work can be considered from the general point of view of formalizations of design notations, which is tackled in different ways, using different formalisms; in the case of our work, focussed on Statecharts and Signal, it is also relevant to refer to approaches to multiple formalisms in reactive systems, which are close to ours.

8.1.1 *Formalizations of design notations.* Industrial embedded systems are designed using a number of problem-oriented graphical methods and notations. Some of them, like SA/RT, explicitly address real-time systems. Others, like the PLC (Programmable Logic Controller) language GRAFCET, have a broad user-base in automated manufacturing. A drawback of such notations is that they often lack any explicit semantics, or if they are provided with one it is informal, ambiguous and varies from tool to tool, or from paper to paper. Hence the various approaches to formalizing these notations, which has the advange of stabilizing interpretation, and of supporting automated and powerful analysis, simulation or verification. The variety comes from the notations studied, the formal models used (e.g., Petri nets, finite state automata, process algebra, other high-level languages provided with a formal semantics, etc.) and the kind of analysis and properties aimed at. The practical tools, automating the formalization by a translation, involve taking care in building formal models following structured methods, where composition of models of parts of a specification is handled well, and complexity of the resulting models (in terms of, e.g., state space size) is limited. For example, formalizations of SA/RT feature the one of [Shi and Nixon 1996] in terms of timed Petri nets. It proposes an automated translation with compositionality and complexity improved with respect to earlier models. In particular, quite similarly to our work, it stresses the locality of the translation, where each component of a specification is modelled

independently, and the composition of subnets has the appropriate effect. Another one [Fencott et al. 1994] proposes a model of SA/RT essential models in terms of a process algebra and using the Z formal notation. The semantic function constructed makes it possible to have an automated tool, but can only be informally validated with regard to the definition of SA/RT, the latter being informal.

A more general approach to the problem is given in [Baresi et al. 1997], where the mapping of popular front-end notations to formal models is examined with a special care for flexibility, particularly in the sense of coping with various interpretations of the notations, or various formal models used, and also for back-diagnosis from the model to the notation. Translations are described by sets of rules, customizable according to different semantics of a notation, based on graph grammars, and implemented as an interpreter, with tools for simulation and animation.

Similarities between these works and our approach can be found in the motivations: the clarification of informal or partially formal semantics, and the enabling of automated tool-support for simulation, analysis, and code generation. We also share the care taken in building compositional models, with a structural translation where locally constructed sub-models can be assembled to form the global one. We address flexibility when we propose possible models of non-determinism and memory management: it is a particular case of flexibility, where some parameterization is possible; this way, the model is applicable to Statecharts-like notations with different semantics, as in UML or StateFlow. On the other hand, similarities also include problems like back-diagnosis: when some error or warning is detected on the formal model, how do we formulate it back in terms of the original specification? Also, justifying correctness with regards to the semantics of the original notation under study is difficult, especially when the latter is informal.

The purpose of the work described in this paper is the construction of a synchronous model of Statecharts, a popular and wide-spread notation, in its industrially relevant semantics, i.e., the Statemate one. As such it places itself as a particular case among works on the more general problem of translations between formalisms. Comparisons between StateCharts and other formal notations are also out of the scope of this paper. Technically, we are placing ourselves in the context of multiple reactive formalisms detailed next in Section 8.1.2.

8.1.2 *Multiple formalisms in reactive systems.* Different attempts were made to mix imperative and declarative synchronous languages. In [Maraninchi and Halbwachs 1996], Maraninchi and Halbwachs 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 [SYNCHRON 1995]. Each state of the hierarchical automaton is associated with a Boolean signal which is *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 and declarative formalisms. The translation covers some basic features of the Statecharts: hierarchical parallel automata with event sending as actions. Then, using the semantics of both languages, Maran-

inchi e.a. prove that the translation preserves the behavior from the point of view of traces. A more recent, and also related work, concerns mode-automata, alternating continuous modes following a discrete automaton [Maraninchi and Rémond 1998]. However, the semantics adopted is the Argos one, which is a kind of purely synchronous semantics of Statecharts different from the Statemate one [Harel and Naamad 1996]. Also, many features of the languages of Statemate are absent from Argos.

