Correct-by-construction asynchronous implementation of modular synchronous specifications

Dumitru Potop-Butucaru¹ Benoît Caillaud IRISA, campus universitaire de Beaulieu, 35042 Rennes cedex, France Email: {dumitru.potop,benoit.caillaud}@irisa.fr

Abstract

We introduce a model for the representation of asynchronous implementations of synchronous specifications. The model covers classical implementations, where a notion of global synchronization is preserved by means of signaling, and globally asynchronous, locally synchronous (GALS) implementations where the global clock is removed. Our model offers a unified framework for reasoning about two essential correctness properties of an implementation: the preservation of semantics and the absence of deadlocks.

1. Introduction

Dealing with concurrency, time and causality in the design of electronic systems has become increasingly difficult as the complexity of the designs grew. The synchronous programming model [1], [2], [3] has had major successes at the specification level because it provides a simpler way to employ the power of concurrency in functional specification. Synchronous languages like ESTEREL, LUSTRE, and SIGNAL, the quasi-synchronous STATECHARTS modeling methodology, and design environments like SIMULINK/STATEFLOW all benefit from the simplicity of the synchronous hypothesis, which (1) lets computations and behaviors be divided into a discrete sequence of computation steps, also called reactions, indexed by a global clock, and (2) requires that inside each reaction the behavioral propagation is "well-behaved" (causal), so that the status of each variable is established prior to being tested or used. This hypothesis founds the conceptual abstraction that computations are infinitely fast, and allows universallyrecognized mathematical models like the Mealy machines and the digital circuits to be used as semantic foundations.

However, if the synchronous assumption simplifies system specification and verification, the problem of deriving a correct physical implementation from it does remain [2]. In particular, difficulties arise when the target implementation architecture has a distributed nature that does not match the synchronous assumption because of large variance in computation and communication speeds and because of the difficulty of maintaining a global notion of time. This is increasingly the case in complex microprocessors and Systems-on-a-Chip (SoC), and for many important classes of embedded applications in avionics, industrial plants, and the automotive industry.

Gathering advantages of both the synchronous and asynchronous approaches, the Globally Asynchronous Locally Synchronous (GALS) architectures are emerging as an architecture of choice for implementing complex specifications in both hardware and software. In a GALS system, locallyclocked synchronous components are connected through asynchronous communication lines. Thus, unlike for a purely asynchronous design, the existing synchronous tools can be used for most of the development process, while the implementation can exploit the more efficient/unconstrained/required asynchronous communication schemes.

We further pursue in this paper our quest for correct-byconstruction deployment of synchronous specifications over GALS architectures.

Informal discussion of the issues. In the synchronous paradigm [1], [2], [3], an execution of the program, also called *trace*, is a sequence of reactions, each reaction assigning a unique value (status) to each variable of the program. Not all variables need to be involved in each reaction. However, this is taken into account by extending the domain of values of all variables with an extra symbol \bot , which denotes absence. Thus, absence can be tested and used to exercise control.

No global clock exists in the *asynchronous paradigm*, meaning that no notion of reaction exists, and that absence (\perp) has no meaning and cannot be sensed. Only the sequences of values on individual channels can be observed, so that an *asynchronous observation* of the execution of a system is a function assigning to each communication channel the sequence of transmitted messages/values. Asynchronously observing a synchronous execution consists of removing the \perp events and the synchronization boundaries of the reactions to obtain an asynchronous observation.

In many cases, applications designed in a synchronous framework will be implemented for use in an asynchronous environment. Two problems arise: First, the synchronous applications must be fitted with wrappers that read the asynchronous inputs and schedule them into reactions before giving them to the program and triggering the program clock (the

¹Supported by the Columbus European project, IST-2001-38314, and the ARTIST network of excellence IST-2001-34820. Dumitru Potop-Butucaru is now at VERIMAG, centre Equation, 3861 Gieres, France, Email: potop@imag.fr

scheduling operation inserts the missing \perp values). As the synchronous paradigm is often used in the development of safety-critical systems, input reading and the system itself must be deterministic, or at least predictable. It is therefore essential to consider classes of synchronous specifications that facilitate the development of efficient wrappers which make input reading deterministic while not restricting the behavior of the system.

Second, the implementation must preserve the semantics of the synchronous specification, meaning that the set of asynchronous observations of the specification must be identical to the set of observations of the implementation. Preservation of semantics is important because the advantages of synchrony lie with specification and verification. We would therefore like each implementation trace to be covered by the verification of the synchronous model. This problem is of particular importance when the synchronous specification must be implemented over a distributed architecture (an operation called *desynchronization*). In such cases, input reading and computation must be coordinated between distributed sites, and doing this without a careful analysis can be very inefficient (in terms of speed, consumption, communication, etc.) or simply incorrect.

This paper addresses the problem of desynchronizing a modular synchronous specification by replacing the communication lines between modules with asynchronous FIFOs. Instead of a single, global wrapper, we shall have one wrapper per system component. The exact problem we address is that of characterizing large classes of synchronous components for which small, simple wrappers² produce deterministic, efficient, and semantics-preserving GALS implementations. These classes of systems can then be considered as the implementation space, and the remaining problem is that of making given synchronous systems belong to these classes (by adding supplementary signaling). Naturally, a larger implementation space covers better solutions that use less synchronization.

Previous work. Previous approaches to implementing modular synchronous specifications over GALS architectures are respectively based on *latency-insensitive systems*, on *Kahn process networks (KPN)*, and on *endo/isochronous systems*.

In the *latency-insensitive systems* of Carloni *et al.* [4], each synchronous component reads every input and writes every output at each reaction. The communication protocols effectively simulate a single-clock system, which is inefficient, but simplifies the implementation.

In a *Kahn process network* [5], requiring that each component has a deterministic input/output behavior implies the determinism of the global system (and thus any wrapper is a good one). Often used, due to its robustness, in the development of embedded systems, the KPN-based approach has been adapted by Caspi *et al.* for the desynchronization of functional dataflow synchronous specifications [6]. Giving the approach its strength, the determinism is also its main drawback, as nondeterminism is often useful in the specification and analysis of concurrent systems. We also mention here the approach of Talpin *et al.* [7], which is based on a bounded version of the Kahn principle.

The approach based on endo/isochronous systems has been proposed by Benveniste et al. [8] in order to support the analvsis of partial specifications (which can be non-deterministic, or incomplete), to exploit execution modes, and to cover truly concurrent and multi-clock implementations. Informally speaking, a synchronous component is endochronous when the presence and absence of each variable can be inferred incrementally during each reaction from the current state and from the values of already present values. An endochronous component knows how to read its inputs, meaning that no wrapper is needed. Unfortunately, endochronous components can exhibit no internal concurrency, which makes endochrony non-compositional (thus, incremental system development is impossible). Isochrony is a semantics-preservation criterion over pairs of synchronous systems. The work of Singh and Theobald on generalized latency-insensitive systems [9] can be seen as implementing endochrony in hardware.

