Safety Critical Embedded Systems Design : the SACRES approach [1] [2]

Albert Benveniste [3]

May 14, 1998

[1]This work is supported by the Esprit project R&D -SACRES (EP 20897).

[2]In addition to the listed authors, all participants to the SACRES project are gratefully acknowledged. SACRES members are : Siemens (Lead partner), i-Logix, TNI (Techniques Nouvelles d'Informatique), OFFIS, INRIA, the Weizmann Institute of Science, British Aerospace, SNECMA.

[3]The following people have also contributed to the writing of this paper. These are : Michael Siegel (Weizmann Institute), Leszek Holenderski and Klaus Winkelmann (Siemens), Eric Sefton (BAe), Eric Rutten, Paul Le Guernic, and Thierry Gautier (Irisa).

Abstract

The SACRES project is an Esprit R&D project, SACRES members are: Siemens (Lead partner), i-Logix, TNI (Techniques Nouvelles d'Informatique), OFFIS, INRIA, the Weizmann Institute of Science, British Aerospace, SNECMA. As its name indicates, SACRES is dealing with the development of new design methodologies and associated tools for safety critical embedded systems. Main targeted users are aeronautics, automobile, process control and energy. Emphasis has been put on formal techniques for modular verification of the specifications, distributed code generation, and generated code validation against specifications. These techniques aim at making more flexible the exploration of the software life cycle. Verification of the specifications and generated code validation aim at helping for certification of the overall design. Distributed code generation aims at reducing the dependency of the design with respect to the target supporting architecture. In all cases, modularity helps reuse of existing designs, and makes it possible to address much larger systems. A central item of SACRES is the DC_+ format for synchronous languages, which provides the common semantic framework for all tools as well as end user specification formalisms (StateMate and Sildex/Signal).
Keywords: embedded systems, safety critical systems, modular verification, distributed code generation, code validation and certification, synchronous languages, Statecharts, Signal.

Contents

Chapter 1

Overall presentation

Here we discuss overall objectives of SACRES, as related to the design of safety critical embedded systems. We briefly review SACRES architecture, and how this architecture helps meeting the objectives. Then we briefly review what is the design process methodology which serves as a reference for SACRES, the main reference here is the DO178B recommandation in aeronautics. We discuss how SACRES methodology conforms with but also slightly deviates from such official recommendations, with aiming at modularity and reuse, higher safety in software design, and more generally higher efficiency.

1.1 SACRES objectives and architecture

The overall objective of the SACRES project is to provide designers of embedded control systems, in particular safety critical systems, with an enhanced design methodology supported by a toolset significantly reducing the risk for design errors and shortening overall design times. Where and how SACRES contribution fits is illustrated in the V-diagram of Figure 1.1.

In this diagram, we have also shown the companion activities related to the design of control and information processing functions. In particular, these activities play a prominent role in shifting from subsystem requirements to software requirements. Helping for a smooth transition from subsystem requirements to software requirements is one major concern of safety critical systems engineering. One contribution of SACRES to this concern is the development of a pivot formalism called $\mathrm{D}\textsc{c}_{+}$ [12], which would facilitate such a transition — $\mathrm{D}\textsc{c}_{+}$ is a natural candidate for modelling the sampled

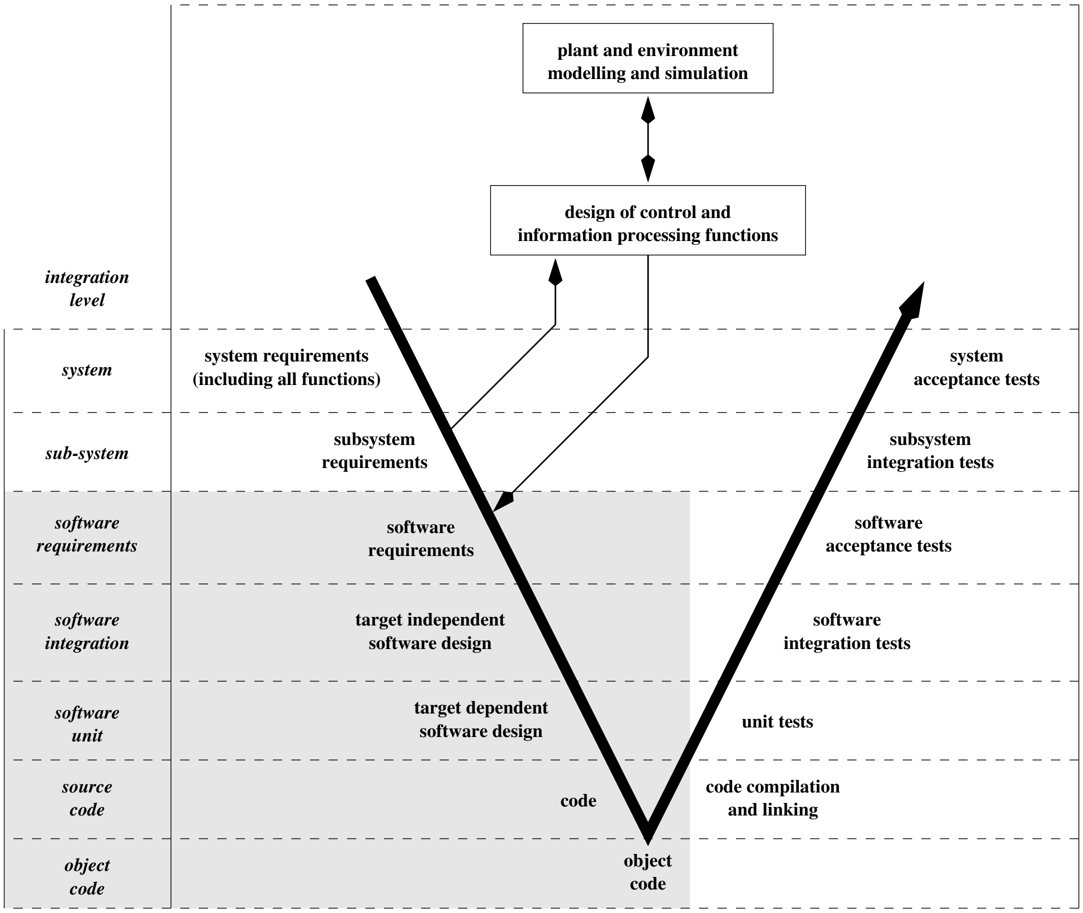


Figure 1.1: *Extended V-diagram.* The scope of SACRES is shown in grey.

time part of control engineering diagrams such as provided by, e.g., Simulink or Matrix$_x$.

Note the following: normally, when drawing the V-diagram, the source code and object code integration levels are left out. Human involvement is not significant in these steps because the translation process is mostly automatic. In the SACRES context, however, because of the emphasis on safety-critical applications and formal verification, the association between source code and object code representations needs to be considered. It is an objective of SACRES to facilitate the certification of the whole design process by means of cross-checking the object code against the source code, this is called *code validation.*

Although test is an important activity in the V-diagram, SACRES will not contribute to it per se. However, by providing improved *automatic code generation* (both target architecture independent and dependent), and by offering improved facilities for formal *verification and validation* at each stage of the left branch of the V-diagram, the cost of subsequent test will be drastically reduced.

Also, as software requirements for embedded systems typically involve a mixture of state oriented and dataflow oriented descriptions, SACRES offers the possibility of *mixed formalism* design, by combining expressive power of Statecharts from the StateMate tool, and Signal based dataflow diagrams from the Sildex tool [1]. Integration is tightly offered thanks to the Dc_+ pivot formalism. Finally, the dataflow diagram editor is also useful for describing target architectures for (possibly distributed) code generation.

SACRES architecture is depicted in Figure 1.2. It is designed according to SACRES objectives. SACRES project is based on various tools. Some were available before the project started, and benefited from the project by various enhancements. Some have been developed entirely within the project. There has also been a lot of work integrating the various tools. Here is a description of the main groups of tools:

- The command-level interface permits the tools to be invoked. Some of the tool lanch facilities are actually available directly through the menus from the specification tools StateMate and Sildex.

- The specification front-end tools are mostly self-contained graphical tools for building models. The ssl function allows to assembling models with components from different specification tools. ssl is available as a textual language, but it is also integrated as part of the graphical editors of some of the specification formalisms — e.g., Sildex diagrams can be used to describe such assembly.

- The Dc_+ pivot formalism is a global model format [2] implementing the paradigm of synchronous programming in its full generality. Dc_+ for-

[1] StateMate is a visual tool offering Statecharts, Activity Charts, and Module Charts for specifying systems, it is marketed by iLogix, Andover, MA, USA, `http://www.ilogix.com`. Sildex is a Signal-based graphical tool marketed by TNI, Brest, France, `http://www.tni.fr`.

[2] Dc_+ is originating from a previous version designed in the framework of the Eureka-Synchron project on synchronous languages.

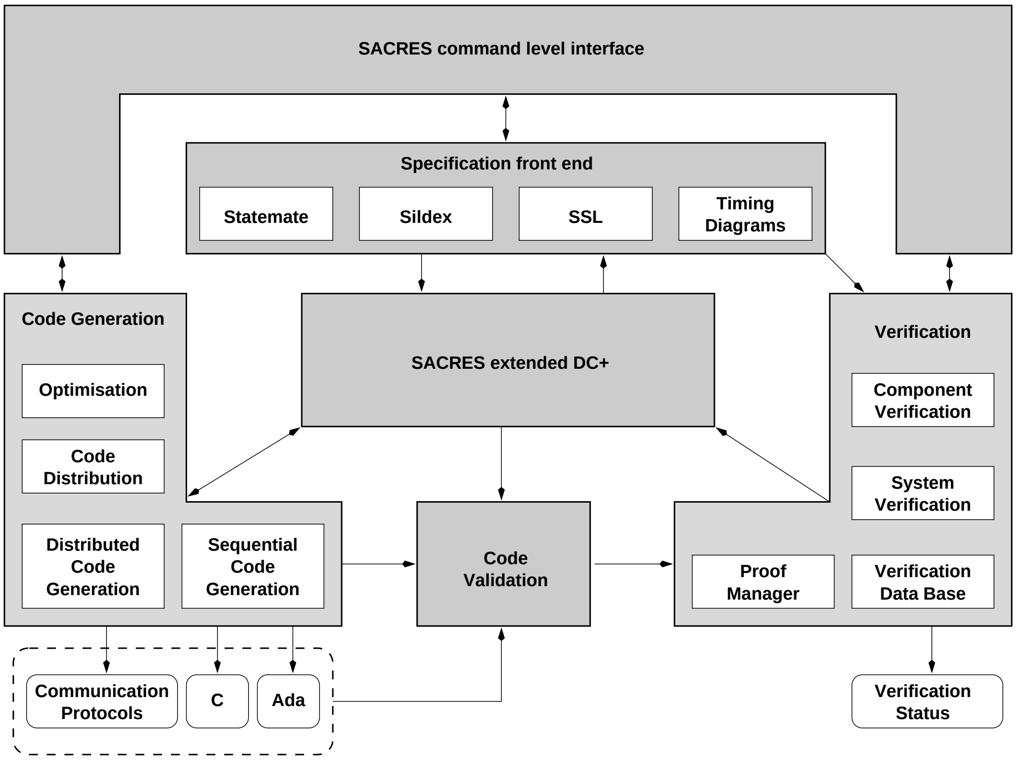


Figure 1.2: SACRES *architecture.*

mat is not expected to be read by the user, just to be passed between tools in the tool-set.

- The code generation tool includes an interface for defining target architecture and code distribution as well as the code-generation engine itself.

- Code validation provides a formal correctness check for the code generation translation.

- The verification tool allows automatic checking of the models that are produced from the specification tools, and manages the results of the proof.

Referring to the V-diagram of Figure 1.1, the matching between SACRES objectives and architecture is clear. The following is worth noticing. The

$\mathrm{D}\textsc{c}_+$ pivot format is a natural candidate for linking control engineering tools to the SACRES tool-set. This would provide a smooth transition along more stages of the V-diagram, by integrating control and software engineering together in the design process.

1.2 SACRES methodology

The SACRES methodology is illustrated in the diagrams of Figures 1.3 and 1.4. The figure 1.3 better explains how the V-diagram design cycle of

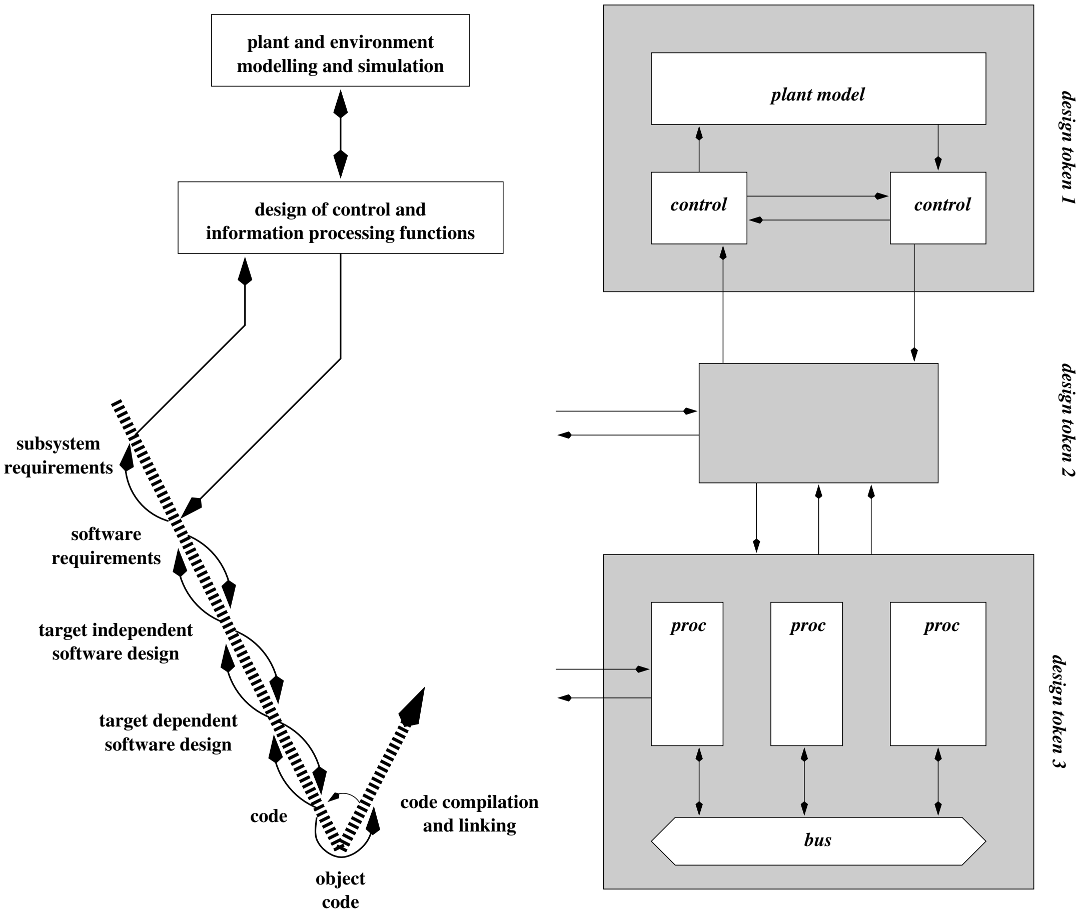


Figure 1.3: *From V-diagram to iterative, and modular design.* Tokens are attached to different subsystems. Tokens move independently along the V-diagram.

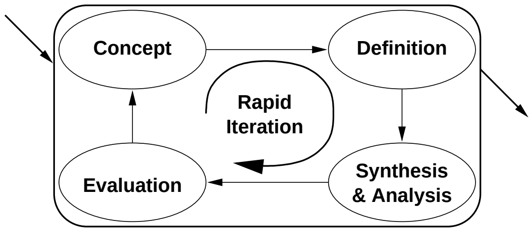


Figure 1.4: *Exploration cycle.* Each token is moved along the V-diagram independently by applying successive exploration cycles.

Figure 1.1 is actually implemented. When creating deliverables within the V-diagram, it is possible to form each deliverable completely, before progressing to the next stage. However, this leads to a situation where the whole breadth of the design is maturated slowly and therefore the discovery of a problem in a late life-cycle phase which was introduced in an earlier phase can cause significant rework of large parts of the design. The iterative (or spiral) development process allows different components of the project to be developed to different levels of detail. The principle behind iterative design is that the hardest or most central elements of the design are explored first, possibly all the way through to implementation.

The diagram of Figure 1.3 illustrates this. While the original V-cycle is shown in dashed, the iterative cycle is depicted in curved arrows, and its detail is shown in Figure 1.4. The idea is to dynamise the V-diagram by attaching a token to each different subsystem, and to move tokens down the V independently, while maintaining consistence of the overall design. To illustrate this, the architecture diagram on the right hand side of the figure 1.3 shows a typical situation in which different subsystems of the project are developed to different levels of detail (thus a design token is attached to each such subsystem). This diagram is a data-flow diagram : three interacting subsystems are shown in grey (top, middle, and bottom), which are at different stages of their design. The grey box in the middle is a component kept at a coarse level of abstraction. The topmost box shows a part of the design in which control engineering is performed : a controller software model is running in closed feedback loop with a model of the plant. The bottom box shows a part of the design in which description occurs at a detailed architectural level : the supporting architecture is depicted (a nice bus is seen) and the functions each processor is implementing is detailed, models of communication media are included at a certain level of abstraction.

To help for the designer to follow this iterative cycle, SACRES offers the following, cf. Figure 1.2:

A semantic based pivot format: DC$_+$. This format implements the paradigm of synchronous programming in its full generality. It plays a central role in maintaining the soundness all the way along the design cycle, and supports the semantic integration of subsystems at different stages of their design. It could be used as a link between the SACRES tool-set and control engineering CAD environments.