In [Berry], Berry gives a semantics of the Esterel synchronous language in terms of electric circuits. First, the substatements of an Esterel statement are individually translated into circuits, then the resulting circuits are combined using appropriate auxiliary gates and wiring. Some aspects of the translation are 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 control signals between the subcircuits.

A tool for the integration of different synchronous languages is being developed in the SYNCHRONIE project [Maffeïs and Poigné 1996]. 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 translation of Statecharts into an equational synchronous model presented here might eventually be supported also by SYNCHRONIE.

An extension of the Signal language called Signal*GTi* offers constructs for hierarchical task preemption. In the declarative synchronous language Signal, a process defines a behavior in terms of an unbounded series of instants. However, there are no explicit language constructs for handling the starting, termination, interruption or sequencing of such processes: they are considered to be all active on the whole duration of the execution. 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. Signal *GTi* was implemented by a preprocessor to the Signal environment [Rutten and Martinez 1995] that generates equations for activity management, using additional control signal similar to the reactive box model of Section 4.1.

A synchronous model of Statecharts in Esterel [Seshia et al. 1999] gives access to compilation and verification of specifications with the Statemate semantics; the Esterel compiler can serve as a diagnostic tool to detect non-determinism, and other tools can be used for dynamic properties. An approach to using the declarative Signal language to model Statemate has been studied [Guéguen 97], dealing with the control aspect of hierarchical concurrent automata, but not actions nor Activitycharts. It models states and the transition relation in the Signal equational style.

Also related are Sequential Function Charts (also called GRAFCET), a graphical language with places and transitions, related to Petri nets; they can be modeled in Signal [Marcé and Le Parc 1993; Marcé et al. 1996]. The Sequential Function Charts are part of the international standard IEC 1131 [IEC 1993] of the International Electrotechnical Commission. The norm concerns different aspects of the control of industrial systems using Programmable Logic Controllers (PLC). Part 3 of this

norm describes several programming languages corresponding to different aspects of the design, or to different cultural backgrounds of the designers:

—graphical languages feature:
  —Function Block diagrams, a block-diagram-based formalism,
  —Ladder Diagrams, inherited from relay ladders with which these industrial controllers were implemented and designed before the massive introduction of micro-controllers,
  —and the Sequential Function Charts (SFC).
—textual languages:
  —Instruction List (IL), an assembly language
  —Structured Text (ST), a Pascal-like sequential imperative language.

These styles can be mixed, functions in one language being called from a program in another language. This mixture of styles, as well as the problems of interoperability arising from their co-existence in the same framework, and the cyclic, reactive nature of the overall behaviour, requires that any attempt to model these systems in a synchronous framework, and in particular Signal [Jiménez-Fraustro and Rutten 2001], should resemble the work presented here on Statemate.

Especially, it should have the same specificities as our approach regarding Statemate, which is that we use the relational nature of Signal, i.e., its capacity to represent exactly non-determinism, to be able to handle multi-clocking, and even up-sampling and "internal acceleration" with regard to inputs and outputs.

8.2 Results

8.2.1 *Model and translation..* 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 permit 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 execution architectures, from Statecharts specifications, using the clock calculus and the BDD's techniques of the Signal compiler. Non-determinism may be modelled 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 give access to the already existing Signal tools from a Statechart design.

This translation provides support for co-execution or 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.

The complexity of the translation algorithm, and the size of the constructed model, in number of equations (or lines of Signal), is linear in the size of the source specification, in number of states, including hierarchical OR-states, transitions and variables. In particular, it is not made costly by combinatorial explosion due to parallel automata, because there is no explicit computation or enumeration of the

global state space. This is an advantage of the structural definition of the translation.

