Project 22703: SYRF

Deliverable 4.1

Code distribution

Contents

1 Motivations

Modularity is advocated as the ultimate solution for the design of large systems, and this holds in particular for embedded systems, and their software and architecture. Modularity allows the designer to scale down design problems, and facilitate reuse of predefined modules.

The mathematical translation of the concept of modularity is often that of *compositionality.* Paying attention to the composition of specifications [MP92] is central to any system model involving concurrency or parallelism. More recently, significant effort has been devoted toward introducing compositionality in verification with aiming at deriving proofs of large programs from partial proofs involving (abstractions of) components [MP95].

Compilation and code generation has been given less attention from the same point of view, however. This is unfortunate, as it is critical for the designer to scale down the design of large systems by 1/ storing modules like black-box "procedures" or "processes" with minimal interface description, and 2/ generating code only using these interface descriptions, while still guaranteing that final encapsulation of actual code within these black-boxes together with their composition, will maintain correctness of the design w.r.t. specification.

This paper is devoted the issues of compositionality aiming at modular code generation, for dataflow synchronous languages. As dataflow synchronous is rather a paradigm more than a few concrete languages or visual formalisms [BB91], it was desirable to abstract from such and such particular language. Thus we have chosen to work with a formalism proposed by Amir Pnueli, that of Symbolic Transition Systems (STS [Pnu97]), which is at the same time very lightweight, and fully general to capture the essence of synchronous paradigm.

Using this formalism, we study composition of specifications, a very trivial topic indeed. Most of our effort is then devoted to issues of compositionality that are critical to code generation. As careless storing of object code for further reuse in systems design fails to work, we first concentrate on what are the additional features needed to abstract programs for the purpose of code generation: we show that a central notion is that of scheduling specification as resulting from a causality analysis of the given program. We pay strong attention to this notion, a whole appendix is devoted to its mathematical analysis, using a technique not unlike the one for the causality analysis of Esterel [Ber95]. Related issues of compositionality are investigated. Then we show that there is some appropriate level of "intermediate code", which at the same time allows us to scale down code generation for large systems, and still maintains correctness at the system integration phase. Finally we discuss the side issue of distributed implementation using an asynchronous medium of communication.

Besides this, let us mention that Amir Pnueli & coworkers [Pnu97] have introduced a more elaborated version of our STS formalism, for the purpose of investigating issues of compositionality in proofs involving liveness properties.

2 The essentials of the synchronous paradigm

There has been several attempts to characterize the essentials of the synchronous paradigm [BB91] [Halb93]. 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.

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[1].

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 [DC96] has been proposed with this objective. 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 programming, however): the STS formalism we describe next has this in its objectives.

3 Specification: Symbolic Transition Systems (STS) [Pnu97]

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 each variable $u \in \mathcal{V}$ a value $s[u]$ over its domain. We denote by Σ 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.

[1]For instance, most of the effort related to the semantics of ESTEREL has been directed toward solving this issue satisfactorily [Ber95].

Following [Pnu97][2], 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 previous and current states s^- and s, by referring to both past 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 that its value in s^-. If $\rho(s^-[V], s[V]) = \mathrm{T}$, we say that state s^- is a ρ-*predecessor* of state s.

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

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

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.

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

- $c, v, w, \ldots$ denote STS *variables.*
- 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:

$$\begin{array}{lll} \textbf{if}\ \ h_v & \textbf{then} & \xi_v = v \\ & \textbf{else} & \xi_v = \xi_v^- \end{array} \qquad (1)$$

State variable ξ_v is assumed to be *local,* i.e., is never shared among different STS, and thus state variables play no role for STS composition. Also, note that ξ_v is always present after the 1st occurrence of v.

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

Examples of Transition Relations :

- a selector :

$$\textbf{if} \quad b \quad \textbf{then} \quad z = u \quad \textbf{else} \quad z = v$$

  Note that the "**else**" part corresponds to the property "$[b = \mathrm{F}] \ \vee \ [b = \perp]$".

- decrementing a register :

$$\textbf{if} \quad h_z \quad \textbf{then} \quad v = \xi_z^- - 1 \quad \textbf{else} \quad v = \perp \quad ,$$

  where ξ_z is the state variable associated with z as in (1), and ξ_z^- denotes its previous value. The more intuitive interpretation of this statement is: $(v_n = z_{n-1} - 1)$, where index "n" denotes the instants at which both v and z are present (their clocks are specified to be equal). The specification of a register would simply be :

$$\textbf{if} \quad h_z \quad \textbf{then} \quad v = \xi_z^- \quad \textbf{else} \quad v = \perp \quad .$$

  Note that both statements imply the equality of clocks :

$$h_z \quad \equiv \quad h_v \ .$$

- testing for a property :

$$\textbf{if} \quad h_v \quad \textbf{then} \quad b = (v \leq 0) \quad \textbf{else} \quad b = \perp \quad .$$

  Note that a consequence of this definition is, again,

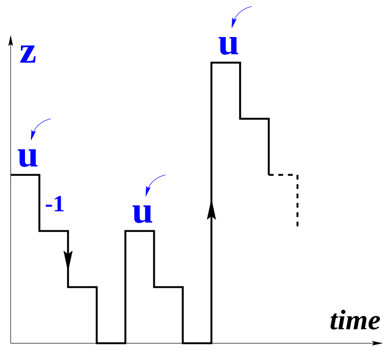
$$h_z \quad \equiv \quad h_v \ .$$

- a synchronization constraint :

$$(b = \mathrm{T}) \quad \equiv \quad (h_u = \mathrm{T}) \ \ ,$$

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

Putting things together yields the STS :



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

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}.

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 [Lam83a, Lam83b]. 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

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

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".

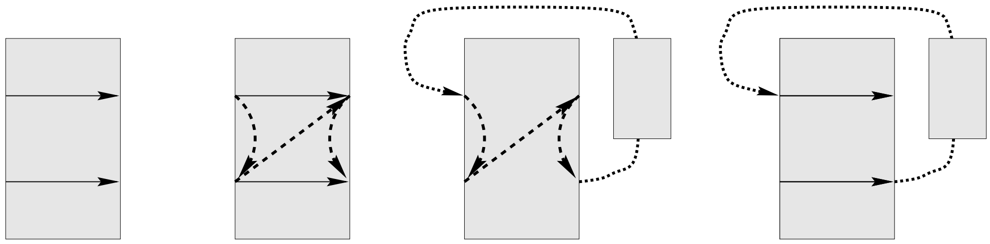
Stuttering invariance is not hardwired into our STS formalism, but a quick inspection of all the statements we have introduced in our example reveals that any composition of them is stuttering invariant. More generally, any STS not involving the always present clock (be it directly or indirectly) is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

4 Compositional reasoning on causality and scheduling specifications

4.1 What is the problem ?

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.

