Clock-Driven Distributed Real-Time Implementation of Endochronous Synchronous Programs

D. Potop-Butucaru
INRIA Rocquencourt, France
dumitru.potop@inria.fr

R. de Simone
INRIA Sophia Antipolis, France
rs@sophia.inria.fr

Y. Sorel
INRIA Rocquencourt, France
yves.sorel@inria.fr

J.-P. Talpin
INRIA Rennes, France
jean-pierre.talpin@inria.fr

ABSTRACT

An important step in model-based embedded system design consists in mapping functional specifications and their tasks/operations onto execution architectures and their ressources. This mapping comprises both temporal scheduling and spatial allocation aspects. Therefore, we promote an approach which starts from loosely-timed/asynchronous models and proceeds by refining them to fully synchronized ones, using so-called clock calculus techniques under the architecture constraints. In this paper we provide a modeling framework based on an intermediate representation format, called *clocked graphs*, for polychronous endochronous specifications, which are the ones that can be safely considered for deterministic distributed real-time implementation using static scheduling techniques. Our formalism allows the specification of both "intrinsic" correctness properties of the specification, such as causality and clock consistency, and "external" correctness properties, such as endochrony, which ensure compatibility with the desired implementation architecture, including both hardware and software aspects. Using this formalism, we define a new method for distributed real-time implementation of synchronous specification. The move from (endochronous) synchronous specification to real-time scheduled implementation is a seamless sequence of model decorations.

Categories and Subject Descriptors: D.3.4 [Programming languages]: Processors–*Code generation, Optimization*; D.4.7 [Operating systems]: Organization and Design–*Distributed systems, Real-time systems and Embedded systems*

General Terms: Algorithms

Keywords: synchronous model, intermediate representation, clock calculus, distributed real-time scheduling

1. INTRODUCTION

Synchronous reactive formalisms [1, 2] are modeling and programming languages used in the specification and analysis of safety-critical embedded systems. They comprise (synchronous) concurrency features, and are based on the Mealy machine paradigm: Input signals can occur from the environment, possibly simultaneously, at the pace of a given *global clock*. Output signals and state changes are then computed before the next clock tick, grouped as one *atomic reaction*, also called *execution instant*. Because common computation instants are well-defined, so is the notion of signal *absence* at a given instant. Reaction to absence is allowed, *i.e.*, a change can be caused by the absence of a signal on a new clock tick. Since component inputs may become local signals in a larger concurrent system, *absent* values may have to be computed and propagated, to implement correctly the synchronous semantics.

When an asynchronous, possibly distributed implementation is meant, where possibly distributed components communicate via message passing, the explicit propagation of all absent values may clog the system to a certain extent. A natural question arises: when can one dispose of such absent signal communications? Sufficient conditions, known as (weak) endochrony [3, 4, 5], have been introduced in the past to figure when the absent values can be replaced in the implementation by actual absence of messages without affecting its correctness and determinism. These conditions establish that compound reactions that are apparently synchronous can be split into independent smaller reactions that are asynchronously feasible in a confluent way (as coined by R. Milner [6]), so that the first one does not discard the second. This is also linked to the Kahn principles for networks [7], where only internal choice is allowed, in order to ensure that overall lack of confluence cannot be caused by input signal speed variations.

In this paper, we use these theoretical results as the foundation of a new method for distributed real-time implementation of synchronous specifications. Following the traditional approach in compiler development, our method is centered around a new intermediate representation format, called *clocked graphs (CG)*. On one hand, this format allows the faithful representation of specifications written in high-level synchronous languages like Esterel [8], Scade/Lustre [9], Signal [4], or discrete Scicos [10], including some structural information that can be used for efficient code generation purposes. On the other hand, a *CG* specification is close enough to the target machine code to allow fine manipulations such as scheduling, allocation, and optimization.

EMSOFT09, October 12–16, 2009, Grenoble, France.
Copyright 2009 ACM 978-1-60558-627-4/09/10 ...$5.00.

We focus in this paper on the real-time implementation of CG specifications on distributed hardware architectures (a large corpus of work exists on the translation of high-level synchronous specifications into intermediate representations related to ours).