Essential improvement is brought by the work by Potop *et al.* [10] on weak endochrony and weak isochrony. Weak endochrony extends endochrony by allowing operations within a component to run independently when no synchronization is necessary. The notion is compositional, allowing incremental development of large systems. Being formulated in a non-causal framework, this approach is also less constrained than the KPN-based one, allowing non-determinism in the less abstract causal model. The non-causal framework is also the main disadvantage, because it hides implementation properties, like the presence of synchronization or communication deadlocks (which are important in practice).

The distribution of synchronous or strongly synchronized specifications has been studied in many other settings. We only mention here the Time-Triggered Architectures of Kopetz [11], the ocrep tool of Girault *et al.* [12], the AAA methodology of Sorel [13], and the desynchronization approach of Cortadella *et al.* [14]. From another point of view, our work is closely related to results concerning the design of delay-insensitive [15] and burst-mode [16] circuits.

Contribution. This paper brings an important improvement over previous work, by allowing us to *reason about concurrency and efficient synchronization in a causal, operational synchronous framework that takes into account the composition through read/write mechanisms.* The approach inherits the advantages of the weak endochrony-based approach: It allows the representation of non-deterministic specifications, takes into account execution and communication modes, and covers concurrent and multi-clock implementations. At the same time, it allows us to reason in a unified model about semantics-preservation and the absence of deadlocks due to synchronization and communication (which are both essential correctness properties of any implementation). As we shall see, *the level of detail is essential in this analysis, as it reveals the strong ties that exist between the two correctness properties,*

 $^{^{2}}$ For instance, wrappers that trigger a transition as soon as the needed input is available.

and simplifies the correctness analysis.

Our main contribution is the definition of a new model for the representation of asynchronous implementations of synchronous specifications. The model covers classical implementations, where a notion of global synchronization is preserved by means of signaling, and globally asynchronous, locally synchronous (GALS) implementations where the global clock is removed. We use this model to derive criteria ensuring the correct deployment of synchronous specifications over GALS architectures.

For space reasons, and in order to keep the presentation clear, we chose not to include the (rather complex) proofs of our results in this paper. They shall soon be available in [17]. **Outline.** The remainder of the paper is organized as follows: Section 2 defines the formal framework used throughout the paper, and section 3 gives intuitive examples and explains why the new structures are adapted to modeling and reasoning about the correctness of GALS implementations of synchronous specifications. Sections 4 defines criteria ensuring correct desynchronization. A short conclusion is given in section 5.

2. The model

This section defines our model of asynchronous implementation of a synchronous specification. We structured its presentation into several parts. The subsections 2.1, 2.2, and 2.3 introduce a rather standard notation for transition systems (variables, transition labels, traces, concurrent transition systems, and composition by synchronized product). Subsection 2.4 is the first to define communication channels, clocks, and the I/O transition systems which form our basic implementation model. Section 2.5 defines the synchronous transition systems – which are I/O transition systems satisfying the synchronous hypothesis. In section 2.6 we explain how transition systems are synchronously and asynchronously composed using FIFO models. Recall that intuitive examples are given later, in section 3.

2.1. Variables and labels

Our components and systems interact with each other and with their environment through *variables*. The *domain* of a variable v is denoted with \mathcal{D}_v . Given V a *finite* set of variables, a *label* over V is a partial valuation of its variables. Formally, the set of all labels over V is $\mathcal{L}_V = \prod_{v \in V} \mathcal{D}_v^{\perp}$, where $\mathcal{D}_v^{\perp} = \mathcal{D}_v \cup \{\perp\}$, and $\perp \notin \mathcal{D}_v$ is a special symbol denoting the *absence* of a value. The *support* of a label $l \in \mathcal{L}_V$ is $supp(l) = \{v \in V \mid l(v) \neq \perp\}$. We denote with \perp_V the label of empty support over V. For simplicity, we shall usually write out a label as the set of its non-absent variable valuations. For instance, $\langle v = 0, u = 1 \rangle_V$ denotes the label over V with support $\{u, v\}$ and which assigns 0 to v and 1 to u. When confusion is not possible, the set V of variables can be omitted from the notation.

If $l \in \mathcal{L}_V$ and V' is another set of variables, then the *image* of l through V' is the label $l \mid_{V'} \in \mathcal{L}_{V'}$ that equals l over $V \cap V'$ and equals \perp on $V' \setminus V$.

The labels $l_i \in V_i$, i = 1, 2, are *non-contradictory*, denoted $l_1 \bowtie l_2$, if for all $v \in V_1 \cap V_2$ such that $l_i(v) \neq \bot, i = 1, 2$ we have $l_1(v) = l_2(v)$. In this case, their *union* is $l_1 \sqcup l_2 \in \mathcal{L}_{V_1 \cup V_2}$, of support $supp(l_1) \cup supp(l_2)$ and which equals l_i over $supp(l_i), i = 1, 2$. The union operator is associative and commutative. If $l_1, l_2 \in \mathcal{L}_V$ are non-contradictory, we also define their *difference* $l_1 \setminus l_2 \in \mathcal{L}_V$ by:

$$l_1 \setminus l_2(u) = \begin{cases} l_1(u), \text{ if } u \notin supp(l_2) \\ \bot, \text{ otherwise} \end{cases}$$

When the non-contradictory labels $l_i \in V_i$, i = 1, 2 are equal over $V_1 \cap V_2$, they are called *synchronizable*. Their union is also called in this case *product* and denoted with $l_1 \otimes l_2$. Note that $l_1 \otimes l_2$ equals l_i on V_i , i = 1, 2. The labels $l_i \in V_i$, i = 1, 2are *disjoint* if $supp(l_1) \cap supp(l_2) = \emptyset$.

Assume that v and v' are variables with the same domain (*i.e.* $\mathcal{D}_v = \mathcal{D}_{v'}$). Given $l \in \mathcal{L}_V$, with $v \in V$ and $v' \notin V \setminus \{v\}$, the name change operator associates $l[v/v'] \in \mathcal{L}_{V \setminus \{v\} \cup \{v'\}}$ with:

$$l[v/v'](u) = \begin{cases} l(u), \text{ if } u \neq v' \\ l(v), \text{ if } u = v' \end{cases}$$

We define the "inclusion" partial order \leq over \mathcal{L}_V : $l_1 \leq l_2$ if $\forall v : (l_1(v) \neq \bot \Rightarrow l_1(v) = l_2(v))$.

2.2. Traces

A trace over the set of variables V is a finite sequence of labels over V. The set of all traces over V is denoted with $Traces(V) = \mathcal{L}_V^* = \{(l_i)_{0 \le i < n} \mid n \in \mathbb{N} \land \forall i : l_i \in \mathcal{L}_V\}.$ Given a trace $\varphi = (l_i)_{0 \le i < n}$ we denote $length(\varphi) = n$ and $\varphi[i] = l_i$. Note that any label is a trace of length 1.