Specification tools offering animation and simulation capabilities. These are the StateMate and Sildex tools.

System verification. Each component or subsystem at any stage of its design can be checked against certain required properties. And properties of the system can be inferred from properties of their subdesigns.

Abstraction mechanisms. Each component or subsystem at any stage of its design can be abstracted to (an approximation of) its interface behaviour. This is made feasible thanks to the particular features of the DC$_+$ format, see chapter 2.

Architectural modelling and distributed code generation. Automatic code generation plays of course a central role in implementing the design cycle of the Figure 1.3. More importantly, target dependent code generation is supported, including distributed asynchronous architectures.

Validating code generation. Official process development models such as DO178B and Def Stan 00-55 impose strong demand on how each move is performed throughout the design process. "Evidence" of correctness must be provided in support of the design. In support for this, SACRES offers code validation, a procedure in which object code can be checked against its associated source code. This involves checking for refinement correctness against compiler optimisations.

1.3 Case studies

Two major user case studies have been conducted within the framework of SACRES, these were 1/ a Generic Stores Management System, proposed by British Aerospace, and 2/ an aircraft Engine Control System and a Digital Control Unit for an helicopter, both proposed by Snecma. For each of them, the following history was more or less followed :

- For each case study, an early definition with coarse objectives was available at the beginning of the project. Focus increased in the course of the project, with both a refined definition of the applications, and more precise definition of the objectives.
- For each application, first experiments involved the part of the SACRES tool-set that was already familiar for the SACRES user who offered the trial. Subsequently, involvment of the remainder of the tool-set occurred, at a later stage of the project.

In addition to the above main case studies, several trials were performed. These are :

- an helicopter alarm monitoring system, proposed by ESG GmbH,
- a pickling line/tandem cold-rolling mill, from steel industry, proposed by Siemens,
- some systems from car body electronics, proposed by BMW, also by Siemens Automotive
- cruise control, proposed by Renault.

The two major case studies are summarized now.

1.3.1 Generic Stores Management System (British Aerospace)

The application proposed by British Aerospace is a generic Stores Management System (SMS), for an aircraft. This system is installed within the airframe and is responsible for the arming, pre-conditioning, release, and jettison of all stores carried by the aircraft (including arms and fuel tanks). Different phases are considered, namely : aircraft on ground, aircraft on flight, aircraft approaches target. Two main operating modes are considered: normal and jettison.

The Generic SMS project is concerned with demonstrating the viability of designing the core of an SMS which can be configured to suit different aircraft and different stores. The ideal Generic SMS would be capable of being fitted to any aircraft and manage any store configuration without any alteration to the SMS. Of course we must recognise that the diverse requirements of both aircraft and stores, the introduction of new aircraft and stores and changing standards would not allow this ideal SMS to be a practical proposition. Rather, the Generic SMS must provide for a degree of configurability which is as large as practicable and an upgrade path that facilitates the addition of new functionality to the SMS. The Generic SMS would be configurable at design time. By this we mean that there will be an SMS design which can in itself cannot be implemented and installed onto an aircraft. Rather, this design will be taken and configured by the designer, by putting values to parameters provided by the generic design and adding new functionality using techniques identified by the generic design.

Hence the Generic SMS case study is typical of a study focussing at the specification phase. Specification was undertaken using StateMate. The main question was thus to check the satisfaction of some properties of the following types: safety, liveness, and bounded time safety. The latter type involves reasoning on quantitative time. As the size of the considered system was too large for a global handling of some of the proofs by the SVE-BDD tool (Siemens Verification Engine [30, 31, 32, 33]), *system verification* was performed, cf. section 3.1. In this way, the requested properties were verified, in a modular way.

Interestingly enough, one property was found which could possibly be violated by the design, this was due to the inadequate handling of a very unlikely configuration for the pilot actions (simultaneity of actions that are normally undertaken in sequence, but could happen simultaneously, although very unlikely so).

1.3.2 A Digital Control Unit application (Snecma)

The DCU is an airframe mounted electronic package with a microprocessor based programmable digital control, for a dual engine helicopter. The DCU interfaces with cockpit controls and displays and has a self-test feature.

The function of the DCU is to provide control of engine operation in order to obtain a safe steady state and a transient engine performance. It shall provide full authority control of fuel flow and power matching, combined with a manual back-up.

DCU is a mock-up which has to provide control of engine operation in a test bench environment. It interfaces the hydro-mechanical unit which will be slightly modified in order to have a full authority control. The DCU will include some technical control functions which are actually completed by the hydro-mechanical unit.

The main functions implemented by this DCU are: Start, Twin engine load sharing, Power turbine speed control, Transient Control, Limitations control, and Bleed valve control.

The control computer is based on a MC68332 chipset, without floating point unit. The operating system software is coded manually with assembler (low level interface hardware) and ADA (middle level interface hardware).

In a first series of experiments, the SIGNAL language was used for specifying the application software and for automatic code generation, using the SILDEX environment. The specification is based on a mixed formalism involving grafcets and SIGNAL dataflow diagrams, both available within the SILDEX editor. The following was performed on this case study:

Distributed code generation, using the above indicated target architecture, cf. chapter 4. The target architecture is Posix (using processes and threads), with C.

Verification of properties at the specification level. As corresponding properties involved only small parts of the system, system verification was not exercised in this first phase.

Code validation. This was the first significant experiment of validating DC_+ format generated by the SILDEX tool, against generated (monolithic, not distributed) C code. The generated C code was a significant program (about 30,000 lines), and it is believed that validating such a large piece of code was never performed before, cf. chapter 5.

In a second stage (not reported here), mixed formalism design (using jointly STATEMATE and SILDEX) with system verification is to be performed on the other Snecma application, a flight engine control.

Chapter 2

Symbolic Transition Systems (STS) : concepts, objects, and notations

From now on, we review the fundamental bases for SACRES. As SACRES relies on a strong formal modelling basis, we spend significant effort discussing related issues in the chapter devoted to the model of Symbolic Transition Systems. This abstract model is used to introduce the basics of the Dc_+ format. Then, using this general framework as a common basis, we review verification, code generation, and code validation, from a more theoretical standpoint. Fully theoretical aspects and results are not given here, however, but appropriate references are provided.

2.1 The essentials of the synchronous paradigm

There has been several attempts to characterize the essentials of the synchronous paradigm [6] [13]. With some experience and after attempts to address the issue of moving from synchrony to asynchrony (and back), we feel the following features are indeed essential for characterizing this paradigm :

1. Programs progress via an infinite sequence of *reactions*:

$$P = R^\omega$$

where R denotes the family of possible reactions[1].

2. Within a reaction, decisions can be taken on the basis of the *absence* of some events, as exemplified by the following typical statements, taken from ESTEREL, LUSTRE, and SIGNAL respectively:

```
present S else 'stat'
y = current x
y := u default v
```

   The first statement is selfexplanatory. The "`current`" operator delivers the most recent value of `x` at the clock of the considered node, it thus has to test for absence of `x` before producing `y`. The "`default`" operator delivers its first argument when it is present, and otherwise its second argument.

3. When it is defined, parallel composition is always given by taking the conjunction of associated reactions:

$$P_1 \| P_2 \;=\; (R_1 \wedge R_2)^\omega$$

   Typically, if specifying is the intention, then the above formula is a perfect definition of parallel composition. In contrast, if programming was the intention, then the need for this definition to be compatible with an operational semantics very much complicates the "when it is defined" prerequisite[2].

Of course, such a characterization of what the synchronous paradigm is makes the class of "synchrony–compliant" formalisms much larger than usually considered. But it has been our experience that these were the key features for the techniques we have developed so far to work.

Clearly, these remarks call for a common format implementing this paradigm, the DC/DC$_+$ format [12] has been proposed with this objective, see section 2.5. Also, this calls for a simplest possible formalism with the above features, on which fundamental questions should be investigated (the purpose of this basic synchronous formalism would not be to allow better specification or

[1]Reactive systems continuously observe their environment, they compute outputs and perform actions in reaction to the reception of stimuli from the environment.

[2]For instance, most of the effort related to the semantics of ESTEREL has been directed toward solving this issue satisfactorily [9].

programming, however): the STS formalism we describe next has this in its objectives.

2.2 Specification: Symbolic Transition Systems (STS) [19]

Symbolic Transition Systems (STS). We assume a vocabulary $\mathcal{V}$ which is a set of typed variables. All types are implicitly extended with a special element $\perp$ to be interpreted as "absent". Some of the types we consider are the type of *pure signals* with domain $\{\mathrm{T}\}$, and *booleans* with domain $\{\mathrm{T}, \mathrm{F}\}$ (recall both types are extended with the distinguished element $\perp$).

We define a *state* s to be a type-consistent interpretation of $\mathcal{V}$, assigning to the set of all variables, a value for it over its domain. We denote by S the set of all states. For a subset of variables $V \subseteq \mathcal{V}$, we define a V-state to be a type-consistent interpretation of V. Thus a V-state assigns to the set V a value $s[V]$ for it over its domain; also, for $v \in V$ a variable, we denote by $s[v]$ its interpretation by state s.

Following [19][3], we define a *Symbolic Transition System* (STS) to be a system

$$\Phi = \langle V, \Theta, \rho \rangle$$

consisting of the following components:

- V is a finite set of typed *variables,*
- $\Theta(V)$ is an assertion characterizing *initial states.*
- $\rho = \rho(V^-, V)$ is the *transition relation* relating past and current states s^- and s, by referring to both past[4] and current versions of variables V^- and V. For example the assertion $x = x^- + 1$ states that the value of x in s is greater by 1 than its value in s^-. If $\rho(s^-[V], s[V]) = \mathrm{T}$, we say that state s^- is a *ρ-predecessor* of state s.

[3] The talented reader will also notice some close relation to the TLA model of L. Lamport.

[4] Usually, variables and *primed* variables are used to refer to current and *next* states. This is equivalent to our present notation. We have preferred to consider s^- and s, just because the formulas we shall write mostly involve current variables, rather than past ones. Using the standard notation would have resulted in a burden of primed variables in the formulas.

A *run* $\sigma : s_0, s_1, s_2, \ldots$ is a sequence of states such that

$$s_0 \models \Theta \;\bigwedge\; \forall i > 0\ ,\ (s_{i-1}, s_i) \models \rho \tag{2.1}$$

The *composition* of two STS $\Phi = \Phi_1 \bigwedge \Phi_2$ is defined as follows:

$$\begin{array}{rcl} V & = & V_1 \cup V_2 \\ \Theta & = & \Theta_1 \wedge \Theta_2 \\ \rho & = & \rho_1 \wedge \rho_2\ , \end{array}$$

the composition is thus the pairwise conjunction of initial and transition relations. Note that, in STS composition, interaction occurs through common variables only. Hence variables that are declared private to an STS will not directly contribute to any external interaction.

Notations for STS: we shall use the following generic notations in the sequel:

- $c, v, w, \ldots$ denote STS *variables,* these are the declared variables of the STS; useful additional variables are the following.
- for v a variable, $h_v \in \{\mathrm{T}, \perp\}$ denotes its *clock*:

$$[h_v \neq \perp] \;\; \Leftrightarrow \;\; [v \neq \perp]$$

- for v a variable, ξ_v denotes its associated *state-variable,* defined by:

$$\begin{array}{lll} \textbf{if}\ \ h_v & \textbf{then} & \xi_v = v \\ & \textbf{else} & \xi_v = \xi_v^- \end{array} \tag{2.2}$$

  Values can be given to $s_0[\xi_v^-]$ as part of the initial condition. Then, ξ_v is always present after the 1st occurrence of v. By convention, ξ_v is private to the STS in which it is used. Thus state-variables play no role in STS-composition. Also, note that $\xi_{\xi_v} = \xi_v$, thus "state-state-variables" need not to be considered.

Transition relations for STS are naturally specified using conjunction of predicates. The following set of predicates will be repeatedly used:

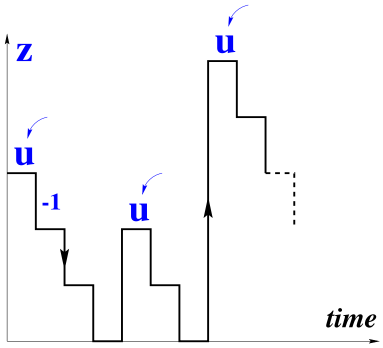
$$\begin{array}{lll} \text{a selector} & : & \textbf{if}\ \ b\ \ \textbf{then}\ \ z = u\ \ \textbf{else}\ \ z = v\ . \\ \text{a register} & : & \textbf{if}\ \ h_z\ \ \textbf{then}\ \ v = \xi_z^-\ \ \textbf{else}\ \ v = \perp \end{array}$$

For the selector, note that the "**else**" part corresponds to the property "$[b = \text{F}] \vee [b = \perp]$". The more intuitive interpretation of the second statement is "$v_n = z_{n-1}$", where index "n" denotes the instants at which both v and z are present (their clocks are equal). Clearly, this models a register. This statement implies the equality of clocks: $h_z \equiv h_v$. The following synchronization constraint is also a predicate of interest:

$$\text{a synchronization constraint} \quad : \quad (b = \text{T}) \quad \equiv \quad (h_u = \text{T}) \ ,$$

meaning that the clock of u is the set of instants at which boolean variable b is true.

Here is an example of STS that we shall study in the sequel:



$$\begin{array}{ll} & \textbf{if } b \textbf{ then } z = u \textbf{ else } z = v \\ \bigwedge & \textbf{if } h_z \textbf{ then } v = \xi_z^- - 1 \textbf{ else } v = \perp \\ \bigwedge & \textbf{if } h_v \textbf{ then } b = (v \leq 0) \textbf{ else } b = \perp \\ \bigwedge & h_v \equiv h_z \equiv h_b \\ \bigwedge & (b = \text{T}) \equiv (h_u = \text{T}) \end{array}$$

A run of this STS for the z variable is depicted on the figure above. Each time u is received, z is reset and gets the value of u. Then z is decremented by one at each activation cycle of the STS, until it reaches the value 0. Immediately after this latter instant, a fresh u can be read, and so on. Note the schyzophrenic nature of the "inputs" of this STS. While the value carried by u is an input, the instant at which u is read is not: reading of the input is on demand-driven mode. This is reflected by the fact that inputs of this STS are the pair {activation clock h, value of u when it is present}.

Using these primitives, dataflow synchronous languages such as LUSTRE [13], SIGNAL [16], and Statecharts [21] are easily encoded.

Open environments and modularity. As modularity is wanted, it is desirable that the pace of an STS is local to it rather than global. Since any STS is subject to further composition in some yet unknown environment, this makes the requirement of having a global pace quite inconvenient. This is why *we prohibit the use of clocks that are always present.* This has several consequences. First, it is not possible to consider the "complement of a clock" or the "negation of a clock" : this would require referring to the always present clock. Thus, as will be revealed by our examples, clocks will always be variables, and we shall be able to relate clocks only using $\wedge$ (intersection of instants of presence), $\vee$ (union of instants of presence), and $\setminus$ (set difference of instants of presence).

Stuttering. In the same vein, it should be permitted, for an STS, to do nothing while the environment is possibly working. This feature has been yet identified in the litterature and is known as *stuttering* invariance or robustness [14, 15]. It is central to TLA, where it is understood that a transition with no event at all and no change of states is always legal.

For an STS Φ, stuttering invariance is defined as follows: if

$$\sigma \ : s_0, s_1, s_2, \ldots$$

is a run of Φ, so is

$$\sigma' \ : s_0, \underbrace{\perp_0, \ldots, \perp_0}_{0 \leq \ \#\{\perp_0\} \ < \infty}, s_1, \perp_1, \ldots, \perp_1, s_2, \perp_2, \ldots, \perp_2, \ldots \ , \tag{2.3}$$

where symbol $\perp_i$ denotes a *silent* state in which all state variables keep the value they had at state s_i, while other variables take the value $\perp$. The number of inserted silent states is ≥ 0 but finite. This models that the considered STS can do nothing for any arbitrary but finite "duration".

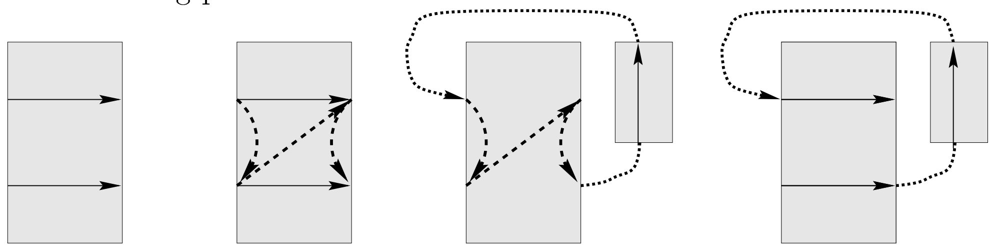
Stuttering invariance is not hardwired into our STS formalism, but a quick inspection of all the statements we have introduced in our example reveals that any composition of them is stuttering invariant. More generally, any STS not involving the always present clock (be it directly or indirectly) is stuttering invariant.

2.3 Enhancing the STS model for the purpose of code generation

2.3.1 Two basic issues

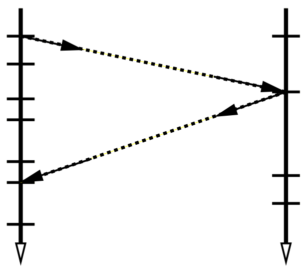
Basically, the problem is twofold: 1/ bruteforce separate compilation can cause deadlocks, and 2/ generating distributed code is generally not compatible with maintaining strict compliance with the synchronous model of computation. We illustrate briefly these two issues next.