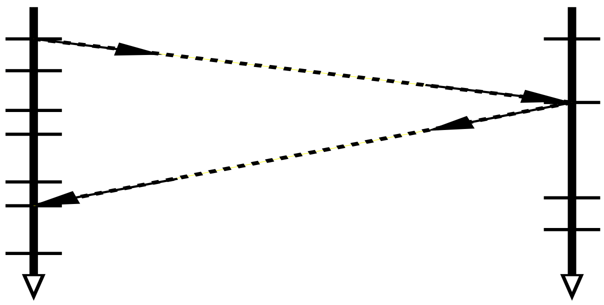
Separate compilation may be dangereous. This is illustrated in the following picture:



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: 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.

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.

Desynchronisation. This is illustrated in the following picture:



This figure depicts a communication scenario: two processors, modelled as sequential machines, exchanges 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.*

4.2 Scheduling specifications

Preorders and partial orders to model causality relations, schedulings, and communications. Causality relations have been investigated for several years in the past in the area of models of distributed systems and computations. The classical approach considers a classical automaton, in which concurrency is modelled via an "independence" equivalence relation among the labels of the transitions. Since independence is generally not a symmetric relation (actions of writing and reading are not symmetric), the theory of traces [AR88] has been extended to so-called "semi-commutations" [CL87], and this technique has been recently applied to the implementation of reactive automata on distributed architectures [CCGJ96]. Causality preorder relations have also been used in a different way in [BCHlG94], from which we borrow the essentials of the present technique. In addition to modelling causality relations, preorders can be used to specify scheduling requirements, they can also be used to model send/receive type of 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.*

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

STS with scheduling specifications. Now we consider STS $\Phi = \langle V, \Theta, \rho \rangle$ as before, but with the following additional feature: as part of

- $\Theta(V)$ (relation defining initial states), and
- $\rho(V^-, V)$ (transition relation),

we have *preorders* denoted by

$$x \preceq y \quad \text{for} \quad x, y \in V \ ,$$

specified via *(possibly cyclic)* directed graphs:

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

STS involving such type of preorder relation shall be called in the sequel STS *with scheduling specifications.* As preorders are just like any other relation, 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,

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

is denoted by

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

Note the following:

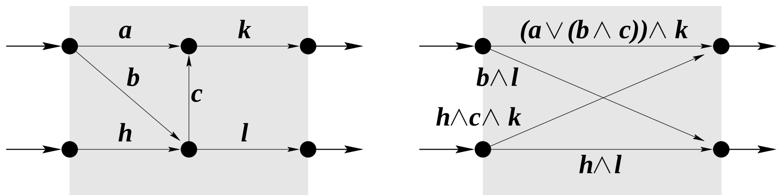
$$\underbrace{u \xrightarrow{\;\;\neg b\;\;} v}_{b=\text{false}} \;\; \neq \;\; \underbrace{u \xrightarrow{\;\;\overline{b}\;\;} v}_{b=\text{false} \,\vee\, b=\perp} \;\; !!!$$

In appendix A, it is shown that scheduling specifications have the following properties:

$$x \xrightarrow{\;\;b\;\;} y \bigwedge y \xrightarrow{\;\;c\;\;} z \;\Rightarrow\; x \xrightarrow{\;\;b \wedge c\;\;} z \tag{2}$$

$$x \xrightarrow{\;\;b\;\;} y \bigwedge x \xrightarrow{\;\;c\;\;} y \;\Rightarrow\; x \xrightarrow{\;\;b \vee c\;\;} y \tag{3}$$

Properties (2,3) can be used to compute input/output abstractions of scheduling specifications:



In this figure, the digram on the left depicts a scheduling specification involving local variables. These are hidden in the diagram on the right, using rules (2,3).

Inferring scheduling specifications from causality analysis. The idea supporting causality analysis of an STS specification is quite simple. On the one hand, a transition relation involving only the types "pure" and "boolean" can be solved by unification and thus made executable. On the other hand, a transition relation involving arbitrary types is abstracted as term rewriting, encoded via directed graphs. For instance, relation $y = 2uv^2$ (involving, say, real types) is abstracted as $(u, v) \longrightarrow y$, since y can be substituted by expression $2uv^2$. On the other hand, the relation $w + uv^2 \geq 0$ (again involving real types) is abstracted as the full directed graph with vertices (u, v, w), since no term rewriting is possible. A difficulty arises from the hybrid nature of general STS, in which boolean variables can be computed from the evaluation of non-boolean expressions (e.g., $b = (x \geq 0)$), and then used as a control variable.

We now provide a technique for inferring schedulings from causality analysis for STS specified as conjunctions of the particular set of primitive statements we have introduced so far. In table (4), 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 table (4), labels of schedulings are expressions involving variables in the domain $\{\bot, \mathrm{F}, \mathrm{T}\}$ ordered by $\{\bot < \mathrm{F} < \mathrm{T}\}$; with this in mind, expressions involving the symbols "$\wedge$" (min) and "$\vee$" (max) have a clear meaning.

$$
\begin{array}{lll}
\text{(R-1)} & & h_u \longrightarrow u \\[2em]
\text{(R-2)} & \begin{array}{l} \textbf{if}\ \ b\ \ \textbf{then}\ \ w = u \\ \qquad\ \ \textbf{else}\ \ \ w = v \end{array} \Rightarrow & \left\{ \begin{array}{l} b \xrightarrow{h_b\ \wedge\ (h_u \vee h_v)} h_w \\[1em] h_u \xrightarrow{b\ \wedge\ h_u} h_w \\ h_v \xrightarrow{\bar{b}\ \wedge\ h_v} h_w \\[1em] u \xrightarrow{b\ \wedge\ h_u} w \\ v \xrightarrow{\bar{b}\ \wedge\ h_v} w \end{array} \right. \\[2em]
\text{(R-3)} & u \xrightarrow{b} w \ \Rightarrow & b \longrightarrow h_w \\[2em]
\text{(R-4)} & \left. \begin{array}{r} w = f(u_1, \ldots, u_k) \\ h_w = h_{u_1} = \ldots = h_{u_k} \end{array} \right\} \Rightarrow & u_i \xrightarrow{h_w} w
\end{array}
\tag{4}
$$

Rules (R-1,...,R-4), as well as the theorem to follow, are formally justified in appendix A. In this appendix, a precise definition of "*deterministic*" is also given.