Any two traces $\varphi_1, \varphi_2 \in Traces(V)$ can be concatenated. The trace φ_1 is a prefix of φ_2 (written $\varphi_1 \preceq \varphi_2$) if, by definition, $\varphi_2 = \varphi_1 \varphi_3$ for some φ_3 . The prefix relation is a partial order over traces. The image operator is extended component-wise on traces: $(l_i)_{0 \le i < n} |_{V'} = (l_i |_{V'})_{0 \le i < n}$. The traces $\varphi_i \in Traces(V_i), i = 1, 2$ are called synchronizable if $length(\varphi_1) = length(\varphi_2)$ and if for all j the labels $l_1[j]$ and $l_2[j]$ are synchronizable. In this case, we can define the product trace $\varphi_1 \otimes \varphi_2 = (l_1[i] \otimes l_2[i])_{0 \le i < length(\varphi_1)}$. The product operator is associative and commutative. The support of a trace φ , denoted $supp(\varphi)$, is the union of the supports of its labels.

2.3. Generalized concurrent transition systems

The generalized concurrent transition systems (GCTS) form our (asynchronous) implementation model. GCTSs are step transition systems where steps are syntactic representions of the concurrency between atomic operations (which assign or test a single variable). They generalize the concurrent transition systems of Stark [18], and can be seen as a subset of the step transition systems of Mukund [19].

Definition 1 (generalized concurrent transition system)

A generalized concurrent transition system (GCTS) is a tuple $\Sigma = (S, \hat{s}, V, \odot \Sigma)$, where S is the set of states (not necessarily finite), $\hat{s} \in S$ is the initial state, V is the finite set of communication variables, and $\hookrightarrow_{\Sigma} \subseteq S \times \mathcal{L}_V \times S$ is a transition relation satisfying:

GCTS1 (void transition): $\forall s \in S : s \circ \frac{l_V}{\Sigma} > s$. GCTS2 (prefix closure): If $s \circ \frac{l}{\Sigma} > s'$ and $l' \leq l$, then there exists $s'' \in S$ such that $s \circ \frac{l'}{\Sigma} > s''$ and $s'' \circ \frac{l \setminus l'}{\Sigma} > s'$.

When there is no ambiguity, Σ can be dropped from the transition relation notation.

We shall say that $\varphi = l_1 \dots l_n \in Traces(V)$ is a trace of the GCTS $\Sigma = (S, \hat{s}, V, \to \Sigma)$ starting in the state $s \in S$ if there exist $s_1, \dots, s_n \in S$ such that $s \circ \stackrel{l_1}{\longrightarrow} s_1 \circ \stackrel{l_2}{\longrightarrow} \dots \circ \stackrel{l_n}{\longrightarrow} s_n$. In this case, we also write $s \xrightarrow{\varphi} s_n$. The set of all traces of Σ starting in s is denoted by $Traces_{\Sigma}(s)$, and the set of all destination states of such traces is: $RSS_s(\Sigma) = \{s' \in S \mid \exists \varphi : s \xrightarrow{\varphi} s'\}$. The reachable state space of Σ is $RSS(\Sigma) = _{def} RSS_{\hat{s}}(\Sigma)$.

Concurrent transition systems are composed by means of synchronized product. If $\Sigma_i = (S_i, \hat{s}_i, V_i, \odot_{\Sigma_i}), i = 1, 2$, then $\Sigma_1 \otimes \Sigma_2 = (S_1 \times S_2, (\hat{s}_1, \hat{s}_2), V_1 \cup V_2, \odot_{\Sigma_1 \otimes \Sigma_2})$, where $(s_1, s_2) \xrightarrow{l}_{\Sigma_1 \otimes \Sigma_2} (s'_1, s'_2) \Leftrightarrow s_i \circ \frac{l|_{V_i}}{\Sigma_i} s'_i, i = 1, 2$. The \otimes operator is well-defined – it preserves the properties GCTS1 and GCTS2. It is also associative and commutative, and $Traces_{\bigotimes_{i=1}^n \Sigma_i} ((s_i)_{1 \le i \le n})$ is the set of all $\bigotimes_{i=1}^n \varphi_i$, where $\varphi_i \in Traces_{\Sigma_i}(s_i)$ are mutually synchronizable.

The variable name change operator is extended to GCTSs. If $\Sigma = (S, \hat{s}, V, \odot \Sigma)$, $v \in V$, $v' \notin V \setminus \{v\}$, and $\mathcal{D}_v = \mathcal{D}_{v'}$, then:

$$\Sigma[v/v'] = (S, \hat{s}, V \setminus \{v\} \cup \{v'\}, \{ s \overset{l[v/v']}{\xrightarrow{\Sigma[v/v']}} s' \mid s \overset{l}{\xrightarrow{\Sigma}} s' \})$$

2.4. I/O causality. Channels and clocks

In real life, the communication between the different components of a system is directed. One component emits a value on a channel, and another reads it. To take this into account, we use directed communication channels that are pairs of directed *variables.* We emit a value on a channel c by assigning the variable !c, and we receive a value by reading the variable ?c. The variables !c and ?c have the same domain, denoted with \mathcal{D}_c . We denote with $\mathcal{C}(V) = \{c \mid | c \in V \lor ?c \in V\}$ the set of channels associated with a set of variables V. To simplify the model, we assume that every channel connects at most one emitter with at most one receiver, meaning that !c is variable of at most one component in the system, the same holding for ?c(a simple renaming technique allows the use of multicast, but we shall not cover the subject here). We further assume that the only variables that are not directed are the clocks of the synchronous components. A clock is a variable whose domain is $\mathcal{D}_{clk} = \{\top\}$ (\top stands for the "clock tick"). Given a set V of variables we shall denote with Clocks(V) the subset of clock variables, and with $Directed(V) = V \setminus Clocks(V)$ the subset of directed variables. To simplify the notations, we abbreviate the clock tick valuation $\tau = \top$ with τ (for any clock variable τ).

Definition 2 (I/O transition system) We say that a GCTS is an I/O transition system when all its variables are either directed or clocks. From now on, this paper only considers I/O transition systems.

