Information and Computation **163**, 125–171 (2000)
doi:10.1006/inco.2000.2898, available online at http://www.idealibrary.com on IDEAL®

Compositionality in Dataflow Synchronous Languages: Specification and Distributed Code Generation[1, 2, 3]

Albert Benveniste, Benoît Caillaud, and Paul Le Guernic

Irisa/inria, Campus de Beaulieu, 35042 Rennes cedex, France
E-mail: albert.benveniste@arisa.fr, benoit.caillaud@irisa.fr, paul.leguernic@irisa.fr

Modularity is advocated as a solution for the design of large systems; the mathematical translation of this concept is often that of *compositionality*. This paper is devoted to the issues of compositionality for modular code generation, in dataflow synchronous languages. As careless reuse of object code in new or evolving system designs 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. Using this notion, we study separate compilation for synchronous programs. An entire section is devoted to the formal study of causality and scheduling specifications. Then we discuss the issue of distributed implementation using an asynchronous medium of communication. Our main results are that it is possible to characterize those synchronous programs which can be distributed on an asynchronous architecture without loosening semantic properties. Two new notions of *endochrony* and *isochrony* are introduced for this purpose. As a result, we derive a theory for synthesizing additional schedulers and protocols needed to guarantee the correctness of distributed code generation. Corresponding algorithms are implemented in the framework of the DC$_+$ common format for synchronous languages, and the V4 release of the Signal language.
© 2000 Academic Press

Key Words: synchronous languages; modularity; distributed code generation; separate compilation; desynchronization.

[1] This paper is a significantly revised version of a preliminary report which appeared under the same title in the Proceedings of the 1997 Malente Workshop on Compositionality, organized by W. P. de Roever and H. Langmaack; these proceedings will be published in Lecture Notes in Computer Science (Springer-Verlag).

[2] This work is or has been supported in part by the following projects: Eureka-SYNCHRON, Esprit R&D-SACRES (Esprit Project EP 20897), and Esprit LTR-SYRF (Esprit Project EP 22703).

[3] In addition to the listed authors, the following people have indirectly, but strongly, contributed to this work: the STS formalism has been shamelessly borrowed from Amir Pnueli; the background on labelled partial orders is mostly acknowledged to Paul Caspi.

0890-5401/00 $35.00

CONTENTS

1. RATIONALE

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

The mathematical translation of the concept of modularity is often that of *compositionality.* Paying attention to the composition of specifications (Manna and Pnueli, 1992) is central to any system model involving concurrency or parallelism. More recently, significant effort has been devoted toward the introduction of compositionality in verification, which aims at deriving proofs of large programs from partial proofs involving (abstractions of) components (Manna and Pnueli, 1995). See also the volume (de Roever *et al.*, 1998) in which a number of papers are devoted to this topic.

Compilation and code generation have been given less attention from this very same point of view. 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 which uses these modules only on the basis of their interface description, while preserving in any case the correctness of the design. This paper is devoted to the issues of compositionality of dataflow synchronous languages, aimed at modular code generation.

Dataflow synchrony is rather a paradigm than a set of concrete languages or visual formalisms (Benveniste and Berry, 1991), and hence it is desirable to abstract from such and such particular language. Thus we have chosen to work with synchronous transition systems (STS), a lightweight formalism proposed by Amir Pnueli, general enough to capture the essence of the synchronous paradigm. This is the topic of Section 2. Using this formalism, we study in Section 2 the composition of specifications.

Most of our effort is then devoted to issues of compositionality that are critical to code generation. Section 3 contains an informal discussion of this problem. It is known that careless storing of object code for further reuse in systems design fails to work. Hence we first concentrate on the additional features that are required to abstract programs for the purpose of code generation and reuse: we show that a central notion is that of scheduling specification as resulting from a causality analysis of the given program. 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.

In Section 4 we formally study desynchronization. We first formalize what we mean by desynchronization. Our theory requires that the communication medium or operating system: (1) shall not lose messages, and (2) shall preserve the total ordering of messages, *for each flow individually* (but, of course, not globally). These assumptions are typically satisfied by services offered by reliable communication media or operating system. Our main result is that *it is possible to check, directly on the original synchronous specification, whether semantic properties will or will not be preserved after desynchronization.* The two fundamental notions are *endochrony*, which guarantees that, for a single STS, desynchronization is a "revertible" transformation, and *isochrony*, which guarantees that, for a pair of STS, desynchronizing communications is also a "revertible" transformation. In some sense formalized in Section 4, semantics is preserved by desynchronization when these conditions are satisfied.

Section 5 is devoted to a formal study of causality. In many respect, this formal study is important. First, it is instrumental in getting executable, deterministic code from a given STS specification. Then, it is a cornerstone of proper abstractions for separate compilation and reuse. We pay strong attention to this study, using a technique not unlike the one used for analyzing causality in ESTEREL (Berry, 1995). Our analysis encompasses the case of arbitrary data types, and suitable abstractions are used for this purpose.

In the Conclusion (Section 6) we discuss how our views on compositionality are modified by this study. We sketch the resulting system design methodology, and we briefly mention the implementation resulting from this theory, mostly developed in the framework of the Esprit-SACRES project.

2. SPECIFICATION

This section discusses compositionality aspects of specifications, first informally, and then formally.

2.1. The Essentials of the Synchronous Paradigm

There have been several attempts to characterize the essentials of the synchronous paradigm (Berry, 1989; Benveniste and Berry, 1991; Halbwachs, 1993). With some experience, we feel that the following features are indeed essential and sufficient for characterizing this paradigm:

1. Programs progress via an infinite sequence of *reactions*, informally written

$$P = R^{\omega},$$

where R denotes the set of legal reactions.[4]

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

3. Communication is performed via instantaneous broadcast. In other words, when it is defined, parallel composition is always given by the conjunction of associated reactions:

$$P_1 \parallel P_2 = (R_1 \wedge R_2)^{\omega}.$$

The above formula is a perfect definition of parallel composition when the intention is specifying. In contrast, if producing executable code was the intention, then this definition has to be compatible with an operational semantics. This very much complicates the "when it is defined" prerequisite.[5]

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

[4] In fact, "reaction" is a slightly restrictive term, as we shall see in the following that "reacting to the environment" is not the only possible kind of interaction a synchronous system may have with its environment.

[5] For instance, most of the effort related to the semantics of ESTEREL has been directed toward solving this issue satisfactorily (Berry, 1995).

Clearly, this calls for the simplest possible formalism comprising the above features, and on which fundamental questions should be investigated. This is one of the objectives of the STS formalism described next.

2.2. Synchronous Transition Systems

Synchronous Transition Systems. We assume a vocabulary $\mathscr{V}$ 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 that both types are extended with the distinguished element $\perp$).

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

We define a *synchronous transition system* (STS) to be a triple

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

consisting of the following components:

- V is a finite set of typed *variables*.
- Θ is an assertion characterizing the set of *initial states*; $\{s \mid s \models \Theta\}$.
- $\rho \subseteq S \times S$ is the *transition relation* relating past and current states denoted by s^- and s, respectively.[6] 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 $(s^-, s) \models \rho$, we say that state s^- is a *ρ-predecessor* of state s.

Runs. A run σ: $s_0, s_1, s_2, \ldots$ is a sequence of states such that

$$s_0 \models \Theta \wedge \forall_i > 0, \qquad (s_{i-1}, s_i) \models \rho. \tag{1}$$

Composition. The *composition* of two STS $\Phi = \Phi_1 \parallel \Phi_2$ is defined as

$$V = V_1 \cup V_2$$
$$\Theta = \Theta_1 \wedge \Theta_2$$
$$\rho = \rho_1 \wedge \rho_2.$$

The composition is thus the pairwise conjunction (denoted by $\wedge$) of initial and transition relations. Composition is thus commutative and associative. Note that, in STS composition, interaction occurs through common variables only.

[6] Usually, states and *primed* states 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.

Notations for STS. For the convenience of specification, STS have a set of *declared* variables, written $V_{\mathbf{d}}$, implicitly augmented with associated *auxiliary* variables: the whole constitutes the set V of variables. We shall use the following generic notation in the following:

- $b, c, v, w, \ldots$ denote STS declared variables, and b, c are used to refer to variables of Boolean type.
- For v a declared variable, $h_v \in \{\mathrm{T}, \bot\}$ denotes its *clock*:

$$[h_v \neq \bot] \Leftrightarrow [v \neq \bot].$$

- For v a declared variable ξ_v denotes its associated *state variable*, defined by

$$\begin{aligned} \textbf{if} \quad h_v \quad \textbf{then} \quad & \xi_v = v \\ \textbf{else} \quad & \xi_v = \xi_v^-. \end{aligned} \tag{2}$$

Values can be given to $s_0[\xi_v]$ as part of the initial condition. Then, ξ_v is always present after the first occurrence of v. Note that $\xi_{\xi_v} = \xi_v$; thus only state variables of declared variables have to be considered.

Stuttering. As modularity is desirable, an STS should be permitted to do nothing while its environment is possibly working. This feature has been yet identified in the litterature and is known as *stuttering* invariance or robustness (Lamport, 1983a, b). Stuttering invariance of an STS Φ is defined as follows: if

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

is a run of Φ, so is

$$\sigma': s_0, \underbrace{\bot_{s_0}, \ldots, \bot_{s_0}}_{0 \leqslant \#\{\bot_{s_0}\} < \infty}, s_1, \bot_{s_1}, \ldots, \bot_{s_1}, s_2, \bot_{s_2}, \ldots, \bot_{s_2}, \ldots, \tag{3}$$

where, for every state s, the symbol $\bot_s$ denotes the *silent state* associated with s, defined by

$$\forall v \in V_{\mathbf{d}} : \begin{cases} \bot_s[v] = \bot \\ \bot_s[\xi_v] = s[\xi_v]. \end{cases}$$

This means that state variables are kept unchanged, whenever their associated declared variables are absent. Note that stuttering invariance allows for runs possessing only a finite number of present states.

We require in the following that all STS that we consider be stuttering invariant. They should indeed satisfy

$$[(s^-, s) \models \rho] \Rightarrow [(s^-, \bot_{s^-}) \models \rho] \wedge [(\bot_{s^-}, s) \models \rho]. \tag{4}$$

By convention, we shall simply write $\bot$, when mentioning a particular state s is not required.

Examples of Transition Relations.

- A selector:

$$\textbf{if} \quad b \quad \textbf{then} \quad z = u \quad \textbf{else} \quad z = v. \tag{5}$$

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

- A register:

$$\textbf{if} \quad h_z \quad \textbf{then} \quad v = \xi_z^- \quad \textbf{else} \quad v = \perp, \tag{6}$$

where ξ_z is the state variable associated with z as in (2), and ξ_z^- denotes its past value. The more intuitive interpretation of this statement is $v_n = z_{n-1}$, where index "n" denotes the instants at which both v and z are present (their clocks are specified to be equal). Decrementing a register would simply be specified by

$$\textbf{if} \quad h_z \quad \textbf{then} \quad v = \xi_z^- - 1 \quad \textbf{else} \quad v = \perp, \tag{7}$$

where z is of integer type. Note that both statements (6) and (7) imply the equality of clocks:

$$h_z = h_v.$$

- Testing for a property:

$$\textbf{if} \quad h_v \quad \textbf{then} \quad b = (v \leqslant 0) \quad \textbf{else} \quad b = \perp. \tag{8}$$

Note that a consequence of this definition is, again,

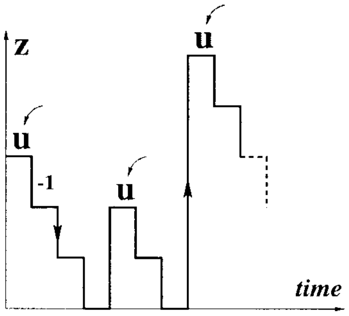
$$h_v = h_b.$$

- A synchronization constraint:

$$(b = \mathrm{T}) = (h_u = \mathrm{T}), \tag{9}$$

meaning that the clock of u is the set of instants where the Boolean variable b is true.

Putting (5), (7), (8), and (9) together yields the STS



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

A run of this STS for the variable z is depicted on the figure above. Each time u is received, z is set to 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, a fresh u can be read, and so on. Note the schizophrenic 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 the primitives (5), (6), (8), and (9), dataflow synchronous languages such as LUSTRE (Halbwachs, 1993) and SIGNAL (LeGuernic *et al.*, 1999) are easily encoded. Note that the primitives (5), (6), (8), and (9) and their composition are stuttering invariant STS; i.e., they satisfy condition (4).

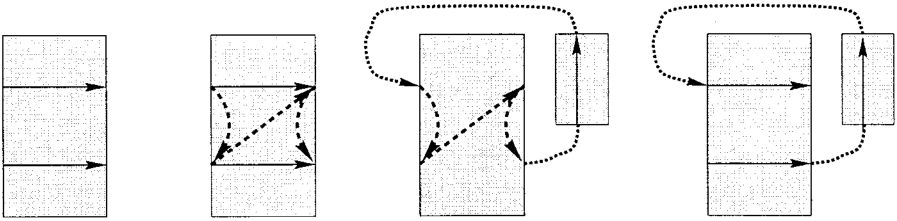
3. COMPOSITIONALITY IN CODE GENERATION: INFORMAL ANALYSIS

In this section, we informally discuss issues of compositionality aiming at code generation. After a brief review of the problems, we acknowledge the importance of extending our basic STS model with preorders; preorders are useful to capture causality, to specify schedulings, and to model communications in a distributed environment. Also, preorders are instrumental in handling abstractions. Then we discuss causality analysis and we analyze a few simple examples. Separate compilation is discussed, using preorders: we show that separate compilation requires a new level of intermediate code which allows us to store and reuse modules in a correct way. Finally we discuss the issue of distributed code generation on an asynchronous architecture.