Theorem 1 (executable STS**)** *For* `P` *an* STS*,*

1. *Apply Rules (R-1,...,R-4) until fixpoint is reached: this yields an* STS *we call* **sched(**`P`**)**.
2. *A* sufficient *condition for* `P` *to have a unique deterministic run is:*
   (a) **sched(**`P`**)** *is provably circuitfree at each instant, meaning that* it is never true that

$$
\begin{array}{c}
x_1 \xrightarrow{b_1} x_2 \xrightarrow{b_2} x_1 \\
\textbf{and} \\
(b_1 \wedge b_2 = \mathrm{T})
\end{array}
$$

   unless $x_1 = x_2$ holds.

(b) **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 P *and the* $\exp_1$ *and* $\exp_2$ *are different expressions, then*

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

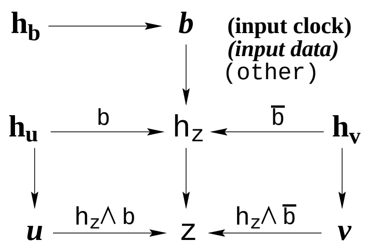
never holds in P*.*

Then P *is said to be* executable, *and* **sched**(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. See the appendix for a formal support of this claim.

A data-driven statement :

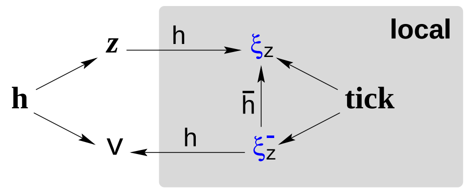
if b **then** $z = u$ **else** $z = v$



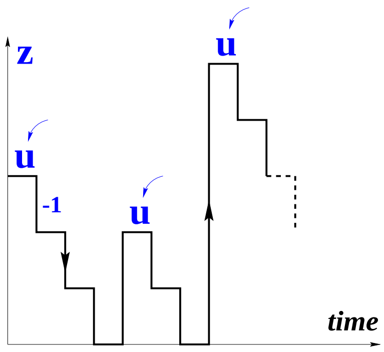
In the above example, input data are pairwise associated with corresponding input clocks : this STS reads its inputs on a purely data-driven mode, input patterns (u, v, b) are free to be present or absent, and, when they are present, their value is free also. We call it a “reactive” STS.

Decrementing a register :

if h_z **then** $v = \xi_z^- - 1$ **else** $v = \perp$

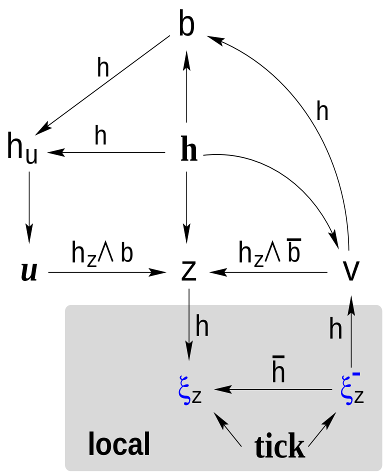


The full example:



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

Applying the rules (R-1,...,R-4) for inferring schedulings from causality, we get:



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

Note the change in control: { input clock, input data} habe 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".

Summary. What do we get at this stage?

1. STS composition is just the conjunction of constraints.
2. Since preorders are just relations, scheduling specifications do compose as well.
3. As causality analysis is based on an abstraction, the rules (R-1,...,R-4) for inferring scheduling from causality are bound to the *syntax* of the STS conjuncts.
4. Hence, in order to maximize the chance of effectively recognizing that an STS `P` is executable, `P` is generally rewritten in a different but semantically equivalent syntax (runs remain the same) while causality analysis is performed[3]. But this latter

[3] This is part of the job performed by the SIGNAL compiler's "clock calculus".

operation is global and not compositional: here we reach the limits of bruteforce universal compositionality.

5 Modular and distributed code generation

Two major issues need to be considered:

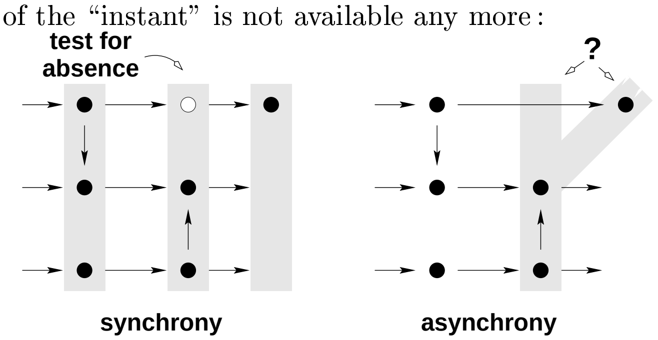
1. *Relaxing synchrony* is needed if distribution over possibly asynchronous media is desired without paying the price for maintaining the strong synchrony hypothesis via costly protocols.

2. *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.

5.1 Relaxing synchrony

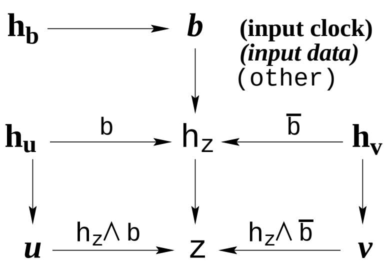
The major problem is that of testing for absence in an asynchronous environment! This is illustrated in the following picture in which the information of "which are the variables present in the considered instant" is lost when passing from left to right hand side, since explicit definition of the "instant" is not available any more:

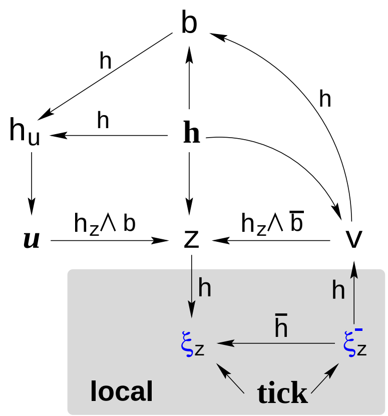


The questionmark indicates that it is generally not possible, in an asynchronous environment, to decide upon presence/absence of a signal relatively to another one. The major problem is that of "testing for absence" as being part of the control. While this is perfectly sound in a synchronous paradigm,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

The solution consists in restricting ourselves to so-called *endochronous* STS [LG97]. Endochronous STS are those for which the control depends only on 1/ the previous state, and 2/ the values possibily carried by environment signals, but not on the presence/absence status of these signals. An endochronous STS can work as a (synchronous) module working in a distributed execution using an asynchronous communication medium, provided that this medium satisfies the two requirements of 1/ not loosing messages, and 2/ not changing the order of messages.