Naive separate compilation may be dangereous. This is illustrated in the following picture:



The first diagram depicts the "dependencies" associated with some STS specification: the 1st output needs the 1st input for its computation, and the 2nd output needs the 2nd input for its computation. The second diagram shows a possible scheduling, corresponding to the standard scheduling: 1/ read inputs, 2/ compute reaction, 3/ emit outputs. This gives a correct sequential execution of the STS. In the third diagram, an additional dependency is enforced by setting the considered STS in some environment which reacts with no delay to its inputs: a deadlock is created. In the last diagram, however, it is revealed that this additional dependency caused by the environment indeed was compatible with the original specification, and no deadlock resulted from applying it. Here, deadlock was caused by the actual implementation of the specification, not by the specification itself.

Desynchronisation. This is illustrated in the following picture:



This figure depicts a communication scenario: two processors, modelled as sequential machines, exchange messages using an asynchronous medium for their communications. The natural structure of time is that of a *partial order,* as derived from the directed graph composed of 1/ linear time on each processor, and 2/ communications. This structure for time does not match the linear time corresponding to the infinite sequence of reactions which is the very basis of synchronous paradigm.

The need for reasoning about causality, schedulings, and communications. This need emerges from the above discussion. In the next subsection, we shall introduce a unique framework to handle these diverse aspects : the formalism of *scheduling specifications.*

2.3.2 Scheduling specifications

Preorders and partial orders to model causality relations, schedulings, and communications. We consider a set V of variables. A *preorder* on the set V is a relation (generically denoted by $\preceq$) which is reflexive ($x \preceq x$) and transitive ($x \preceq y$ and $y \preceq z$ imply $x \preceq z$). To $\preceq$ we associate the equivalence relation $\asymp$, defined by $x \asymp y$ iff $x \preceq y$ and $y \preceq x$. If equivalence classes of $\asymp$ are singletons, then $\preceq$ is a *partial order.* Preorders are naturally specified via *(possibly cyclic)* directed graphs :

$$x \rightarrow y \;\; \text{for} \;\; x, y \in V \; .$$

The *conjunction* of two preorders is the minimal preorder which is an extension of the two considered conjuncts.

A *labelled preorder* on V is a preorder, together with a domain for each $v \in V$. Call $\mathbf{dom}_V$ the domain of the set V of variables. Denote by Λ_V the set of all labelled preorders on V. A *state* s is a type consistent interpretation of the labelled preorder, i.e., a preorder on V together with a value $s[V]$ for the set of all variables belonging to V. Denote by $\mathbf{dom}_{\Lambda_V}$ the domain in which states take their value.

STS with scheduling specifications. Now we consider STS $\Phi = \langle V, \Theta, \rho \rangle$ as before, but with the following modification for the transition relation $\rho = \rho(V^-, V)$:

$$\rho \;\; \subset \;\; \mathbf{dom}_V \; \times \; \mathbf{dom}_{\Lambda_V} \; , \tag{2.4}$$

i.e., transition relations relate the value for the t-uple of previous variables to the current state. As before, runs are sequences $s_0, s_1, s_2, \ldots$ that are consistent with transition relation (2.4).

STS involving such type of preorder relation shall be called in the sequel STS *with scheduling specifications.* STS with scheduling specifications are just like any other STS, hence they inherit their properties, in particular *they can be composed.*

Notations for scheduling specifications : for b a variable of type $bool \cup \{\perp\}$, and u, v variables of any type, the following generic conjunct will be used:

$$\textbf{if} \quad b \quad \textbf{then} \quad u \longrightarrow v \quad , \text{resp.} \quad \textbf{if} \quad b \quad \textbf{else} \quad u \longrightarrow v$$

also written:

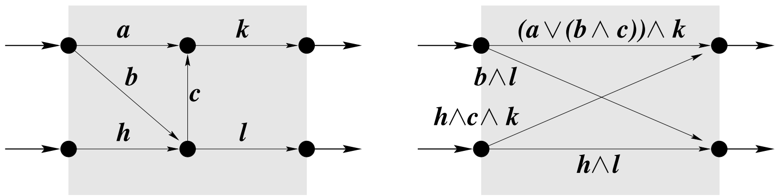
$$u \xrightarrow{b} v \quad \text{resp.} \quad u \xrightarrow{\overline{b}} v$$

In [1], it is shown that scheduling specifications have the following properties:

$$x \xrightarrow{b} y \bigwedge y \xrightarrow{c} z \Rightarrow x \xrightarrow{b \wedge c} z \tag{2.5}$$

$$x \xrightarrow{b} y \bigwedge x \xrightarrow{c} y \Rightarrow x \xrightarrow{b \vee c} y \tag{2.6}$$

Properties (2.5,2.6) can be used to compute input/output abstractions of scheduling specifications:



In this figure, the diagram on the left depicts a scheduling specification involving local variables. These are hidden in the diagram on the right, using rules (2.5,2.6).

Inferring scheduling specifications from causality analysis. We now provide a technique for inferring schedulings from causality analysis for STS specified as conjunctions of the particular set of generic conjuncts we have introduced so far. Considering this restricted set of generic conjuncts is justified by the fact that 1/ all known synchronous languages can be encoded using this set of basic conjuncts [2], and even more, 2/ these primitives allow to express the most general synchronisation mechanisms that are compatible with the paradigm of perfect synchrony [3]. We recall next this set of basic conjuncts for the sake of clarity :

$$
\begin{array}{c}
\begin{array}{lll}
\textbf{if} \;\; b & \textbf{then} & w = u \\
 & \textbf{else} & w = v
\end{array}
\\[2ex]
u \xrightarrow{\;\;b\;\;} w
\\[2ex]
\left.\begin{array}{rcl}
w & = & f(u_1, \ldots, u_k) \\
h_w = h_{u_1} & = & \ldots = h_{u_k}
\end{array}\right\}
\end{array}
\qquad (2.7)
$$

In addition to the set (2.7) of primitives, state-variable ξ_v associated to variable v can be used on the right hand side of each of the above primitive statements. The third primitive involves a conjunction of statements that are considered jointly. Later on, in the examples, we shall freely use nested expressions such as **if** b **then** $w = expr$, where "$expr$" denotes an expression built on the same set of primitives. It is understood that such expressions need to be expanded prior to applying the rules of formulas (2.8) given next.

In formulas (2.8), each primitive statement has a scheduling specification associated with it, given on the corresponding right hand side of the table. Given an STS specified as the conjunction of a set of such statements, for each conjunct we add the corresponding scheduling specification to the considered STS. Since, in turn, scheduling specifications themselves have scheduling specifications associated with them, this mechanism of adding scheduling specifications must be applied until fixpoint is reached. Note that applying these rules until fixpoint is reached takes at most two successive passes. In formulas (2.8), labels of schedulings are expressions involving variables in the domain $\{\perp, \mathrm{F}, \mathrm{T}\}$ ordered by $\{\perp < \mathrm{F} < \mathrm{T}\}$; with this in mind, expressions involving the symbols "$\wedge$" (min) and "$\vee$" (max) have a clear meaning.

(R-1) $$\forall u \qquad h_u \longrightarrow u$$

(R-2) $$\begin{array}{lll}\textbf{if} \;\; b & \textbf{then} & w=u \\ & \textbf{else} & w=v\end{array} \;\; \Rightarrow \;\; \left\{\begin{array}{l} b \xrightarrow{h_b \,\wedge\, (h_u \vee h_v)} h_w \\ \\ h_u \xrightarrow{b \,\wedge\, h_u} h_w \\ h_v \xrightarrow{\bar{b} \,\wedge\, h_v} h_w \\ \\ u \xrightarrow{b \,\wedge\, h_u} w \\ v \xrightarrow{\bar{b} \,\wedge\, h_v} w \end{array}\right. \tag{2.8}$$

(R-3) $$u \xrightarrow{b} w \;\; \Rightarrow \;\; b \longrightarrow h_w$$

(R-4) $$\left.\begin{array}{c} w \;=\; f(u_1,\ldots,u_k) \\ h_w = h_{u_1} = \ldots = h_{u_k} \end{array}\right\} \;\; \Rightarrow \;\; u_i \xrightarrow{h_w} w$$

Rules (R-1,...,R-4) are formally justified in [1]. We briefly report the corresponding results. For `P` an STS, first apply Rules (R-1,...,R-4) until fixpoint is reached: this yields an STS we call **sched(`P`)**. Then, a *sufficient* condition for `P` to have a unique deterministic run is:

1. **sched(`P`)** is provably circuitfree at each instant, meaning that *it is never true that*

$$x_1 \xrightarrow{b_1} x_2 \xrightarrow{b_2} x_1$$

and

$$(b_1 \wedge b_2 = \mathrm{T})$$

   unless $x_1 = x_2$ holds.

2. **sched(`P`)** has provably no multiple definition of variables at any instant, meaning that, whenever

$$\begin{array}{ll} & \textbf{if } b_1 \textbf{ then } x = \exp_1 \\ \wedge & \textbf{if } b_2 \textbf{ then } x = \exp_2 \end{array}$$

holds in $\mathtt{P}$ and the $\exp_1$ and $\exp_2$ are different expressions, then

$$b_1 \wedge b_2 \quad = \quad \mathrm{T}$$

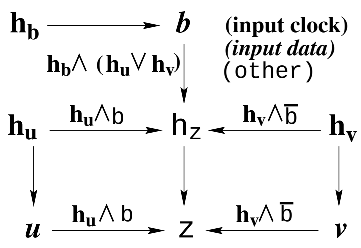
never holds in $\mathtt{P}$.

Then $\mathtt{P}$ is said to be *executable,* and **sched**($\mathtt{P}$) provides (dynamic) scheduling specifications for this run.

Examples. We show here some STS statements and their associated scheduling as derived from causality analysis. In the following figures, vertices in boldface denote input clocks, vertices in bold-italic denote input data, and vertices in courier denote other variables. It is of interest to split between these two different types of inputs, as input reading for an STS can occur with any combination of data- and demand-driven mode. Note that, for each vertex of the graph, the labels sitting on the incoming branches are evaluated prior to the considered vertex. Thus, when this vertex is to be evaluated, it is already known which other variables are needed for its evaluation.

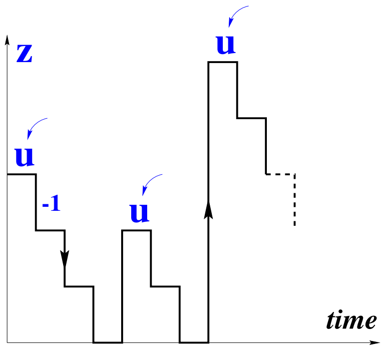
A reactive statement :

if b **then** $z = u$ **else** $z = v$



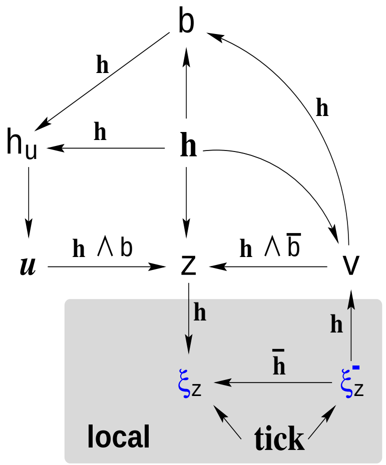
In the above example, input data are pairwise associated with corresponding input clocks : this STS reads its inputs on a purely data-driven mode, input patterns (u, v, b) are free to be present or absent, and, when they are present, their value is free also. We call it a "reactive" STS.

The full example, a proactive STS :



$$
\begin{array}{rl}
 & \textbf{if } b \textbf{ then } z = u \textbf{ else } z = v \\
\wedge & \textbf{if } h_z \textbf{ then } v = \xi_z^- - 1 \textbf{ else } v = \perp \\
\wedge & \textbf{if } h_v \textbf{ then } b = (v \leq 0) \textbf{ else } b = \perp \\
\wedge & h_v \equiv h_z \equiv h_b \\
\wedge & (b = \mathrm{T}) \equiv (h_u = \mathrm{T})
\end{array}
$$

Applying the rules (R-1,...,R-4) for inferring schedulings from causality, and then performing some straightforward simplifications, we get the result shown in Table 2.1. Note the change in control: { input clock, input data} have



$$
\begin{array}{rl}
 & \textbf{if } b \textbf{ then } z = u \textbf{ else } z = v \\
\wedge & \textbf{if } h_z \textbf{ then } v = \xi_z^- - 1 \textbf{ else } v = \perp \\
\wedge & \textbf{if } h_v \textbf{ then } b = (v \leq 0) \textbf{ else } b = \perp \\
\wedge & h_v \equiv h_z \equiv h_b \equiv_{\text{def}} h \\
\wedge & (b = \mathrm{T}) \equiv (h_u = \mathrm{T})
\end{array}
$$

Table 2.1: *Scheduling from causality analysis for the example.*

been drastically modified from the " **if** b **then** $z = u$ **else** $z = v$" statement to the complete STS: inputs now consist of the pair $\{h, v_u\}$, where v_u refers to the value carried by u when present. Reading of the u occurs on demand, when condition b is true. Thus we call such an STS "proactive".

2.4 Simplifying the STS model for the purpose of system verification

Clocks are useful for code generation and validation, they can also be used for verification. However, most verification tools (and even some code generation tools for synchronous languages) do not use clocks explicitly. For this reason, the DC format [11] was formerly developed within the Eureka-Synchron project ; it essentially corresponds to STS (possibly with scheduling specifications) having a single clock. A notion of an "activation clock" was offered instead, see Section 2.5 for this notion in the framework of DC_+. While restricting to single clocked STS can be too strong a restriction, the following simplification of our STS model was proved useful, and has been in particular used for verification.

The subclass of STS we consider is that for which we have two different classes of variables, namely

variables : variable x has clock h_x specified by the true occurrences of a boolean variable b_x. Such boolean variables b_x etc., are called "boolean clocks".

boolean clocks : boolean clocks have all the same clock, we call "**tick**" this clock.

In other words, clocks have a flat structure, they are not nested, unlike in the full STS model. The intesrest of this submodel is that 1/ it offers a restricted notion of clock, but 2/ it can easily be translated into BDD's using a don't care coding for the status "absent". A subformat of DC_+ has been defined to support this model.

2.5 DC_+ and its STS Semantics

As code validation is based on actual code, not just a model, we present here a sketch of the DC_+ format for which code validation is performed. We refer the reader to [12] for a detailed description of the DC_+. In the following, we describe how to construct an STS Φ_P corresponding to a given DC_+ program P.

A DC_+ program describes a reactive system whose behavior along time is an infinite sequence of *instants* which represent reactions, triggered by external or internal events. The main objects manipulated by a $\text{D}\textsc{c}_+$ program are *flows*, which are sequence of values synchronized with a *clock*. A flow is a typed object which holds a value at each instant of its clock. The fact that a flow is currently absent is represented by the bottom symbol $\perp$ (cf. section 2.2). Clocks are boolean flows, assuming the values $\{\text{T}, \perp\}$. A clock has the value T if and only if the flow associated with the clock holds a value at the present instant of time. Actually, any expression *exp* in the language has its corresponding clock h_{exp} which indicates whether the value of the expression at the current instant is different from $\perp$.

Besides *external* flows (input/output flows), which determine the interface of the $\text{D}\textsc{c}_+$ program with its environment, also *internal* flows are used and manipulated by the program.

A sketch of $\text{D}\textsc{c}_+$ primitives. We list in Table 2.2 the $\text{D}\textsc{c}_+$ primitives. Explanations and intuitive semantics is given now. In these primitives,

expression extended to sequences	:	$f(u_1, \ldots, u_n)$
assignment extended to flows	:	**equ** : $w\ v$ **at** c **init** w_0
memorization	:	**memo** : $w\ v$ **at** c **init** w_0
data dependent downsampling	:	$v =$ **when** (u, b)
selector with priority	:	$w =$ **pcond**(b, u, v)
left_is_true	:	$d =$ **left_tt**(b, c)
h is the clock of u	:	$h =$ **event** (u)
scheduling specification	:	v **depends** : u **at** b

Table 2.2: $\text{D}\textsc{c}_+$ *primitives.*

u, v, w, b denote typed *flows*, i.e., sequences of values in the considered type extended with the special symbol $\perp$. In writing these instructions, we should have written rather statements such as " **equ** : w *expr* **at** c **init** w_0 " where "*expr*" stands for any $\text{D}\textsc{c}_+$ expression, and then we should have defined " **when** (u, b) ", etc., as $\text{D}\textsc{c}_+$ expressions. Our presentation will however make the writing of the STS semantics simpler. By convention, the b, c, d flows are assumed boolean, while the other ones are of any type. In the " **at** c " statements, the boolean flow c is an *activation condition* as will explained later in the semantics. Finally, the last statement is the scheduling specification which states that v depends on u when b is present and true: it is thus exactly the corresponding STS statement. Dc_+ program composition and closure are inherited from their respective STS counterparts. The intuitive as well as formal semantics of these instructions will be given subsequently. Next, we give the semantics of these instructions.

System Variables. The system variables of Φ are given by $V = U$, where U are the Dc_+ *flows* explicitly declared and manipulated in P. An auxiliary variable by the name of ζ_v is included in X whenever needed, this will be made explicit in giving the STS semantics of Dc_+ subsequently. The value of ζ_v is intended to represent the value of v at the previous instance (present excluded) that v was different from $\perp$. The ζ_v is closely related to our ξ_v, but is not identical, hence the different notation for it.