3.1. What Is the Problem?

Basically, the problem is twofold: (1) bruteforce separate compilation can be the source of deadlock, 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.

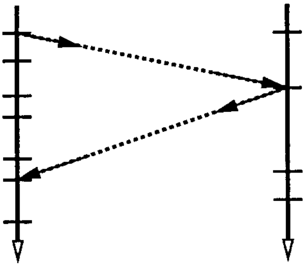
Naive Separate Compilation may be Dangerous. This is illustrated in the following:



The first diagram depicts the "dependencies" associated with some STS specification: the first output needs the first input for its computation, and the second output needs the second input for its computation. The second diagram shows a possible scheduling, corresponding to the standard scheduling: (1) read inputs, (2) compute reaction, and (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.

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 later see that this does not need to be the case, however.

Desynchronization. This is illustrated in the following:



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 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 the 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.*

3.2. Scheduling Specifications

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 (Aabelsberg and Rozenberg, 1988) has been extended to so-called "semi-commutations" (Clerbout and Latteux, 1987), and this technique has been recently applied to the implementation of reactive automata on distributed architectures (Caillaud *et al.*, 1997). Causality preorder relations have also been used in a different way in (LeGuernic and Gautier, 1991), and also in (Benveniste *et al.*, 1994), from which we borrow the essentials of the present technique. In addition to modelling causality relations, preorders can be used to specify scheduling requirements, and they can also be used to model send/receive type of communications.

STS *with Scheduling Specifications.* We consider a set V of variables. A *preorder* on the set V is a relation (generically denote by $\preccurlyeq$) which is reflexive ($x \preccurlyeq x$) and transitive ($x \preccurlyeq y$ and $y \preccurlyeq z$ imply $x \preccurlyeq z$). To $\preccurlyeq$ we associate the equivalence relation $\asymp$, defined by $x \asymp y$ iff $x \preccurlyeq y$ and $y \preccurlyeq x$. If equivalence classes of $\asymp$ are singletons, then $\preccurlyeq$ is a *partial order*. Preorders are naturally specified via (*possibly cyclic*) directed graphs, denoted

$$x \to y \qquad \text{for} \quad x, y \in V, \tag{10}$$

by defining $x \preccurlyeq z$ iff there is a path originating from x and terminating in z. The *supremum* of two preorders, written

$$\preccurlyeq_1 \vee \preccurlyeq_2, \tag{11}$$

is the least preorder which is an extension of $\preccurlyeq_1$ and $\preccurlyeq_2$. The set of all preorders on V is denoted Λ_V.

A *labelled preorder* on V is a preorder on V, together with a value $s[v]$ for each $v \in V$ over its domain. A *state* $\vec{s}$ is a labelled preorder. The set of all states is denoted $\vec{S}$. As before for STS, we denote by S the set of all type-consistent interpretations of V. Thus $\vec{S} = S \times \Lambda_V$, and a state $\vec{s}$ decomposes as

$$\vec{s} = (s, \preccurlyeq_V). \tag{12}$$

An STS *with scheduling specifications* is a triple $\vec{\Phi} = \langle V, \Theta, \vec{\rho} \rangle$, where V, Θ are as before, and

$$\vec{\rho} \subset S \times \vec{S} = S \times S \times \Lambda_V; \tag{13}$$

i.e., $\vec{\rho}$ relates the value for the tuple of previous variables to the current state.

By convention, transition relation $\vec{\rho}$ is trivially extended to a transition on $\vec{S}$, i.e., a subset of $\vec{S} \times \vec{S}$, and runs are sequences $\vec{s}_0, \vec{s}_1, \vec{s}_2, ...$ that are consistent with transition relation (13).

We shall denote by ρ the transition relation on S obtained by projecting $\vec{\rho}$ on $S \times S$, i.e., by ignoring the preorder component. Note that $\Phi = \langle V, \Theta, \rho \rangle$ is an ordinary STS. The *composition* of two STS with scheduling specifications

$$\vec{\Phi} = \vec{\Phi}_1 \parallel \vec{\Phi}_2 \tag{14}$$

is defined as follows:

1. Associated underlying STS (without scheduling specifications) are simply composed:

$$\Phi = \Phi_1 \parallel \Phi_2 \tag{15}$$

Then we need to define how preorders are combined.

2. For s a state for Φ, for $i=1,2$ let s_i be the restriction of s to V_i; we know that s_i is a state for Φ_i. Let $\vec{s}_i=(s_i, \prec_{V_i})$ be the corresponding state for $\vec{\Phi}_i$; cf. (12). Define

$$\preccurlyeq_V =_{\text{def}} \preccurlyeq_{V_1} \vee \preccurlyeq_{V_2} (\text{cf. } (11)) \tag{16}$$

$$\vec{s} =_{\text{def}} (s, \preccurlyeq_V). \tag{17}$$

Thus (15), (16), and (17) define how states of the components $\vec{\Phi}_i$ are combined together, building up the states and runs of $\vec{\Phi}=\vec{\Phi}_1 \parallel \vec{\Phi}_2$. Again, composition $\parallel$ as extended to STS with scheduling specifications is commutative and associative.

Notation for Scheduling Specifications. We now introduce convenient notation for the graphs generating the above-introduced preorders. The notation $u \rightarrow v$ corresponds to the edge (10). For b a variable of type $bool \cup \{\perp\}$, and u, v variables of any type, the following generic conjunct will be used to specify preorders,

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

also written

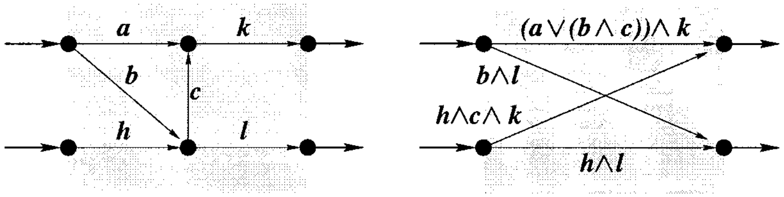
$$u \xrightarrow{b} v, \quad \text{resp.} \quad u \xrightarrow{\bar{b}} v.$$

In Subsection 5.1, it is shown that scheduling specifications have the following properties:

$$u \xrightarrow{b} y \parallel y \xrightarrow{c} z \Rightarrow x \xrightarrow{b \wedge c} z \tag{18}$$

$$u \xrightarrow{b} y \parallel x \xrightarrow{c} y \Rightarrow x \xrightarrow{b \vee c} y. \tag{19}$$

Properties (18) and (19) 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 (18) and (19).

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 conjunct 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, and even more, (2) these primitives allow one to express the most general synchronization mechanisms that are compatible with the paradigm of perfect synchrony (Benveniste *et al.*, 1992). We recall next this set of basic conjuncts for the sake of clarity:

$$\begin{gathered}
\textbf{if} \quad b \quad \textbf{then} \quad w = u \quad \textbf{else} \quad w = v \\[1ex]
u \xrightarrow{b} w \\[1ex]
\left.\begin{aligned} w &= f(u_1, ..., u_k) \\ h_w = h_{u_1} &= \cdots = h_{u_k} \end{aligned}\right\} .
\end{gathered} \tag{20}$$

In addition to the set (20) 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 (21) given next.

In formulas (21), 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 (21), labels of schedulings are expressions involving variables in the domain $\{\bot, \text{F}, \text{T}\}$ ordered by $\{\bot < \text{F} < \text{T}\}$; with this in mind, expressions involving the symbols "$\wedge$" (min) and "$\vee$" (max) have a clear meaning:

$$\begin{array}{llcl}
\text{(R-1)} & \qquad \forall u & & h_u \rightarrow u \\[2ex]
\text{(R-2)} & \begin{array}{l} \textbf{if} \quad b \quad \textbf{then} \quad w = u \\ \qquad\quad \textbf{else} \quad w = v \end{array} & \Rightarrow & \left\{ \begin{array}{l} b \xrightarrow{h_b \wedge (h_u \vee h_v)} h_w \\[1ex] h_u \xrightarrow{b \wedge h_u} h_w \\ h_v \xrightarrow{\bar{b} \wedge h_v} h_w \\[1ex] u \xrightarrow{b \wedge h_u} w \\ v \xrightarrow{\bar{b} \wedge h_v} w \end{array} \right. \\[2ex]
\text{(R-3)} & \qquad u \xrightarrow{b} w & \Rightarrow & b \rightarrow h_w \\[2ex]
\text{(R-4)} & \left.\begin{aligned} w &= f(u_1, ..., u_k) \\ h_w = h_{u_1} &= \cdots = h_{u_k} \end{aligned}\right\} & \Rightarrow & u_i \xrightarrow{h_w} w.
\end{array} \tag{21}$$

Note that there is no rule involving variables of the form ξ_z^-, as previous state variables are available prior to starting the current reaction and thus do not participate in the causality calculus. Rules (R-1)–(R-4) are formally justified in Section 5. 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 circuit-free at each instant, meaning that *it is never true that*

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

$$\textbf{and}$$

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

where x_1 and x_2 are distinct variables

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

$$\begin{array}{ll} & \textbf{if } b_1 \textbf{ then } x = \mathrm{exp}_1 \\ \wedge & \textbf{if } b_2 \textbf{ then } x = \mathrm{exp}_2 \end{array}$$

holds in P and the exp_1 and exp_2 are different expressions, then

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

never holds in P.

Then P is said to be *executable*, and **sched**(P) provides (dynamic) scheduling specifications for this run. Note that proof obligations resulting from the above two conditions are generally no automatically provable; therefore abstractions may have to be considered.

Summary. What do we have at this stage?

1. STS composition is just the conjunction of constraints.

2. Scheduling specifications do compose as well.

3. Since causality analysis is based on an abstraction, Rules (R-1)–(R-4) for inferring scheduling from causality are bound to the *syntax* of the STS conjuncts. 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.[7] But this latter operation is global and not compositional: here we reach the limits of ideal compositionality.

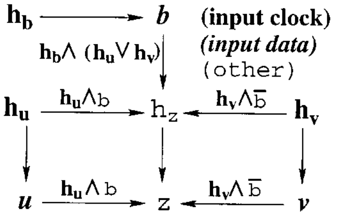
[7] This is part of the job performed by the SIGNAL compiler's "clock calculus."

3.3. Causality Analysis: 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 boldface italic denote input data, and vertices in Courrier 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 siting on the incoming branches are evaluated prior to the considered vertex. Thus, when this vertex is to be evaluated, the other variables needed for its evaluation are already known. Resulting directed graphs (which are labelled with Booleans) specify the set of all legal schedulings for the execution of the considered STS; this is formalized in Section 5.

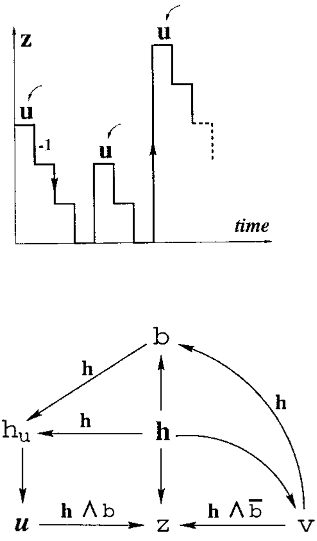
A reactive STS.

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



In the above example, input data are associated with their 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.



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

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

FIG. 1. Scheduling from causality analysis for the example.

Applying scheduling rules (R-1)–(R-4) and then performing some straightforward simplifications, we get the result shown in Fig. 1. Note the change in control: {input clock, input data} have 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 u occurs on demand, when condition b is true. We propose to call such an STS "proactive."

3.4. Generating Scheduling for Separate Modules

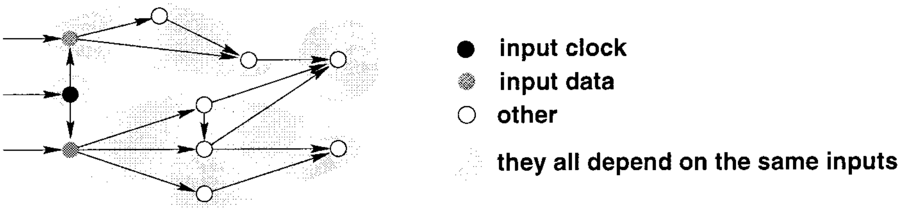
Relevant target architectures for embedded applications are typically (1) purely sequential code (such as C-code), (2) code using a threading or tasking mechanism provided by some kind of a real-time OS (here the threading mechanism offers some degree of concurrency), or (3) DSP-type multiprocessor architectures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

On the other hand, the scheduling specifications we derive from causality rules (R-1)–(R-4) still exhibit maximal 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 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.

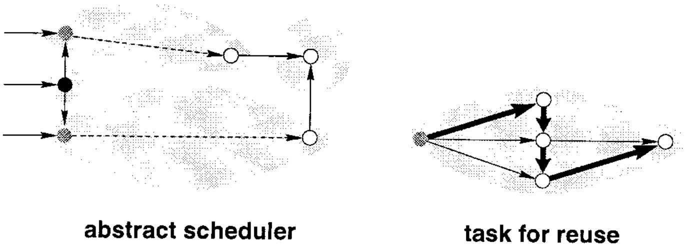
We shall however see that this does not need to be the case. Instead, a careful use of the scheduling specifications of an STS will allow us to decompose it into modules that can be stored as object code for further reuse, whatever the actual environment and implementation architecture will be.