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:

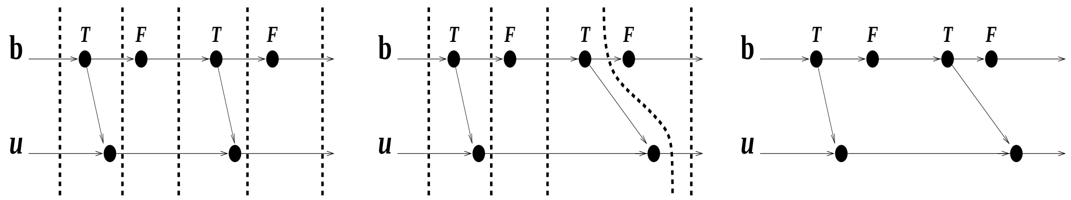




In the diagram on the left hand side, three different clocks are source nodes of the directed graph. This means that the primary decision in executing a reaction consists in deciding upon relative presence/absence of these clocks. In contrast, in the diagram on the right hand side, only one clock, the 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.

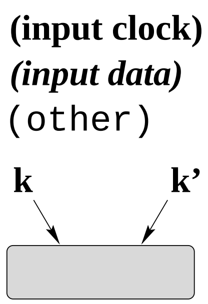
How endochrony allows us to desynchronize an STS is illustrated in an intuitive way on the following diagram, which depicts the scheduling specification associated with the pseudo-statement

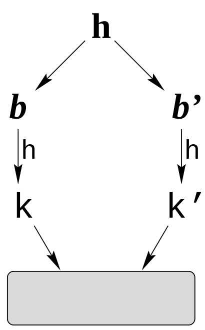
" **if** b **then** get_u" :



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 $b = \mathrm{T}$, and thus it is only needed to wait for b in order to know whether u is to be waited for also.

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.

5.2 Generating scheduling for separate modules

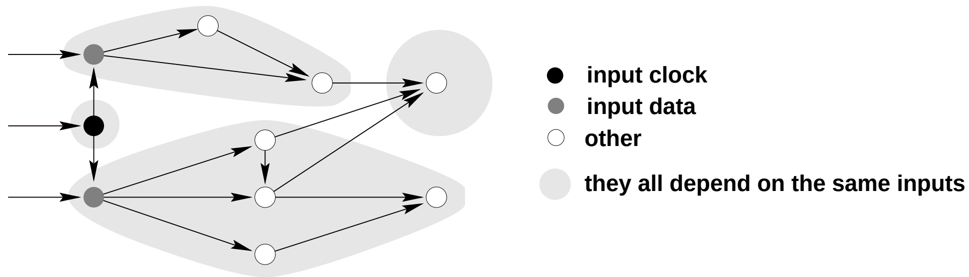
Relevant target architectures for embedded applications typically are 1/ purely sequential code (such as C-code), 2/ code using a threading or tasking mechanism provided by some kind of real-time OS (here the threading mechanism offers some degree of concurrency), or 3/ DSP-type multiprocessor architectures 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 4.1. 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.

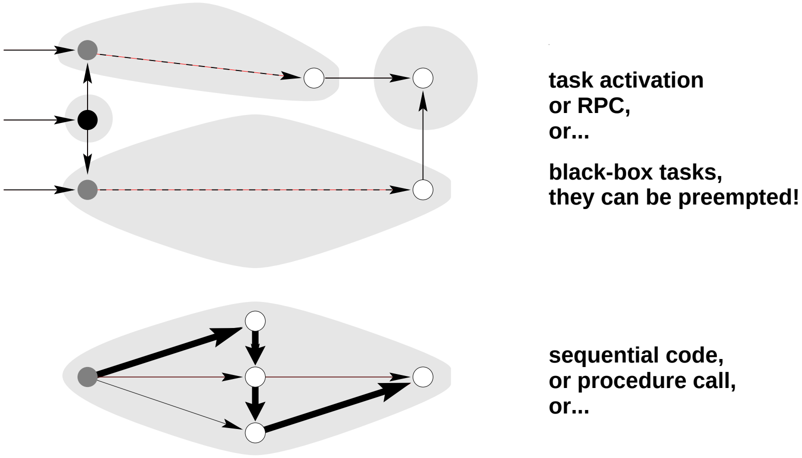
We shall however see that this does not need to be the case, however. Instead, a careful use of the scheduling specifications of an STS will allow us to decompose it into modules that can be stored as object code for further reuse, whatever the actual environment and implementation architecture will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

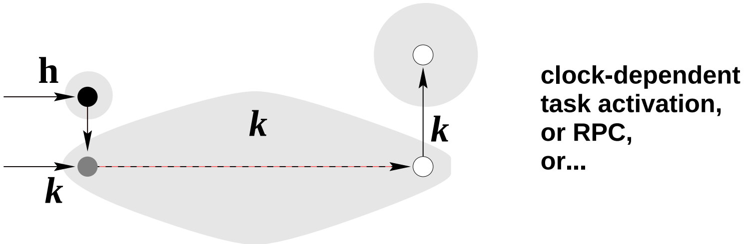
In the next picture, we show how the actual implementation will be prepared :



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 general case of multiple-clocked STS**.** The generalization is illustrated in the following picture :



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. In the above example, the considered STS is activated by some clock h, which in turns defines the activation clock k of the depicted task. The bottom line is :