8.2.2 *Implementation and code generation..* An implementation of such a translation has been done in C++ in the context of the SAfety CRitical Embedded Systems (Sacres) European Project [SACRES Consortium 1998]. The translation is done in a variant of Signal called DC+ [SACRES Consortium 1997a] which is a common format of the synchronous languages. The Statemate tool from *i-Logix* is used to draw the Statecharts, then the automatic translator uses an API of the Statemate tools in order to extract the needed information from the Statechart design and generate the DC+. This translator is at a prototype level, and a set of small examples have been used for validation of the translation schemes. This work is evolving following a change of focus towards separate compilation and distribution of activities, for which the code can be generated by Statemate. Methods and techniques initially developed for the distribution and semantics-preserving desynchronization of synchronous programs can be applied to Statecharts [Talpin et al. 1999].

Concerning the code generated, we have already described how the translation into Signal keeps the structural hierarchy present in the Statemate specification. The communication of control information along this hierarchy is made explicit with the introduction of new variables, that did not appear explicitly in the Statemate program. However, we have observed that the size of the resulting Signal model, in number of defined signals, is linear in the size of the source specification, in number of states, transitions and defined variables. The code generated from the Signal program follows the general structure of code generated from synchronous programs. It is a single loop, and shows no use of recursion (this is a common requisite for embedded systems). The syntactic structure of the specification is not preserved during the compilation transformations; this is unless it is explicitly required by the user that some part of the structure of activities must be kept in the generated code. Otherwise, it is replaced by the hierarchical semantic structure resulting from the clock calculus. This allows to compute values of variables only at the instants where they need to be calculated. In addition, many optimizations, based on the clock calculus and on rewriting of the conditional dependency graph of the specification can be applied (e.g., elimination of redundant variables, retiming, etc.).

8.3 Perspectives

Perspectives presently worked upon concern: extending coverage of the languages, in order to converge towards a full translator, validating the translation, and treating real-size applications, in order to analyse quantitatively the complexity of the translation. Also, the actual connection of the translation to other tools in the environment brings the question of back-diagnosis, i.e., constructing from the diagnosis of an analysis tool (error message, constraint on signals, etc.) a feedback to the user in terms of the original specification (identifiers, location in the source specification, etc.). Other features of the Statemate languages that should be considered are for instance that 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 [SACRES Consortium 1997b]. 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.

Using the synchronous technology for code reuse and distributed code generation is also possible by modelling the coordination of modules of code generated by Statemate-Magnum, independently of DC+, and represent modules by their input-output profiles, including control information regarding their clocks [Benveniste et al. 1998]. The results obtained in experiments, although relatively limited, have convinced users of the possible industrial validity of the approach: in the ESPRIT project SafeAir (Advanced Design Tools for Aircraft Systems and Airborne Software), which is a successor of the Sacres project, it has been decided to follow a comparable approach from Statemate to SCADE, the industrial version of the synchronous language Lustre, to take advantage of the SCADE *qualified* code generator.

As mentioned earlier, the results of modelling Statemate can be of use when considering other contexts where an equational model is useful, i.e., other graphical, state-based languages, or other contexts where different languages can interoperate; for example, the languages of the international standard IEC 1131 [IEC 1993] of the International Electrotechnical Commission. Another perspective of this work, that follows a comparable but different approach, is to give a semantics of UML state-machines using the synchronous model [Wang et al. 2000]. This modelling results in yet another translation of Statecharts into Signal [Wang 2000].

ACKNOWLEDGMENTS

We wish to thank William W. Wadge for reading and discussing the paper with us, as well as the anonymous reviewers for their contructive comments.