For the sake of clarity, we restrict our discussion to the case of single-clocked STS, i.e., an STS in which all declared variables have the same clock. The issue is illustrated in the following figure, in which the directed graph defining the circuit-free scheduling specification of some single-clocked STS is depicted:



In the above figure, the gray zones group all variables which depend on the same subset of inputs; let us cal them "tasks." Tasks are not subject to the risk of creating fake deadlocks from implementation, unlike the example from Subsection 3.1. In fact, as all variables belonging to the same task depend on the same inputs, each task can be executed safely according to the following scheme: (1) collect inputs and (2) execute task.

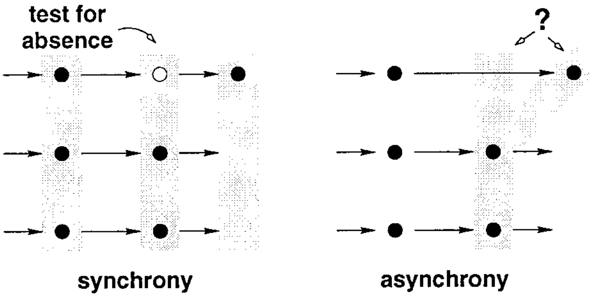
In the next figure, we show how the actual implementation is prepared:



The thick arrows inside the task depicted on the right 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 tasks 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 mechanisms 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. Using our notion of scheduling specification, the above approach easily extends to general STS, in which several different clocks are involved.

3.5. Relaxing Synchrony

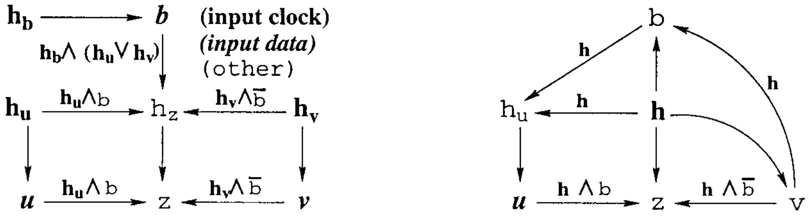
Loosening Synchrony. The major problem is that of testing for absence in an asynchronous environment. This is illustrated in the following figure in which information about the presence of variables in the considered instant is lost when passing from the left- to right-hand side, since an explicit definition of the "instant" is not available anymore:



The question mark indicates that it is generally not possible, in an asynchronous environment, to decide upon the presence/absence of a signal relative to another one. While testing for absence is perfectly sound in a synchronous paradigm, it is meaningless in an asynchronous one.

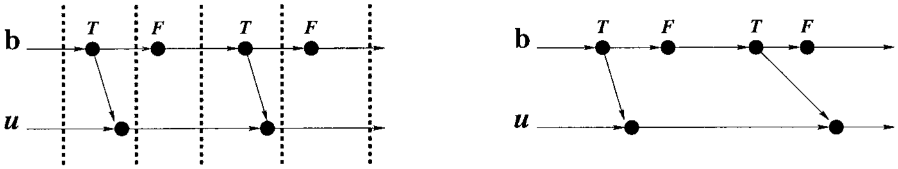
The solution consists in restricting ourselves to so-called *endochronous* STS. Endochronous STS are those for which the control depends only on (1) the past state and (2) the values possibly carried by environment signals, but not on the presence/absence status of these signals. For an endochronous STS, losing the synchronization barriers that define the successive reactions will not result in changing its semantics; this is formalized in Subsection 4.2.

An example of an STS which is “exochronous” is the “reactive” STS given on the left-hand side of the following figure, whereas the “proactive” STS shown on the right-hand side is endochronous:



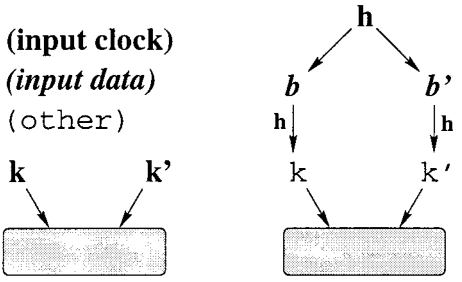
In the diagram on the left-hand side, three different clocks are source nodes on the directed graph. This means that the first decision in executing a reaction consists in deciding upon the relative presence/absence of these clocks. In contrast, in the diagram on the right-hand side, only one clock,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

How endochrony allows us to desynchronize an STS is illustrated in an intuitive way in the following diagram, which depicts the scheduling specification associated with the (endochronous) pseudo-statement “**if** *b* **then** *get*_*u*”:



In the diagram on the left, a history of this statement is depicted, showing the successive instants (or reactions) separated by thick dashed lines. In the right-hand side diagram, thick dashed lines have been removed. Clearly, no information has been lost: we know that *u* should happen exactly when *b* = T, and thus awaiting for the value of *b* is enough for deciding whether *u* is to be waited for. A formal study of desynchronization and endochrony is presented in Section 4.

Moving from exochronous programs to endochronous programs can be performed; we only show one typical but simple example:



The idea is to add to the considered STS a monitor which delivers the presence/absence information via two Boolean variables b, b' with identical clocks h, and such that $[k = \mathrm{T}] = [b = \mathrm{T}]$, 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 Subsection 4.5.

Loosening Synchronous Composition. The second question is that of preserving the semantics of synchronous composition when an asynchronous communication medium is used. In the synchronous programming paradigm, communication occurs via instantaneous broadcast, meaning that all components must agree on (1) which variable is present/absent in the considered reaction, and then (2) what is the value carried by each present variable. Again this protocol is meaningless in an asynchronous communication medium. In Subsection 4.3, it is shown that the condition for semantics preserving desynchronization of the communication is that the considered pair of STS should be *isochronous.*

Isochrony is a property of the synchronous composition $P \parallel Q$ of two STS. Roughly speaking, a pair of STS is isochronous if every pair of reactions, of P and Q, respectively, which agree on *present* common variables, also agree on *all* common variables. Thus, again, common agreement for composition of reactions can disregard absence.

Endochrony and isochrony are the basic concepts for our theory of desynchronization. For this theory to hold, requirements for the communication medium are: (1) it should not lose messages and (2) it should not change the order of messages associated with each given variable.

3.6. Modular Design, GALS Architectures

From the theory informally presented in the previous subsections, the following approach results for modular design and distributed implementations of reactive systems. The target architecture is globally asynchronous, locally synchronous (GALS) by nature. The whole approach is summarized in Fig. 2, where the considered

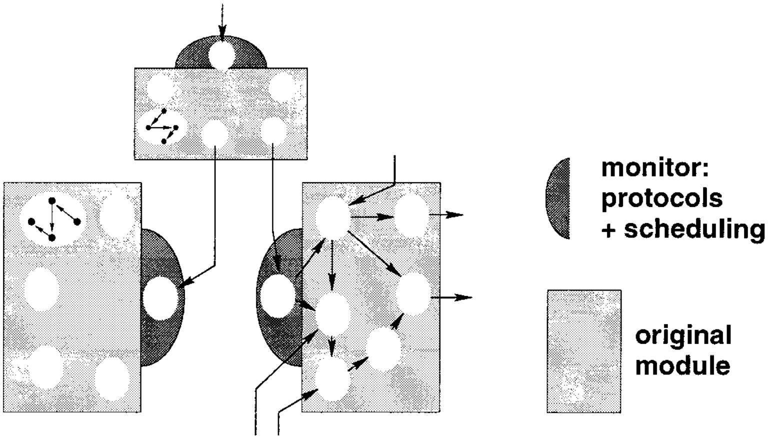


FIG. 2. Implementation architecture.

STS is assumed to possess a unique, deterministic execution; i.e., it satisfies the correctness criteria stated in Subsection 3.2. 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 $\mathtt{P} = \mathtt{P}_1 \parallel \mathtt{P}_2 \parallel \mathtt{P}_3$. We assume here that this partitioning has been given by the designer, based on functional and architectural considerations.

White bubbles inside the gray rectangles depict the structuration into tasks as discussed in Subsection 3.4. The black half-ellipses denote the monitors. Monitors are in charge of (1) providing the additional protocols 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 that they respect the send–receive abstract communication model and conform to the scheduling constraints set by the monitors.

4. FORMAL STUDY OF DESYNCHRONIZATION

How far/close is indeed synchrony from asynchrony has already been discussed in the literature, thus questioning the oversimplified vision of "zero time" computation and instantaneous broadcast communication. An early paper (Benveniste and Berry, 1991) informally discussed the link between perfect synchrony and token-based asynchronous dataflow networks; see in particular Section V therein. The first formal and deep study is that of Caspi (1992): a precise relation is established between so-called well-clocked synchronous functional programs and the subset of Kahn networks amenable to "bufferless" evaluation.

Distributed code generation from synchronous programs requires one to address the issue of the relationship between synchrony and asynchrony in some way or another. Mapping synchronous programs to a network of automata, communicating asynchronously via unbounded fifos, has been proposed in Caillaud *et al.* (1997). Mapping SIGNAL programs to distributed architectures was proposed in Maffeis and LeGuernic (1994) and Aubry (1997), based on an early version of the theory we present in this paper. The SYNDEX tool (Sorel and Lavarenne; Sorel, 1996) also implements a similar approach. Recent work (Berry and Sentovich, 1998) on the POLIS system proposes to reuse the "constructive semantics" approach for the ESTEREL synchronous language, with CFSM (codesign finite state machines) as a model of synchronous machines which can be desynchronized.

Independently, another route to relate synchrony and asynchrony has been followed. In Benveniste and LeGuernic (1990) and LeGuernic *et al.* (1991) it was shown how *nondeterministic* SIGNAL programs can be used to model asynchronous communication media such as queues and buffers. *Reactive modules* were proposed (Alur and Henzinger, 1996) as a synchronous language for hardware modelling, in which asynchrony is emulated by the way of nondeterminism. Although this is of interest, we believe that this approach is no suited to analyzing true asynchrony, in which no notion of a global state is available, unlike for synchrony.

We first informally discuss the essentials of asynchrony. Synchronous transition systems were defined in Subsection 2.2, and their asynchronous counterpart is defined in Subsection 4.1, where desynchronization is also formally defined. The rest of this section is devoted to the analysis of desynchronization and its inverse, namely resynchronization.

4.1. Desynchronizing STS, and Two Fundamental Problems

We first start with an informal discussion, following the discussion of Subsection 2.1. Keeping in mind the essentials of the synchronous paradigm, we are now ready to discuss informally how asynchrony relates to synchrony. Referring to points 1, 2, and 3 of the discussion of Subsection 2.1, the following can be stated about asynchrony:

1. Reactions cannot be observed anymore: as no global clock exists, the global synchronization barriers which indicate the transition from one reaction to the next one are no longer available. Instead, we only assume a reliable distributed communication medium, in which messages are not lost, and messages within each individual channel are sent and delivered in the same order. We call a *flow* such a totally ordered sequence of messages.

2. Absence cannot be sensed, and thus cannot be used to exercise control.

3. Composition occurs by means of separately unifying each common flow of the two components. This models in particular the communications via asynchronous unbounded fifos, such as used, say, in Kahn networks. Rendezvous type of communication can also be abstracted in this way.

From the definition (1) of a run of an STS, we can say that a run is a sequence of tuples of values in domains extended with the extra symbol $\perp$. Desynchronizing a run amounts to discarding the synchronization barriers defining the successive reactions. Hence, for each variable $v \in V$, we only know the ordered sequence of *present* values. Thus desynchronizing a run amounts to mapping a *sequence of tuples* of values in domains extended with the extra symbol $\perp$, into a *tuple of sequences* of present values, one sequence per each variable. This is formalized next.

For σ: $s_0, s_2, s_2, \ldots$ a run for Φ, we decompose state s_k as

$$s_k = (s_k[v])_{v \in V}.$$

Thus we can rewrite run σ as

$$\sigma = (\sigma[v])_{v \in V},$$

where

$$\sigma[v] = s_0[v], s_1[v], \ldots, s_k[v], \ldots.$$

Now, compress each $\sigma[v]$ by deleting those $s_k[v]$ that are equal to $\perp$. Formally, we denote by $k_0, k_1, k_2, \ldots$ the subsequence of $k = 0, 1, 2, \ldots$ such that $s_k[v] \neq \perp$. Then we set

$$\sigma^a = (\sigma^a[v])_{v \in V},$$

where

$$\sigma^a[v] = s_{k_0}[v], s_{k_1}[v], s_{k_2}[v], ...$$

This defines the *desynchronization mapping*

$$\sigma \longmapsto \sigma^a, \tag{22}$$

where each

$$\sigma^a[v] = s_{k_0}[v], s_{k_1}[v], s_{k_2}[v], ...$$

is called a *flow* in the sequel.