To reason about desynchronization properties, we shall need the following function, which removes the clock synchronization barriers, so that only messages (and not absence) are visible, along with message ordering on each channel: $\delta : (\mathcal{D}_v^{\perp})^* \to \mathcal{D}_v^*$, defined by $\delta(\epsilon) = \epsilon$ and:

$$\delta(v\varphi) = \begin{cases} v\delta(\varphi), \text{ if } v \neq \bot \\ \delta(\varphi), \text{ otherwise} \end{cases}$$

Using this notation, we extend the relation \leq (first defined on labels) to a preorder over Traces(V): Given $\varphi_1, \varphi_2 \in Traces(V)$, we write $\varphi_1 \leq \varphi_2$ whenever we have $\delta(\varphi_1 |_{\{v\}}) \preceq \delta(\varphi_2 |_{\{v\}})$ for all $v \in Directed(V)$. If $\varphi_1 \leq \varphi_2$ and $\varphi_2 \leq \varphi_1$, then we say that φ_1 and φ_2 are asynchronously equivalent, denoted $\varphi_1 \sim \varphi_2$. When for all $v \in Directed(V)$ we have $\delta(\varphi_1 |_{\{v\}}) \preceq \delta(\varphi_2 |_{\{v\}})$ or $\delta(\varphi_2 |_{\{v\}}) \preceq \delta(\varphi_1 |_{\{v\}})$, then we say that φ_1 and φ_2 are asynchronously non-contradictory, and write $\varphi_1 \bowtie \varphi_2$. Note that \bowtie extends the non-contradiction relation over labels. Moreover, we can extend the label difference operator to non-contradictory traces by inductively defining the asynchronous difference of traces $\varphi_1 \setminus \varphi_2$:

$$\begin{cases} \varphi_1 \setminus (l\varphi_2) = (\varphi_1 \setminus l) \setminus \varphi_2\\ (l_1\varphi_1) \setminus l_2 = (l_1 \setminus l_2)(\varphi_1 \setminus (l_2 \setminus l_1)) \end{cases}$$

If φ is a trace of an I/O transition system, then we denote with $|\varphi|$ the number of assignments of non-clock variables contained in φ .

2.5. Synchronous transition systems

Our synchronous transition systems represent causal synchronous specifications or, equivalently, implementations of synchronous specifications where the global clock is preserved by means of signalization. A synchronous transition system, as defined below, is an I/O transition system with a single clock variable, and satisfying the synchronous hypothesis and a stuttering-invariance property (which is necessary if we want to derive GALS implementations).

Definition 3 (synchronous transition system) A microstep synchronous transition system (μ STS) is a tuple $\Sigma = (S, \hat{s}, V, \tau, \infty)$ where all the variables of V are directed, where τ is a clock variable (the clock of the component), and where $(S, \hat{s}, V \cup {\tau}, \infty)$ is a GCTS satisfying:

 μ STS1 (clock transitions): if $s \xrightarrow{l} s'$ and $l(\tau) \neq \bot$ then $l \mid_V = \bot_V$. μ STS2 (stuttering-invariance):

$$\hat{s} \xrightarrow{\langle \tau \rangle} \hat{s} \text{ and } (s \xrightarrow{\langle \tau \rangle} s' \Rightarrow s' \xrightarrow{\langle \tau \rangle} s')$$

 μ STS3 (single assignment): two assignments of a same variable must be separated by a clock transition. More exactly, if $s_0 \stackrel{l_1}{\longrightarrow} s_1 \stackrel{l_2}{\longrightarrow} \cdots \stackrel{l_n}{\longrightarrow} s_n$ and $\forall i : l_i \neq \tau$, then l_1, \ldots, l_n are mutually disjoint.

Note that axiom μ STS1 identifies the *clock transitions* – with label $< \tau > -$ which are the only transitions where the clock variable is present. Such transitions separate synchronous reactions during which a variable cannot be assigned more than once (cf. axiom μ STS3). A state which is destination of a clock transition is called synchronizing state. Given a trace φ of a synchronous system, we can decompose it into reactions $\varphi = Step_0(\varphi) < \tau > Step_1(\varphi) < \tau > \dots$ where each reaction $Step_i(\varphi)$ contains no clock transition. As the transitions of each $Step_i(\varphi)$ are disjoint, we can denote with $\langle Step_i(\varphi) \rangle$ the union of all its labels. We shall say that a trace φ is *complete* if it ends with a $\langle \tau \rangle$ transition. We say that a μ STS is *non-blocking* if from any reachable state there is a path towards a stuttering state. Note that in a nonblocking μ STS any trace can be completed. Blocking systems are considered incorrect.

2.6. Synchronous and asynchronous composition

As earlier mentioned, we simplify the model by only allowing point-to-point communication, and we enforce this rule by syntactic means³.

Definition 4 (composable transition systems) We say that the I/O transition systems $\Sigma_i, i = \overline{1, n}$ are composable if their variable sets are mutually disjoint.

Note that the definition requires not only point-to-point communications (no directed variable is shared by two or more systems), but also the non-overlapping of clock sets (which is natural). Also note that a system can have both !c and ?c as variables, thus allowing the representation of systems obtained by composition.

The composition of synchronous and asynchronous systems is defined by means of synchronized product, using FIFO models to represent communication through synchronous and asynchronous channels. To represent synchronous communication, we use 1-place synchronous FIFO models (which are μ STSs themselves). The FIFO model associated with a channel *c* is:

$$\begin{aligned} SFIFO(c,\tau) &= \\ (\{c_0,c_1\} \cup \bigcup_{x \in \mathcal{D}_c} \{c_x\}, c_0, \bigcup_{x \in \mathcal{D}_c} \{!c = x, ?c = x\}, \tau, \circ \to_S) \end{aligned}$$

where the transition relation is defined by:

$$\tau \underbrace{\overset{\circ}{\smile}}_{c_0} \underbrace{\overset{}{\longrightarrow}}_{c_x} \underbrace{\overset{c=x}{\longrightarrow}}_{c_1} c_1 \\ \times \underbrace{\overset{<}{\longleftarrow}}_{<\tau>} c_> \quad , x \in \mathcal{D}_c$$

 3 However, broadcast can be simulated by replicating and renaming variables.

Note that modeling multicast communication (a feature that will not be addressed in this paper), is simply done by renaming channel read variables in a component-wise fashion, and then modifying the FIFO model to allow the concurrent read of the value from different sites.

Asynchronous communication involves infinite asynchronous FIFO models (which are not μ STSs):

$$AFIFO(c) = (\mathcal{D}_c^*, \epsilon, \bigcup_{x \in \mathcal{D}_c} \{ !c = x, ?c = x \}, \circ \to_A)$$

where the transition relation contains all the transitions of the form:

$$x_1 \dots x_n \xrightarrow{} x_1 \dots x_n x_{n+1} \xrightarrow{} x_2 \dots x_{n+1}$$

Definition 5 (synchronous composition of μ **STSs**) Let

 $\Sigma_i = (S_i, \hat{s}_i, V_i, \tau_i, \dots, \Sigma_i), i = 1, 2$ be composable μ STSs and let τ be a clock variable. Then, the synchronous composition of Σ_1 and Σ_2 over the base clock τ is:

$$\Sigma_1 \mid^{\tau} \Sigma_2 = \Sigma_1[\tau_1/\tau] \otimes \Sigma_2[\tau_2/\tau] \otimes \bigotimes_{c \in \mathcal{C}(V_1) \cap \mathcal{C}(V_2)} SFIFO(c,\tau)$$

Lemma 1 (synchronous composition properties) The synchronous composition of the μ STSs Σ_1 and Σ_2 over the base clock τ is a μ STS of clock τ . The result of the synchronous composition is unique upto isomorphism⁴ of μ STSs (so that we can discard τ from the notation). Moreover, the operator | is associative and commutative (again, modulo isomorphism).