APPENDIX

A. COMPLEMENTS TO THE TRANSLATION OF STATEMATE INTO SIGNAL

A.1 Memorization of events

If `shift` is used with events where there is no value to memorize, a specific version is used, called `shift_event`, where a Boolean is used to memorize presence of the input event:

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

```
  | y := when shift_instant_x
  | Stable := not ^x default tick
  |)
 where boolean instant_x, shift_instant_x;
 end;
```

Here, the shifted event `y` is present at the shifted clock `shift_instant_x`, and `shift_event` behaves like:

```
   y := when yb
 | (yb,Stable) := shift( (x default not tick), tick )
```

A.2 Events associated to variables

We have:

—for all variables:

—`read(X)` (abbreviated in `rd(X)`) emitted when variable `X` is read by action `read_data(X)`: to the basic process `shift`, we must add an input `read_data_X` and the equation:

```
(read_X, Stable) := shift_event(read_data_X, Tick)
```

—`written(X)` (abbreviated in `wr(X)`) emitted when variable `X` is written by action `write_data(X)` or by an assignment (i.e., the clock of presence of `X`): the basic process `shift` needs to be added an input `write_data_X` and the equation:

```
(written_X, Stable) := shift_event(
                       write_data_X default event X, Tick)
```

—`changed(X)` (abbreviated in `ch(X)`) emitted when variable `X` has a value different from that memorized in the `shift` process: the basic process `shift` needs to be added an output `changed_X` and the equation:

```
(changed_X, Stable) := shift_event(
                when not (shift_value_x = value_x), Tick)
```

Concerning the first value of `X`, it is the initialization of `y` in the process `shift` which decides whether or not there is a change. In Statemate-Magnum, integers for example are initialized to 0, hence if the first value given to `X` is different from 0, then there is a change.

—for Booleans only:

—`true(C)` (abbreviated in `tr(C)`) emitted when the condition `C` becomes *true*:

```
(true_X, Stable) := shift_event(
            when not (shift_value_x when value_x), Tick)
```

—`false(C)` (abbreviated in `fs(C)`) emitted when the condition `C` becomes *false*:

```
false_X := shift_event(
            when (shift_value_x when not value_x), Tick)
```

These control events make as much sense in Signal as in Statemate, and it is foreseen that future versions of the Signal compiler will support the computation, based on the clock calculus, of the clocks at which a signal is actually used, or produced. Having these facilities in the compiler would simplify the modelling of the Statemate events.

A.3 Instantaneous States

An instantaneous state may be simultaneously entered and exited (in the same instant). Figure 19 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.

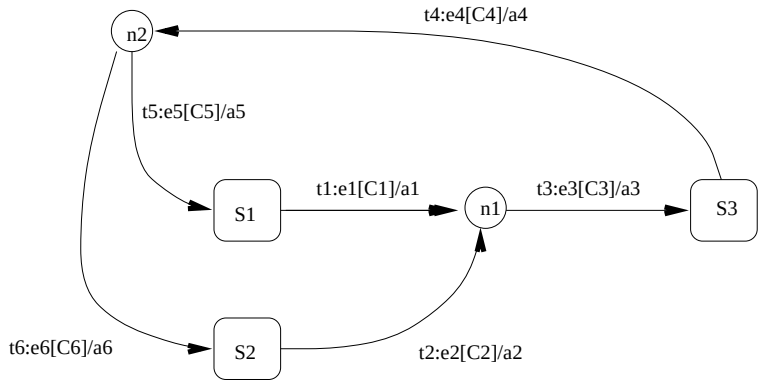

Fig. 19. *A Statechart containing instantaneous states n1 and n2*

In some Statecharts semantics (Statemate for instance [Harel and Naamad 1996]), instantaneous states exist ,and we could handle them in the translation as shown below for the example of Figure 19:

```
  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 ^control) default localclock)
| (c,nc) := nextstate {Sinit} (tick, localclock, control,
                                    time, t3 default t5 default t6)
```

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`), instead 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 19: `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.

A.4 The clock of actions: special cases

Actions on default transitions: these are to be executed on entering the super-state, i.e., the state of which the automaton considered is a sub-automaton. At that instant, the state is re-initialized, and the actions on the default transition executed. This instant is characterized by the `Control` signal as follows:
`clockaction := when Control=Start`

Actions on entering or exiting a state: these are also related to the `Control` signal of the state which is entered or exited. Their mechanism is similar to that of actions associated with the activation or deactivation of an activity, in the way shown in Figure 20.