For $\Phi = \langle V, \Theta, \rho \rangle$ an STS, we define

$$\Phi^a =_{\text{def}} \langle V, \Sigma^a \rangle, \tag{23}$$

where Σ^a is the family of all σ^a, for σ ranging over the set of runs of Φ. For $\Phi_i = \langle V_i, \Theta_i, \rho_i \rangle$, $i = 1, 2$, we define

$$\Phi_1^a \parallel^a \Phi_2^a =_{\text{def}} \langle V, \Sigma^a \rangle, \qquad \text{where} \quad \begin{cases} V = V_1 \cup V_2 \\ \Sigma^a = \Sigma_1^a \wedge^a \Sigma_2^a \end{cases} \tag{24}$$

and $\wedge^a$ denotes the conjunction of sets of asynchronous runs, which we define now. For $\sigma_i^a \in \Sigma_i^a$, $i = 1, 2$, we say that σ_1^a and σ_2^a are *unifiable*, written

$$\sigma_1^a \bowtie^a \sigma_2^a, \tag{25}$$

if the following condition holds:

$$\forall v \in V_1 \cap V_2 : \sigma_1^a[v] = \sigma_2^a[v] \text{ holds.}$$

If condition (25) holds, then we define $\sigma^a =_{\text{def}} \sigma_1^a \wedge^a \sigma_2^a$ as

$$\forall v \in V_1 \cap V_2 : \sigma^a[v] = \sigma_1^a[v] = \sigma_2^a[v]$$
$$\forall v \in V_1 \backslash V_2 : \sigma^a[v] = \sigma_1^a[v]$$
$$\forall v \in V_2 \backslash V_1 : \sigma^a[v] = \sigma_2^a[v].$$

Finally, Σ^a is the set of the so-defined σ^a. Thus asynchronous composition proceeds via unification of shared flows.

Synchrony vs Asynchrony? At this point two natural questions arise, namely:

QUESTION 1 (Desynchronizing a Single STS). *Is resynchronization feasible and uniquely defined? More precisely, is it possible to uniquely reconstruct the original run σ for our* STS *from its desynchronized version σ^a as defined in* (22)?

Question 2 (Desynchronizing a Communication). *Does communication behave equivalently for both the synchronous and asynchronous compositions? More precisely, does the following property hold*:

$$\Phi_1^a \parallel^a \Phi_2^a = (\Phi_1 \parallel \Phi_2)^a ? \tag{26}$$

If Question 1 had a positive answer, then we could desynchronize a run of the considered sts, and then still recover the original synchronous run. Thus a positive answer to Question 1 would guarantee the preserving of the synchronous semantics when performing desynchronization, for a single sts.

On the other hand, if the question (26) had a positive answer, then we could interpret our sts composition equivalently as synchronous or asynchronous.

Unfortunately, neither 1 nor 2 has positive answers in general, due to the possibility of exercising control by way of absence in synchronous composition $\parallel$. In the following section, we show that Questions 1 and 2 have positive answers under certain sufficient conditions, in which the two notions of *endochrony* (for point 1) and *isochrony* (for point 2) play a central role.[8]

4.2. Endochrony and Resynchronization

4.2.1. Formal Results

In this section, we use notation from Subsection 2.2. For $\Phi = \langle V, \Theta, \rho \rangle$ an sts, and s a reachable state of Φ, we denote by s^h the clock abstraction of s, defined by

$$\forall v \in V : s^h[v] \in \{\bot, \top\}, \qquad \text{and} \qquad s^h[v] = \bot \Leftrightarrow s[v] = \bot. \tag{27}$$

For $\Phi = \langle V, \Theta, \rho \rangle$ and sts, s^- a reachable previous state for Φ, and $W' \subseteq W \subseteq V$, we say that W' is a *clock inference of W given* s^-, written

$$W' \hookrightarrow_{s^-} W, \tag{28}$$

if, for each state s reachable from s^- for Φ, knowing the presence/absence and actual value carried by each variable belonging to W' allows us to determine exactly the presence/absence for each variable belonging to W. In other words,

$$s[W'] \text{ determines } s^h[W] \tag{29}$$

If $W' \hookrightarrow_{s^-} W_1$ and $W' \hookrightarrow_{s^-} W_2$ hold, then $W' \hookrightarrow_{s^-} (W_1 \cup W_2)$ follows; thus there exists a greatest W such that $W' \hookrightarrow_{s^-} W$ holds. Hence we can consider the unique increasing chain, for s^- given,

$$\emptyset = V(0) \hookrightarrow_{s^-} V(1) \hookrightarrow_{s^-} V(2) \hookrightarrow_{s^-} \cdots \tag{30}$$

[8] Endochronous, from ancient Greek *ενδο*, inside, and *χρονος*, time; isochronous, from ancient Greek *ισο*, identical, and *χρονος*, time. It is sometimes nice to remember that ancient Greeks were great scientists, and thus honor them by reusing their words in our context.

of subsets of V such that, for each k, $V(k)$ is the greatest set of variables such that $V(k-1) \hookrightarrow_{s^-} V(k)$ holds. As $\emptyset = V(0)$, $V(1)$ consists of the subset of variables that are present as soon as the considered STS gets activated.[9] Of course chain (30) must become stationary at some finite $k_{\max}$: $V(k_{\max}+1) = V(k_{\max})$. In general, we only know that $V(k_{\max}) \subseteq V$. Chain (30) is called the *synchronization chain* of Φ.

DEFINITION 1 (Endochrony). STS Φ is said to be *endochronous* if, for each state s^- reachable for Φ, $V(k_{\max}) = V$, i.e., if the following condition is satisfied: the synchronization chain

$$\textbf{(E)} \qquad \emptyset = V(0) \hookrightarrow_{s^-} V(1) \hookrightarrow_{s^-} V(2) \hookrightarrow_{s^-} \cdots \text{ converges to } V. \tag{31}$$

Condition (31) expresses that the presence/absence of all variables can be inferred *incrementally* from already known values carried by present variables and state variables of the STS in consideration. Hence no test for presence/absence on the environment is needed. The following theorem justifies our approach:

THEOREM 1. *Consider an* STS $\Phi = \langle V, \Theta, \rho \rangle$.

1. *Conditions* (a) *and* (b) *are equivalent, where*:

(a) *Φ is endochronous*;

(b) *for each $\delta \in \Sigma^a$, we can reconstruct the corresponding synchronous run σ such that $\sigma^a = \delta$, in a unique way up to silent reactions.*

2. *Assume that Φ is endochronous and stuttering invariant. If $\Phi' = \langle V, \Theta, \rho' \rangle$ is another endochronous and stuttering invariant* STS *then*

$$(\Phi')^a = \Phi^a \Rightarrow \Phi' = \Phi. \tag{32}$$

Proof. We prove successively points 1 and 2.

1. We fix the previous state s^- and prove the result by induction. Pick a $\delta \in \Sigma^a$, and assume for the moment that we were able to decompose it as

$$\underbrace{s_1, s_2, ..., s_n}_{n\text{—initial segment of }\sigma} \quad , \delta_n, \tag{33}$$

i.e., into a finite sequence of length n composed of nonsilent states s_i (the head of the synchronous run σ we wish to reconstruct), followed by the tail of the asynchronous run δ, which we denote by δ_n, and we assume that such a decomposition is unique. Then we claim that

$$(33) \text{ is also valid with } n \text{ substituted by } n+1. \tag{34}$$

[9] Of course we assume here that no variable is absent in every reachable state.

To prove (34), we note that, when STS Φ gets activated, then we know that variables belonging to $V(1)$ will be present in the considered state. By assumption, the clock-abstracted state $s^h_{n+1}[V(1)]$, having $V(1)$ as variables, is uniquely determined. In the following we write $s^h_{n+1}(1)$ for short instead of $s^h_{n+1}[V(1)]$. Thus, the presence/absence of variables for state $s_{n+1}(1)$ is known; the values carried by the variables present remain to be determined.

For $v \in V_1$, we simply pick the value carried by the minimal element of the sequence associated with variable v in δ_n. Values carried by corresponding state variables are updated accordingly. Thus we know all of $s_{n+1}(1)$.

Next we move on constructing $s_{n+1}(2)$. From $s_{n+1}(1)$ we know $s^h_{n+1}(2)$. Thus we know how to split V_2 into present and absent variables for the considered state. Pick the present ones, and repeat the same argument as before to get $s_{n+1}(2)$.

Repeat this argument until $V(k) = V$ for some finite k (by endochrony assumption). This proves claim (34).

Given the initial condition for δ, we get from (34), by induction, the desired proof that (a) $\Rightarrow$ (b).

Next, we prove (b) $\Rightarrow$ (a). We assume that Φ is not endochronous, and show that condition (b) cannot be satisfied. If Φ is not endochronous, there must be some reachable state s^- for which chain (31) does not converge to V. Thus again we pick a $\delta \in \Sigma^a$, decomposed as for case 1, cf. formula (33),

$$\underbrace{s_1, s_2, ..., s_n}_{n\text{—initial segment of } \sigma} \quad , \delta_n,$$

and we assume in addition that $s_n = s^-$, the given state for which endochrony is violated. We now show that (34) is disproved. Let $k_* \geqslant 0$ be the smallest index such that $V(k) = V(k+1)$, we know $V_{k_*} \neq V$. Thus we can apply the algorithm of case 1 for reconstructing the reaction, until variables of V_{k_*}. Then presence/absence for variables belonging to $V \backslash V_{k_*}$ cannot be determined based on the knowledge of variables belonging to V_{k_*}.Thus there are several possible extensions for $s^h_{n+1}(k_* + 1)$ and thus the $(n+1)$th reaction is not determined in a unique way. Hence condition (b) is falsified.

2. Assume that Φ is endochronous, and consider Φ' as in point 2 of the theorem. As both Φ and Φ' are stuttering invariant, point 2 is an immediate consequence of point 1.

Comments. 1. For an STS, endochrony is not decidable in general. It is decidable for STS involving, say, only finite domains for their variables, and model checking can be used for that. For general STS, model checking can be used, in combination with abstraction techniques. The case of interest is when the chain $V(0), V(1), ...$ does not depend upon the particular state s^-, and we write simply $V(k) \hookrightarrow V(k+1)$ in this case.

2. The proof of this theorem in fact provides an effective algorithm for the on-the-fly reconstruction of the successive reactions, for a desynchronized run of an endochronous program.

(Counter)examples.

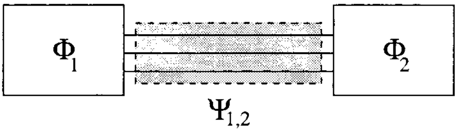
Examples.

- A single-clocked STST.
- STS "**if** $b=\mathrm{T}$ **then** *get* u," where b, u are the two inputs, and b is Boolean. The clock of b coincides with the activation clock for this STS, and thus $V(1)=\{b\}$. Then, knowing the value for b indicates whether or not u is present, and thus $V(2)=\{b, u\}=V$.

Counterexample. STS "**if** ([**present** a] $\|$ [**present** b]) **then**..." is not endochronous, as the environment is free to offer any combination of presence/absence for the two inputs a, b. Thus $\emptyset=V(0)=V(1)=V(2)=\cdots \overset{\subset}{\neq} V$, and endochrony does not hold.

4.2.2. Practical Consequences

A first use of endochrony is shown in the following figure:



In this figure, a pair (Φ_1, Φ_2) of STS is depicted, with W as a set of shared variables. Rewrite their composition as

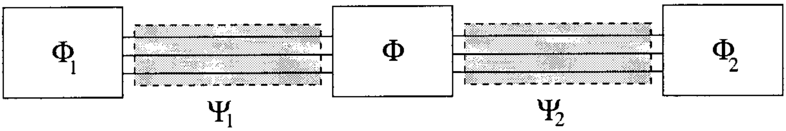
$$\Phi_1 \| \Phi_2=\Phi_1 \| \Psi_{1,2} \| \Phi_2,$$

where $\Psi_{1,2}$ is the restriction of $\Phi_1 \| \Phi_2$ to W; hence $\Psi_{1,2}$ models the synchronous communication channel. Using the property $\Phi \| \Phi=\Phi$ for every STS Φ, we get

$$\Phi_1 \| \Phi_2=\underbrace{(\Phi_1 \| \Psi_{1,2})}_{\tilde{\Phi}_1} \| \underbrace{(\Psi_{1,2} \| \Phi_2)}_{\tilde{\Phi}_2}=\tilde{\Phi}_1 \| \tilde{\Phi}_2. \tag{35}$$

Assume now that channel model $\Psi_{1,2}$ is endochronous, and composition $\Phi_1 \| \Phi_2$ is implemented as the (equivalent) composition $\tilde{\Phi}_1 \| \tilde{\Phi}_2$. Then, as $\tilde{\Phi}_1$ knows channel $\Psi_{1,2}$ and the latter is endochronous, then communication can be equivalently implemented according to perfect synchrony or full asynchrony.

This is fine, but it does not extend to networks of STS involving more than two nodes. The following figure shows an example:



Assume that Ψ_1, Ψ_2 are both endochronous. Then communication between Φ_1 and Φ on the one hand, and Φ and Φ_2 on the other hand, can be desynchronized. Unfortunately, communication between Φ_1 and Φ_2 via Φ cannot as it is not true in general that $\Psi_1 \parallel \Phi \parallel \Psi_2$ is endochronous. The problem is that endochrony is not compositional; hence even ensuring in addition that Φ itself is endochronous would not do. Thus we would need to ensure that Ψ_2, Ψ_2 as well as $\Psi_1 \parallel \Phi \parallel \Psi_2$ are all endochronous, not an elegant solution when networks are considered! Thus we move on to introducing the alternative notion of *isochrony*, which focusses on communication, and is compositional.