In addition, note that synchronizing states of $|_{i=1}^{n} \Sigma_{i}$ have void communication lines (all synchronous FIFO models are in their unique synchronizing state).

Definition 6 (asynchronous composition of I/O systems)

Let $\Sigma_i = (S_i, \hat{s}_i, V_i, \odot \Sigma_i), i = 1, 2$ be composable I/O transition systems. Then, the asynchronous composition of Σ_1 and Σ_2 is:

$$\Sigma_1 \mid\mid \Sigma_2 = \Sigma_1 \otimes \Sigma_2 \otimes \bigotimes_{c \in \mathcal{C}(V_1) \cap \mathcal{C}(V_2)} AFIFO(c)$$

Lemma 2 (asynchronous composition properties) The

asynchronous composition of I/O transition systems results in another I/O transition system. The || operator is associative and commutative. The asynchronous composition of two μ STSs is not a μ STS.

⁴An isomorphism λ between the GCTSs $\Sigma_i = (S_i, \hat{s_i}, V_i, \bigoplus_{\Sigma_i}), i = 1, 2$ consists of two bijections $\lambda^S : S_1 \to S_2$ and $\lambda^V : V_1 \to V_2$ having the properties: (i) $\forall v : \mathcal{D}_v = \mathcal{D}_{\lambda^V(v)}$, (ii) $\lambda^S(\hat{s_1}) = \hat{s_2}$, and (iii) $s \circ \frac{l}{\Sigma_1} \succ s' \Leftrightarrow \lambda^S(s) \circ \frac{\lambda(l)}{\Sigma_2} \bigstar^S(s')$, where $\lambda(l)$ denotes the label obtained from l by renaming v with $\lambda^V(v)$ for all $v \in V_1$. If Σ_1 and Σ_2 are l/O transition systems, we say that λ is an isomorphism of l/O transition systems if λ^V maps read variables onto read variables, write variables, the λ is

an isomorphism of μ STSs if it is an isomorphism of I/O transition systems.

2.7. Product states and product traces

Note that the state of a synchronous or asynchronous product of I/O systems is not only given by the state of the components, but also by the state of its communication channels. Indeed, given the composable $\mu \text{STSs } \Sigma_i, i = \overline{1,n}$, connected through the channels $c_i, i = \overline{1,m}$, the state of $|_{i=1}^n \Sigma_i$ is $((s_i)_{i=\overline{1,n}}, (c_i^s)_{i=\overline{1,m}})$, and the state of $||_{i=1}^n \Sigma_i$ is $((s_i)_{i=\overline{1,m}})$, where c_i^s denotes states of $SFIFO(c_i, \tau)$ and c_i^a denotes states of $AFIFO(c_i)$.

Nevertheless, our channels are fully deterministic, so that the evolution of the system is completely determined by the transitions of the components Σ_i . Therefore, when we work on traces starting in given states we can drop the FIFO states from the global system state notation (the destination state is fully determined by the initial state and by the trace).

As should be expected, the synchronous composition binds tighter than the asynchronous one. Indeed, given the composable μ STSs $\Sigma_i = (S_i, \hat{s}_i, V_i, \tau_i, \odot_{\Sigma_i}), i = \overline{1, n}$, we can map the state space of $|_{i=1}^n \Sigma_i$ onto the state space of $||_{i=1}^n \Sigma_i$:

$$\iota: RSS(|_{i=1}^n \Sigma_i) \hookrightarrow RSS(||_{i=1}^n \Sigma_i)$$

by mapping for each communication channel c the state of $SFIFO(c, \tau)$ onto the state of AFIFO(c) using: $c_0 \mapsto \epsilon$, $c_1 \mapsto \epsilon$, and $\forall x \in \mathcal{D}_c : c_x \mapsto x$. Similarly, we can define for any $s_i \in S_i, i = \overline{1, n}$ the injective "inclusion morphism" that maps traces of the synchronous product into traces of the asynchronous product:

$$\iota: Traces_{|_{i=1}^{n}\Sigma_{i}}(s) \hookrightarrow Traces_{|_{i=1}^{n}\Sigma_{i}}(\iota(s))$$

defined inductively by $\iota(\epsilon) = \epsilon$, by $\iota(\varphi_1 \varphi_2) = \iota(\varphi_1)\iota(\varphi_2)$, and (for labels) by:

$$\iota(l) = \begin{cases} l \mid_{\bigcup_{i=1}^{n} V_i \cup \{\tau_i \mid i = \overline{1, n}\}}, \text{ if } l \neq <\tau > \\ <\tau_1, \dots, \tau_n >, \text{ if } l = <\tau > \end{cases}$$

where τ stands by the base clock of the synchronous composition. With these notations we have:

$$s \xrightarrow{\varphi}_{|_{i=1}^n \Sigma_i} s' \Rightarrow \iota(s) \xrightarrow{\iota(\varphi)}_{||_{i=1}^n \Sigma_i} \iota(s')$$

3. Modelling and correctness of GALS implementations

This section starts by illustrating our definitions with a number of small, but intuitive examples. Based on this intuition, we are define in section 3.2 the formal correctness criterion. Section 3.3 explains why our model is useful in solving the GALS implementation problem.

3.1. Examples

The following μ STS represents a system that emits a message on channel *a* and then awaits for one message from either channel *b* or *r* (*e.g.* for whichever comes first). Data is uninterpreted (not important), therefore not represented. The

clock of the system is τ_1 , and we shall assume that the directed variable set of Σ_1 is $\{!a, ?b, ?r\}$:



In a more classical macrostep framework, like that of [10], this system would be represented by:



The correspondence between the microstep and macrostep representations of a system is straightforward: The states of the macrostep system are the synchronizing states of the microstep one. The macrostep transitions correspond to full reactions connecting synchronizing states (after forgetting the direction of the signals and the causality between successive labels).

We compose Σ_1 with the μ STS Σ_2 , which has the clock τ_2 and the directed variable set $\{?a, !b\}$:

$$\Sigma_2: <\tau_2 > \bigcirc t_0 \overset{<\tau_2 >}{\longrightarrow} t_1 \overset{<\tau_2 >}{\underset{<\tau_2 >}{\longrightarrow}} t_2 \overset{}{\longrightarrow} t_3 \overset{<\tau_2 >}{\longrightarrow} <\tau_2 >$$

The synchronous composition $\Sigma_1 \mid \Sigma_2$ is done using two synchronous FIFOs, corresponding to the variables/channels a and b:

Note again that data is uninterpreted, only write/read causality and clock synchronization is considered. The composed synchronous system is (we simplified for space reasons the label notations):



Note that we simplified the notation by not representing the state of the two FIFOs (the initial state having void FIFOs, the status of the FIFOs is fully determined in each state). However, note that the composed system is blocked in state (s_3, t_3) because $SFIFO(b, \tau)$ cannot take a clock transition (data has been written on it, but not read). The system $\Sigma_1 \mid \Sigma_2$ is blocking, thus incorrect.