	Activitycharts	Statecharts
	mini-spec	static reaction
activation	started	entering
deactivation	stopped	exiting

Fig. 20. *Events emitted upon activation and deactivation*

The relation of these events with `Control` goes as follows:
α(entering) = `when Control=Start`
α(exiting) = `when Control=Stop`
α(started) = `when Control=Start`
α(stopped) = `when Control=Stop`
Figure 21 shows a Statechart with two states, for which Figure 22 shows the occurrences of these special events.

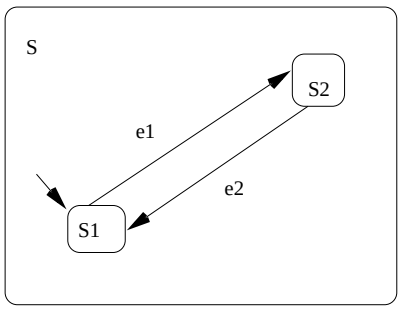

Fig. 21. *Example of a two-state Statechart*

In this example, one sees that:
—`LocalClock` respectiveley for `S1` and for `S2` are complementary, i.e., exclusive, and their union is the `LocalClock` of `S`,
—`exiting` for the state which is left is simultaneous with `entering` for the new state,
—`in(x)` correspond to `LocalClock` for the process encoding state x,
—`entering` and `exiting` are present at the same times as *Start* and *Stop*,

	Tick	t	t	t	t	t	t	t	t	t	...
in S	LocalClock	t	t	t	t	t	t	t	t	t	...
(future)	nc	S1	S1	S2	S2	S2	S2	S1	S1	S1	...
(present)	c	S1	S1	S1	S2	S2	S2	S2	S1	S1	...
(for transition)	c_t	S1	S1	S1	S2	S2	S2	S2	S1	S1	...
	e1			t							...
	e2							t			...
in S1	LocalClock	t	t	t					t	t	...
	Control			Stop				Start			...
	entering							t			...
	exiting			t							...
	in(S1)	t	t	t					t	t	...
	en(S1)								t		...
	ex(S1)				t						...
in S2	LocalClock				t	t	t	t			...
	Control			Start				Stop			...
	entering			t							...
	exiting							t			...
	in(S2)				t	t	t	t			...
	en(S2)				t						...
	ex(S2)								t		...

Fig. 22. *Occurrences of special events (`entering`, `exiting`, etc.) in Figure 21*

—events `en(x)` and `ex(x)` are shifted of one `Tick` w.r.t. `entering` and `exiting` for process `x`.

Actions relative to time: timeouts and schedules are noted: `timeout(e,d)` and `schedule(a,d)`. In Statemate-Magnum, the time referential is given when generating code; it can be an external physical clock (second, millisecond, etc.). In order to support different time units, we have introduced a signal `Time`, propagated through the structural Signal model.
The translation of `timeout(e,d)` uses a counter initialized to `d` upon occurrence of `e`, and then decremented by one at each occurrence of `Time`. The resulting event `timeout(e,d)` is emitted when this couter reaches 0. The counter remains at the value `-1` until afterwards. The translation uses instanciations of the process:

```
process Timeout =
( ? event Time, e; integer d;
  ! event timeout;)
(| timeout := when (counter = 0)
 | counter := d when e default
              (zcounter - 1) when (zcounter > 0)
              default -1
 | zcounter := counter $1 init (-1)
 | counter ^= Time
 |)
;
```

When the environment delivers not an event to be counted but a date, then the process `Dateout` should be used instead:

```
process Dateout =
( ? integer date, e; integer d;
  ! event dateout;)
(| dateout := when (date >= deadline)
 | deadline := (date when e) + (d when e) default
              zdeadline when (0 < date < zdeadline) default -1
 | zdeadline := deadline $1 init (-1)
 | dealine ^= date
 |)
;
```

Modelling `schedule` is less simple because they can be started in unbounded numbers, which falls out of the scope of what can be modelled in a synchronous framework. However, a restriction allowing for only one `schedule` at a time on a given action, makes it possible to have a translation re-using the `Timeout` process:

```
clock_a:=Timeout(Time, schedule, d)
```

where event `schedule` is generated when executing the action `schedule(a,d)`.