Initial Condition. The initial condition for Φ is given by

$$\Theta\colon \quad \bigwedge_{u \in U} u = \perp \quad \wedge \quad \bigwedge_{v \in U} \zeta_v^- = v_o$$

As will result from our STS translation of SIGNAL and Dc_+ programs they are all stutteringly robust. Consequently, we can simplify things by assuming that the first state in every run of the system is a stuttering state.

Transition Relation. The transition relation ρ will be a conjunction of assertions, where each Dc_+ statement gives rise to a conjunct in ρ. We list the statements of Dc_+ and, for each statement S, we present the conjunct contributed to ρ by S.

- Consider the Dc_+ expression $v = f(u_1, \ldots, u_n)$, where f is a state-function. Its contribution to ρ is given by:

$$\begin{array}{rl} & h_v \;\equiv\; h_{u_1} \;\equiv\; \ldots \;\equiv\; h_{u_n} \\ \wedge & h_v \neq \perp \;\;\rightarrow\;\; v = f(u_1, \ldots, u_n)) \end{array}$$

  This formula requires that the signals $v, u_1, \ldots, u_n$ are present at precisely the same time instants, and that at these instants $v = f(u_1, \ldots, u_n)$.

- The contribution of the statement

$$\textbf{equ} \;:\; w \; v \;\; \textbf{at} \; c \; \textbf{init} \, w_0$$

requires the introduction of an auxiliary variable ζ_v with initial value w_0, it is given by:

$$\begin{array}{rrcl} & h_w & \equiv & h_v \ \equiv \ h_c \\ \wedge & \zeta_v & = & \left(\begin{array}{lll} \textbf{if} & c = \mathtt{T} & \textbf{then} \ \ v \\ \textbf{else} & \zeta_v^- & \end{array}\right) \\ \wedge & w & = & \textbf{if } h_v \textbf{ then } \zeta_v \textbf{ else } \perp \end{array}$$

Roughly speaking, this statement delivers the most recent value of v.

- The contribution of the statement

$$\textbf{memo} \ : \ w \ v \ \ \textbf{at} \ c \ \textbf{init} \, w_0$$

requires the introduction of an auxiliary variable ζ_v with initial value w_0, it is given by:

$$\begin{array}{rrcl} & h_w & \equiv & h_v \ \equiv \ h_c \\ \wedge & \zeta_v & = & \left(\begin{array}{lll} \textbf{if} & c = \mathtt{T} & \textbf{then} \ \ v \\ \textbf{else} & \zeta_v^- & \end{array}\right) \\ \wedge & w & = & \textbf{if } h_v \textbf{ then } \zeta_v^- \textbf{ else } \perp \end{array}$$

This statement delivers the last value of v (Compare with the semantics of **equ**).

- The contribution of the statement

$$v = \ \textbf{when} \ (u, b)$$

is given by:

$$v = \textbf{if} \ (b = \mathtt{T}) \ \textbf{then} \ \ u \ \textbf{else} \perp.$$

- The contribution of the statement

$$w = \ \textbf{pcond}(b, u, v)$$

is given by:

$$w \ = \ \left(\begin{array}{llll} \textbf{if} & h_u \textbf{ and } b = \mathtt{T} & \textbf{then} & u \\ \textbf{else if} & h_v \textbf{ and } b = \mathtt{F} & \textbf{then} & v \\ \textbf{else} & \perp & & \end{array}\right)$$

- The contribution of the statement

$$d = \ \mathbf{left\_tt}(b, c)$$

is given by:

$$d \quad = \quad \left(\begin{array}{lll} \textbf{if} & b = \text{T} & \textbf{then} \ \ \text{T} \\ \textbf{else if} & h_b \ \textbf{or} \ h_c & \textbf{then} \ \ \text{F} \\ \textbf{else} & \perp & \end{array} \right)$$

- The contribution of the statement

$$h = \ \textbf{event} \ (u)$$

is given by:

$$h = \textbf{if} \ h_u \ \textbf{then} \ \ \text{T} \ \textbf{else} \perp .$$

- The contribution of the statement

$$v \ \textbf{depends} \ \ : u \ \textbf{at} \ b$$

is given by:

$$u \xrightarrow{\ b\ } v.$$

2.6 Integration of StateMate specifications

As StateMate is a rich and powerful specification formalism with a strong state orientation, it is worth discussing how it translates into our sts model. The actual concrete target for translation is Dc_+ (or Signal), but explaining the principles of translation into sts will be sufficient, as Dc_+ is very close to sts. Among the different semantics of Statecharts [24], the translation proposed here follows the StateMate one [23]. It is implemented in the tool Magnum, designed by i-Logix. We assume the reader is familiar with Statecharts, and we assume some familiarity with the StateMate tool; missing information is found in [23]. In SACRES work on the sts modelling of StateMate has been performed for the purpose of StateMate verification [26, 27, 28, 29] and StateMate2Dc_+ translation [20]. Here we discuss the principles of the latter one.

2.6.1 Statecharts as in StateMate

The Statecharts formalism has been introduced by Harel [21]. It is a graphical language based on hierarchical and concurrent automata, with so-called **and** and **or** states, that provide the state hierarchy, and **basic**-states. It is integrated in the StateMate environment, along with another language called Activitycharts, which is block-diagram oriented.

The transitions between states are labeled by reactions of the form: `e[C]/a`, where `e` is an event that possibly triggers the transition, `C` is a boolean guard condition that has to be true to pass the transition, and `a` is the resulting action.

The basic evolution of a Statechart consists in a step, where given the events currently present and the current values of variables, triggers and conditions are evaluated, and actions are carried out. In StateMate the effects (event generation, variable modifications) of the actions carried out in one step are sensed only at the next step. This makes a difference with other semantics [24], e.g. strictly synchronous as in Argos [25].

In the so-called "step semantics" of StateMate, external inputs are sensed at each step. In the "superstep semantics", supersteps are composed of a sequence of steps for which only effects of internal actions are sensed. The superstep terminates when no more transition can be fired. Then, new external inputs are sensed, and the next superstep begins.

Besides the Statecharts, another language of the StateMate environment is Activitycharts [22]: it provides the designer with a notion of multi-level data-flow diagrams, each of the blocks in the hierarchy represents an activity. Activities can be started, suspended, resumed, and stopped.

2.6.2 Translation principles

Main objectives are:

- to keep the nested structure of states visible;
- to have a structural translation (have sts prototypes for a state, for a transition, for computing maximal nonconflicting sets of enabled transitions, for computing resulting actions...);
- to propagate in a standardised way control signals to capture preemption mechanisms;

- to have an easily tunable semantics, from which “step” and “superstep” semantics are easily derived ;
- to have a translation which takes advantage of the multiclock features for subsequent exploitation in the code generation phase.

Variables in StateMate are of two kinds: events or data-items. Only events are clocked, data-items are valued and always present. In our STS translation, events will be translated as STS clocks, while data-items will be translated into (clocked) STS variables.

The reactive box model

It aims at propagating the needed control signals in a strandardised way, thus ensuring compositionality of the translation. Our reactive box model is shown in Figure 2.1.

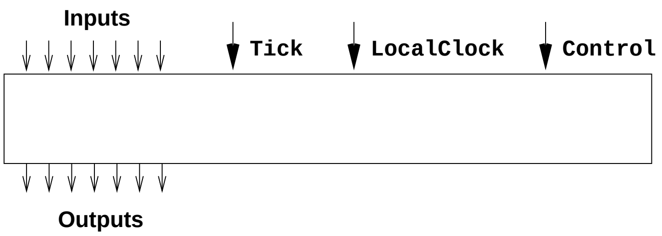


Figure 2.1: *The reactive box model*

A reactive box is a box containing inputs and outputs. Some of these interface variables 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).
- `LocalClock` is the clock of the box. This clock is present whenever the corresponding Statechart or Activitychart component is active.
- `Control` is an enumerated-typed signal ranging in the set of values *Start, Stop, Resume, LocalResume*. It is used inside the box to know when and how the box is (re)entered.

The clock of the data-items is computed using the signals `Tick` and `LocalClock`.

Some useful predefined STS

A transition. To check if a given transition τ is triggered, cf. Figure 2.2, a specific STS, also called τ, is defined, which indicates if transition τ

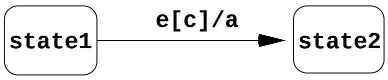


Figure 2.2: *A transition in Statecharts*

can be fired. STS τ has two parameters: $\mathtt{s}_1, \mathtt{s}_2$ (for `state1,state2`), the origin and target states of the considered transition. It has two inputs: s_o (for `origin_state`), indicating which state was left by the actual move of the Statechart, and t (for `trigger`), the triggering event of transition τ, a boolean variable. It has one output s_t (for `target_state`), indicating if a move to the target state of τ is actually performed. Its STS specification is:

$$s_t \quad = \quad \left(\begin{array}{llllll} \textbf{if} & s_o = \mathtt{s}_1 & \textbf{then} & \textbf{if} \quad t = \mathrm{T} & \textbf{then} & \mathtt{s}_2 \\ & & & & \textbf{else} & \perp \\ & & \textbf{else} & \perp & & \end{array} \right)$$

A 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 variable ranging on an enumerated type with n different values: one value per sub-state. Call *"state-var"* such an STS variable.

For a general Statechart, a tree of state-var is built: for each **and**-level of the Statechart configuration, one state-var is attached to each **and**-state. The value of a given state-var indicates which **or**-state is active at the subsequent level of the configuration. Thus, to each configuration we attach a tree of state-var with nested clocks, this is illustrated in Figure 2.3. In this figure, four state-vars are shown. The state-var s_1 is present when the global state is active, it takes its values in the enumerated set $\{1, 2, 3\}$. The same holds for s_2. Next, $s_{2,1,1}$ and $s_{2,1,2}$ are attached to the **and**-states at the finest level: both are present when the father **or**-state is active. As the states of a Statechart form a tree, we have a tree of associated state-vars with nested clocks. Thus, in our translation scheme, the structure of the original Statechart will be kept visible.

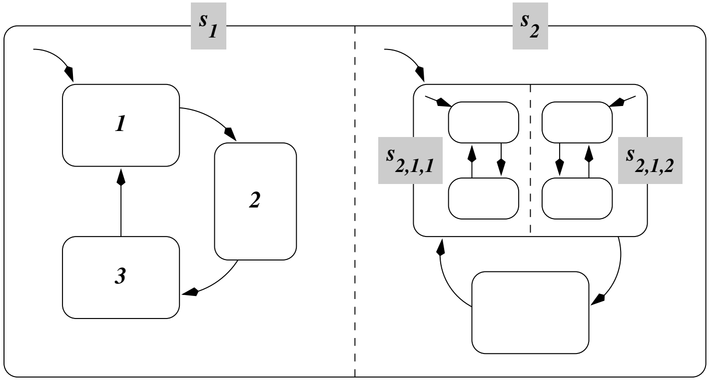


Figure 2.3: *A Statechart and its associated state-vars.*

For every level of the Statecharts hierarchy, an instance of the following STS σ is used to update the configuration variable and compute the next state. σ has one parameter: $\mathtt{s}_i$ (for `initial_state`), the initial state at the considered level of the hierarchy. Also, σ has three inputs: the event h_l (the `local_clock`, a boolean variable), the state s_{pn} (the `proposed_new_state`) which is the proposed new value for the state as computed from the transition STS, the control variable c which indicates if the operating mode is the normal one or if restart or resume is being performed. And it has one output, the new value s_n (the `new_state`) for the state. Its STS specification is:

$$s_n \quad = \quad \left(\begin{array}{llllll} \textbf{if} \; h_l = \perp & \textbf{then} \; \perp & & & & \\ & \textbf{else} \; \textbf{if} \; s_{pn} \neq \perp & \textbf{then} \; s_{pn} & & & \\ & & \textbf{else} \; \textbf{if} \; c = \mathit{start} & \textbf{then} \; \mathtt{s}_i & & \\ & & & \textbf{else} \; s_n^- & & \end{array} \right)$$

Delaying the effect of internally produced events and actions. The StateMate semantics of Statecharts [23] 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". A run of the corresponding STS with integer values for `x` is shown figure 2.4. If it is wanted to have a *perfect synchrony hypothesis* [24], the shift should be removed and then input and corresponding output would occur at the same time.

Tick	•	•	•	•	•	•	•	•	•	•	•
x	1		2				3	4	5		
y		1		2				3	4	5	

Figure 2.4: *Trace of the shift process (Integer values for x)*

Handling actions and nondeterminism. At this point, we model Statecharts by combining STS models for transitions and for computing new states, as well as conveniently delaying the effect of internally created events and actions. Firing a transition would normally result in performing the corresponding action. As several fired transitions can cause conflicting assignments for the same data-item (racing condition), nondeterminism results in the StateMate semantics [5] and this is easily modelled in STS (and in the Dc_+ format) as nondeterminism is naturally captured in this framework. Similarly, conflicting sets of transitions can result from the enabling conditions and events ; again, the resulting "nondeterministic" interpretation of these situations in StateMate can be captured in STS.

When all these issues of conflicts and nondeterminism are captured, one choice is arbitrarily chosen for the nondeterministic transition, and the resulting set of actions globally results.

Step vs. superstep. As our generic model of a reactive box has a `Tick` explicitly available as an input to model the *step* for the considered design, we can explicitly relate it in our STS model to the clock `io_Tick` of external communications. In the so-called *step* semantics, the two clocks `Tick` and `io_Tick` coincide, whereas they differ for the case of the *superstep* semantics. Actually, `io_Tick` is a subclock of `Tick` corresponding to the satisfaction of the "stability" condition.

[5]Note that it would be more natural, from the synchronous STS point of view, to view conflicting situations as contradicting constraints rather than nondeterminism. But this is not choice done by StateMate.

Chapter 3

Verification : verifying large, complex systems

Both component and system verification have been addressed in SACRES. Component verification consists in checking properties for a monolithic software component considered as a whole. Our technology basis for component verification is a BDD-package developed at Siemens [30, 31, 32, 33]. While modern BDD packages are extremely powerful and optimized, their processing capabilities is still below what is frequently requested for verifying software for embedded systems. Two answers can be considered to meet this challenge of complexity :

1. Usually, software for verification was produced using some particular specification formalism. For instance, in the SACRES context, STATEMATE and SILDEX are the specification tools. Particularities of *source* programs generated using this particular formalism can be taken advantage of, in order to facilitate the work of BDD-based model checking. This technique is for instance advocated and used in the context of the ESTEREL language, see [36]. It is in fact also heavily used in the SIGNAL compilation technique, both for the TNI and the Inria compilers. As for the Inria compiler, synthesizing the control for the object C code requires performing the so-called *clock calculus.* The clock calculus consists in synthesizing the overall synchronisation of the program from the source SIGNAL specification given in the form of local synchronisation constraints. This is indeed a restricted form of a "formal verification" type of task. For doing this, a canonical arborescent form

for families of clocks is computed [5]. This is a special data structure to which several (in fact, many) boolean predicates are attached in the form of BDD representations. In this way, large programs can be handled, much larger than pure BDD-based representations would allow. In addition, computing reachability sets – a basic activity in performing model checking – is drastically facilitated. We are not going to elaborate any further on this approach in the sequel.

2. The alternative approach for reducing complexity consists in performing compositional, or incremental, verification. This amounts to considering that *systems* are at hand, that are composed of several components, and so on, in a hierarchical manner. The overall proof would then be splitted into (hopefully) smaller proofs for subsystems or components. The intent is that proofs for components would be accessible to basic BDD-technology, while *assembling* partial proofs to derive larger proofs would also result in affordable tasks for the BDD-package. This is called *system verification* and is one of the major developments within SACRES.

Two approaches for system verification have been considered in SACRES. The first one [26, 27, 28, 29] is based on an *assumption/commitment* type of approach, it was developed at OFFIS. The second one is a simple and systematic way of performing incremental proofs using synchronous observers, based on a decomposition of a synchronous reactive program into its constitutive components, it has been designed and experimented at Siemens.

3.1 The assumption/commitment approach

See [26, 27, 28, 29] for this approach, for which we only provide general hints. Efficient verification of STATEMATE designs raises difficulties per se. Thus this problem has been addressed as such in SACRES, and component & system verification for STATEMATE was an important objective from the beginning, and a direct route for this has been developed. Subsequently as the project progressed, component & system verification for mixed designs via the DC_{+} format was developed.

In this section, we focus on discussing challenges from system verification. The general aim was to provide verification with the following features:

1. It should allow to express temporal logic type of properties in a user friendly way. This includes untimed safety and liveness properties, as well as some subclass of timed safety properties. User friendliness would require not requesting the user to express her/himself using the mathematical jargon of some temporal logic dialect. A more familiar graphical frontend should be offered instead. This is why *Timing Diagrams* (TD) have been considered in SACRES.

2. As system verification involves reasoning on *open* (sub)systems, in which the environment can be either the systems environment, or the interactions with other subsystems, it was needed to consider verification for *open* (sub)systems, thus room was needed in the reasoning for all possible interference from the environment.