The asynchronous composition $\Sigma_1 || \Sigma_2$ is done using the two asynchronous FIFOs, figured below:

$$AFIFO(b): \underbrace{\epsilon \overset{<|b>}{\searrow} b}_{b} \qquad AFIFO(a): \underbrace{\epsilon \overset{<|a>}{\searrow} a}_{a}$$

Recall that in the general case the asynchronous FIFO models are infinite. However, Σ_1 and Σ_2 can emit at most one message on any of the two channels, so our choice does not affect the result of the composition:

 $\Sigma_1 \mid\mid \Sigma_2$:



It is essential no note that $\Sigma_1 || \Sigma_2$ has traces, like $\langle !a \rangle \langle ?a \rangle \langle \tau_2 \rangle \langle !b \rangle \langle ?b \rangle$, that are not asynchronously equivalent to any of the synchronous traces of $\Sigma_1 || \Sigma_2$. Such traces are not covered by the verification done on the synchronous model, meaning that the GALS implementation does not preserve the semantics of the specification.

It is also important to note that requiring a one-to-one correspondence between synchronous and asynchronous traces is not a good idea, because for large classes of systems it can be highly inefficient. Consider, for instance, the following system:



and its synchronous and asynchronous composition with Σ_2 :



As expected, the synchronous composition binds tighter than the synchronous one, but for any trace of $\Sigma_3 || \Sigma_2$ going from (s_0, t_0) to (s_4, t_3) we can find an asynchronously equivalent trace in $\Sigma_3 || \Sigma_2$. Such a GALS implementation is obviously correct, because it does not introduce new behaviors. Exploiting the concurrency between different computations (as we do here) to allow the systems to evolve at different rates is a desirable feature because it minimizes communication and consumption. The difference between Σ_1 and Σ_3 is that in Σ_3 the transitions $\langle ?b \rangle$ and $\langle ?r \rangle$ are concurrent in state s_1 , while in Σ_1 there is a non-deterministic choice between them (meaning that if messages come on both channels, only one will be read, in an unpredictable fashion).

3.2. Formal correctness criterion

The intuitive notion of correctness of a GALS implementation *w.r.t.* its microstep synchronous specification is formalized in the following criterion:

Criterion 1 (correct desynchronization) Let Σ_i , $i = \overline{1, n}$ be composable μ STSs. Then, we shall say that the GALS implementation $||_{i=1}^n \Sigma_i$ is correct w.r.t. the synchronous specification $|_{i=1}^n \Sigma_i$ if for all synchronizing state s of $|_{i=1}^n \Sigma_i$ and for all trace $\varphi \in Traces_{||_{i=1}^n \Sigma_i}(\iota(s))$ there exist $\tilde{\varphi} \in Traces_{||_{i=1}^n \Sigma_i}(\iota(s))$ and $\overline{\varphi} \in Traces_{|_{i=1}^n \Sigma_i}(s)$ such that $\varphi \preceq \tilde{\varphi}$ and $\tilde{\varphi} \sim \iota(\overline{\varphi})$.

In other words, the GALS implementation is correct if any of its traces can be completed with a finite number of transitions to a trace that is asynchronously equivalent to a complete synchronous trace. Our criterion is akin to previous correctness criteria [8], [10] defined in a macrostep setting. Most important, criterion 1 allows us to exploit (like in $\Sigma_3 || \Sigma_2$) the concurrency of the synchronous specification to support GALS implementations that are weakly synchronized, yet correct. Important differences exist, though, as our criterion is formulated in a micro-step operational framework that simplifies, as we shall see in section 4.2, the definition of sufficient conditions for correctness.

As explained in the introduction, our purpose is now to find sufficient conditions for correctness (in the formal sense of criterion 1) that cover large classes of implementations. We do not cover here the synthesis problem of transforming given systems to satisfy the correctness criterion. However, we use two examples to give the intuition of future synthesis techniques: First, to correct the composition of Σ_1 with Σ_2 , we can simply prevent Σ_2 from firing the transition labeled < !b > by guarding it with a condition that is never fulfilled:

$$\Sigma'_{2}: \tau_{2} \bigcirc t_{0} \stackrel{?a}{\hookrightarrow} t_{1} \stackrel{?a}{\longrightarrow} t_{2} \stackrel{?d}{\hookrightarrow} t'_{3} \stackrel{!b}{\hookrightarrow} t_{3} \stackrel{\tau_{2}}{\bigcirc} \tau_{2}$$

More interesting is the case where we compose Σ_1 with a process that non-deterministically chooses between emitting *b* or doing something else. In this case, the solution is to signal the non-deterministic choice to Σ_1 , so that it can adapt its behavior:



Here, we assumed that the non-deterministic choice between $\langle !b \rangle$ and $\langle !x \rangle$ is an essential feature of the specification Σ_4 , which must be preserved. To make the composition correct we need to make this choice visible from its asynchronous environment, under the form of a choice over the value of a new signal (d). Component Σ'_1 can use this signal to decide which message to wait for.

3.3. Modeling issues

The I/O transitions systems can be viewed either as microstep specifications, or as asynchronous implementation models. A sub-class of I/O transition systems satisfy the synchronous hypothesis – they have a single clock variable, which determines clock transitions, and no variable is assigned twice between successive clock transitions. Thus, they can be seen as microstep synchronous specifications. The only hypothesis that departs from the classical synchronous model is stuttering-invariance. However, we see stuttering-invariance as a prerequisite for the efficient multi-rate GALS deployment.

If we compare our model to macro-step models like those of [8], [10], every macrostep specification (incomplete automaton) has (at least) a microstep implementation. Like many macro-step models, our formalism does not explicitly represent the reaction to signal absence. This does dot influence the expressivity of the model, as reaction to signal absence can be represented using non-deterministic choice. The composition through point-to-point links is not an essential restriction, as it is easy to define FIFO models that cover multicast.

The synchronous and asynchronous composition operators reflect the assumption that an emitted signal must not be left unread by the receiver. This hypothesis reflects in an operational fashion the rendez-vous-like synchronized product composition from macro-step formalisms.

Composing the μ STSs Σ_i , $i = \overline{1, n}$ using the || operator intuitively corresponds to implementing $|_{i=0}^n \Sigma_i$ as a GALS system where all the communication lines have been replaced with asynchronous FIFOs. The components are still clocked, but individual clocks are independent, and the components are only synchronized by the FIFO causality rules. In the GALS implementation the clock of one component can be triggered concurrently with another clock or an assignment of another component. The GALS implementation can function in a multi-rate fashion, as no constraint relates the occurrence of clock transition in different components.