4.3. Isochrony, and Synchronous and Asynchronous Compositions

The next result addresses the question of when property (26) holds true. We are given two STS $\Phi_i = \langle V_i, \Theta_i, \rho_i \rangle$, $i = 1, 2$. Denote by $W = V_1 \cap V_2$ the set of their common variables, and by $\Phi = \Phi_1 \parallel \Phi_2$ their synchronous composition. For s a reachable state in Φ, we denote by $s_1 =_{\mathrm{def}} s[V_1]$ and $s_2 =_{\mathrm{def}} s[V_2]$ the restrictions of state s to Φ_1 and Φ_2, respectively. Note that, for $i = 1, 2$, s_i is a reachable state for Φ_i. Corresponding notations s^-, s_1^-, s_2^- for past states will be used accordingly.

Definition 2 (Isochrony). Consider a pair (Φ_1, Φ_2) of STS. Transitions of Φ_i, $i = 1, 2$, are written (s_i^-, s_i). Consider the following conditions on pairs $((s_1^-, s_1), (s_2^-, s_2))$ of transitions for (Φ_1, Φ_2):

(i) 1. $s_1^- = s^-[V_1]$ and $s_2^- = s^-[V_2]$ hold for some reachable state s^- for Φ; in particular s_1^- and s_2^- are unifiable;

2. none of the states s_i, $i = 1, 2$, are silent on the common variables; i.e., it is not the case that, for some $i = 1, 2$, $s_i[v] = \perp$ holds $\forall v \in W$;

3. s_1 and s_2 coincide over the set of *present* common variables,[10] i.e.,

$$\forall v \in W : (s_1[v] \neq \perp \text{ and } s_2[v] \neq \perp) \Rightarrow s_1[v] = s_2[v],$$

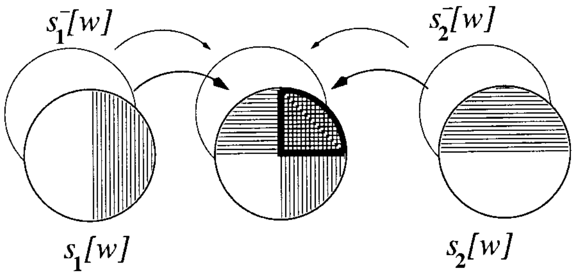
(ii) States s_1 and s_2 coincide over the *whole set* of common variables, i.e., states s_1 and s_2 are unifiable:

$$s_1 = s[V_1] \text{ and } s_2[V_2] \text{ hold for some state } s \text{ for } \Phi.$$

The pair (Φ_2, Φ_2) is called *isochronous* if condition (i) implies condition (ii), for each pair $((s_1^-, s_1), (s_1^-, s_2))$ of transitions for (Φ_1, Φ_2).

Comment. Roughly speaking, the condition of isochrony expresses that unifying over *present* common variables is enough to guarantee the unification of the two considered states s_1 and s_2. The condition of isochrony is illustrated on the following figure:

[10] By convention this is satisfied if the set of present common variables is empty.



The figure depicts, for unifiable previous states s_1^-, s_2^-, corresponding states s_1, s_2, where (s_i^-, s_i) is a valid transition for Φ_i. It shows the interpretation of s_1 (circle on the left) and s_2 (circle on the right) over shared variables W. White and hatched areas represent absent and present values, respectively. The two left and right circles are superimposed in the middle circle. In general, vertically and horizontally hatched areas do not coincide, even if s_1 and s_2 unify over the subset of shared variables that are present for both transitions (crosshatched area). Pictorially, unification over the crosshatched area does not imply in general that the hatched areas coincide. Isochrony indeed requires that unification over the crosshatched area does imply that hatched areas coincide; hence unification of s_1 and s_2 follows.

The following theorem justifies introducing this notion of isochrony.

THEOREM 2. 1. *If the pair* (Φ_1, Φ_2) *is isochronous, then it satisfies property* (26).

2. *Conversely, assume in addition that* Φ_1 *and* Φ_2 *are both endochronous. If the pair* (Φ_1, Φ_2) *satisfies property* (26), *then it is isochronous.*

Thus, isochrony is sufficient for (26) *to holds, and it is also in fact necessary when the components are endochronous.*

Comments. 1. We already discussed the importance of guaranteeing property (26). Now, why is this theorem interesting? Mainly because it replaces condition (26), which involves infinite runs, by condition (i) of isochrony, which only involves a single reaction for the considered pair of STS.

2. Comment 1 for endochrony also applies here.

Proof. We successively prove points 1 and 2.

1. *Isochrony implies Property* (26). We proceed into two steps.

1. The desynchronization of Φ, defined by (23), is denoted by Φ^a, and we denote by δ a run of Φ^a. For each $\delta \in \Sigma^a$, there is at least one corresponding synchronous run σ for Φ such that $\delta = \sigma^a$. Any such σ is clearly the synchronous composition of two unifiable runs σ_1 and σ_2 for Φ_1 and Φ_2, respectively. Hence associated asynchronous runs σ_1^a and σ_2^a are also unifiable, and their asynchronous composition $\sigma_1^a \wedge^a \sigma_2^a$ belongs to $\Sigma_1^a \wedge^a \Sigma_2^a$. Thus we always have the inclusion

$$\Phi_1^a \parallel^a \Phi_2^a \supseteq (\Phi_1 \parallel \Phi_2)^a, \tag{36}$$

which proves the first part of (26). So far we have used only the definition of desynchronization and asynchronous composition; isochrony has not yet been used.

2. To prove the opposite inclusion, we need to prove that, when moving from asynchronous to synchronous composition, the additional need for a reaction-per-reaction matching of unifiable runs will not result in rejecting pairs of runs that otherwise would be unifiable in the asynchronous sense. This is where condition (i) of isochrony enters the game.

Pick a pair (δ_1, δ_2) such that $\delta_1 \bowtie^a \delta_2$ (cf. (25)): they can be combined while performing the asynchronous composition $\Phi_1^a \parallel^a \Phi_2^a$ to form some δ (cf. (24)); this is denoted by $\delta_1 \wedge^a \delta_2 = \delta$. By definition of desynchronization (cf. Subsection 4.1), there exist a (synchronous) run σ_1 for Φ_1 and a (synchronous) run σ_2 for Φ_2 such that δ_i is obtained by desynchronizing σ_i, $i = 1, 2$ (as we do not assume endochrony at this point, run σ_i is not uniquely determined). Thus each run σ_i is a succession of states. Clearly, inserting finitely many silent states between successive states of σ_i would also provide valid candidates for recovering δ_i after desynchronization. We shall show, by induction over successive states, that

properly inserting such a silent state in the appropriate
component will provide two runs which are
unifiable in the synchronous sense. (37)

This will show that, from a pair (δ_1, δ_2) such that $\delta_1 \bowtie^a \delta_2$, we can reconstruct (at least) one pair (σ_1, σ_2) of runs for Φ_1 and Φ_2 that are unifiable in the synchronous sense, and thus will prove the alternative inclusion

$$\Phi_1^a \parallel^a \Phi_2^a \subseteq (\Phi_1 \parallel \Phi_2)^a. \tag{38}$$

From (36) and (38) we then deduce property (26). We prove (37) now, by induction over successive states.

We are given a pair (δ_1, δ_2) such that $\delta_1 \bowtie^a \delta_2$. Pick a σ_1 such that $\sigma_1^a = \delta_1$, and similarly for σ_2. For $s_1, s_2, ..., s_n$ a finite run, we say that another run $s'_1, s'_2, ..., s'_m$ is a *stretching* of $s_1, s_2, ..., s_n$, written

$$s'_1, s'_2, ..., s'_m = (s_1, s_2, ..., s_n)^{\uparrow} \tag{39}$$

if there is a strictly increasing subsequence $k_1, ..., k_n$ of $1, ..., m$ such that $s'_{k_j} = s_j$, $j = 1, ..., n$, and $s'_k = \perp$ for $k \neq k_1, ..., k_n$. Note that (39) implies $m \geqslant n$. Using notation (39) we introduce the following hypothesis, for use in our inductive reasoning: for $i = 1, 2$, run σ_i decomposes as

$$\sigma_i = \underbrace{s_{i,1}, s_{i,2}, ..., s_{i,n_i}}_{\text{initial segment of length } n_i}, \sigma_{i,n_i} \tag{40}$$

and there are stretchings such that

$$\begin{aligned} s'_{i,1}, s'_{i,2}, ..., s'_{i,n} &= (s_{i,1}, s_{i,2}, ..., s_{i,n_i})^{\uparrow} && \text{for} \quad i = 1, 2 \\ s'_{1,m} &\bowtie s'_{2,m} && \text{for} \quad m = 1, ..., n. \end{aligned} \tag{41}$$

Note that (41) implies $\sigma^a_{1,n_1} \bowtie^a \sigma^a_{2,n_2}$. Define index

$$\zeta(n) = \min\{n_1, n_2\},$$

where n_i is defined in (40). To perform the proof by induction, we need to extend (40) and (41) in such a way that index $\zeta(n)$ grows to infinity.

To this end, decompose the tail σ_{i,n_i} into

$$\sigma_{i,n_i} = s_{i,n_i+1}, \sigma_{i,n_i+1}.$$

The following cases can occur:

Case 1. None of the two states s_{1,n_1+1} and s_{2,n_2+1} is silent over the common W variables. Concentrate one those $v \in W$ variables that are present in both states s_{1,n_1+1} and s_{2,n_2+1}. As $\delta_1 \bowtie^a \delta_2$ holds, then we must have $s_{1,n_1+1}[v] = s_{2,n_2+1}[v]$ for any such v. Thus points 1, 2, and 3 of condition (i) of isochrony are satisfied. Hence s_{1,n_1+1} and s_{2,n_2+1} are indeed unifiable in this case, *by isochrony*. Therefore, in this case, hypothesis (40), (41) extends in such a way that $\zeta(n+1) = \min\{n_1+1, n_2+1\} = \zeta(n)+1$ holds.

Case 2. Both states s_{1,n_1+1} and s_{2,n_2+1} are silent over the common W variables. They are unifiable. Again, hypothesis (40), (41) extends in such a way that $\zeta(n+1) = \zeta(n)+1$ holds.

Case 3. One and only one of the two states s_{1,n_1+1} and s_{1,n_1+1} is silent over the common W variables, say $\forall v \in W: s_{1,n_1+1}[v] = \perp$. In this case we unify state s_{1,n_1+1} with the silent state $\perp$ for Φ_2. Thus the matching hypothesis (41) is extended as

$$\begin{aligned} s'_{1,1}, s'_{1,2}, ..., s'_{1,n}, s'_{1,n+1} &= (s_{1,1}, s_{1,2}, ..., s_{1,n_1}, s_{1,n_1+1})^{\uparrow} \\ s'_{2,1}, s'_{2,2}, ..., s'_{2,n}, \underbrace{\perp}_{s'_{2,n+1}} &= (s_{2,1}, s_{2,2}, ..., s_{2,n_2})^{\uparrow} \end{aligned}$$

$$s'_{1,m} \bowtie s'_{2,m} \qquad \text{for} \quad m = 1, ..., n+1. \tag{42}$$

Therefore $\zeta(n+1) = \min\{n_1+1, n_2\}$ and we cannot infer that $\zeta(n+1) > \zeta(n)$ holds in this case.

Given the analysis above, we only need to show that

Case 3 cannot occur for infinitely many successive induction steps (43)

Assume that (43) does not hold. Then this implies that the whole tail σ_{1,n_1} is silent over the common W variables, while σ_{2,n_2} is not. But on the other hand we should have $\sigma^a_{1,n_1} \bowtie^a \sigma^a_{2,n_2}$, see (41), whence a contradiction. This finishes the induction proof; hence (38) follows.

2. *Under Endochrony of the Components, Property* (26) *Implies Isochrony.* This is easy. From Theorem 1 we know that, in our argument for proving point 1 of Theorem 2, the synchronous runs σ_i are uniquely defined, up to silent states, from their desynchronized respective versions σ^a_i. Now, focus on Case 1 of this argument. If isochrony is not satisfied, then, for some pair $\sigma^a_1 \bowtie^a \sigma^a_2$ of unifiable asynchronous runs, and some decomposition (40) of them, it follows that points 1, 2, and 3 of condition (i) of isochrony are satisfied, but states $s_{1,n+1}$ and $s_{2,n+1}$ are *not* unifiable. As our only possibility is to try to insert silent states for one of the two components—not feasible in Case 1—our process of incremental unification on a per reaction basis fails. Thus (38) is violated, and so is property (26). This finishes the proof of the theorem. ▮

The following result is instrumental in proving compositionality of isochrony.

Lemma 1. *If pairs* (Ψ, Φ_1) *and* (Ψ, Φ_2) *are isochronous, then so is pair* $(\Psi, \Phi_1 \parallel \Phi_2)$.

Proof. Let (s^-, s) and (t^-, t) be pairs of successive states, for Ψ and $\Phi_1 \parallel \Phi_2$, respectively, satisfying condition (i) for isochrony; see Definition 2. Let t be the unification of the two states s_1 and s_2 for Φ_1 and Φ_2, respectively. By point 2 of (i), at least one of these two states is not silent; assume that s_1 is not silent. From point 3 of (i), s and s_1 coincide over the set of *present* common variables, and thus, since pair (Ψ, Φ_1) is isochronous, states s and s_1 coincide over the *whole* set of common variables for Ψ and Φ_1. Thus s and s_1 are unifiable. But, on the other hand, s_1 and s_2 are also unifiable since they are just restrictions of the same global state t for $\Phi_1 \parallel \Phi_2$. Thus states s and t are unifiable, and thus pair $(\Psi, \Phi_1 \parallel \Phi_2)$ is isochronous. This proves Lemma 1. ▮

An interesting immediate by-product is the extension of the results on desynchronization, to networks of communicating synchronous components:

Corollary 1 (Desynchronizing a Network of Components). *We are given a finite family* $(\Phi_k)_{k=1,\dots,K}$ *of* STS. *Assume that each pair* $(\Phi_k, \Phi_{k'})$ *is isochronous. Then*

1. *For each disjoint subsets I and J of set* $\{1, \dots, K\}$, *the pair*

$$(\parallel_{k \in I} \Phi_k, \parallel_{k' \in J} \Phi_{k'}) \tag{44}$$

is isochronous. Thus isochrony is compositional.