While point 2 is (relatively) easily handled when one restricts to safety properties, see Section 3.2, it becomes much more difficult when more general properties are considered. Consider the open STS from Figure 3.1. In this

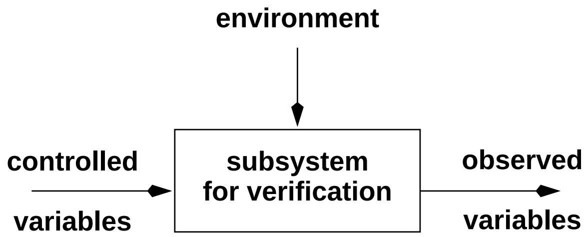


Figure 3.1: *An open (sub)system.*

figure, the system shown has observed outputs, and two types of inputs. Controlled inputs are those which can be chosen by the system, i.e., they correspond to the possible choices among the different executions of the program. Environment inputs are those inputs which are not under control of the considered system. Thus, environment inputs must always be considered under universal quantifier "$\forall$" when expressing properties. In contrast, both existential and universal quantifiers can be considered about controlled inputs. And this holds for both safety and liveness properties. As for liveness properties, environment can always block a reaction by just refusing to synchronise with it. Thus usual liveness should be replaced by *"liveness assuming the environment is willing to synchronise".*

Figure 3.2 depicts how this difficulty is handled in the particular case of Statecharts as implemented by StateMate. The environment can block

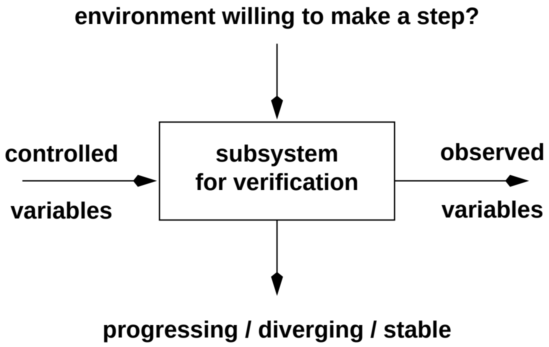


Figure 3.2: *An open (sub)system for Statecharts.*

a Statechart by refusing to synchronise on a step, this is depicted by the "environment input". To contribute to such an environment input for other subsystems, the subsystem in consideration must itself output its current status as to whether it is progressing, diverging, or willing to make a step. This is to be compared with the "reactive box" model of Figure 2.1, used as a generic STS interface for the translation StateMate2Dc$_+$. This is the approach taken for system verification in SACRES.

This is in contrast with the ATL approach of Alur, Henzinger, and Kupferman [37], in which a game theoretic approach is also taken, but with no predefined adversary inputs; in this approach,environment may act at will in a totally unpredictable way.

To summarize, when translating Statecharts into the restricted form of STS formalism needed by the Siemens Verification Engine (SVE, [30, 31, 32, 33]), the needed auxiliary variables are added for each considered component. The same is performed when translating Statecharts in the full Dc$_+$ format.

Of course, the major methodological difficulty consists in decomposing the overall proof obligation into several partial proofs involving subsystems and their environment. Also, from a technical point of view, as partial properties (call them "lemmas") are used for establishing other lemmas as well as the overall desired property, dependencies between partial properties as well as subsystems need to be traced. The Dc$_+$ format is of help for this.

3.2 Incremental verification of safety properties via synchronous observers

See [35] for details on this approach. Safety properties of reactive systems can convieniently be specified in assert-commit style, by Hoare triples $\alpha P \beta$, where α and β are formulas in some logic and P is a process (or a system). α (assertion) specifies a property of the environment in which P is executed, and β (commitment) specifies a property guaranteed by P. More precisely, $\alpha P \beta$ is valid, denoted by $\models \alpha P \beta$, iff $E \,\|\, P$ satisfies β, for any environment E which satisfies α.

Our approach is based on the observation that under the above interpretation of Hoare triples, the following inference rule is both sound and (relatively) complete for synchronous parallel composition of processes P and Q:

$$\frac{\alpha P \gamma, \quad \gamma Q \beta}{\alpha(P \,\|\, Q)\beta}$$

Thus, in order to prove $\alpha(P \,\|\, Q)\beta$, it suffices to find an intermediate formula γ such that $\alpha P \gamma$ and $\gamma Q \beta$.

The above rule is analogous to the well-known rule for the sequential composition of iterative programs:

$$\frac{\alpha P \gamma, \quad \gamma Q \beta}{\alpha(P; Q)\beta}$$

We draw on this analogy, by considering the strongest post-condition of α w.r.t. P, denoted by $\alpha \overrightarrow{P}$, and the weakest pre-condition of β w.r.t. Q, denoted by $\overleftarrow{Q}\beta$, in order to automate the task of finding the intermediate formula γ.

When verifying complex systems, especially software safety-critical systems, one is usually interested in verifying many relatively simple commitments under the same assertion. By a relatively simple commitment we mean the property which only a small part of the system contributes to. In this typical scenario, synthesis of γ as $\overleftarrow{Q}\beta$ turns out to be particularly useful: by considering successive weakest pre-conditions $\overleftarrow{Q}\beta$ and $\overleftarrow{P}(\overleftarrow{Q}\beta)$, one in fact considers automatically how the component processes contribute to property β.

On the way, we develop a simple logical framework to study proof strategies for $\alpha(P \,\|\, Q)\beta$. This simple framework leads to a simple calculus in which such strategies can be derived formally, just by manipulating logic formulas.

The logical framework is developed in two steps. First we consider the general case in which we are not concerned with any particular notation used to specify components of $\alpha(P \,\|\, Q)\beta$. Next we instantiate the general framework in the context where processes P and Q are specified by boolean automata (generalised synchronous sequential circuits), and formulas α and β are specified by synchronous observers [39]. Since synchronous observers can also be specified by boolean automata, all the components of $\alpha(P \,\|\, Q)\beta$ can be represented by circuits, and this allows to simplify the proof of $\alpha(P \,\|\, Q)\beta$ via manipulations on circuits.

3.2.1 Logical framework

We do not distinguish between STS regarded as programs, and STS regarded as logical formulas. From our more abstract point of view, both define just sets of legal runs. Thus, parallel composition of programs (written $\|$), conjunction of formulas (written $\wedge$), and interconnection of synchronous circuits, are equally modelled as STS composition (written $\bigwedge$). With this kept in mind, we are ready to discuss our approach.

Assert-commit formulas

A property of a reactive process can be regarded as a set of traces which have the property. As usual, we say that process P has property X iff $[\![P]\!] \subseteq X$, where $[\![P]\!]$ denotes the semantics of P, i.e., its set of runs. We assume that properties are specified by formulas in some logic $\mathcal{L}$. As usual, we say that process P satisfies formula α, denoted by $P \models \alpha$, iff $[\![P]\!] \subseteq [\![\alpha]\!]$.

In fact, we will specify behaviours of processes in the assert-commit style, by formulas of the form $\alpha P \beta$, where $\alpha, \beta \in \mathcal{L}$. Intuitive meaning of such Hoare triples is the following: $\alpha P \beta$ is valid iff P has property β whenever executed in an environment which has property α. More formally,

$$\models \alpha P \beta \quad \text{iff} \quad \forall E (\text{if } E \models \alpha \text{ then } E \,\|\, P \models \beta) \tag{3.1}$$

The meaning of triple $\alpha P \beta$ can also be formalized in logic $\mathcal{L}$, as

$$\models \alpha P \beta \quad \text{iff} \quad \models \alpha \wedge \mathcal{P} \rightarrow \beta \tag{3.2}$$

or equivalently, as

$$\models \alpha P \beta \quad \text{iff} \quad \models \mathcal{P} \rightarrow (\alpha \rightarrow \beta) \tag{3.3}$$

The semantical and logical characterization of validity of $\alpha P \beta$ are (almost) equivalent:

Fact 1 $\models \alpha \wedge \mathcal{P} \rightarrow \beta$ *implies* $\forall E($*if* $E \models \alpha$ *then* $E \,\|\, P \models \beta)$.

Proof: If $E \models \alpha$ then $E \,\|\, P \models \alpha \wedge \mathcal{P}$, hence $E \,\|\, P \models \beta$. $\square$

Fact 2 *If logic* $\mathcal{L}$ *is* realizable*, in the sense that any formula* $\alpha \in \mathcal{L}$ *is a characteristic formula of some process, then* $\forall E($*if* $E \models \alpha$ *then* $E \,\|\, P \models \beta)$ *implies* $\models \alpha \wedge \mathcal{P} \rightarrow \beta$.

Proof: Take as E the process characterized by α, then $E \,\|\, P \models \beta$ is equivalent to $[\![\alpha]\!] \cap [\![P]\!] \subseteq [\![\beta]\!]$ which is equivalent to $\models \alpha \wedge \mathcal{P} \rightarrow \beta$. $\square$

In the rest of the paper we assume that the meaning of $\alpha P \beta$ is defined by (3.2) or (3.3).

Strongest post-condition and weakest pre-condition

Intuitively, the strongest post-condition of α w.r.t. P, denoted by $\alpha\overrightarrow{P}$, is the strongest property guaranteed by P when P is executed in the environment which has property α. Formally, $\alpha\overrightarrow{P}$ is a formula such that $\models \alpha\overrightarrow{P} \rightarrow \beta$ iff $\models \alpha P \beta$, for any β. Since $\models \alpha P \beta$ is equivalent to $\models \alpha \wedge \mathcal{P} \rightarrow \beta$, it follows that

$$\alpha\overrightarrow{P} = \alpha \wedge \mathcal{P} \tag{3.4}$$

Dually, the weakest pre-condition of β w.r.t. P, denoted by $\overleftarrow{P}\beta$, is the weakest property of the environment in which P is executed, in order for P to guarantee the property β. Formally, $\overleftarrow{P}\beta$ is a formula such that $\models \alpha \rightarrow \overleftarrow{P}\beta$ iff $\models \alpha P \beta$, for any α. Since $\models \alpha P \beta$ is equivalent to $\models \alpha \rightarrow (\mathcal{P} \rightarrow \beta)$, it follows that

$$\overleftarrow{P}\beta = \mathcal{P} \rightarrow \beta \tag{3.5}$$

The inference rule

The following rule is both sound and complete:

$$\frac{\alpha P \gamma, \quad \gamma Q \beta}{\alpha (P \,\|\, Q) \beta}$$

Soundness of the rule follows directly from (3.2), by a simple propositional reasoning to justify that $\models \alpha \wedge \mathcal{P} \wedge \mathcal{Q} \to \beta$ (equivalent to $\models \alpha(P \parallel Q)\beta$) follows from $\models \alpha \wedge \mathcal{P} \to \gamma$ (equivalent to $\models \alpha P \gamma$) and $\models \gamma \wedge \mathcal{Q} \to \beta$ (equivalent to $\models \gamma Q \beta$).

For completeness it suffices to take as γ either $\alpha \overrightarrow{P}$ or $\overleftarrow{Q}\beta$. For example, let $\gamma := \alpha\overrightarrow{P} = \alpha \wedge \mathcal{P}$. On one hand, $\models \alpha P \gamma$ follows from $\models \alpha \wedge \mathcal{P} \to \alpha \wedge \mathcal{P}$. On the other hand, $\models \gamma Q \beta$ (equivalent to $\models \alpha \wedge \mathcal{P} \wedge \mathcal{Q} \to \beta$) follows from the assumption that $\models \alpha(P \parallel Q)\beta$ (also equivalent to $\models \alpha \wedge \mathcal{P} \wedge \mathcal{Q} \to \beta$).

Proof strategies

In this section, we analyse proof strategies for establishing validity of $\alpha(P \parallel Q)\beta$. The analysis can be done in a very simple way, using the simple logical framework we have developed so far.

Directly from definition 3.2,

$$\models \alpha(P \parallel Q)\beta \quad \text{iff} \quad \models \alpha \wedge \mathcal{P} \wedge \mathcal{Q} \to \beta$$

and formula $\alpha \wedge \mathcal{P} \wedge \mathcal{Q} \to \beta$ is equivalent to the following formulas:

$$\begin{array}{ll} \text{(monolithic)} & \mathcal{P} \wedge \mathcal{Q} \to (\alpha \to \beta) \\ \text{(forward)} & ((\alpha \wedge \mathcal{P}) \wedge \mathcal{Q}) \to \beta \\ \text{(backward)} & \alpha \to (\mathcal{P} \to (\mathcal{Q} \to \beta)) \end{array}$$

The first formula represents a proof strategy which we call a monolithic proof of $\models \alpha(P \parallel Q)\beta$. The formula can be directly rewritten as $P \parallel Q \models \alpha \to \beta$, and this can be proved by using a model checker for the model $P \parallel Q$ and property $\alpha \to \beta$.

The second formula represents a proof strategy which we call a forward proof of $\models \alpha(P \parallel Q)\beta$. The formula can be directly rewritten as $((\alpha\overrightarrow{P})\overrightarrow{Q}) \to \beta$, and this represents the method of pushing α forward, towards β, via the chain $P \parallel Q$.

The third formula represents a proof strategy which we call a backward proof of $\models \alpha(P \parallel Q)\beta$. The formula can be directly rewritten as $\alpha \to (\overleftarrow{P}(\overleftarrow{Q}\beta))$, and this represents the method of pulling β backward, towards α, via the chain $P \parallel Q$.

The three strategies seem to be equally difficult since the formulas which characterise them have the same complexity, in terms of their size. However, the forward and backward strategies can easily be improved, and thus have an advantage over the monolithic strategy.

In case of the forward proof, instead of forming the whole implication $((\alpha \wedge \mathcal{P}) \wedge \mathcal{Q}) \rightarrow \beta$ in one step, one can first form $\alpha \wedge \mathcal{P}$ (thus, $\alpha\overrightarrow{P}$), simplify it to some γ, then form $\gamma \wedge \mathcal{Q}$ (thus, $\gamma\overrightarrow{Q}$), simplify it to some γ', and finally form the simpler implication $\gamma' \rightarrow \beta$.

Similarly, in case of the backward proof, instead of forming the whole implication $\alpha \rightarrow (\mathcal{P} \rightarrow (\mathcal{Q} \rightarrow \beta))$ in one step, one can first form $\mathcal{Q} \rightarrow \beta$ (thus, $\overleftarrow{Q}\beta$), simplify it to some γ, then form $\mathcal{P} \rightarrow \gamma$ (thus, $\overleftarrow{P}\gamma$), simplify it to some γ', and finally form the simpler implication $\alpha \rightarrow \gamma'$.

The improved forward and backward strategies are very similar. However, the backward proof has an advantage over the forward proof, under the verification scenario presented in the Introduction.

3.2.2 Synchronous observers

In the rest of this paper we will analyse the backward proof strategy in the context where both α and β, in $\alpha P \beta$, are synchronous observers [39].

A synchronous observer is a process, say O, with one distinguished output, say o. It is denoted by $(O\!:\!o)$. Observers specify safety properties of processes. Intuitively, process P satisfies property $(O\!:\!o)$ iff $P \,\|\, O$ always emits o. Formally,

$$\begin{array}{rll} P \models (O\!:\!o) & \text{iff} & P \,\|\, O \models \Box o \\ & \text{iff} & \models \mathcal{P} \wedge \mathcal{O} \rightarrow \Box o \\ & \text{iff} & \models \mathcal{P} \rightarrow (\mathcal{O} \rightarrow \Box o) \\ & \text{iff} & P \models \mathcal{O} \rightarrow \Box o \end{array}$$

Thus, the property specified by the observer $(O\!:\!o)$ can be defined by formula $\mathcal{O} \rightarrow \Box o$. We call this semantics a *weak* interpretation of observer $(O\!:\!o)$. Another semantics of observer $(O\!:\!o)$ can be given by formula $\mathcal{O} \wedge \Box o$. We call this semantics a *strong* interpretation of observer $(O\!:\!o)$.

Strong observers are used to specify assertions and weak observers are used to specify commitments. In other words, we only consider triples $(A \rhd a)P(B \lhd b)$, for some observers $(A\!:\!a)$ and $(B\!:\!b)$. In this formula, $(A \rhd a)$ denotes a strong observer and $(B \lhd b)$ denotes a weak observer. Such triples are characterised by formula

$$\mathcal{A} \wedge \Box a \wedge \mathcal{P} \rightarrow (\mathcal{B} \rightarrow \Box b)$$

which is equivalent to

$$\mathcal{A} \wedge \mathcal{P} \wedge \mathcal{B} \rightarrow (\Box a \rightarrow \Box b)$$

It turns out that the construction of the strongest post-condition $\alpha\overrightarrow{P}$ and the weakest-precondition $\overleftarrow{P}\beta$ is surprisingly easy, when α and β are observers, and P, α and β are specified by circuits. This is in contrast with the methods of synthesising the weakest-precondition via manipulations on Mealy machines, as presented in [39, 38].

Let $(O : o)$ be an observer of process P and consider the observer $(O \,\|\, P : o)$ obtained by putting process O in parallel with process P.

Fact 3 *For* $\alpha = (O \rhd o)$ *and* $\beta = (O \lhd o)$, $\alpha\overrightarrow{P} = (O \,\|\, P \rhd o)$ *and* $\overleftarrow{P}\beta = (P \,\|\, O \lhd o)$.

Proof: $\alpha\overrightarrow{P} = \alpha \wedge \mathcal{P} = \mathcal{O} \wedge \Box o \wedge \mathcal{P} = (\mathcal{O} \wedge \mathcal{P}) \wedge \Box o = (O \,\|\, P \rhd o)$
$\overleftarrow{P}\beta = \mathcal{P} \rightarrow \beta = \mathcal{P} \rightarrow (\mathcal{O} \rightarrow \Box o) = \mathcal{P} \wedge \mathcal{O} \rightarrow \Box o = (P \,\|\, O \lhd o)$ □

The algorithm for incremental proof. The proof of $\models (A \rhd a)(P \,\|\, Q)(B \lhd b)$ can proceed as follows:

1. Form the weakest observer

$$\overleftarrow{Q}\beta := (Q \,\|\, B \lhd b)$$

   and simplify $Q \,\|\, B$ to $Q' \,\|\, B'$, using some *sequential circuit optimizer,* such as [42]. In addition, one can use some special-purpose algorithms, for example [41, 40], to further reduce the number of latches in $Q \,\|\, B$.