1. Different tasks can have different activation clocks.

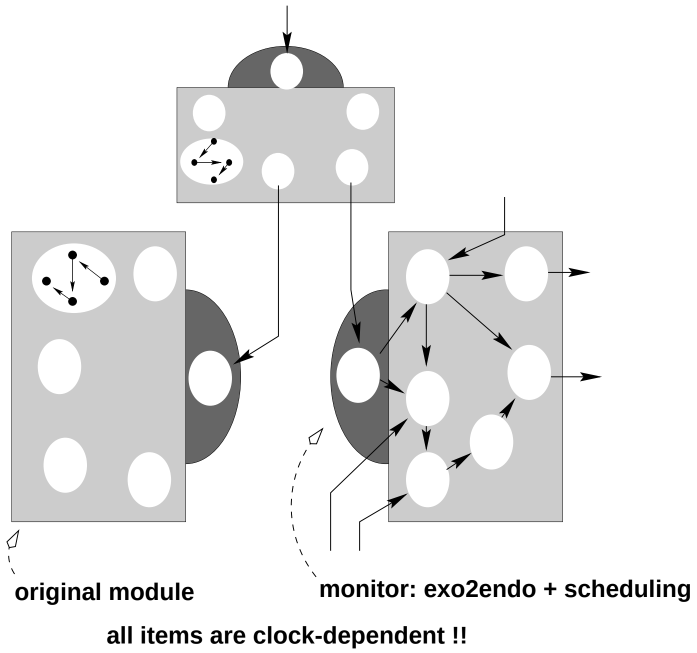
2. For each task, the internal scheduling generally depends on some internal clocks, i.e., on some predicates involving internal states of the task. Thus the internal scheduling is dynamic, but can be precomputed at compile time.

3. Also, the task abstraction has a clock-dependent scheduling specification. This is another source of dynamic scheduling, also computed at compile time: it is implemented in the form of a "software monitor" which opens/forbids the possibility of executing a given task at a given instant, depending on the current status of the STS clocks.

4. This software monitor can be combined with the interruption mechanism of the supporting real-time kernel intended to optimize response time of the execution, without impairing conformity of the object code to its specification.

5.3 The bottom line: modular and distributed code generation

The whole approach is summarized in the following diagram:



In this diagram, gray rectangles denote three modules $\mathtt{P}_1, \mathtt{P}_2, \mathtt{P}_3$ of the source STS specification, hence given by $\mathtt{P} = \mathtt{P}_1 \wedge \mathtt{P}_2 \wedge \mathtt{P}_3$. We assume here that this partitioning has been given by the designer, based on functional and architectural considerations. Note that the idempotence of STS composition ($\mathtt{Q} \wedge \mathtt{Q} = \mathtt{Q}$) allows us to duplicate source code at convenience while partitioning the specification.

Then, white bubbles inside the gray rectangles depict the structuration into tasks as discussed before.

Finally the black half-ellipses denote the monitors. Monitors are in charge of 1/ providing the additional control to make the considered module endochronous if asynchronous communication media are to be used, and 2/ specifying the scheduling of the abstract tasks.

In principle, communication media and real-time kernels do not need to be specified here, as they can be used freely provided they respect the send-receive abstract communication model and conform to the scheduling constraints set by the monitors.

b \ x	**?**	$\perp$	$\top$
?	**?**	**?**	**?**
$\perp$			
F			
T	**?**		

Table 1: *Definition of the dependency* $x \xrightarrow{b} y$. This table gives the result of this multivalued function for its output y. When nothing is written, this means that any value is accepted. If x is boolean, then $\top$ is to be refined as any of the two values $\{\mathrm{F}, \mathrm{T}\}$.

6 Conclusion

The above approach is supported by the DC_+ common format for synchronous languages [DC96]. The DC_+ format is one possible concrete implementation of our STS model, including scheduling specifications. It serves as a basis for the code generation suite in the Esprit SACRES project.

A Appendix: formal study of causality and scheduling specifications

A.1 Dependencies

In this appendix, as an abstraction of arbitrary domains, we consider the following domain of values $\mathcal{D}$ and its two orderings $\prec$ and $<$:

$$\mathbf{?} \quad \prec \quad \overbrace{\{\perp, \underbrace{\mathrm{F}, \mathrm{T}}_{\top}\}}^{\text{¿}} \tag{5}$$

$$\perp < \mathrm{F} < \mathrm{T} \tag{6}$$

In these formulæ, symbols **?** (resp. **¿**) indicate "undefined" (resp. "defined"). "Undefined" should not be confused with "absent" ($\perp$): "absent" is a perfectly defined status, while "undefined" is intended to model that a variable has not been produced yet at the current instant. Finally, non-boolean types are abstracted as the single distinguished element $\top$. Thus, for booleans, the pair $\{\mathrm{F}, \mathrm{T}\}$ can be seen as a refinement of the symbol $\top$, this is shown by the underbrackets. And $\{\perp, \top\}$ is a refinement of **¿**. this is shown by the overbrackets.

Definition of dependencies: Relation $x \xrightarrow{b} y$ is defined in table 1, where it is specified in the form of a multivalued function. Its main feature is that it forbids, when $b = \mathrm{T}$, that y gets defined while x is not.

Properties of dependencies. The following properties hold:

$$x \xrightarrow{\quad b \quad} y \bigwedge y \xrightarrow{\quad c \quad} z \;\Rightarrow\; x \xrightarrow{\quad b \wedge c \quad} z \tag{7}$$

$$x \xrightarrow{\quad b \quad} y \bigwedge x \xrightarrow{\quad c \quad} y \;\Rightarrow\; x \xrightarrow{\quad b \vee c \quad} y \tag{8}$$

In these equations, $b \wedge c$ and $b \vee c$ are respectively defined as the infimum (resp. supremum) w.r.t. relation "$<$" defined in (6) when both values belong to the subdomain $\{\perp, \mathtt{T}, \mathtt{F}\}$. Otherwise, they are both equal by definition to **?**.

A.2 Circuitfree schedulings

We are given a set $\mathtt{X}_1, \ldots, \mathtt{X}_n$ of variables with some of them being boolean. The boolean variables will also be denoted by $b_1, b_2, \ldots$, and will be used as labels in dependencies. Then we are given 1/ a set of constraints on the boolean variables of the form $C(b_1, \ldots, b_k)$ as restricted to the subdomain $\{\perp, \mathtt{T}, \mathtt{F}\}$ of defined and noncontradictory values, and 2/ a set of dependencies defined on the set $\mathtt{X}_1, \ldots, \mathtt{X}_n$. We call the whole a *scheduling* of $\mathtt{X}_1, \ldots, \mathtt{X}_n$.