2. *Also, desynchronization extends to the network*

$$(\Phi_1 \parallel \cdots \parallel \Phi_K)^a = \Phi^a_1 \parallel^a \cdots \parallel^a \Phi^a_K. \tag{45}$$

Proof. 1. Property (44) follows from Lemma 1 via obvious induction on the cardinal of sets I, J.

2. The second statement is proved via induction on the cardinal of the number of components,

$$\begin{aligned}(\Phi_1 \parallel \cdots \parallel \Phi_K)^a &= ((\Phi_1 \parallel \cdots \parallel \Phi_{K-1}) \parallel \Phi_K)^a \\ &= (\Phi_1 \parallel \cdots \parallel \Phi_{K-1})^a \parallel^a \Phi_K^a,\end{aligned}$$

and the induction step follows from (44). ▮

The next corollary expresses that isochrony is a "local" property.

COROLLARY 2 (Locality of Isochrony). *Assume that the pair* (Φ_1, Φ_2) *is isochronous, and the pair* (Ψ_1, Ψ_2) *is such that* Ψ_1 *has no common variable with* $\Phi_2 \parallel \Psi_2$ *and* Ψ_2 *has no common variable with* $\Phi_1 \parallel \Psi_1$. *Then the pair* $(\Psi_1 \parallel \Phi_1, \Phi_2 \parallel \Psi_2)$ *is also isochronous.*

Proof. This follows directly from Lemma 1. ▮

This is a useful result; it says that, in order for a pair $(\parallel_{k \in I} \Phi_k, \parallel_{k' \in J} \Phi_{k'})$ to be isochronous, it is enough to check isochrony for pairs $(\Phi_k, \Phi_{k'})$ of *interacting components.*

Note however that, in order for a pair $(\Psi_1 \parallel \Phi_1, \Phi_2 \parallel \Psi_2)$ to be isochronous, it is not necessary, but only sufficient, that the pair (Φ_1, Φ_2) be isochronous.

(*Counter*)*examples.*

Examples.

- A single-clocked communication between two STS.
- The pair $(\tilde{\Phi}_1, \tilde{\Phi}_2)$ of formula (35).

Counterexample. Assume that an STS communicates with another one according to the synchronous protocol "**await** $x \parallel$ **await** y"; the resulting pair of STS is not isochronous.

4.4. Getting GALS Architectures

In practice, only partial desynchronization of networks of communicating STS may be considered. This means that we really want to have *locally synchronous* components communicating via a *globally asynchronous* communication medium—this is referred to as GALS architectures.

In fact, Theorems 1 and 2 provide the adequate solution. Let us assume that we have a finite collection Φ_i of STS such that:

1. each Φ_i is endochronous, and
2. each pair (Φ_i, Φ_j) is isochronous.

Then, from Corollary 1 and Theorem 1, we know that

$$(\Phi_1 \parallel \cdots \parallel \Phi_K)^a = \Phi_1^a \parallel^a \cdots \parallel^a \Phi_K^a$$

and each Φ_k^a is in one-to-one correspondence with its synchronous counterpart Φ_k. Here is the resulting running mode for this GALS architecture:

- For communications involving a pair (Φ_i, Φ_j) of STS, each flow is preserved individually, but global synchronization is lost.
- Each STST Φ_i reconstructs its own successive reactions by just observing its (desynchronized) environment, and then locally behaves as a synchronous STS.
- Note that it is allowed, for each Φ_i, to have an internal activation clock which is *faster* than communication clocks. Resulting local activation clocks evolve asynchronously from one another.

4.5. Handling Endo/Isochrony in Practice

While we have given criteria for endochrony and isochrony, we did not propose a practical algorithm for checking these criteria. We do this now. Our aim is to prepare for GALS architectures such as discussed in Subsection 4.4. In particular, throughout this subsection, a network of STS satisfying conditions 1 and 2 of Subsection 4.4 will be called *endo/isochronous.*

In this subsection, we shall indicate (1) how a (tight) sufficient condition for endo/isochrony can be actually tested, and (2) how making an STS endo/isochronous can be performed. As both the DC$_+$ format and the SIGNAL language can be considered concrete instances of our STS model, we shall rely for our explanation on tools and algorithms already developed in these environments.

4.5.1. Checking Endo/Isochrony

As one of the modules of the existing DC$_+$ or SIGNAL compiler, the data structure shown in Fig. 3 is computed, for a given program P: In this figure, b, c denote Boolean variables, and $[b], [c]$ denote clocks composed of the instants at which $b, c = \text{T}$ holds, respectively. Finally, h, k are also clocks. The down arrows $h_0 \rightarrow b_1$, $[b_1] \rightarrow b_2$, $[b_2] \rightarrow b_3$, etc., indicate that Boolean variable b_1 has a clock equal to h_0 and only needs variables with clock h_0 for its evaluation, and so on. Roots of the trees are related by clock equations, depicted, for instance, by the bidirectional arrow relating h_0 and k_0. This defines a tree under each clock $h_0, k_0, ...$, and yields the so-called *clock hierarchy* in the form of a "forest," i.e., a collection of trees related by clock equations. This structure is detailed in Amagbegnon *et al.* (1994, 1995), where it is shown to be a canonical representation of the combination of clock equations and scheduling specifications of a program. Now, considering this clock hierarchy, one easily proves the following:

THEOREM 3. *Assume that program P has a clock hierarchy consisting of a* single *tree. Also assume that it is decomposed as $P = P_1 \parallel \cdots \parallel P_K$, and, for each k, the*

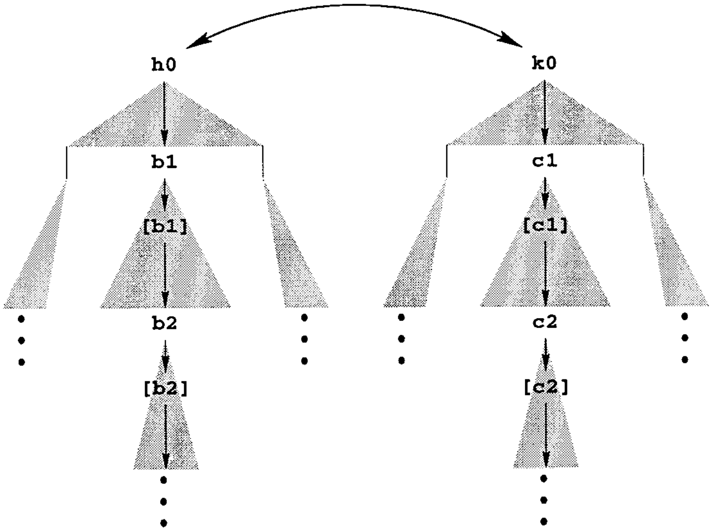


FIG. 3. The clock hierarchy computed by the DC_+ or Signal compiler.

clock hierarchy of component P_k is a subtree of the clock tree of P. Then the corresponding network of STS *is endo/isochronous.*

Theorem 3 is an immediate corollary of Theorem 1 of Section 4; it only states a sufficient condition. In computing a clock hierarchy, the abstractions performed are twofold: (1) inferring dependencies from causality analysis, and (2) abstracting Boolean variables which result from the evaluation of a predicate involving a non-Boolean expression. In practice, we shall use the clock hierarchy as the practical criterion for checking endo/isochrony.

4.5.2. Enforcing Endo/Isochrony

Assume that we have an STS P having a clock hierarchy which is not a tree, and we still *want* it to be a tree. What can we do? As revealed by inspecting the previous figure, it is sufficient to make the roots $h_0, k_0, ...$ of the clock hierarchy belonging to some single clock tree. In other words, we can concentrate on the roots of the clock hierarchy. Thus the problem can be restated as follows:

We are given a set $h_1, ..., h_k$ of clocks, which are related by a set of clock equations of the form

$$\begin{array}{c} p_1(h_1, ..., h_k) \neq \mathrm{F} \\ \cdots \\ p_q(h_1, ..., h_k) \neq \mathrm{F}. \end{array} \tag{46}$$

This corresponds to having a collection $p_1, ..., p_q$ of predicates on clocks, which are Boolean-valued expressions that are either true or absent. Note that being always true is the case for predicates in classical Boolean logic, while in our case, due to the requirement for stuttering robustness, we must accept the possibility for a "clock predicate" to be absent. Systems of equations of the form (46) can be solved for their variables $h_1, ..., h_k$, meaning that we can find a set $h_1^0, ..., h_l^0$ of clocks, and a set $p_1^0, ..., p_k^0$ of clock expressions, such that equation system

$$\begin{aligned} h_1 &= p_1^o(h_1^o, ..., h_l^o) \\ &\cdots \\ h_k &= p_k^o(h_1^o, ..., h_l^o) \end{aligned} \tag{47}$$

has the same set of solutions for $h_1, ..., h_k$ as the original system (46), and new clocks $h_1^0, ..., h_l^0$ are free, i.e., unconstrained by the system of equations (47). Finally, we introduce Boolean variables $b_1^0, ..., b_l^0$, and a "master clock" h^0, such that

$$\begin{aligned} h_1^o = [b_1^o], ..., h_l^o = [b_l^o] \\ h_{b_1^o} = \cdots = h_{b_l^o} = h. \end{aligned} \tag{48}$$

The bottom line is:

1. System of clock equations (46) is equivalent to (47), (48) after hiding auxiliary variables $h, b_1^0, ..., b_l^0$.
2. System (47), (48) is a clock tree.

Discussion. Basically, building (48), (47) from (46) intuitively corresponds to equipping the original P program with a suitable *communication protocol* Q in such a way that the compound program $P \parallel Q$ is endo/isochronous. This is not surprising indeed, for it is known in the area of distributed systems that components in a distributed system must be equipped with suitable protocols for their communications.

Finally, the way we moved from (46) to (47) reveals one unpleasant feature of this technique, namely, that this part of the process is not unique, and thus there are possibly many different correct protocols.

5. FORMAL STUDY OF CAUSALITY

In this section we develop a formal theory of causality for STS. Our basic tool is that of scheduling specifications and labelled preorders. We first formalize this, by adding the value *unknown* to our domains, like in the Constructive Boolean logic used in Berry and Sentovich (1998). Using this extended domain, we are able to formally state and prove our criterion that circuit-freeness implies executability. Then we formalize Rules (R-1)–(R-4) of (21), and we finally show how correct deterministic execution results from a successful causality analysis.

5.1. Encoding Scheduling Specifications Using an Algebraic Domain

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

$$\mathscr{D} = \{\mathbf{?},\ \overbrace{\bot,\ \underbrace{\text{F},\ \text{T}}_{\top}}^{\boldsymbol{¿}}\} \tag{49}$$

$$\mathbf{?} \prec \bot, \text{F}, \text{T}, \qquad \bot > \text{F} > \text{T}. \tag{50}$$

In these formulas, the symbol **?** (resp. **¿**) indicates that the value is "unknown" (resp. "known"). The "unknown" status should not be confused with absence ($\bot$): absence is a perfectly known status, while "unknown" is intended to model hat a variable has not been produced yet in the current reaction. Non-Boolean types are abstracted as the single distinguished element $\top$; hence, for Booleans, the pair $\{\text{F}, \text{T}\}$ can be seen as a refinement of the symbol $\top$: this is shown by the under-brackets. And $\{\bot, \top\}$ is a refinement of **¿**; this is shown by the overbrackets. Ordering $<$ has already been introduced, and the additional partial order $\prec$ is the Scott information ordering: $\mathbf{?} \prec \bot, \text{F}, \text{T}$, the three values $\bot, \text{F}, \text{T}$ being incomparable with respect to $\prec$.

Definition. 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, whenever $b = \text{T}$, that y becomes know while x is not.

Properties of Scheduling Specifications. The following properties hold:

$$\text{if } b, c \neq \mathbf{?}, \text{ then} \quad \begin{array}{lcl} x \xrightarrow{b} y \wedge y \xrightarrow{c} z & \Rightarrow & x \xrightarrow{b \wedge c} z \\ x \xrightarrow{b} y \wedge y \xrightarrow{c} z & \Rightarrow & x \xrightarrow{b \vee c} y. \end{array} \tag{51}$$

TABLE 1

Definition of the Dependency $x \xrightarrow{b} y$

b \ x	**?**	$\bot$	$\bot$
?	**?**	**?**	**?**
$\bot$			
F			
T	**?**		

Note. 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 $\{\text{F}, \text{T}\}$.

In these equations, $b \wedge c$ and $b \vee c$ are respectively defined as the infimum (resp. supremum) w.r.t. relation "$<$" defined in (50) when both values belong to the subdomain $\{\perp, \mathrm{T}, \mathrm{F}\}$. In fact, we do not need formulas (51) in case b or c is unknown, because the label of a branch is known prior to its extremity, in executable programs equipped with their scheduling specifications as inferred from rules (R-1)–(R-4).