The CG representation is based on the separation of computations, under the form of a *dataflow graph*, from control, under the form of *clocks* which identify the synchronous execution instants where the dataflow elements are executed. The two parts are interconnected, as all computations and communications are associated a clock defining their execution condition, and clocks may depend on values computed by the dataflow.

We stress the clock-driven specificity of our method. Clocks are logical activation condition defining the sequence of synchronous execution instants where some computation or communication takes place. While such activation conditions are traditionally represented with *events*, the activation events associated with our clocks are *logical*, in the sense that they are computed by the system itself at each execution instant, and not received from the exterior (*e.g.* under the form of interrupts). This model is nowadays common in computer science and engineering, even at hardware level where *clock gating* techniques are used to minimize power consumption.

The implementation of our CG specifications is defined as a sequence of transformations which gradually add information to the representation. The first implementation step is to make explicit the dependencies between the computations of the various logical clocks in a *clocked graph with dependencies (CGd)*. Then, we provide conditions determining when such a graph is endochronous. Endochrony being a scheduling-independence criterion, we know that for *endochronous clocked graphs (CGe)*, untimed asynchronous simulation is deterministic and equivalent with the synchronous semantics.

The last step of our implementation process produces a real-time schedule by assigning real-time dates and execution resources (processors and buses) to each element of a CGe. This approach, inspired from the AAA/SynDEx methodology [11], results in a *scheduled clocked graph (CGsch)* that describes the real-time scheduling of one clock cycle. The execution of the scheduled implementation is an infinite repetition of such execution instants. The difficulty here is to ensure the consistency between the logical clocks of the CGe specification and the real-time dates and resource allocations of the $CGsch$ scheduled implementation. We provide a scheduling algorithm ensuring by construction these consistency properties, and we compare it with the existing one of SynDEx. For simplicity, we restrict ourselves to simple target hardware architectures formed of a unique asynchronous message-passing broadcast bus that connects a set of (possibly different) sequential processors. We also make the simplifying assumption that the bus is reliable, which is realistic in certain settings for real-life buses such as CAN [12, 11], pending the use of fault tolerant communication libraries and a reliability analysis that are not covered in this paper. Our scheduling technique also works for static TDMA buses such as TTA or FlexRay (static segment), but does not take full advantage of the time-triggered nature of these architectures. Future work will cover better implementations on time-triggered and heterogenous architectures and more complex interconnect topologies.

2. RELATED WORK

Our work is closely related to the work of Benveniste *et al.* on tagged systems [13]. Indeed, the main originality of our work – the combination of logical clocks and "real" time in a single formalism – occurs in the form of a time refinement, which can be modeled using tag refinements. The difference is that we also need to deal with spatial allocation issues, causality, and that we focus on specific time models in order to give efficient implementation algorithms.

The second main inspiration of this work is the AAA/SynDEx methodology for distributed real-time implementation of synchronous specifications [11]. The scheduling approach we use here is much inspired from the SynDEx one. The difference is that SynDEx does not treat clocks as first-class citizens, allowing their definition only through structured dataflow constructs. By consequence, execution conditions of a specification are often pessimistic, which also pessimizes the implementation. Our work provides, on one hand, a more flexible language supporting better specifications and implementations and on the other hand, a more general formal framework for reasoning about the correctness of such implementation processes.

The third main inspiration of our work is the previous work on endochrony, already mentioned above. Our work extends this line by proposing a formalism combining the manipulation of endochronous logical clocks with that of "real" time. In this sense, it goes beyond results on the implementation of the Signal/Polychrony language [14]. Also related to Signal are the results of Kountouris and Wolinski [15] on the scheduling of *hierarchical conditional dependency graphs*, a formalism allowing the representation of data dependencies and execution conditions. The main difference with our work is that we focus on timed and distributed implementation, whereas Kountouris and Wolinski focus on optimizing mono-processor untimed implementations.

The intermediate representation proposed in this paper is also inspired from the large corpus of work concerning clock analysis and the compilation of synchronous languages.

We also mention here the large corpus of work on optimized distributed scheduling of dataflow specifications onto time-triggered architectures [16, 17]. The work of Caspi *et al.* on the distributed scheduling of Scade/Lustre specifications onto TTA-based architectures [18] is closest to ours in terms of proximity of the specification formalism. The main difference is that in TTA-based systems communications must be realized in time slots that are statically assigned to the various processors.