The conjunction of the corresponding tables 1, together with boolean constraints of the form $C(b_1, \ldots, b_k)$, define constraints on the product domain $\mathcal{D}^n$, denote by $\mathcal{S}$ the set of points satisfying these constraints. Points in $\mathcal{D}^n$ are written $x, y, \ldots$ and their coordinates are denoted by $x = (x_1, \ldots, x_n)$. In $\mathcal{S}$, some values x_i are set to **?**, and the other ones are free, or alternatively constrained by boolean equations for boolean variables.

Lemma 1 *Dependency* $\mathtt{X} \xrightarrow{\quad b \quad} \mathtt{Y}$ *is part of the considered scheduling iff, for variables* $(\mathtt{X}, \mathtt{Y}, b)$*, none of the two triples of values* $(\mathbf{?}, \perp, \mathtt{T}), (\mathbf{?}, \top, \mathtt{T})$ *belongs to* $\mathcal{S}$.

Proof: The "only if" part follows by inspection of table 1. As for the "if" part, we note that boolean constraints of the form $C(b_1, \ldots, b_k)$ do not involve the **?** value, so that the condition that triples $(\mathbf{?}, \{\perp, \top\}, \mathtt{T})$ do not belong to $\mathcal{S}$ can only result from the conjunction of dependencies, and we see that $\mathtt{X} \xrightarrow{\quad b \quad} \mathtt{Y}$ is implied by this conjunction, and can thus be added without changing $\mathcal{S}$. $\diamond$

Two points of $\mathcal{S}$ are said to be *neighbours* if they differ exactly by one coordinate. We call a *path* in $\mathcal{S}$ any finite sequence $x(1), x(2), \ldots, x(K)$ of neighbouring points.

For x and y two neighbouring points of $\mathcal{S}$, we write $x \prec y$ (resp. $x \preceq y$) if their discriminating coordinates, say the i-th one, satisfy the relation $x_i \prec y_i$ (resp. $x_i \preceq y_i$) defined in (5). A path $x(1), x(2), \ldots, x(K)$ such that $x(k) \prec x(k+1)$ (resp. $x(k) \preceq x(k+1)$) is called *increasing* (resp. *nondecreasing*).

A scheduling $\mathcal{S}$ is called *circuitfree* if *it is never true in* $\mathcal{S}$ *that*

$$\begin{array}{c} \mathtt{X}_{i_1} \xrightarrow{\quad b_1 \quad} \mathtt{X}_{i_2} \xrightarrow{\quad b_2 \quad} \mathtt{X}_{i_3} \ldots \mathtt{X}_{i_p} \xrightarrow{\quad b_p \quad} \mathtt{X}_{i_1} \\ \textbf{and} \\ (b_1 \wedge \ldots \wedge b_p = \mathtt{T}) \end{array} \tag{9}$$

Lemma 2 (circuitfree schedulings) *A scheduling is circuitfree iff, for every* $x \in \mathcal{S}$ *satisfying* $\forall i : x_i \neq \mathbf{?}$*, there is an increasing path linking* $(\mathbf{?}, \ldots, \mathbf{?})$ *to* x.

Proof: We first prove the "if" part. Assume (9) is violated for some circuit $\mathtt{X}_{i_1} \xrightarrow{b_1} \mathtt{X}_{i_2} \xrightarrow{b_2} \mathtt{X}_{i_3} \ldots \mathtt{X}_{i_p} \xrightarrow{b_p} \mathtt{X}_{i_1}$, i.e., $(b_1 \wedge \ldots \wedge b_p = \mathrm{T})$ is possible for this circuit in $\mathcal{S}$. Restrict $\mathcal{S}$ to those points for which, in addition, $(b_1 \wedge \ldots \wedge b_p = \mathrm{T})$ holds, call $\mathcal{S}_{(b_1 \wedge \ldots \wedge b_p = \mathrm{T})}$ this subset. Then, on $\mathcal{S}_{(b_1 \wedge \ldots \wedge b_p = \mathrm{T})}$, due to condition $\mathtt{X}_{i_1} \xrightarrow{b_1} \mathtt{X}_{i_2} \xrightarrow{b_2} \mathtt{X}_{i_3} \ldots \mathtt{X}_{i_p} \xrightarrow{b_p} \mathtt{X}_{i_1}$, one has:

$$\mathtt{X}_{i_1} \preceq \mathtt{X}_{i_2} \preceq \ldots \preceq \mathtt{X}_{i_p} \preceq \mathtt{X}_{i_1} ,$$

and thus the $\mathtt{X}_{i_j}$'s are, either all $= \mathbf{?}$, or alternatively all defined (equal to one of the values $\perp, \mathrm{T}, \mathrm{F}$). Thus, when $(b_1 \wedge \ldots \wedge b_p = \mathrm{T})$ holds, there is no increasing path leading to any defined point belonging to $\mathcal{S}_{(b_1 \wedge \ldots \wedge b_p = \mathrm{T})}$. This proves the "if" part.

We now prove the "only if" part. Assume $\exists x \in \mathcal{S}$ satisfying $\forall i : x_i \neq \mathbf{?}$, such that there is no increasing path linking $(\mathbf{?}, \ldots, \mathbf{?})$ to x. Denote by $b_1, \ldots, b_p$ the boolean variables such that $(b_1 \wedge \ldots \wedge b_p = \mathrm{T})$ holds at point x. And denote by X the maximal (for set theoretic union), connected subset of $\mathcal{S}$, such that 1/ $(b_1 \wedge \ldots \wedge b_p = \mathrm{T})$, 2/ every $y \in X$ satisfies: $\forall i, y_i \neq \mathbf{?}$ and there is no increasing path linking $(\mathbf{?}, \ldots, \mathbf{?})$ to y, and finally 3/ $X \ni x$. Clearly, X is not empty. Let ∂X denote the boundary of X, i.e., the set of points not belonging to X, that are neighbour to X. Since ∂X does not satisfy the scheduling, there are some variables, say $\mathtt{X}_{j_1}, \mathtt{X}_{j_2}, \ldots, \mathtt{X}_{j_q}$ such that, on $\mathcal{S}_{(b_1 \wedge \ldots \wedge b_p = \mathrm{T})}$, one has:

$$\begin{array}{rcl} \text{on } \mathcal{S}_{(b_1 \wedge \ldots \wedge b_p = \mathrm{T})} \setminus X & : & \mathtt{X}_{j_1} = \mathtt{X}_{j_2} = \ldots = \mathtt{X}_{j_q} = \mathbf{?} \ , \\ \text{on } X & : & \mathtt{X}_{j_1}, \mathtt{X}_{j_2}, \ldots, \mathtt{X}_{j_q} \text{ are defined.} \end{array} \tag{10}$$