5.2. Circuit-Free Schedulings

We are given a set of variables $x_1, ..., x_n$. Some of them are Boolean; for the sake of readability, Boolean variables used as labels in scheduling specifications will be generically denoted by $b_1, b_2, ...$ Then we are given (1) a set of constraints of the form $C(b_1, ..., b_k)$ on boolean variables restricted to subdomain $\{\perp, \mathrm{T}, \mathrm{F}\}$ of known values, and (2) a set of scheduling specifications defined on $x_1, ..., x_n$. Constraints $C(b_1, ..., b_k)$ are extended to the "unknown" value by simply assuming that $G(b_1, ..., b_k)$ is satisfied as soon as at least one of the variables $b_1, ..., b_k$ is "unknown."

Each dependency is interpreted as specified in Table 1. Thus, together with the Boolean constraints of the form $C(b_1, ..., b_k)$, they specify a subdomain of the product domain $\mathscr{D}^n$ of all possible states. The set of states satisfying these constrains is denoted by $\mathscr{S}$, and we call it a *scheduling* of $x_1, ..., x_n$. States in $\mathscr{S}$ are written $s, t, ...$ and corresponding interpretations are denoted by $s_1, ..., s_n$ for short instead of $s[x_1], ..., s[x_n]$, and similarly for t. The "totally unknown state,"

$$\forall i,\ s_i = \mathbf{?}, \tag{52}$$

is denoted by $s_{\mathbf{?}}$.

Two states of $\mathscr{S}$ are said to be *neighbours* if they differ in exactly one variable; we call it their *discriminating* variable. We call a *path* in $\mathscr{S}$ any finite sequence $s(1), s(2), ..., s(K)$ of neighbouring states belonging to $\mathscr{S}$.

For s and t two neighbouring states of $\mathscr{S}$, we write $s \prec t$ if their respective values for their discriminating variable x_i satisfy the relation $s_i \prec t_i$ defined in (49). A path $s(1), s(2), ..., s(K)$ such that $s(k) \prec s(k+1)$ is called *increasing*.

A scheduling $\mathscr{S}$ is called *circuit-free* if *it is never true in* $\mathscr{S}$ *that*

$$\begin{gathered} x_{i_1} \xrightarrow{b_1} x_{i_2} \xrightarrow{b_2} x_{i_3} \cdots x_{i_p} \xrightarrow{b_p} x_{i_1} \\ \textbf{and} \\ (b_1 \wedge \cdots \wedge b_p = \mathrm{T}). \end{gathered} \tag{53}$$

THEOREM 4 (Circuit-Free Schedulings). *A scheduling is* circuit-free *iff, for every state* $s \in \mathscr{S}$ *satisfying* $\forall i$: $s_i \neq \mathbf{?}$, *there is an* increasing path *linking* $s_{\mathbf{?}}$ *to* s.

The intuitive interpretation of this theorem is that, for an STS with a circuit-free scheduling, it is possible to compute sequentially without deadlock all variables, starting from the inputs. Each increasing path mentioned in Theorem 4 corresponds to one possible sequential execution.

Proof. We first prove the "if" part by contradiction. Assume that (53) is violated for some circuit $x_{i_1} \xrightarrow{b_1} x_{i_2} \xrightarrow{b_2} x_{i_3} \cdots x_{i_p} \xrightarrow{b_p} x_{i_1}$; i.e., $b_1 \wedge \cdots \wedge b_p = \mathrm{T}$ is possible for this circuit in $\mathscr{S}$. We want to deduce from this assumption that there are states for which all variables are known, but there is no increasing path originating from $s_?$ and terminating at the states in consideration. Without loss of generality, we can restrict $\mathscr{S}$ to those states for which

$$\begin{gathered}\forall i = 1, ..., p : [b_i = \textbf{?} \ \textbf{or} \ b_i = \text{T}] \text{ holds;} \\ \text{the set of such states is called } \mathscr{S}_{(b_1 \wedge \cdots \wedge b_p = \text{T})}.\end{gathered} \tag{54}$$

By Table 1, condition $x_{i_1} \xrightarrow{b_1} x_{i_2} \xrightarrow{b_2} x_{i_3} \cdots x_{i_p} \xrightarrow{b_p} x_{i_1}$ implies that, on $\mathscr{S}_{(b_1 \wedge \cdots \wedge b_p = \text{T})}$, the following holds,

$$x_{i_1} \succcurlyeq x_{i_2} \succcurlyeq \cdots \succcurlyeq x_{i_p} \succcurlyeq x_{i_1},$$

and thus the x_{i_j}'s are either all unknown, or alternatively all known. Thus there is no increasing path originating from $s_?$ and leading to any known state belonging to $\mathscr{S}_{(b_1 \wedge \cdots \wedge b_p = \text{T})}$. This proves the "if" part.

Next, we prove the "only if" part, also by contradiction. Beforehand, we need a lemma. Two states s and s' are said to be *complementary* if, for each variable x,

$$\text{either} \quad s[x] = \textbf{?} \qquad \text{or} \quad s'[x] = \textbf{?}.$$

Two states s and s' are said *compatible* if, for each variable x,

$$\text{either} \quad s[x] = \textbf{?} \qquad \text{or} \quad s'[x] = \textbf{?} \quad \text{or} \quad s'[x] = s[x].$$

Complementary states are also compatible. For two compatible states s and s', we define their sum $s \uplus s'$ by

$$(s \uplus s')[x] \quad = \quad \textbf{if } s[x] \neq \textbf{?} \quad \textbf{then} \quad s[x] \quad \textbf{else } s'[x].$$

LEMMA 2 (Monotonicity). *Let t_0 and t_1 be two neighbouring states belonging to $\mathscr{S}$, such that $t_0 \prec t_1$. Let t be a state such that*

1. *t_1 and t are complementary,*
2. *$t_0 \uplus t \in \mathscr{S}$,*
3. *there is an increasing path contained in $\mathscr{S}$ originating from t_0 and terminating in $t_0 \uplus t$, and*
4. *$t_1 \uplus t$ satisfies the Boolean constraints $C(b_1, ..., b_k)$ which contribute to the definition of $\mathscr{S}$.*

Then $t_1 \uplus t \in \mathscr{S}$ and there is an increasing path contained in $\mathscr{S}$ originating from t_1 and terminating in $t_1 \uplus t$.

Proof. Note that $t_0 \uplus t$ is well defined, since t_0 and t are also complementary. Let $t_0 \to t_0 \uplus t$ denote the path referred to in item 3. Denote by $\tilde{t}$ the state such that (1) $\tilde{t}$ and t_0 are complementary and (2) $t_1 = t_0 \uplus \tilde{t}$; such a state exists and is unique. Denote by $t_0 \uplus \tilde{t} \to t_0 \uplus t \uplus \tilde{t}$ the increasing path obtained by complementing each state belonging to path $t_0 \to t_0 \uplus t$ by $\tilde{t}$. This is possible since each intermediate state of path $t_0 \to t_0 \uplus t$ and $\tilde{t}$ are complementary. We claim that

$$\text{path } t_0 \uplus \tilde{t} \to t_0 \uplus t \uplus \tilde{t} \text{ is contained in } \mathcal{S}. \tag{55}$$

Clearly, claim (55) is equivalent to the conclusion of the lemma. To prove (55), using item 4, we first note that each state belonging to path $t_0 \uplus \tilde{t} \to t_0 \uplus t \uplus \tilde{t}$ satisfies the Boolean constraints $C(b_1, ..., b_k)$ which contribute to the definition of $\mathcal{S}$. We thus only need to check that they also satisfy the dependencies contributing to the definition of $\mathcal{S}$, but the latter results from an inspection of Table 1. This proves the lemma. ▮

We now return to the proof of Theorem 4 and proceed by steps.

1. Assume $\exists s^* \in \mathcal{S}$ satisfying $\forall i$: $s_i^* \neq \mathbf{?}$, such that there is no increasing path linking $s_?$ to s^*. Denote by $b_1, ..., b_p$ the Boolean variables such hat $b_1 \wedge \cdots \wedge b_p = \mathrm{T}$ holds at state s^*. Denote by S the set of states $s \in \mathcal{S}$ such that $s \preccurlyeq s^*$. We have $s_? \in S$ and $s^* \in S$. States belonging to S are all compatible.

2. Let $s, s' \in S$ be two states such that increasing paths $s_? \to s$ and $s_? \to s'$ are both contained in S. Then we claim that

$$\begin{aligned} s'' = s \uplus s' \in S, \text{ and there exists an increasing path contained in } S,\\ \text{originating from } s_?,\\ \text{and terminating in } s''. \end{aligned} \tag{56}$$

As all $s \in \mathcal{S}$ satisfy $s \preccurlyeq s^*$, they satisfy in particular the Boolean constraints $b_1 \wedge \cdots \wedge b_p = \mathrm{T}$. Thus we only need to verify the dependencies. There is a unique state $s_0 \in S$ such that (1) $s_0 \in [s_? \to s] \cap [s_? \to s']$, and (2) $[s_0 \to s] \cap [s_0 \to s'] = \{s_0\}$, meaning that s_0 is the latest point at which the two considered paths deviate from each other. Let s_1 be the neighbour state of s_0 belonging to path $[s_0 \to s']$. Apply Lemma 2 with the following substitutions: $t_0/s_0, t_1/s_1, t/\tilde{s}$ such that $s = s_0 \uplus \tilde{s}$. We deduce that path $[s_? \to s \uplus s_1] \subseteq S$. Then, let s_2 be the neighbour state of s_1 belonging to path $[s_1 \to s']$; we can repeat the same argument. And we proceed repeatedly in the same way until we prove the claim (56).

3. Consider the set of $s \in S$ for which there exists an increasing path $[s_? \to s] \subseteq S$. From (56) we know that this set has a unique maximal element $s_{\max}$ for partial order $\prec$. By hypothesis we have $s_{\max} \prec s^*$, $s_{\max} \neq s^*$. Thus there are at least two variables, denote them by x and x', such that $s_{\max}[x] = s_{\max}[x'] = \mathbf{?}$, but $s[x] = s[x'] \neq \mathbf{?}$ for every $s \in S \backslash [s_? \to s_{\max}]$. Hence, the following holds at each state belonging to S:

$$x \xrightarrow{b} x' \xrightarrow{b} x, \qquad \text{where} \quad b = b_1 \wedge \cdots \wedge b_p = \mathrm{T}.$$

Hence the condition of circuit-freeness is violated on S, and thus it can be violated on $\mathcal{S}$. This finishes the proof of Theorem 4. ▮

In the following, for Φ and STS with scheduling specifications, we shall consider its associated scheduling

$$\mathscr{S}_{\Phi} \tag{57}$$

which is obtained by keeping, from the set of predicates defining the transition relation of Φ,

1. the scheduling specifications, and

2. the assertions involving only Boolean variables and clocks, and discarding the other ones.

5.3. Deriving Scheduling Specifications as Causality Constraints

In this subsection, we formally justify rules (21). The principles we follow for our abstraction mechanism are given next:

(P-1) For x not a Boolean variable, we abstract its domain $\mathscr{D}_x$ as the singleton $\{\top\}$, and then extend $\{\top\}$ with 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 $\{\perp, \text{F}, \text{T}\}$ to the single value ¿ (known). Note the asymmetry of this abstraction principle: for the statement "**if** b **then** $y = x$," where x, y are Booleans, we abstract y but not x.

(P-3) Since we are interested in causality constraints, we only need to keep track of configurations for which y cannot be known; i.e., $y = \mathbf{?}$ is the only allowed possibility. For other configurations, we weaken the constraint on y to "y unconstrained," which is depicted in the tables by an empty box.

We now proceed in deriving the scheduling associated to each primitive statement, using **(P-1)–(P-3)**. We use the notation $\mathbf{?}, \perp$ to indicate that, for the considered configuration, either $y = \mathbf{?}$ or $y = \perp$ holds, and similarly for other cases.

LEMMA 3. *The following holds*:

$$x \overset{b}{\longrightarrow} y \;\Rightarrow\; b \longrightarrow h_y .$$

Proof. By inspection of Table 1.

LEMMA 4. *The following holds*:

$$h_x \rightarrow x.$$

Proof. By inspection of the following tables (the first table relates x to h_x, as extended to unknown values):

h_x	$\mathbf{?}$	$\perp$	T
x	$\mathbf{?}$	$\mathbf{?}, \perp$	$\mathbf{?}, \top$

abstracted as (using (P-2)) :

h_x	?	⊥	T
x	?	?, ¿	?, ¿

which is equal to :

h_x	?	⊥	T
x	?		

which turns out to be equivalent to $h_x \to x$ by Table 1.

Lemma 5. *The following holds*:

$$(f):\begin{cases} y=f(u,v) \\ h_u=h_v=h_y \end{cases} \quad \Rightarrow \quad (u,v)\xrightarrow{h_y} y.$$

Proof. By inspection of Table 1 and of the following tables (# denotes a prohibited value):

abstraction of (f), using (P-1) :

u v	?	⊥	⊤
?	?	?, ⊥	?
⊥	?, ⊥	⊥	#
⊤	?	#	⊤

using (P-2) :

u v	?	⊥	⊤
?	?	?, ¿	?
⊥	?, ¿	¿	#
⊤	?	#	¿

using (P-3) :

u v	?	⊥	⊤
?	?		?
⊥			
⊤	?		

which is equivalent to the formulas of the conclusion of the rule 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. These tables define the possible values, of x and h_x, respectively, for $[\textbf{if } b \textbf{ then } x = u] \wedge [\textbf{if } b \textbf{ then } h_x = h_u]$:

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

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

Applying Principles **(P-2)** and **(P-3)** then yields the formulas corresponding to the conclusion of the rule of Lemma 6. 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.