2. Form the weakest observer

$$\overleftarrow{P}(\overleftarrow{Q}\beta) := \overleftarrow{P}(Q' \,\|\, B' \lhd b) = (P \,\|\, Q' \,\|\, B' \lhd b)$$

   and simplify $P \,\|\, Q' \,\|\, B'$ to $P' \,\|\, Q'' \,\|\, B''$, as above.

3. Prove $\models \alpha \rightarrow \overleftarrow{P}(\overleftarrow{Q}\beta)$ by using either a tautology checker on formula

$$\mathcal{A} \wedge \Box a \rightarrow (\mathcal{P}' \wedge \mathcal{Q}'' \wedge \mathcal{B}'' \rightarrow \Box b)$$

   or a model checker on

$$A \,\|\, P' \,\|\, Q'' \,\|\, B'' \models \Box a \rightarrow \Box b$$

Some properties for checking can get true just by circuit optimization, so neither the tautology checker nor the model checker is needed.

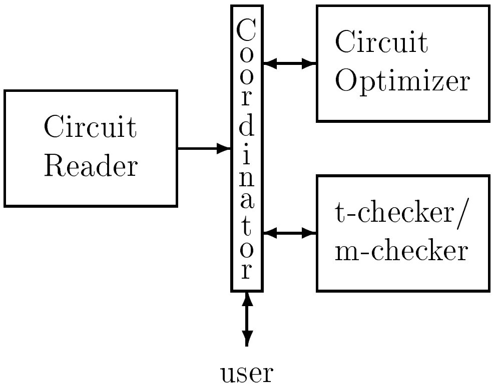


Figure 3.3: The software tool

3.2.3 The software tool

The software tool which implements our incremental method is quite simple. It is fed with the formula $(A \triangleright a)(P_1 \parallel \cdots \parallel P_n)(B \triangleleft b)$, given as $n+2$ files which contain descriptions of respective circuits, and returns either a confirmation that the formula is valid, or a counter-example (a trace) if the formala is not valid.

The tool consists of four main software components arranged in the structure depicted in figure 3.3. The input circuits are read in by `CircuitReader` and converted to the form suitable as input to `CircuitOptimizer`. `Coordinator` implements the loop in which the successive weakest observers are formed and fed to the optimizer. The loop is finished if either the optimizer returns the "empty" observer *true*, in which case the input formula is valid, or there are no more processes to be processed. In the later case, the coordinator invokes the tautology (or model) checker (component `t-checker/m-checker`) and reports the result.

Notice that this simple architecture allows to parameterize the tool, in a relatively easy way, with different combinations of `CircuitOptimizer` and `t-checker/m-checker`.

The tool can be used to experiment with different heuristics (expressed as scripts for hardware optimizer) for improving efficiency of proving $\alpha(P \parallel Q)\beta$, compared to the usual monolithic strategy.

Chapter 4

Code generation

4.1 Distributed code generation : SACRES method

The overall method is illustrated in the figures 4.1, 4.2, and 4.3.

Figure 4.1 shows what the designer has to do. The designer has on her/his

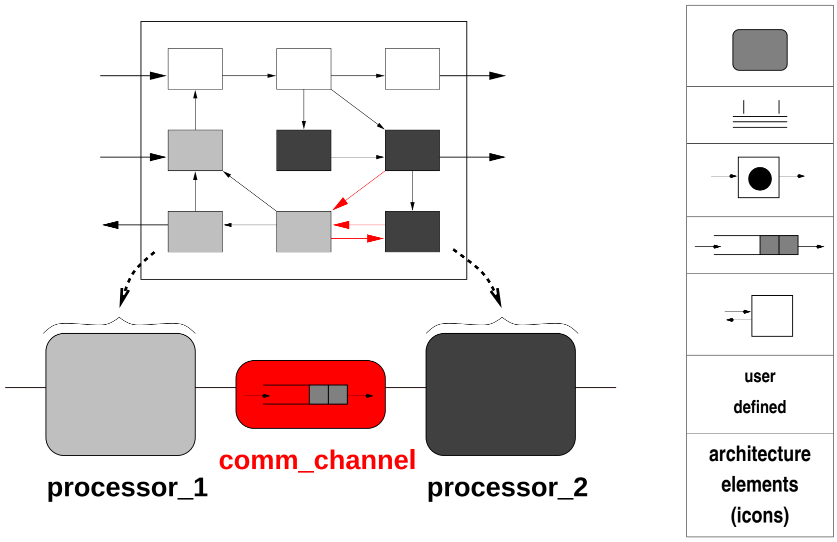


Figure 4.1: *Code generation : what the designer does.*

screen (at least) three windows. The first window – top left – is the (SIGNAL, or DC_+, or activitycharts in STATEMATE) program editor. In this window, a dataflow diagram is depicted. The arrows would typically depict flows of data, but also could correspond to scheduling requests. In the top right window, icons are shown which allow the designer to specify her/his target architecture. This architecture has two types of constitutive elements. The first one (on the top) is a *processor,* i.e. a machine that complies with the synchronous model of execution, in which a run is a sequence of atomic reactions. Thus processors can be pieces of sequential code (C/C$_{++}$-procedures, threads, etc.), or alternatively parallel machines running according to the model of perfect synchrony (e.g., synchronous hardware). Other icons refer to (generally *a*synchronous) communication media, they are discussed in section 4.4. Using these two windows, the designer builds, in the third window, her/his execution architecture: the source dataflow diagram is partitioned as shown in the figure, and corresponding subdiagrams are mapped onto "processors" by click-and-point. Also, models of communication links are specified by the designer, by clicking-and-pointing to the appropriate icon.

Figure 4.2 shows what the tool generates. From the specifications pro-

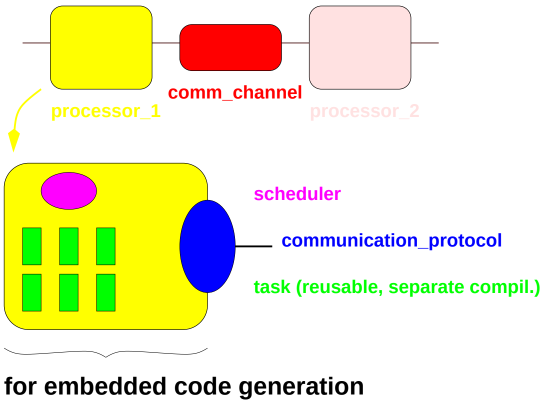


Figure 4.2: *Code generation: what the tool generates.*

vided by the designer as in Figure 4.1, the tool generates, for embedding into each processor, the following: 1/ a suitable communication protocol which guarantees that the semantics of synchronous communication will be preserved even though an asynchronous communication medium is used (the **first problem**, cf. below); 2/ a structuring of the code into pieces of sequential code (the so-called "tasks") and a scheduler, aiming at guaranteing separate compilation and reuse (the **second problem**, cf. below).

Figure 4.3 shows the use of the overall architecture model for simulation and profiling. The whole model of this figure (processors and channel models)

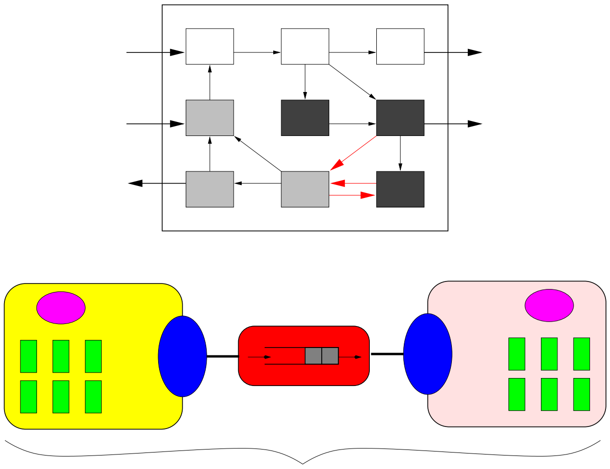


Figure 4.3: *Code generation : simulation and profiling.*

can be used for architecture simulation. In addition, it can be enhanced for the purpose of profiling as follows. Duplicate the architecture model: the two models share their inputs. For x an input variable of the architecture model, we associate its sequence of *dates of availability* δ^x : if x_n is the nth present occurrence of x, then δ^x_n delivers the date of availability for x_n, measured in terms of some underlying physical quantitative clock (say, in microseconds, or in basic machine cycle when this makes sense). For each x_n delivered to the original model, we deliver δ^x_n to its copy. Then, in the copy, computations or actions specified are replaced by their corresponding quantitative time subsumption (when can the outputs be delivered, given the date of availability of the inputs). Then, the copy of our architecture model would deliver, for each output y of interest in the architecture model, a corresponding sequence δ^y of earliest dates of availability.

A particular case of this whole approach is implemented by TNI within the SILDEX tool, it is discussed in section 4.4, and shown in Figure 4.7.

As indicated before, two major issues need to be considered :

1. **First problem :** *relaxing synchrony* is needed if distribution over possibly asynchronous media is desired without paying the price for maintaining the strong synchrony hypothesis via costly protocols.

2. **Second problem :** *designing modules equipped with proper interfaces*

for subsequent reuse, and generating a correct scheduling and communication protocol for these modules, is the key to modularity.

We consider these two issues next.

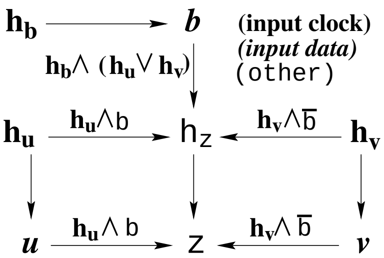
4.2 Relaxing synchrony

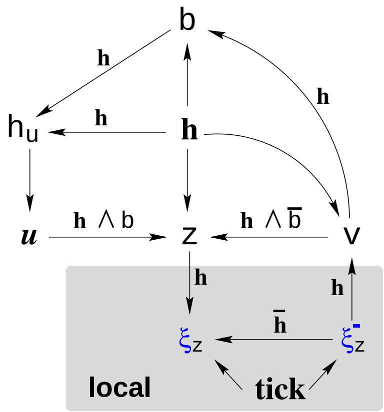
The major problem is that of testing for absence in an asynchronous environment! It is generally not possible, in an asynchronous environment, to decide upon presence/absence of a signal relatively to another one. The major problem is that of "testing for absence" as being part of the control. While this is perfectly sound in a synchronous paradigm, this is meaningless in an asynchronous one.

4.2.1 Desynchronising while keeping the semantics of perfect synchrony

The solution consists in restricting ourselves to so-called *endochronous* STS. Endochronous STS are those for which the control depends only on 1/ the past state, and 2/ the values possibly carried by environment signals, but not on the presence/absence status of these signals. For an endochronous STS, loosing the synchronisation barriers that define the successive reactions will not result in changing its semantics; this is formalized in [1].

An example of an STS which is "exochronous" (i.e., not endochronous) is the "reactive" STS given on the left hand side of the following picture, whereas the "proactive" STS shown on the right hand side is an endochronous STS:

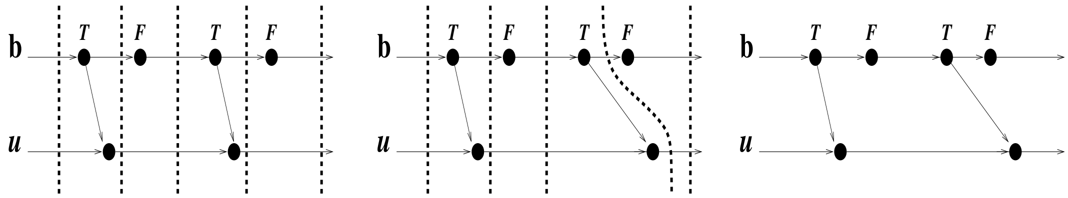




In the diagram on the left hand side, three different clocks are source nodes of the directed graph. This means that the primary decision in executing a reaction consists in deciding upon relative presence/absence of these clocks. In contrast, in the diagram on the right hand side, only one clock, the activation clock h, is a source node of the graph. Hence no test for relative presence/absence is needed, and the control only depends on the value of the boolean variable b, which is computed internally.

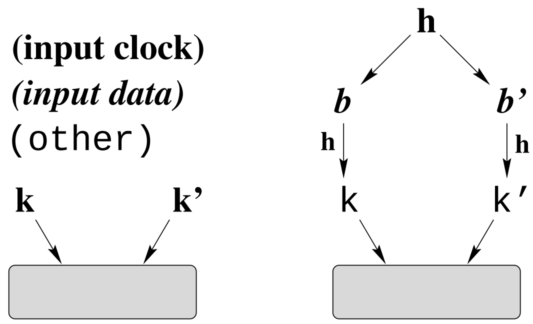
How endochrony allows us to desynchronize an STS is illustrated in an intuitive way on the following diagram, which depicts the scheduling specification associated with the (endochronous) pseudo-statement

" **if** b **then** get_u" :



In the left diagram, a history of this statement is depicted, showing the successive instants (or reactions) separated by thick dashed lines. In the middle, an instant has been twisted, and in the last one, thick dashed lines have been removed. Clearly, no information has been lost : we know that u should be got exactly when $b =$ T, and thus it is only needed to wait for b in order to know whether u is to be waited for also. A formal study of desynchronization and endochrony is presented in [1].

Moving from exochronous to endochronous is easily performed, we only show one typical but simple example :



The idea is to add to the considered STS a monitor which delivers the information of presence/absence via the b, b' boolean variables with identical clock h, i.e., $\{k = \mathrm{T}\} \equiv \{b = \mathrm{T}\}$, and similarly for k', b'. The resulting STS is endochronous, since boolean variables b, b' are scrutinized at the pace of activation clock h. Other schemes are also possible, this is discussed in [1].

4.2.2 Maintaining synchronous semantics of composition while using *a*synchronous communication media

The second question is that of preserving the semantics of synchronous composition when an asynchronous communication medium is used. Requirement for such a medium is that: 1/ it should not loose messages, and, 2/ it should not change the order of messages. In [1], it is shown that the condition for this is that of the *isochrony* of the considered pair of STS. Roughly speaking, a pair of STS is isochronous if it is enough for two respective reactions of each STS to agree on *present* common variables for their carried values, in order to guarantee that these reactions agree on *all* common variables for their presence/absence status as well as carried values when they are present. Thus, again, common agreement for reaction composition can disregard absence.

4.3 Generating scheduling for separate modules

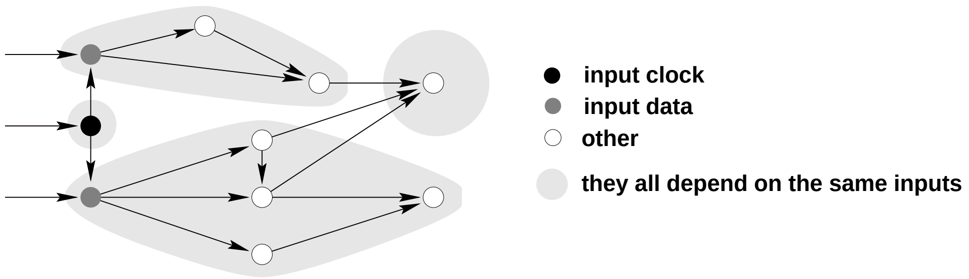
Relevant target architectures for embedded applications typically are 1/ purely sequential code (such as C-code), 2/ code using a threading or tasking mechanism provided by some kind of a real-time OS (here the threading mechanism offers some degree of concurrency), or 3/ DSP-type multiprocessor architectures with associated communication media.

On the other hand, the scheduling specifications we derive from rules (R-1,...,R-4) of causality analysis still exhibit maximum concurrency. Actual implementations will have to conform to these scheduling specifications. In general, they will exhibit less (and even sometimes no) concurrency, meaning that further sequentialization has been performed to generate code.

Of course, this additional sequentialization can be the source of potential, otherwise unjustified, deadlock when the considered module is reused in the form of object code in some environment, this was illustrated in subsection 2.3.1. The traditional answer to this problem by the synchronous programming school has been to refuse considering separate compilation : modules for further reuse should be stored as source code, and combined as such before code generation.

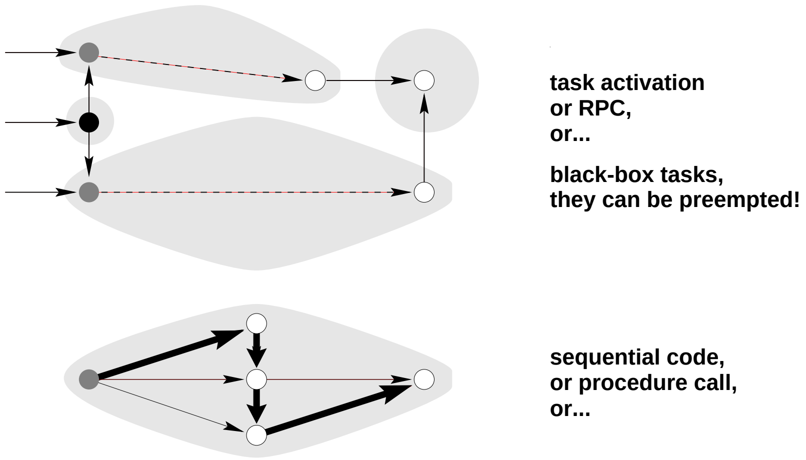
We shall however see that this does not need to be the case, however. Instead, a careful use of the scheduling specifications of an STS will allow us to decompose it into modules that can be stored as object code for further reuse, whatever the actual environment and implementation architecture will be.