REFERENCES

Amagbegnon, T. P., Besnard, L., and Le Guernic, P. 1995. Implementation of the data-flow synchronous language signal. In *Proceedings of the ACM Symp. on Programming Languages Design and Implementation, PLDI'95* (1995), pp. 163–173. ACM.

Armstrong, J. 1996. Safecharts handbook. Technical Report DCSC/TR/95/7, Dependable Computing Systems Centre, University of York.

Armstrong, J. 1998. Industrial integration of graphical and formal specification. *Journal of Systems Software 40*, 3, 211–225.

Baresi, L., Orso, A., and Pezzè, M. 1997. Introducing formal specification methods in industrial practice. In *Proceedings of the International Conference on Software Engineering, ICSE'97,* Boston, Massachussets (1997), pp. 56–66.

Benveniste, A., Gautier, T., Le Guernic, P., and Rutten, E. 1998. Distributed code generation of dataflow synchronous programs: the SACRES approach. In *Proceedings of The Eleventh International Symposium on Languages for Intensional Programming, ISLIP'98* (Sun Microsystems, Menlo Park, California (USA), May 1998). `http://www.inria.fr/ep-atr`, publications.

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

Berry, G. and Gonthier, G. 1992. The esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming 19*, 87–152.

Fencott, P., Galloway, A., Lockyer, M., O'Brien, S., and Pearson, S. 1994. Formalising the semantics of ward/mellor sa/rt essential models using a process algebra. In *Proceedings of Formal Methods Europe, FME'94*, Volume 873 of *Lecture Notes in Computer Science* (1994), pp. 681–702. Springer-Verlag.

Gautier, T., Guernic, P. L., and Maffeïs, O. 1994. For a new real-time methodology. Research Report 2364 (Oct.), INRIA. `http://www.inria.fr/RRRT/RR-2364.html`.

Grazebrook, A. 1997. Sacres - formalism for real projects. In F. Redmill and T. Anderson Eds., *Safer Systems* (London, 1997). Springer-Verlag.

Guéguen, H. 97. Mixing statecharts and signal for the specification of control. In IFAC Ed., *IFAC Workshop AARTC97 : Algorithm and Architecture for real time control, Vilamoura* (Avril 97).

HALBWACHS, N., CASPI, P., RAYMOND, P., AND PILAUD, D. 1991. The synchronous dataflow programming language Lustre. *Proc. of the IEEE 79*, 9 (Sept.), 1305–1320.

HAREL, D. 1987. Statecharts : A visual formalism for complex systems. *Science of Computer Programming 8*, 231–274.

HAREL, D. AND NAAMAD, A. 1991. The languages of Statemate. *i-Logix Inc.*

HAREL, D. AND NAAMAD, A. 1996. The Statemate semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology 5*, 4 (Oct.), 293–333.

IEC. 1993. International standard for programmable controllers: Programming languages. Technical Report IEC 1131 part3, IEC (International Electrotechnical Commission).

JIMÉNEZ-FRAUSTRO, F. AND RUTTEN, E. 2001. A synchronous model of the iec 61131 plc languages fbd and st. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems, ECRTS'01,* June 13th-15th, 2001, Delft, The Netherlands (2001).

LE GUERNIC, P., GAUTIER, T., LE BORGNE, M., AND LE MAIRE, C. 1991. Programming real-time applications with Signal. *Proceedings of the IEEE 79*, 9 (Sept.), 1321–1336.

M. VON DER BEECK. 1994. A Comparison of Statecharts Variants. In H. LANGMAACK, W.-P. DE ROEVER, AND J. VYTOPIL Eds., *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Volume 863 of *Lecture Notes in Computer Science* (Lübeck, Germany, Sept. 1994), pp. 128–148. Springer-Verlag.