Compared to classical macro-step approaches, our model brings a level of detail which is essential in deciding the correctness of actual implementations. Composing Σ_5 and Σ_6 results in a blocking system:

$$\Sigma_{5}: \quad \underbrace{\tau_{1}}^{\tau_{1}} 0 \stackrel{a}{\longrightarrow} 1 \stackrel{b}{\longrightarrow} 2 \stackrel{c}{\longrightarrow} 3 \stackrel{\tau_{1}}{\bigcirc} \tau_{1}$$
$$\Sigma_{6}: \quad \underbrace{\tau_{2}}^{\tau_{2}} 0 \stackrel{c}{\longrightarrow} 1 \stackrel{b}{\longrightarrow} 2 \stackrel{\sigma}{\frown} \tau_{2}$$

However, this problem cannot be observed in macro-step settings, where the system does not block and can even fire the transition of label *abc*. Indeed, the microstep model is better suited for analysis akin to causality checks performed in synchronous languages like Esterel. In fact, we shall see in section 4.2, that non-blocking correctness and semantics preservation are truly related.

4. Correct desynchronization criteria

Following the goal fixed in the introduction, we now define criteria that characterize a large class of synchronous components for which small, simple wrappers produce deterministic, efficient, and semantics-preserving GALS implementations. We recall that the proofs of our results will be available in [17].

4.1. Microstep weak endochrony

Microstep weak endochrony (or, simply, weak endochrony) is the property guaranteeing that a given synchronous component (μ STS) knows how to read its inputs, so that no asynchronous wrapper is needed. Weak endochrony requires that all internal choice of the component is visible as a choice over the value (and not presence/absence status) of a directed variable (either input or output). Thus, the behavior of the system becomes predictable in any asynchronous environment, because choices can be observed.

With this requirement, the implementation space delimited by weak endochrony is nonetheless very large: Concurrent behaviors are not affected by the previous rule, so that independent system parts can evolve at different speeds. Weak endochrony does not require I/O determinism. Instead, a weakly endochronous component must inform the environment about non-deterministic decisions (the variable used to do so behaves like an oracle that is visible from outside).

Definition 7 (weak endochrony) We say that the $\mu STS \Sigma =$ $(S, \hat{s}, V, \tau, \infty)$ is weakly endochronous if it satisfies the following axioms:

- μ WE1(determinism): $s \xrightarrow{l} s_i$, $i = 1, 2 \Rightarrow s_1 = s_2$ (from now on, we shall denote with $s.\varphi$ the unique state of Σ having the property $s \xrightarrow{\varphi} s.l$, and the notation is extended to traces).
- μ **WE2(independence):** if l_1 and l_2 are disjoint and if $l_1, l_2 \neq \tau$, then:



 μ **WE3(clock properties):** assume that $s_0 \xrightarrow{<\tau>} s_1$ and $\varphi \in Traces_{\Sigma}(s_0)$ with $\tau \notin supp(\varphi)$. Then:

- 1) $\varphi \in Traces_{\Sigma}(s_1)$
- 2) if $\varphi < \tau \rangle \in Traces_{\Sigma}(s_0)$, then $\varphi < \tau \rangle \in$ $Traces_{\Sigma}(s_1)$ and $s_0.\varphi < \tau > = s_1.\varphi < \tau >$
- 3) if $\varphi \psi < \tau > \in Traces_{\Sigma}(s_1)$, then there exists $\psi' \leq \psi \text{ such that } \varphi \psi' < \tau \rangle \in Traces_{\Sigma}(s_{0}).$ 4) if $\varphi < \tau >$, $\theta < \tau > \in Traces_{\Sigma}(s_{0})$ and $\varphi \bowtie$
- θ , then $\varphi(\theta \setminus \varphi) < \tau > \in Traces_{\Sigma}(s_0)$
- μ **WE4(choice):** if $\varphi_i < v = x_i > \in Traces_{\Sigma}(s), i = 1, 2$ and $\varphi_1 \bowtie \varphi_2$, then $\varphi_1 < v = x_2 > \in Traces_{\Sigma}(s)$.

Similar in intuition and in function to its macrostep counterpart [10], weak endochrony is nevertheless specific to our more concrete causal, microstep framework. Thus, while choice can only occur at the level of atomic variable assignments, concurrency (more precisely confluence) must also deal with full reactions and clock transitions (through axioms μ WE2 and μ WE3, and their consequences of lemma 3). Axiom μ WE4 insures that a choice between two concurrent execution

paths does not hide a "real" choice between non-concurrent assignments.

Lemma 3 (confluence) Let $\Sigma = (S, \hat{s}, V, \tau, \circ \rightarrow)$ be a weakly endochronous μ STS, let $s \in S$, and let $\varphi_i \in Traces_{\Sigma}(s), i =$ 1, 2 such that $\varphi_1 \bowtie \varphi_2$. Then:

- 1) If $\tau \notin supp(\varphi_i), i = 1, 2$, then $s.\varphi_1(\varphi_2 \setminus \varphi_1)$ and $s.\varphi_2(\varphi_1 \setminus \varphi_2)$ are defined and equal.
- 2) If φ_i complete, i = 1, 2, then $s.\varphi_1(\varphi_2 \setminus \varphi_1)$ and $s.\varphi_2(\varphi_1 \setminus \varphi_2)$ are defined and equal.

Note that the classical independence (full commutation) results are recovered by considering the case where $supp(\varphi_1)$ and $supp(\varphi_2)$ share no directed variable. However, the finer microstep notion allows us to consider systems like Σ_3 where the classical macrostep independence does not apply (in state s_0 , the macrostep transitions ab and ar do not commute, yet the system is I/O deterministic). The confluence properties of an endochronous system are even stronger, as stated by the following:

Theorem 4 (determinism) (a) Let $\Sigma = (S, \hat{s}, V, \tau, T)$ be a weakly endochronous μ STS, $s \in S$, and let φ_1, φ_2 be complete traces of $Traces_{\Sigma}(s)$ such that $\varphi_1 \sim \varphi_2$. Then, $s.\varphi_1 = s.\varphi_2$. (b) Moreover, if $\varphi_3 \in Traces_{\Sigma}(s)$ with $\varphi_3 \leq \varphi_1$, then there exists φ'_3 complete such that $s.\varphi_3\varphi'_3 = s.\varphi_1$ and $\varphi_1 \sim \varphi_3\varphi'_3$.

In fact, these strong confluence properties allow us to put any trace of of a weakly endochronous system in normal form, in which every transition is maximal and the number of reactions minimal (but we shall not develop the subject here).

Instead, we conclude the presentation of weak endochrony by stating the very important compositionality result that allows us to incrementally build complex weakly endochronous systems.

Theorem 5 (compositionality) Let Σ_i , $i = \overline{1, n}$ be composable weakly endochronous μ STSs. Then, $|_{i=1}^n \Sigma_i$ is weakly endochronous.

Weak endochrony is illustrated by the μ STSs Σ_2 , Σ_3 , and $\Sigma_3 \mid \Sigma_2$ of section 3.1, and by all the examples of the sections 3.2 and 3.3. The μ STS Σ_1 is not weakly endochronous because the non-deterministic choice in state s_1 cannot be observed in an asynchronous environment, so that other components, like Σ_4 , cannot adjust their behavior to preserve the synchronous semantics. The transformation of Σ_1 in Σ'_1 illustrates the type of instrumentation required to transform a general μ STS into a weakly endochronous one.