The case of single-clocked STS. We first discuss the case of single-clocked STS, in which all variables have the same clock. The issue is illustrated in the following picture, in which the directed graph defining the circuitfree scheduling specification of some single-clocked STS is depicted :



In the above picture, the gray zones group all variables which depend on the same subset of inputs, let us call them "*tasks*". Tasks are not subject to the risk of creating fake deadlocks from implementation. In fact, as all variables belonging to the same task depend on the same inputs, each task can be executed according to the following scheme : 1/ collect inputs, 2/ execute task. The actual way the task is executed is arbitrary, provided it conforms the scheduling specification.

In the next picture, we show how the actual implementation will be prepared :



The thick arrows inside the task depicted on the bottom show one possible fully sequential scheduling of this task. Then, what should be really stored as source code for further reuse is only *the abstraction consisting of the task viewed as black-boxes, together with their associated interface scheduling specifications.*

In particular, if the supporting execution architecture involves a real-time tasking system implementing some preemption mechanism in order to dynamically optimize scheduling for best response time, tasks can be freely suspended/resumed by the real-time kernel, without impairing conformity of the object code to its specification. The generalization to multiple-clocked STS is easy. The only new point is that all items discussed before are now clock-dependent. Thus the computations of the "tasks", their internal scheduling, and their abstraction, must be performed using scheduling specifications labelled by booleans.

4.4 Using channel models for distributed code generation

In Figure 4.4, basic channel prototypes are depicted, together with their associated partial order models. In this figure we depict the different types of channel models we consider. From top to down : a bus, synchronous rendezvous (figured by a token), an unbounded asynchronous fifo, a bounded fifo, and a shared memory. We do not mean that these are the only type of communication we can handle, we only say that these are different types

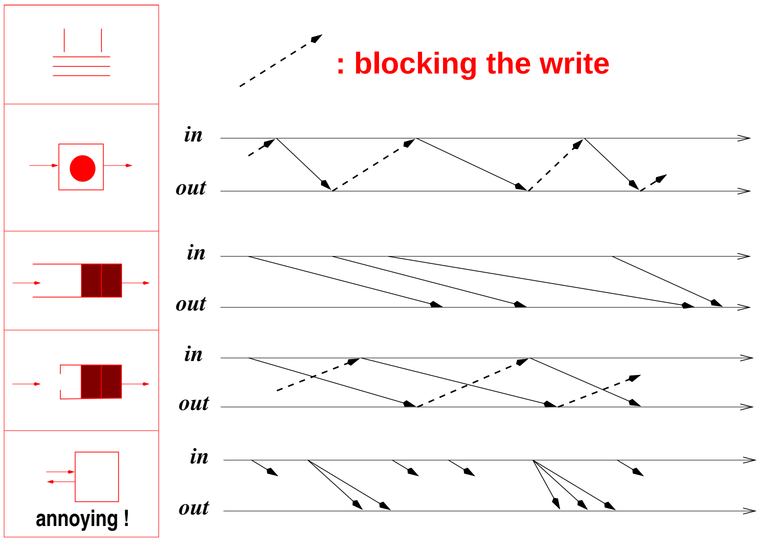


Figure 4.4: *Channels and partial order models.*

of communication services that can be found in most real-time executives or communication layers. On the right hand side, the behaviour of these channels is specified using scenarios. Horizontal lines figure physical time elapsing. Downgoing arrows (from *in* to *out*) depict the effective transfer of data. Upgoing arrows (from *out* to *in*, dashed) show the blocking of the writing exercised by each considered type of channel. No such blocking is exercised neither by the unbounded asynchronous fifo, nor by the shared memory. Also, for the shared memory, the scenario reflects the fact that multiple writings and readings can occur.

In Figure 4.5, the partial order channel models of Figure 4.4 are redrawn according to the paradigm of perfect synchrony. This means that communication takes no time: the input-to-output arrows are now vertical. Of course, the upgoing arrows (dashed) are redrawn accordingly. Let us focus on the first case of the rendez-vous. Rename, for convenience, $in : x$ and $out : y$. Then, our scenario reads $\forall n > 1 : x_n \rightarrow y_n$ and $y_{n-1} \rightarrow x_n$. Setting $z_n = y_{n-1}$ (formally: "**if** h_y **then** $z = \xi_y^-$ **else** $z = \perp$", we can translate this scenario using STS scheduling specifications as:

$$\begin{array}{ll} & x \xrightarrow{h} y \\ \bigwedge & z \xrightarrow{h} x \end{array}$$

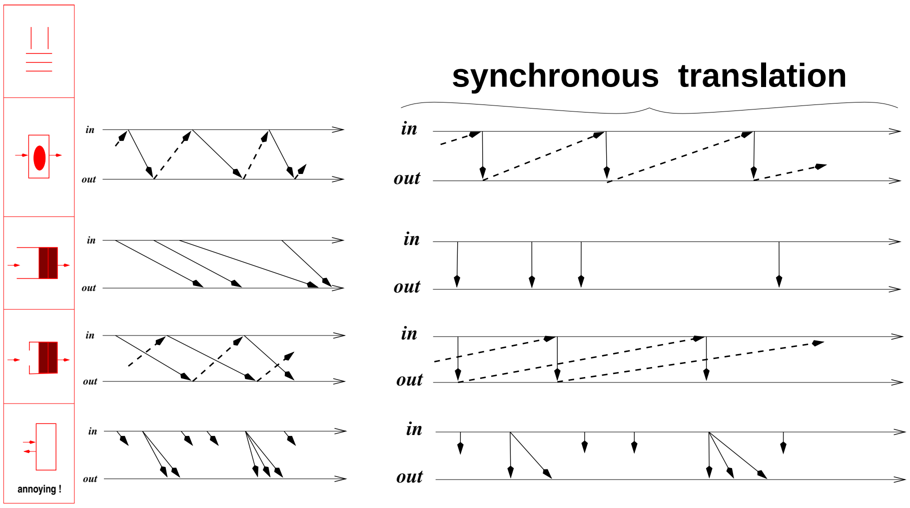


Figure 4.5: *Channels : synchronous partial order models.*

$$
\begin{array}{ll}
\bigwedge & h = h_x = h_y = h_z \\
\bigwedge & \textbf{ if } h \textbf{ then } z = \xi_y^- \textbf{ else } z = \perp \\
\bigwedge & \textbf{ if } h \textbf{ then } y = x
\end{array} \tag{4.1}
$$

For an unbounded fifo, we simply get :

$$
\begin{array}{l}
\qquad x \xrightarrow{h} y \\
\bigwedge \quad h = h_x = h_y \\
\bigwedge \quad \textbf{ if } h \textbf{ then } y = x
\end{array}
$$

For a 2-bounded fifo, we use the same specification as in (4.1), where z is now the output of a 2-register with input y. The scenario for the shared memory reflects nondeterminism of this medium : clocks h_x and h_y are independent and the STS model is :

$$\textbf{if } h_y \textbf{ then } y = \xi_x \textbf{ else } y = \perp .$$

Thus some of the x's are lost (when $h_x = \text{T}$ but $h_y \perp$), which models multiple writings. And multiple readings of the same value also occur. Regarded as an STS with x as input and y as output, this is a nondeterministic STS.

Clearly, user defined models of channels can be handled in the same way.

Then, assume we have been able to ensure the following for the architecture model of Figure 4.3 :

($\mathbf{C}_1$) Each of the two processors P_1, P_2 is endochronous.

($\mathbf{C}_2$) The pair (P_1, P_2) is isochronous.

($\mathbf{C}_3$) The compound model $P_1 \bigwedge C \bigwedge P_2$ (where C is the channel model according to Figure 4.5) is executable — in particular it is non blocking and deterministic.

Note that **($\mathbf{C}_1$,$\mathbf{C}_2$)** only involve the embedded code for each processor, not the channel model. Also, note that **($\mathbf{C}_3$)**rules out the shared memory type of channel, due to its nondeterminism. Then :

1. As channel models of rendez-vous or fifo type guarantee message losslessness and order preserving, properties **($\mathbf{C}_1$,$\mathbf{C}_2$)** guarantee that the STS semantics is preserved even under asynchrony of communications.

2. For the case of rendez-vous or bounded fifo, bounded desynchronisation is guaranteed, by **($\mathbf{C}_3$)**. This guarantees use of bounded memory (including channel memory), as well as bounded response time, an important issue for real-time implementations.

The synchronisation of such an architecture is shown in Figure 4.6 In this

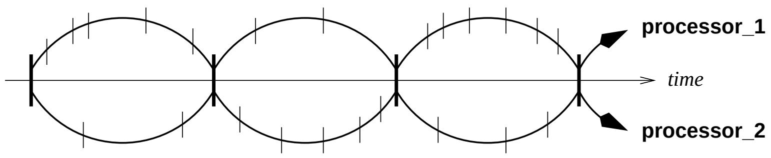


Figure 4.6: *Structure of the synchronisation.* Thick vertical bars indicate synchronisation points, while thin vertical bars indicate reactions that are local to each processor : the latter occur freely and asynchronously.

figure, two runs are depicted, one for each processor of our two-processor architecture. Synchronisation occurs from time to time, and in between, each processor evolves freely, asynchronously from the other processor.

A particular case of this whole approach is implemented by TNI within the SILDEX tool, it is shown in Figure 4.7. Requested channel properties are

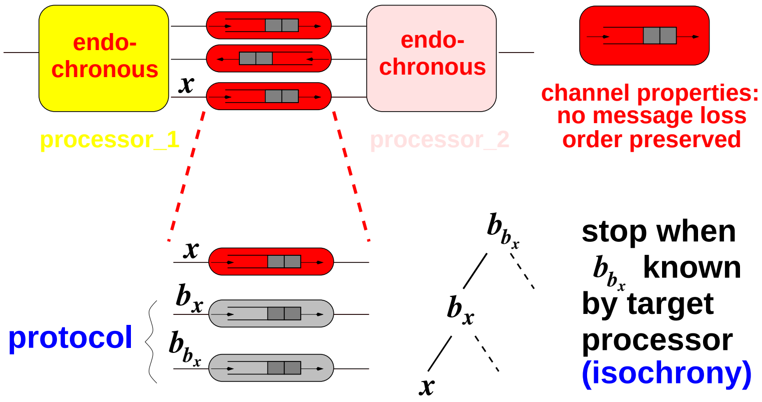


Figure 4.7: *Implementation of the approach within the* SILDEX *tool.*

indicated. For each variable, say x, involved in an asynchronous communication, the added protocol is indicated in gray. It consists of transmitting a boolean variable b_x whose true occurrences indicate when x is present. If the clock of b_x is not known to the receiver, then it is again transmitted in the form of the true occurrences of a boolean variable b_{b_x}, and so on (this process terminates). Then, optimizations can occur in order to remove unnecessary transmission of boolean variables. In doing so, clearly, isochrony of the pair of processes, with their associated protocol included, is guaranteed. Thus distributed implementation is correct by construction. Bounded memory and bounded reaction time is then a separate issue, which can be checked independently.

Chapter 5

Code validation

Translation validation is an alternative to the verification of translators (compilers, code generators). Rather than proving in advance that the compiler always produces a target code which correctly implements the source code (compiler verification), each individual translation (i.e. a run of the compiler) is followed by a validation phase which verifies that the target code produced on this run correctly implements the submitted source program. In order to be a practical alternative to compiler verification, a key feature of this validation is its *full automation*.

Since the validation process attempts to "unravel" the transformation effected by the translators, its task becomes increasingly more difficult (and necessary) with the increase of sophistication and variety of the optimizations methods employed by the translator.

In the context of SACRES, we address the practicability of translation validation for highly optimizing, industrial code generators from $\mathrm{D}\textsc{c}_+$, a widely used intermediate format for synchronous languages, to C. We introduce new abstraction techniques as part of the automation of our approach. The reader is referred to [43] for additional details.

In this chapter we explain the theory underlying (fully automatic) translation validation for two industrial compilers from $\mathrm{D}\textsc{c}_+$ to C. These compilers – which apply more than 100 optimization rules during code generation [50] – are developed in the ESPRIT project SACRES by the French company TNI and by Inria (Rennes) and are used by Siemens, SNECMA and British Aerospace. Their formal verification is prohibitive due to their size (more than 20.000 lines of code each) and the fact that they are constantly improved/extended.

While developed in the context of code generators for synchronous languages, the proposed method has wider applicability. The main feature which enables us to perform the validation task algorithmically is that the source language has a restricted explicit control structure. This is also represented by the fact that the resulting C-code consists of a single main loop whose body is a loop-free program. Source languages with these features can benefit from the method proposed in this paper. For example, the language UNITY [47] which comes from the world of asynchronous distributed systems is a possible customer for the proposed method.

A major advantage of a carefully designed translation validation tool is that it can replace the need for correctness proofs for *various* compilers if these compilers are based on the same definition of "correct code generation". This is the case for the TNI and the Inria compiler and, indeed, TVT is used to validate code originating from either of these two compilers.

Related work: In [49] we addressed translation validation for a compiler from Signal to C which did not perform optimizations while generating the C code. The revision of this work to deal with the optimizing compilers from TNI and Inria is the topic of this paper.

The work in [48] performs translation validation on a purely syntactic level. Their method is based on finding a bijection between abstract and concrete instruction sets (resp. variables) because they are considering a *structural* translation of one sequential program into another sequential program. Since we are dealing with optimizing compilers we have to employ a purely *semantic* approach which is far more involved than finding a syntactic correspondence between programs. We are not aware of any other translation validation approaches for industrial compilers.

5.1 Compilation of Multi-clocked Synchronous Languages

The compilation scheme for multi-locked synchronous languages (e.g. Signal, DC_+) to imperative, sequential languages (e.g. C, ADA) looks as follows. The set of equation and memorization statements of a program D form a *linear equation system* LES on the flows of D and their associated clocks. Solutions of LES for a given set of input/register values determine the next state of the system. The compiler derives from D an imperative program C

which consists of one main loop whose task is to repeatedly compute such solutions of the LES.

For these solutions to be efficiently computable without expensive fixpoint iteration, only programs which induce an *acyclic dependency relation* on flows are compiled. This dependency relation is not static but rather depends on which flows are currently present. Thus, the compiler computes from the source program another linear equation system CC – the, so called, *clock calculus* [45] – which records the dependencies amongst clocks and a *conditional* dependency graph CDG on flows.

If CC has a unique solution for all possible inputs and CDG is acyclic, then system LES is uniquely solvable for all possible input/register values, i.e. the program D is deadlock free and determinate in its input/register variables. A formal definition of determinacy is given in Section 5.2.1. The program is rejected by the compiler if one of the conditions is not satisfied. Otherwise the produced code contains statements originating from the clock calculus and assignments to variables (representing the flows of D) whose order must be consistent with the dependency graph CDG. These assignments are performed if the corresponding flow is currently present in the source program, i.e. the clocks of flows determine the control structure of the generated program.

Example: A part of the (non-optimized) translation of a Dc$_+$ program into C code is given below:

```
.....                                    WHILE true DO
r_in =m= in              { .....
out = when(2*r_in,in>10)                 read(in);
.....                                    c_out = (in>10);
                                         IF c_out { .......... ;
                                                    out = 2*r_in;
                                                    write(out);
                                                  }
                                         r_in = in;
                                         ........
                                         }
```

For the translation validation process also the C programs are translated into the STS formalism. Since the generated C code uses in the body of the main loop only a small fragment of ANSI C (e.g. no pointers, no loops), the translation is straightforward. Note, however that the C programs use *persistent* variables (i.e. variables which are never absent) to implement Dc$_+$

programs which use volatile variables. This has to be taken into account when defining the notion of "correct implementation" in the next section.

5.2 Correct Implementation: Refinement

Our approach to establish that "the C-code correctly implements the DC_+ source" is based on the notion of *refinement*. The presented concepts have been approved by TNI and Inria.

5.2.1 Refinement and Refinement Mappings

For A an STS, we shall denote by $[\![A]\!]$ the semantics of A, i.e., the set of all its runs. Consider the two STSs $A = \langle V_A, \Theta_A, \rho_A \rangle$ and $C = \langle V_C, \Theta_C, \rho_C \rangle$, with $IO_A = IO_C$, to which we refer as the *abstract* and *concrete* system, respectively. We say that C *refines* A, denoted by C ref A, if for any $\sigma = s_0, s_1, s_2, \ldots$ in $[\![C]\!]$ there exists a $\tau = t_0, t_1, t_2, \ldots$ in $[\![A]\!]$ such that

$$\forall x \in IO_A . \forall i \in \mathbb{N}.\ s_i[x] = t_i[x] \text{ or } \ t_i[x] = \perp.$$

In order to establish this notion of refinement for two given systems we have to *construct* for each concrete computation $\sigma \in [\![C]\!]$ the corresponding abstract computation $\tau \in [\![A]\!]$ such that the above property is satisfied. Such constructions are usually done by means of *refinement mappings* [44]. Rather than the standard static correspondence between concrete and abstract variables, we need a more general mechanism which relates *persistent* variables of the STS-representation of the C-code (denoted C-STS from now on) to *volatile* variables of the STS-representation of the DC_+ program (DC+STS).

Definition 4 *Given systems* $A = \langle V_A, \Theta_A, \rho_A \rangle$ *and* $C = \langle V_C, \Theta_C, \rho_C \rangle$ *with* $IO_A = IO_C$*. A mapping* $f : \Sigma_C \to \Sigma_A$ *is a* clocked refinement mapping *from* C *to* A *if it satisfies the requirements of*

- *Initiation:* $s \models \Theta_C$ *implies* $f(s) \models \Theta_A$*, for all* $s \in \Sigma_C$.
- *Propagation:* $(s, s^-) \models \rho_C$ *implies* $(f(s), f(s^-)) \models \rho_A$*, for all* $s, s^- \in \Sigma_C$.
- *Preservation of Observation:* $\forall x \in IO_A . \forall s \in \Sigma_C .\ f(s)[x] = s[x]$ *or* $f(s)[x] = \perp$.