MAFFEÏS, O. AND POIGNÉ, A. 1996. Synchronous automata for reactive, real-time or embedded systems. Technical report (Jan.), GMD. no 967.

MARANINCHI, F. AND HALBWACHS, N. 1996. Compiling ARGOS into Boolean equations. In B. JONSSON AND J. PARROW Eds., *Formal Techniques in Real-Time and Fault-Tolerant Systems, Uppsala, Sweden*, Volume 1135 of *Lecture Notes in Computer Science* (Sept. 1996), pp. 72–90. Springer-Verlag.

MARANINCHI, F. AND RÉMOND, Y. 1998. Mode-automata: about modes and states for reactive systems. In C. HANKIN Ed., *Programming Languages and Systems, Proceedings of the 7th Eropean Symposium on Programming, ESOP'98, Lisbon, Portugal, march–april 1998*, Volume 1381 of *Lecture Notes in Computer Science* (March 1998), pp. 185–199. Springer-Verlag.

MARCÉ, L., L'HER, D., AND LE PARC, P. 1996. Modelling and verification of temporized grafcet. In *Proceedings of The IEEE/SMC Symposium on Discrete Events and Manufacturing Systems, Lille (France)* (July 1996).

MARCÉ, L. AND LE PARC, P. 1993. Defining the semantics of languages for programmable controllers with synchronous processes. *Control Engineering Practice 1*, 1 (Feb.).

NEBUT, M. 1998. Modélisation de statemate en signal : le langage impératif des actions. Master's thesis, Université de Rennes 1, IFSIC.

PERRAUD, J., ROUX, O., AND HUON, M. 1992. Operational semantics of a kernel of the language ELECTRE. *Theoretical Computer Science 97*, 1 (April), 83–104.

RUTTEN, E. AND MARTINEZ, F. 1995. Signal GTi, implementing task preemption and time intervals in the synchronous data flow language Signal. In *Seventh Euromicro Workshop on Real-Time Systems* (June 1995), pp. 176–183. (IEEE Publ.) `http://www.inria.fr/ep-atr`, publications.

SACRES CONSORTIUM. 1997a. The common format of synchronous languages - the declarative code DC+ version 1.4. Technical report (November), Esprit Project SACRES EP 20897.

SACRES CONSORTIUM. 1997b. The semantic foundations of SACRES. Technical report (March), Esprit Project SACRES EP 20897.

SACRES CONSORTIUM. 1998. Task I1.1.A: Statemate integration — the *stm2dcplus* translator. Technical report (Nov.), Esprit Project SACRES EP 20897.

SESHIA, S., SHYAMASUNDAR, R., BHATTACHARJEE, A., AND DHODAPKAR, S. 1999. A translation of statecharts to esterel. In *Proceedings of the World Congress on Formal Methods, FM'99, Volume II, Toulouse, France, September 20-24, 1999*, Number 1709 in Lecture Notes in Computer Science (LNCS) (1999), pp. 983–1007. Springer Verlag. LNCS nr. 1709.

SHI, L. AND NIXON, P. 1996. An improved translation of SA/RT specifications model to high-level timed Petri nets. In *Proceedings of Formal Methods Europe, FME'96*, Volume

1051 of *Lecture Notes in Computer Science* (1996), pp. 518–537. Springer-Verlag.

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

Talpin, J., Benveniste, A., Caillaud, B., and Guernic, P. L. 1999. Hierarchic normal forms for desynchronization. Research Report 3822 (Dec.), INRIA.

Wang, Y. 2000. Compilation of state-machines using behavior expression. In *Proceedings of the* Workshop PhDOOS2000 in 14th European Conference on Object-Oriented Programming (2000).

Wang, Y., Talpin, J., Benveniste, A., and Guernic, P. L. 2000. Compilation and distribution of state-machines using spots. In *Proceedings of the* 16th IFIP World Computer Congress 2000 (Aug. 2000).