4.2. Correctness results

Weak endochrony is compositional, but the weak endochrony of the components does not guarantee the correctness (non-blocking) of the global synchronous specification, nor the correctness (semantics preservation) of the GALS implementation model. This can be easily checked on the systems formed by composing Σ'_1 and Σ_2 .

The most important result of this paper is the following theorem, which states that the correctness of the synchronous composition implies the correctness of the GALS implementation.

Theorem 6 (correctness) Let $\Sigma_i, i = \overline{1,n}$ be composable weakly endochronous μ STSs. If $|_{i=1}^n \Sigma_i$ is non-blocking, then $||_{i=1}^n \Sigma_i$ is correct w.r.t. $|_{i=1}^n \Sigma_i$ in the sense of criterion 1.

The result goes in fact beyond the requirements of criterion 1, by also ensuring a strong confluence property (given by the following lemma):

Lemma 7 (completion, GALS) Let $\Sigma_1, \ldots, \Sigma_n$ be composable weakly endochronous μ STSs, let s be a synchronizing state of $|_{i=1}^n \Sigma_i$, and let $\psi \in Traces_{|_{i=1}^n \Sigma_i}(s)$, complete, and $\varphi \in Traces_{|_{i=1}^n \Sigma_i}(\iota(s))$ such that $\varphi \leq \iota(\psi)$. Then, there exists $\theta \in Traces_{|_{i=1}^n \Sigma_i}(\iota(s).\varphi)$ such that $\varphi \theta \sim \iota(\psi)$ and $\iota(s).\varphi \theta = \iota(s).\iota(\psi)$

Our result implies that for large classes of components for which simple wrappers exist, the correctness of the GALS implementation is implied by the correctness of the global synchronous specification. Thus, no extra signalization is needed to insure semantics preservation (and no costly synthesis algorithms). The GALS implementation is correct by construction.

5. Conclusion

We introduced a new model for the representation of asynchronous implementations of synchronous specifications. The model covers implementations where a notion of global synchronization is preserved by means of signaling, and GALS implementations, where global synchronization is relaxed. The model takes into account computation and communication causality, and allows us to reason about semantics-preservation and absence of deadlocks in the GALS deployment of synchronous specifications. As the model captures the internal concurrency of the synchronous specification, our correctness criteria support implementations that are less constrained and more efficient than existing ones.

The results of section 4 suggest that our model offers a good abstraction level for reasoning about desynchronization. In particular, the level of detail is essential in revealing the intricate relation between (1) causal dependencies, concurrency and conflicts in the micro-step semantics of a synchronous specification and (2) the correctness (semantics preservation) of its GALS implementation.

Future work. Thanks to this new model, we are exploring the development of GALS circuits made of synchronous IPs. Our work aims at using asynchronous logic wrappers to encapsulate the components of a modular synchronous circuit into delay insensitive components. Our model seems wellsuited to analyze designs involving both synchronous and asynchronous circuit specifications.

We are also considering symbolic analysis techniques that would allow us to translate the theory detailed in this paper to high-level synchronous languages like Signal or Esterel, instead of simple finite state automata. The objective is to derive efficient algorithms transforming general high-level specifications into weakly endochronous ones.

A third research direction concerns the (still not sufficiently clear) relations between classical, macro-step synchronous models and more operational models like ours, or like the ones covering the implementations of various languages, especially when desynchronization is involved.

References

- N. Halbwachs, Synchronous programming of reactive systems. Kluwer Academic Publishers, 1993.
- [2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The Synchronous Languages Twelve Years Later," *Proceedings of the IEEE*, 2003.
- [3] D. Potop-Butucaru, R. de Simone, and J.-P. Talpin, "The synchronous hypothesis and synchronous languages," in *The Embedded Systems Handbook*, R. Zurawski, Ed., 2005, CRC Press. To appear.
- [4] C. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "The theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059– 1076, 9 2001.
- [5] N. Lynch and E. Stark, "A proof of the Kahn principle for input/output automata," *Information and Computation*, vol. 82, no. 1, pp. 81–92, 7 1989.
- [6] P. Caspi, A. Girault, and D. Pilaud, "Automatic distribution of reactive systems for asynchronous networks of processors," *IEEE Trans. on Software Engineering*, vol. 25, no. 3, pp. 416– 427, may/june 1999. [Online]. Available: http://www.inrialpes.fr/popart/people/girault/Publications/Tse99
- [7] J.-P. Talpin, P. Le Guernic, S. K. Shukla, R. Gupta, and F. Doucet, "Formal refinement checking in a system-level design methodology," *Fundamenta Informaticae*, 2004.
- [8] A. Benveniste, B. Caillaud, and P. Le Guernic, "Compositionality in dataflow synchronous languages: Specification and distributed code generation," *Information and Computation*, no. 163, pp. 125–171, 2000.
- [9] M. Singh and M. Theobald, "Generalized latency insensitive systems for gals architectures," in *Proceedings FMGALS2003*, Pisa, Italy, 2003.
- [10] D. Potop-Butucaru, B. Caillaud, and A. Benveniste, "Concurrency in synchronous systems," in *Proceedings ACSD 2004*, Hamilton, Canada, 2004.
- [11] H. Kopetz, Real-Time Systems, Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers, 1997.
- [12] P. Caspi, A. Girault, and D. Pilaud, "Automatic distribution of reactive systems for asynchronous networks of processors," *IEEE Transactions* on Software Engineering, vol. 25, no. 3, pp. 416–427, 1999.
- [13] T. Grandpierre and Y. Sorel, "From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *Proceedings MEMOCODE'03*, Mont Saint-Michel, France, 2003.
- [14] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. P. Sotiriou, "Handshake protocols for de-synchronization," in *Proceedings Async04*, Crete, Greece, 2004, pp. 149–158.
- [15] J. Cortadella, M. Kishinevsky, A. Kondratiev, L. Lavagno, and A. Yakovlev, *Logic Synthesis of Asynchronous Controllers and Interfaces.* Springer, 2002.
- [16] K. Yun and D. Dill, "Automatic synthesis of extended burst-mode circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 2, pp. 101–132, feb. 1999.
- [17] D. Potop-Butucaru and B. Caillaud, "Correct-by-construction asynchronous implementation of modular synchronous specifications," IRISA, Rennes, France, Tech. rep. PI1670, 2004, http://www.irisa.fr/bibli/publi/pi. To appear.
- [18] E. Stark, "Concurrent transition systems," *Theoretical Computer Science*, vol. 64, no. 3, pp. 221–269, 1989.
- [19] M. Mukund, "Petri nets and step transition systems," International Journal of Foundations of Computer Science, vol. 3, no. 4, pp. 443– 478, 1992.