The idea of this definition is, that in each time instant and for each observable variable $x \in IO_A = IO_C$ either x is present in the abstract system and $f(s)[x]$ coincides with $s[x]$ or x is absent in $f(s)$. In the following presentation we omit the qualifier "clocked" if it is clear from the context. Refinement mapping is illustrated in the Figure 5.1.

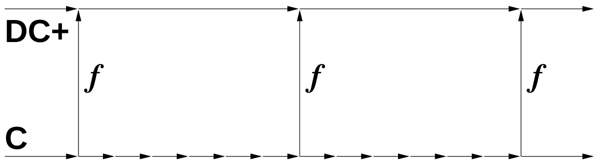


Figure 5.1: *Refinement mapping.*

Theorem 1 *If there exists a clocked refinement mapping from C to A then C* ref *A.*

Usually, finding such a mapping f is left to the ingenuity of the verifier. In the context of translation validation it is essential that f can be *automatically constructed* from the source and target programs.

In order to facilitate the generation of suitable refinement mappings, we perform a transformation of the transition relation of C-STS. Originally, the transition relation in C-STS updates variables *successively* till finally the result of the *simultaneous* update of variables in DC+STS has been computed (these steps correspond to one iteration of the main loop of the C-program, cf. Example 5.1). We construct a new transition relation which reflects the *accumulated* effect of the individual steps in the execution of the loop's body. This compression of the transition relation ρ of C-STS is done by a standard substitution algorithm (note, that the body of the main loop is itself loop-free). The size of the resulting transition relation is smaller than the size of the original transition relation. From now on we consider the generation of refinement mappings for DC+STS and C-STS where C-STS uses the compressed transition relation.

The main idea in [49] was to generate refinement mappings which reconstruct the values of *all* abstract variables. In order to do so, it was necessary to extract from the structure of the C-code the information whether an abstract variable is currently present/absent, i.e. we reconstructed the clocks of these variables. With this information about clocks we could define the correct values of abstract variables from the values of their concrete counterparts. Such a reconstruction of all abstract variables is not possible in the case of the optimizing code generators, because:

1. Internal variables are possibly eliminated for space efficiency during compilation; so there are no corresponding variables in the C-code from which we could automatically reconstruct their values.

2. The reconstruction of the clocks of abstract variables was based in [49] on the assumption that an abstract variable is present iff the corresponding concrete variable has been updated in the current iteration (cf. Example 5.1). The optimizing compilers *move assignments* between if-blocks in the C-code such that neither the fact that a concrete variable has been written implies that its abstract counterpart is actually present nor does the presence of an abstract variable implies that its concrete counterpart is written in the current iteration.

Since the code generators cannot eliminate IOR variables without producing incorrect code, we can exploit the property of *determinacy* – which is a central property in synchronous programs, – to *implicitly* reconstruct local abstract variables.

Definition 5 *An* STS $S = \langle V, \Theta, \rho \rangle$ *is* determinate in $V \subseteq V_S$ *if the following holds:*

$$\forall s_1, s_2, s_3 \in \Sigma_S . \, ((s_1, s_2) \models \rho \wedge (s_1, s_3) \models \rho \wedge s_2[V] = s_3[V]) \Rightarrow s_2[V_S \backslash V] = s_3[V_S \backslash V]$$

Determinacy of S in V says that, after a transition, the value of variables in set $V_S \setminus V$ are uniquely determined once the values for the variables in V have been fixed. $\text{D}\textsc{c}_+$ programs generated from Esterel, Lustre etc. are by construction determinate in their IRM variables. Thus, from now on we always assume that the considered STS representations of $\text{D}\textsc{c}_+$ programs are determinate in their set of IRM variables.

In order to determine corresponding abstract states it thus suffices to reconstruct these IRM variables by the refinement mapping. Besides this we have to reconstruct the values of abstract output variables to check whether the generated abstract and concrete outputs indeed coincide. Consequently we now concentrate on a refinement approach which allows to reconstruct the abstract IORM variables. Fortunately, the clock generation scheme as presented in [49] – and briefly recalled in Section 5.2.2 – can still be applied for IOR variables [50]. The reason is that a "re-timing" of IOR variables (i.e. moving assignments to other if-blocks as explained in point 2) tends to an incorrect implementation since either inputs/register variables would be provided at wrong time instants or/and outputs would be generated at instants where they are not supposed to occur. For memorization variables no clock has to be constructed since they always carry non-bottom values by definition.

Technically we eliminate all local variables in DC+STS $= \langle V, \Theta, \rho \rangle$ by removing them from V, removing their initializations from Θ and hiding them from ρ by existential quantification. The result of applying this transformation to some STS A is denoted by $A^{\exists}$. Determinacy of A in IRM_A implies that it suffices to construct an inductive refinement mapping from C to $A^{\exists}$ to actually prove that C correctly implements A, i.e. C ref A.

However, there is one remaining problem with the reconstruction of the values of register variables. Registers are updated during one iteration of the main loop *after* they have been used in assignments of other variables, cf. Example 5.1. Thus, at the end of an iteration, register variables are already updated for the *next* iteration. So, the values of abstract register variables have to be reconstructed from the values of the corresponding variables at the *beginning* of the iteration while input/output/memorization variables can be reconstructed from the values of corresponding variables at the *end* of the iteration. This situation is handled by automatically inserting a *history variable* $h.r$ [44] into the C-code for each register variable r. This is done by inserting the assignment $h.r = r$ directly before each assignment to r in the C code. These history variables recall the previous value of register variables.

The constructions explained above (introduction of history variables for register variables, compression of ρ_c, hiding of local variables in ρ_a) allow us to use refinement mappings as introduced in Definition 4 to establish the correctness of the generated C-code w.r.t. the $\text{D}{\rm C}_+$ source program.

5.2.2 Syntactic Representation and Proof Rule for Refinement

In the quest for automating the translation validation process, we present in this section a *syntactic representation* of clocked refinement mappings and the associated proof rule. In this, we follow the ideas in [49]. Then, we describe how the components used in the proof rule can be computed, so that the translation validation process can be carried out fully automatically.

Consider two STSs A and C with $IO_A = IO_C$. Let $\alpha : V_A \longrightarrow \mathcal{E}(V_C)$ be a *substitution* that replaces each abstract variable $v \in V_A$ by an expression $\mathcal{E}_v$ over the concrete variables V_C. Such a substitution α induces a mapping between states, denoted by f_α. Let s_C be some concrete state in Σ_C. The abstract state $s_A \stackrel{\text{def}}{=} f_\alpha(s_C)$ corresponding to s_C under substitution α assigns to each variable $v \in V_A$ the value of expression $\mathcal{E}_v$ evaluated in s_C. In this way, a refinement mapping can be syntactically defined by means of an appropriate substitution α. Such a substitution α is defined to be *observation preserving* if $\forall v \in IO_A.\ \models (v)[\alpha] = v \vee (v)[\alpha] = \bot$, cf. Definition 4.

Let $\alpha : V_A \longrightarrow \mathcal{E}(V_C)$ be an observation preserving substitution		
R1.	$\Theta_C \Rightarrow \Theta_A[\alpha]$	Initiation
R2.	$\rho_C \Rightarrow \rho_A[\alpha]$	Propagation
	C ref A	

Rule ref: Proving Refinement

The proof rule can be easily adapted to incorporate an invariant of the concrete system. However, after compressing the concrete transition relation no invariant is needed since the compressed concrete transition relation and the abstract transition relation also coincide (if the C-code is indeed correct) on non-reachable states. The reason for this is, that code generators can not exploit any reachability information for optimizations. This is one of the key observations which facilitate automatic refinement proofs.

In order for rule ref to be useful in a fully automatic translation validation process, an appropriate substitution α has to be generated automatically.

Based on the previous explanations we can define the following generic substitutions α.

Definition 6 *Given $A \in$ STS, representing the* DC$_+$ *program where local variables have been eliminated, and $C \in$ STS, representing the compressed C-code. We define $\alpha : V_A \longrightarrow \mathcal{E}(V_C)$ by:*

$$\begin{array}{lll}
\alpha(v) = & \textit{if}\ \ \mathrm{clk}_c(v)\ \ \textit{then}\ \ v\ \ \textit{else}\ \ \bot & \textit{for all } v \in IO_A (= IO_C) \\
\alpha(r) = & \textit{if}\ \ \mathrm{clk}_c(r)\ \ \textit{then}\ \ h.r\ \ \textit{else}\ \ \bot & \textit{for all } r \in R_A (= R_C) \\
\alpha(x.r) = r & & \textit{for all } x.r \in M_A
\end{array}$$

This specific definition of α automatically yields observation preserving substitutions.

The algorithm for computing the clock expressions above works as follows. The construction is based on viewing the body of the main loop of the C-code as a (cyclic) directed graph where every edge e is labeled by either a *guard* $\gamma(e)$ (originating from if-conditions) or an action which can be a *read* of an input variable, a *write* of an output variable, or an assignment to a variable (cf. Example 5.1.

The clock expression $clk_c(v)$ is computed by considering the guards along paths leading to assignments to v (resp. read/write statements in case of input/output variables). Let $path(v)$ be the conjunction of all guards along a fixed path leading to an assignment of v and let $written(v)$ be the disjunction of the predicates $path(v)$ for all possible paths leading to an assignment of v. Then $clk_c(v)$ is obtained from $written(v)$ by replacing all register variables r by $h.r$ (recall that the register variables are already updated for the next iteration). The combination of the techniques and constructions mentioned above allow us to automatically extract two first order logic formulas (corresponding to **R1.** and **R2.** in rule ref) which state the correctness of the generated code if these formulas can be shown to be valid.

We would like to emphasize that the presented approach is immune against the optimizations performed by the industrial code generators that we consider. The proof technique exploits, in contrast to our previous work [49], only minimal knowledge about the code generation process. We only assume that $IORM$ variables are reconstructible which is the minimal requirement for the C-code to be a correct implementation of the $\mathrm{D}\textsc{c}_{+}$ source [50].

The generated proof obligations are infinite state assertions, thus just invoking a BDD-based decision procedure obviously does not work. Directly supplying them to a theorem prover such as PVS and starting proof strategies turned out to be far too slow.

Ref. [43] explains the theoretical basis for an efficient BDD-based validation of the proof obligations on the basis of *uninterpreted functions.* The introduced concepts are expected to have a wider applicability as a general decision procedure for fragments of first order logic.

Bibliography

[1] A. Benveniste, P. Le Guernic and P. Aubry. Compositionality in Dataflow Synchronous Languages: Specification & Code Generation, Inria Research Report RR-3310, Nov. 1997, `http://www.inria.fr/RRRT/publications-eng.html`, see also a revised version co-authored with B. Caillaud, Mar. 1998.

[2] SACRES consortium. EP 20897 Deliverable report : The semantic foundations of SACRES, March 1997.

[3] A. Benveniste, P. Le Guernic, Y. Sorel, and M. Sorine. A denotational theory of reactive synchronous systems. *Information and Computation*, 99 n^{o}2:192–230, 1992.

[4] T.P. Amagbegnon, L. Besnard and P. Le Guernic, "Arborescent canonical form of boolean expressions", Inria Research Report n^{o}2290, June 1994.

[5] T.P. Amagbegnon, L. Besnard and P. Le Guernic, "Implementation of the dataflow language SIGNAL", in *Programming Languages Design and Implementation,* ACM, 163–173, 1995.

[6] A. Benveniste, G. Berry, "Real-Time systems design and programming", *Another look at real-time programming*, special section of *Proc. of the IEEE,* vol. 9 n^o 9, September 1991, 1270–1282.

[7] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL languages and its semantics. *Sci. Comp. Prog.*, 16:103–149, 1991.

[8] A. Benveniste, P. Caspi, N. Halbwachs, and P. Le Guernic, Data-flow synchronous languages. In *A Decade of Concurrency, reflexions and per-*

spectives, REX School/Symposium, pages 1–45, LNCS Vol. 803, Springer Verlag, 1994.

[9] Gérard Berry, *The Constructive Semantics of Esterel,* Draft book, `http://www.inria.fr/meije/esterel`, December 1995.

[10] B. Caillaud, P. Caspi, A. Giraud, and C. Jard, “Distributing automata for asynchronous networks of processors”, *European Journal on Automated Systems (JESA)*, Hermes, 31(3), 503–524, May 1997.

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

[12] Sacres consortium, The Declarative Code DC+, Version 1.2, May 1996; Esprit project EP 20897: Sacres.

[13] N. Halbwachs, *Synchronous programming of reactive systems,*. Kluwer Academic Pub., 1993.

[14] L. Lamport, Specifying concurrent program modules, *ACM Trans. on Prog. Lang. and Sys.,* 5(2):190-222, 1983.

[15] L. Lamport, What good is temporal logic? In *Proc. IFIP 9th World Congress,* R.E.A. Mason (Ed.), North Holland, 657-668, 1983.

[16] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire, “Programming real-time applications with SIGNAL”, *Another look at real-time programming*, special section of *Proc. of the IEEE,* vol. 9 n^o 9, September 1991, 1321–1336.

[17] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1992.

[18] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Safety.* Springer-Verlag, New York, 1995.

[19] A. Pnueli, personal communication.

[20] J-R. Beauvais, R. Houdebine, P. Le Guernic, E. Rutten, and T. Gautier. A Translation of Statecharts and Activitycharts into Signal Equations, Inria Research Report RR-3397, April 1998, `http://www.inria.fr/RRRT/publications-eng.html`.

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

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

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

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

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

[26] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In W.-P. de Roever, editor, *Proceedings COMPOS'97,* Lecture Notes in Computer Science. Springer-Verlag, 1998.

[27] Werner Damm, Bernhard Josko, and Rainer Schlör. Specification and verification of VHDL-based system-level hardware designs. In E. Börger, editor, *Specification and Validation Methods,* pages 331-410. Oxford University Press, 1995.

[28] U. Brockmeyer and G. Wittich. Real-Time Verification of STATEMATEDesigns, in: Alan Hu and Moshe Vardi, edts., *Proc. CAV 98,* to appear.

[29] U. Brockmeyer and G. Wittich. Tamagotchis Need Not Die - Verification of STATEMATE Designs. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems,* Lecture Notes in Computer Science Vol 1384, pages 217-231, 1998.

[30] Th. Filkorn. Applications of Formal Verification in Industrial Automation and Telecommunication. FemSys, München, 1997

[31] F. Regensburger and Ae. Barnard. Formal Verification of SDL systems at the Siemens Mobile Phone Department. TACAS 1998. Lecture Notes in Computer Science, Springer Verlag 1998.

[32] M. Hölzlein, Th. Filkorn, P. Warkentin and M. Weiss. Eine Verifikationskomponente für HiGraph. Vol 1397 of VDI-Berichte, VDI VErlag, Düsseldorf, 1998.

[33] J. Bormann, J. Lohse, M. Payer and G. Venzl. Model Checking in Industrial Hardware Design. Proceedings 32nd Design Automation Conference, June 1995.

[34] K. Feyerabend and B. Josko. A visual formalism for real-time requirement specifications. In Miquel Bertran and Teodor Rus, Eds, *Transformation-based reactive systems development, Proc., 4th Int. AMAST Workshop on real-time systems and concurrent and distributed software, ARTS'97,* LNCS 1231, 156–168, Springer Verlag, 1997.

[35] L. Holenderski. Compositional verification of synchronous reactive systems. Draft Siemens report, Feb. 1998.

[36] G. Berry. The Foundations of Esterel, To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner,* G. Plotkin, C. Stirling and M. Tofte, editors, MIT Press, 1998.

[37] R. Alur, T. A. Henzinger, O. Kupferman, Alternating-time Temporal Logic, In W.-P. de Roever, editor, *Proceedings COMPOS'97,* Lecture Notes in Computer Science. Springer-Verlag, 1998.

[38] A. Aziz, F. Balarin, R. Brayton and A. Sangiovanni-Vincentelli, Sequential synthesis using S1S, *Proc. International Conf. on Computer-Aided Design ICCAD'95*, 1995.

[39] N. Halbwachs, F. Lagnier and P. Raymond, Synchronous observers and the verification of reactive systems, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Workshops in Computing, Springer Verlag, Twente, June 1993.

[40] E. Sentovich, H. Toma, and G. Berry, Efficient Latch Optimization Using Incompatible Sets, *International Digital Automation Conference DAC'97*, Anaheim, 1997.

[41] H. Toma, E. Sentovich, and G. Berry, Latch Optimization in Circuits Generated from High-Level Descriptions, *Proc. International Conf. on Computer-Aided Design ICCAD'96*, 1996.

[42] SIS: a system for sequential circuit synthesis, *Tech. Rep. UCB/ERL M92/41*, `ftp://ic.eecs.berkeley.edu/pub/Sis`, 1–45, May 1992.

[43] A. Pnueli, O. Shtrichman and M. Siegel. Translation validation for synchronous languages, Report Weizmann Institute of Science, 1998.

[44] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2), 1991.

[45] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language. *Science of Computer Programming*, 16, 1991.

[46] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem.* Springer, 1996.

[47] K. M. Chandy and J. Misra. *Parallel Program Design: a Foundation.* Addison-Wesley, 1988.

[48] A. Cimatti, F. Giunchiglia, and P. Pecchiari et al. A provably correct embedded verifier for the certification of safety critical software. In *CAV*, number 1254 in LNCS. Springer, 1997.

[49] A. Pnueli, M. Siegel, and E. Singermann. Translation validation. In *TACAS 98: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS. Springer-Verlag, 1998.

[50] private communications with TNI (BREST), Siemens (Munich) and Inria (Rennes).