Using lemma 1 we conclude that

$$\begin{array}{rcl} & & \mathtt{X}_{j_1} \xrightarrow{\quad b \quad} \mathtt{X}_{j_2} \xrightarrow{\quad b \quad} \mathtt{X}_{j_3} \ldots \mathtt{X}_{j_q} \xrightarrow{\quad b \quad} \mathtt{X}_{j_1} \text{ holds, where} \\ b & = & (b_1 \wedge \ldots \wedge b_p) \text{ , and } b = \mathrm{T} \text{ is possible in } \mathcal{S}. \end{array} \tag{11}$$

This finishes the proof. ◇

The intuitive interpretation of this lemma is that, for an STS with a circuitfree scheduling, it is possible to sequentially compute all variables from the inputs with no deadlock. Each increasing path mentioned in Lemma 2 corresponds to one possible sequential execution.

A.3 Deriving dependencies as causality constraints

In this section, we infer dependencies as causality constraints from STS statements. The principles we follow for our abstraction mechanism are given next:

(P-1) For x not a boolean variable, we abstract its domain $\mathcal{D}_x$ as the singleton $\{\top\}$, and then extend it to the additional values $\{\mathbf{?}, \perp\}$.

(P-2) Within equations of the form "$y = exp$" or **if** b **then** $y = exp_1$ **else** $y = exp_2$ we shall further abstract y by mapping the set $\{\perp, \mathrm{F}, \mathrm{T}\}$ to the single value ¿. This is where asymmetry between left– and right hand side of equations occurs in our abstraction technique.

(P-3) Since we are interested in causality constraints, we only need to keep track of configurations for which y cannot be defined. Hence, when $y = ¿$ results from applying **(P-2)**, we weaken it to "y unconstrained".

We now proceed on deriving the scheduling associated to each primitive statement.

Lemma 3 *The following holds :*

$$x \xrightarrow{\ b\ } y \quad \Rightarrow \quad b \longrightarrow h_y$$

Proof : by inspection of table 1.

Lemma 4 *The following holds :*

$$h_x \longrightarrow x$$

Proof : by inspection of table 1 and of the following table :

h_x	?	$\bot$	$\top$	
x	?	$\bot$	?	$\top$

,

abstacted as :

h_x	?	$\bot$	$\top$
x	?	¿	

, ,

weakened as :

h_x	?	$\bot$	$\top$
x	?		

which turns out to be equivalent to $h_x \longrightarrow x$.

Lemma 5 *The following holds :*

$$y = f(u,v) \quad \Rightarrow \quad (u,v) \xrightarrow{\ h_y\ } y$$

Proof : by inspection of table 1 and of the following table for the abstracted relation derived from $y = f(u,v)$ using principles **(P-1,P-2,P-3)** (# denotes a prohibited value) :

v \ u	?	$\bot$	$\top$
?	?	#	?
$\bot$	#	$\bot$	#
$\top$	?	#	$\top$

, weakened as :

v \ u	?	$\bot$	$\top$
?	?		?
$\bot$			
$\top$	?		

,

which is equivalent to the formulæ of lemma 5.

Lemma 6 *The following holds :*

$$[\ \textbf{if}\ b\ \textbf{then}\ x = u] \wedge [\ \textbf{if}\ b\ \textbf{then}\ h_x = h_u] \quad \Rightarrow \quad \begin{cases} u \xrightarrow{\ b\ \wedge\ h_u\ } x \\ b \xrightarrow{\ h_b\ \wedge\ h_u\ } h_x \\ h_u \xrightarrow{\ b\ \wedge\ h_u\ } h_x \end{cases}$$

Proof: by inspection of table 1 and of the following two tables yielding respectively x and h_x for [**if** b **then** $x = u$] $\wedge$ [**if** b **then** $h_x = h_u$]:

b \ u	?	$\perp$	$\top$
?	?	$\perp$	?
$\perp$	$\perp$	$\perp$	$\perp$
T	?	$\perp$	$\top$
F	$\perp$	$\perp$	$\perp$

b \ h_u	?	$\perp$	T
?	?	$\perp$	?
$\perp$	$\perp$	$\perp$	$\perp$
T	?	$\perp$	$\top$
F	$\perp$	$\perp$	$\perp$

,

which is equivalent to the formulæ of lemma 6 after weakening by removing the $\perp$'s. Note the asymmetry between x and u, while statements $x = u$ and $u = x$ are clearly identical. This asymmetry is due to principle **(P-2)** for STS abstraction.

A.4 Correct programs

In this section, we prove the following refinement of Theorem 1:

Theorem 2 (correct programs) *Let* `P` *be an* STS *satisfying the following conditions:*

1. *Each statement in* `P` *has its associated set of derived dependencies as derived from lemmas 3, 4, 5, 6 as part of* `P`.
2. *Resulting scheduling is circuitfree.*
3. *The is provably no multiple definition of a variable, meaning that, whenever*

$$\begin{array}{ll} & \textbf{if } b_1 \textbf{ then } x = u_1 \\ \wedge & \textbf{if } b_2 \textbf{ then } x = u_2 \\ \wedge & \dots \\ \wedge & \textbf{if } b_p \textbf{ then } x = u_p \end{array}$$

   is part of `P`, *then*

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

   has to be provably impossible.