5.4. Correct Programs

In this subsection, we formally state and prove the result establishing the link between circuit-freeness and executable STS.

Theorem 5 (Correct Programs). *Let* $\mathtt{P}$ *be an* STS *satisfying the following conditions*:

1. *For each statement of* $\mathtt{P}$, *the scheduling specifications derived from applying the rules of Lemmas* 3, 4, 5, *and* 6 *are also statements of* $\mathtt{P}$.
2. *The scheduling* $\mathscr{S}_{\mathtt{P}}$ (*cf.* (57)) *defined by* $\mathtt{P}$ *is circuit-free.*
3. *There is no multiple definition of a variable*, *meaning that*, *whenever*

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

is part of $\mathtt{P}$, *then*

$$b_1 \wedge b_2 = \mathrm{T} \textit{ never holds}$$

Then:

1. *As far as* control *is concerned, the inputs of* P *are the source nodes of the dependency graph.*

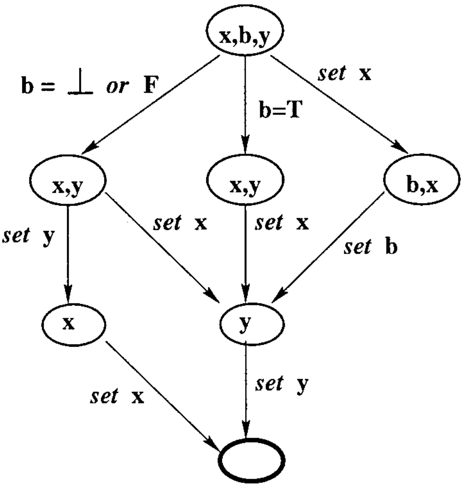
2. *Input* values *are those variables which never occur on the left-hand side of statements of the form* "$x = \text{exp}$."

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.

Note. Clearly, Theorem 5 provides us with a sufficient condition; this condition is not necessary. Furthermore, the rules for inferring scheduling specifications 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 5 for a given syntactic form of P, it may satisfy them after a proper rewriting into a semantically equivalent form. Here, semantic equivalence means identical runs when scheduling specifications are discarded.

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 $\mathscr{D}_x \cup \{\perp\}$. 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. Conversely, any correct evaluation scheme for triple (x, b, y) 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 constrains 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 Theorem 4 we know that, for each tuple of variables which satisfies the specification, there is a path of the product automaton which originates from its initial state and terminates at the final state in which all variables are set, meaning that all variables of the considered tuple are sequentially set.

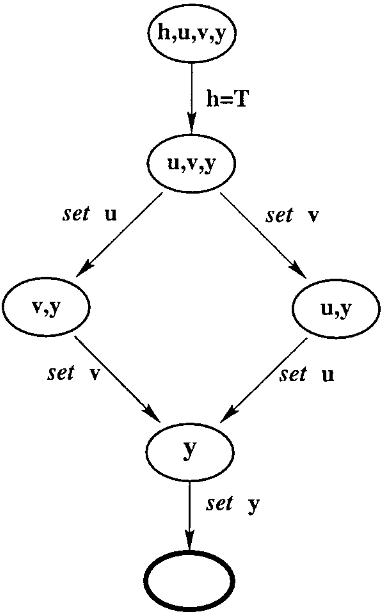
4. Finally, we refine the transition labels of the form "set x," etc., by assigning to x, etc., the value specified by the program. As source nodes of the dependency graph are set first, they appear as inputs of P for its control. Also, variables u that are set and do not occur on the left-hand side of any statement $u = expression$ must be read from the environment: their values are inputs of the considered program P. Finally, thanks to condition 3 of Theorem 5, actions of the form "set x," etc., are refined into single writings. This finishes the proof of the theorem. ▮

We illustrate this technique on the following simple STS:

$$y = f(u, v) \wedge h_u = h_v = h_y =_{\text{def}} h.$$

The causality constraint and associated execution automaton are

$$\begin{aligned} & h \longrightarrow (u, v, y) \\ \wedge\ & (u, v) \xrightarrow{h} y \\ \wedge\ & h_u = h_v = h_y =_{\text{def}} h \end{aligned}$$



Clock h is the activation clock. The refined execution automaton is obtained by replacing *set* u and *set* v by *read* u and *read* v, and *set* y by the assignment $y := f(u, v)$.

6. CONCLUSION

Our contribution can be summarized as follows:

- We have proposed STS with scheduling specifications as a paradigm for causality analysis, STS abstraction, separate compilation, and reuse.
- We have characterized those STS for which asynchronous and synchronous semantics are equivalent in some precise meaning.

We advocate system design methodology based on the synchronous paradigm, possibly followed by a provably correct desynchronization. Advantages of this approach are numerous, and they are listed below according to the different phases of the design:

Specification. Designing within the synchronous paradigm allows the designer to exploit the simplicity and elegance of compositionality of synchronous specifications. In addition, specification can be performed independently from the execution architecture; therefore, upgrading an execution architecture does not require redesigning the specifications.

Verification.

- In the synchronous paradigm, composition of specifications and composition of properties are both performed by using the composition "$\|$" of STS. This facilitates reasoning in general, and in particular compositional reasoning.
- For endo/isochronous STS, proofs based on the synchronous semantics carry over without modifications to asynchrony. For such systems, verifications can be performed within the synchronous framework. This allows one to avoid state explosion resulting from the use of the asynchronous interleaving semantics.

Abstraction, Modularity, and Reuse.

- Scheduling specifications provide the adequate notion of abstraction for separate compilation. It allows the designer to check the correctness of component encapsulation at the systems integration phase.
- STS with scheduling specifications can be composed using a proper generalization of the composition "$\|$" of STS. Thus advantages of compositionality naturally extend to STS with scheduling specifications.
- The structuration of specifications into scheduler and tasks allows us to define proper reusable modules. Of course, if assumptions are available on the possible behaviours of the environment, then larger modules can be stored as object code for further reuse.

GALS *Network.* The elegant feature is that isochrony is a *local* property within a network of components. As isochrony is compositional, adding a new component

Φ_{new} to a preexisting GALS network $(\Phi_i)_{i=1,\ldots,n}$ while preserving its GALS nature only requires one to check whether pairs $(\Phi_{\text{new}}, \Phi_i)$ are isochronous, for each Φ_i having direct communication with Φ_{new} in the extended network. Thus GALS designs can be built *compositionally*, and it is not necessary to desynchronize at once the whole synchronous design.

Thanks to the outcomes of the SACRES project, the above approach is supported by the SIGNAL-V4 language,[11] and by the Dc_+ common format for synchronous languages (Sacres Consortium, 1996). (SIGNAL-V4 and the DC_+ format are both concrete implementations of our STS model. This includes scheduling specifications, which are available as primitive statements in both formalisms.

In particular, the 1999 release of SILDEX (TNI, 1999)[12] implements distributed code generation based on the approach presented in this paper. The target architectures above all else are POSIX-compliant real-time OS.

The new SIGNAL-V4 compiler developed at Inria implements the whole methodology, including separate compilation. Services for architecture generation are also provided, using our notion of abstraction.

Research perspectives. Further work is needed to show that the above principles are viable for generating architectures built up from preexisting C/C++/*Java*/... modules. Then, not all communication media or operating systems provide services satisfying the requirements of our theory of desynchronization, namely, no loss of messages and first-in/first-out semantics for each individual channel. Additional work is needed for getting a full implementation on each different type of distributed architecture; this can be very easy (writing a few generic drivers, e.g., for POSIX), or can be more demanding when adequate services are not provided by the architecture, and thus need to be emulated.

ACKNOWLEDGMENTS

The authors are gratefully indebted to Michael Siegel for a thorough reading and detailed comments, and in particular the discovery of several inconsistencies in an earlier version of this manuscript in the formal study of causality.

Received November 28, 1997; final manuscript received August 31, 1999; published online October 20, 2000

REFERENCES

Aabelsberg, I. J., and Rozenberg, G. (1988), Theory of traces, *Theoret. Comput. Sci.* **60**, 1–82.

Alur, R., and Henzinger, T. A. (1996), Reactive modules, *in* "Proceedings 11th IEE Symposium on Logic in Computer Science 9LICS," pp. 207–218, extended version submitted for publication.

[11] Logic Besnard and other members of the "EpAtr" team at IRISA are gratefully acknowledged for the development of this environment.

[12] The SILDEX tool is a commercial tool for reactive systems design based on the SIGNAL language. It is marketed by TNI, Brest, France.

Amagbegnon, T. P., Besnard, L., and Le Guernic, P. (June 1994), "Arborescent Canonical Form of Boolean Expressions," Inria Research Report, 2290.

Amagbegnon, T. P., Besnard, L., and Le Guernic, P. (1995), Implementation of the dataflow language Signal, *in* "Programming Languages Design and Implementation," pp. 163–173, Assoc. Comput. Mach., New York.

Aubry, P. (1997), "Mises en œuvre distribuées de programmes synchrones," Ph.D. thesis, Univ. Rennes I.

Benveniste, A., and Le Guernic, P. (1990), Hybrid dynamical systems theory and the Signal language, *IEEE Trans. Automat. Control* **35**, No. 5, 535–546.

Benveniste, A., and Berry, G. (1991), Real-time systems design and programming, *in* "Another Look at Real-Time Programming," special section of "Proceedings of the IEEE", Vol. 9, pp. 1270–1282, IEEE, New York.

Benveniste, A., Le Guernic, P., and Jacquemot, C. (1991), Synchronous programming with events and relations: The SIGNAL languages and its semantics, *Sci. Comput. Programming* **16**, 103–149.

Benveniste, A., Le Guernic, P., Sorel, Y., and Sorine, M. (1992), A denotational theory of synchronous communicating systems, *Inform. and Comput.* **99**, 192–230.

Benveniste, A., Caspi, P., Halbwachs, N., and Le Guernic, P. (1994), Data-flow synchronous languages, *in* "A Decade of Concurrency, Reflexions and Perspectives, REX School/Symposium," Lecture Notes in Computer Science, Vol. 803, pp. 1–45, Springer-Verlag, Berlin/New York.

Berry, G. (1989), Real time programming: Special purpose or general purpose languages, *in* "IFIP World Computer Congress, San Francisco."

Berry, G. (Dec. 1995), "The Constructive Semantics of Esterel," Draft book, http://www.inria.fr/meije/esterel.

Berry, G., and Sentovich, E. M. (Nov. 1998), An implementation of constructive synchronous programs in polis, manuscript.

Caillaud, B., Caspi, P., Giraud, A., and Jard, C. (1997), Distributing automata for asynchronous networks of processors, *Eur. J. Automat. Systems* (*JESA*) **31**, No. 3, 503–524.

Caspi, P. (1992), Clocks in dataflow languages, *Theoret. Comput. Sci.* **94**, 125–140.

Clerbout, M., and Latteux, M. (1987), Semicommutations, *Inform. and Comput.* **73**, 59–74.

de Roever, W.-P., Langmaack, H., and Pnueli, A. (1998), Compositionality: The significant difference, *in* "Proceedings International Symposium COMPOS'97, Bad Malente, Germany," Lecture Notes in Computer Science, Vol. 1536, Springer-Verlag, Berlin/New York.

Le Guernic, P., and Gautier, T. (1991), Dataflow to von Neumann: The Signal approach, *in* "Advanced Topics in Dataflow Computing" (L. Biv, and J.-L. Gaudiot, Eds.), pp. 413–438, Prentice–Hall, New York.

Halbwachs, N. (1993), "Synchronous Programming of Reactive Systems," Kluwer Academic, Dordrecht/Norwell, MA.

Lamport, L. (1983), Specifying concurrent program modules, *ACM Trans. Programm. Languages Systems* **5**, No. 2, 190–222.

Lamport, L. (1983), What good is temporal logic?, *in* "Proceedings, IFIP 9th World Congress" (R. E. A. Mason, Eds.), pp. 657–668, North-Holland, Amsterdam.

Le Guernic, P., Gautier, T., Le Borgne, M., and Le Maire, C. (1991), Programming real-time applications with Signal, *in* "another Look at Real-Time Programming," special section of "Proceedings of the IEEE," Vol. 9, pp. 1321–1336, IEEE, New York.

Maffeis, O., and Le Guernic, P. (1994), Distributed implementation of Signal: Scheduling and graph clustering, *in* "3rd International School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems," Lecture Notes in Computer Science, Vol. 863, pp. 149–169, Springer-Verlag, Berlin/New York.

Manna, Z., and Pnueli, A. (1992), "The Temporal Logic of Reactive and Concurrent Systems: Specification," Springer-Verlag, New York.

Manna, Z., and Pnueli, A. (1995), "The Temporal Logic of Reactive and Concurrent Systems: Safety," Springer-Verlag, New York.

Sacres Consortium, (May 1996), The Declarative Code DC$_+$, Version 1.2; Esprit project EP 20897: Sacres, see http://www.tni.fr/sacres/

Sorel, Y., and Lavarenne, C. "SynDEx v4.2 User Guide"; http://www-rocq.inria.fr/syndex/.articles/doc/doc/SynDEx42.html.

Sorel, Y. (Sept. 1996), Sorel: Real-time embedded image processing applications using the A3 methodology, *in* "Proceedings IEEE International Conference on Image Processing, Lausanne."

TNI, (1999), SILDEX tool; see http://www.tni.fr/indexgb.htm.