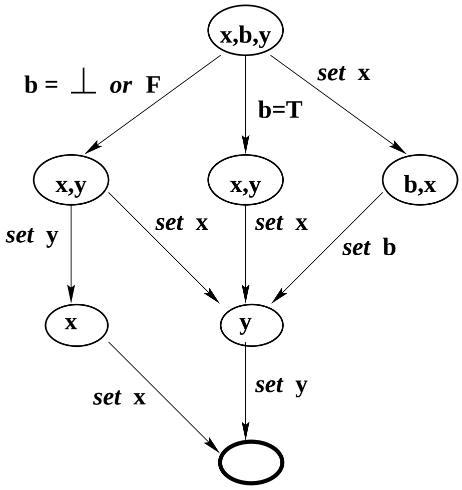
Then:

1. *As far as control is concerned, the inputs of* `P` *are the source nodes of the dependency graph.*
2. *Input values are those variables which never occur on the left hand side of statements such as $x =$expression.*
3. *For each given input control history of* `P` *and compatible input value history, there is exactly one run of* `P`, *i.e.,* `P` *is deterministic.*

NOTA: Clearly, theorem 2 provides us with a sufficient condition, this condition is not necessary. Furthermore, the rules for inferring dependencies as causality constraints is bound to the syntax, not to the semantics of the program. In particular, from statement " **if** b **then** $x = u$", we choose to infer dependency $u \xrightarrow{b \wedge h_u} x$ but not the symmetric one in which x and u are exchanged. This means that, while `P` may not satisfy the assumptions of theorem 2 for a given synctactic form of `P`, it may satisfy them after proper rewriting into a syactactically different, but semantically equivalent form. Here, semantically equivalent means that the values of variables that satisfy each program (considered as sets of constraints) are identical.

Proof: It is organized into several steps.

1. With the formula $x \xrightarrow{b} y$ we associate the following automaton:



Transitions are labelled with actions. Label "*set* x" indicates that variable x is set to an arbitrary value of its (extended) domain $\mathcal{D}_x \cup \{\perp\}$. States are labelled with those variables that are **?**, i.e., have not been set. This automaton is the most permissive one with the following properties:

(a) states are valued with configurations of the triple (x, b, y) that are compatible with the scheduling constraint $x \xrightarrow{b} y$.

(b) Variables are set sequentially.

(c) All variables are eventually set.

Thus each path of this automaton specifies an evaluation scheme for the triple (x, b, y) which is compatible with the considered scheduling specification. And all of them can be specified in this way. We call this automaton the *execution automaton* associated with scheduling specification $x \xrightarrow{b} y$.

2. To each primitive statement we associate the conjunction of its causality constraints and possible constraints involving clocks and boolean variables, and we take the product of associated execution automata. The paths of the resulting automaton

specify all correct schedulings to evaluate the involved variables. We call the resulting product automaton the *execution automaton* associated with the considered primitive.

3. Then we take the product of the execution automata associated with each statement. By Lemma 2 we know that, for each tuple of variables which satisfies the specification, there is a path of the product automaton which sequentially sets all variables of the considered tuple.

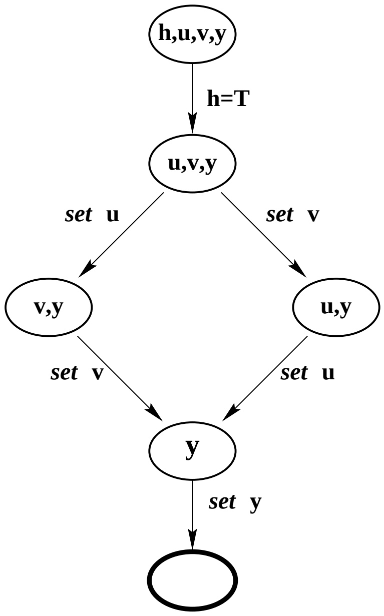
4. Finally, we refine the transition labels of the form "*set* x" etc., by assigning to x etc the value specified by the program. Thanks to condition 3 of theorem 2, actions of the form "*set* x" etc., are refined into single writings. This finishes the proof of the theorem. $\diamond$

We illustrate this technique on the following simple STS:

$$y = f(u,v) \quad \bigwedge \quad h_u \equiv h_v \equiv h_y \equiv_{\text{def}} h \ .$$

The causality constraint and associated execution automaton are:

$$\begin{array}{ll} & h \longrightarrow (u,v,y) \\ \bigwedge & (u,v) \xrightarrow{h} y \\ \bigwedge & h_u \equiv h_v \equiv h_y \equiv_{\text{def}} h \end{array}$$



The refined execution automaton is obtained by replacing *set* u and *set* v by *read* u and *read* v, and *set* y by $y := f(u,v)$.

References

[AR88] I.J. Aabelsberg, and G. Rozenberg, *Theory of traces,* Theoretical Computer Science, 60, 1–82, 1988.

[BB91] 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.

[BlGJ91] 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.

[BCHlG94] A. BENVENISTE, P. CASPI, N. HALBWACHS, AND P. LE GUERNIC, Dataflow synchronous languages. In *A Decade of Concurrency, reflexions and perspectives, REX School/Symposium*, pages 1–45, LNCS Vol. 803, Springer Verlag, 1994.

[Ber95] GÉRARD BERRY, *The Constructive Semantics of Esterel,* Draft book, `http://www.inria.fr/meije/esterel`, December 1995.

[CCGJ96] B. CAILLAUD, P. CASPI, A. GIRAUD, AND C. JARD, *Un modèle pour la répartition d'automates réactifs sur réseau asynchrone de processeurs,* Colloque AFCET sur la modélisation des systèmes réactifs, Brest, March 1996.

[CL87] M. CLERBOUT, AND M. LATTEUX, *Semi-commutations,* Information and Computation, 73, 59–74, 1987.

[DC96] SACRES CONSORTIUM, The Declarative Code DC+, Version 1.2, May 1996; Esprit project EP 20897: Sacres.

[Halb93] N. HALBWACHS, *Synchronous programming of reactive systems,*. Kluwer Academic Pub., 1993.

[KP96] Y. KESTEN AND A. PNUELI, An αSTS-based common semantics for SIGNAL and STATECHARTs, March 1996. Sacres Manuscript.

[Lam83a] L. LAMPORT, Specifying concurrent program modules, *ACM Trans. on Prog. Lang. and Sys.,* 5(2):190-222, 1983.

[Lam83b] L. LAMPORT, What good is temporal logic/,? In *Proc. IFIP 9th World Congress,* R.E.A. Mason (Ed.), North Holland, 657-668, 1983.

[LG97] P. LE GUERNIC, private communication, 1997.

[MP92] Z. MANNA AND A. PNUELI, *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1992.

[MP95] Z. MANNA AND A. PNUELI, *The Temporal Logic of Reactive and Concurrent Systems: Safety.* Springer-Verlag, New York, 1995.

[Pnu97] A. PNUELI & CO-WORKERS, these proceedings.