3. INTERMEDIATE REPRESENTATION

This section defines the syntax of clocked graphs. A clocked graph is a dataflow graph $\mathcal{G}$ whose elements (nodes and arcs) are labelled with clocks. We first introduce the clock definition language, which is the focus of our approach. Then, we introduce the dataflow constructs, and conclude with an example.

3.1 Clocks

Clocks represent execution conditions defining when a computation or communication is performed, or when some data is available to be used in computations. In our purely synchronous setting, clocks are functions associating to each execution instant a value of 1 (*true*, active) or 0 (*false*, inactive). A computation or communication whose clock is c

instant	1	2	3	4	5	6	7	8	9
output port x	5	4	3	2	1	0	-1	-2	-3
true	1	1	1	1	1	1	1	1	1
$x > 0$	1	1	1	1	1	0	0	0	0
$(01)^*$	0	1	0	1	0	1	0	1	0
$0(01)^*$	0	0	1	0	1	0	1	0	1
$(x > 0) \wedge (01)^*$	0	1	0	1	0	0	0	0	0
$(01)^*.0(01)^*$	0	0	0	0	1	0	0	0	1

Figure 1: Examples of clocks

will be executed in the execution instants where c is *true*. We define here the syntax allowing the definition of clocks.

Our choice of operators was determined by the need to represent inside our format certain constructs used at specification and implementation time. Regular repetitions of active and inactive instants, such as data-independent counters, are specified using ultimately periodic infinite Boolean words of the form $w_t(w_p)^*$, where w_t, w_p are finite words over $\{0,1\}$, and $w_t(w_p)^*$ stands for the infinite word starting with w_t and continuing with an infinite sequence of w_p. For instance, $11(001)^*$ stands for $11001001001\ldots$. The constant clocks are denoted $\mathit{true} = (1)^*$ and $\mathit{false} = (0)^*$.

Conditional data-dependent computations are represented using Boolean expressions over the output ports of the dataflow nodes.[1] For instance, if o is an integer output port of dataflow node n, $o = 3$ is the clock defining the execution instants where o has a value of 3. Similarly, $o_1 = o_2$ intuitively defines execution instants where o_1 equals o_2. Much effort will be dedicated in the following sections to ensure that the arguments needed to compute such clocks are available when the Boolean expressions are evaluated.

These elementary clocks (constants, ultimately periodic words, and Boolean atoms) can be composed using:

- The Boolean combinators $\wedge$, $\vee$, and $\neg$, which work instant-wise. For instance, $c_1 \wedge c_2$ is true at execution instants where both c_1 and c_2 are true. We also denote with $c_1 \setminus c_2 = c_1 \wedge \neg c_2$ the difference operators on Booleans and clocks.

- The subclock operator $c_1.c_2$, which evaluates c_2 only on instants where c_1 is true. The subclock operator is different from a simple conjunction. When c_2 is a ultimatelly periodic word, we advance in c_2 only when c_1 is true. When c_2 is a Boolean expression, we need its arguments only when c_1 is true.

Fig. 1 gives examples of clocks, both elementary and composed.

3.1.1 Clock semantics

In the synchronous model, each variable (output port of a dataflow node[1]) is either absent in an execution instant, or is assigned a unique value that will be used in computations throughout the instant. To facilitate notations, we shall append to the data domains of all the output ports of the specification a special value $\perp$ that denotes the absence of a value in a given execution instant. When a dataflow node is not executed in a given instant, all its outputs are set to $\perp$. Given a domain $\mathcal{D}$, we shall denote $\mathcal{D}^\perp = \mathcal{D} \cup \{\perp\}$.

Using this notation, we can represent each clock c with a predicate defining the instants where the clock is active:

$$c : \mathbb{N} \times \prod_{o \in \bigcup_{n \in \mathcal{N}} \mathcal{O}(n)} \mathcal{D}_o^\perp \to \{0,1\}$$

where $\mathcal{N}$ is the set of all dataflow nodes of the specification, $\mathcal{O}(n)$ is the set of output ports of node n (defined in Section 3.2), and $\mathbb{N}$ is the set of positive integer indices of execution instants.

This interpretation of clocks as predicates naturally organizes the clocks in a Boolean algebra where the Boolean combinators have their usual meaning.[2] We denote with $\leq$ the partial order between clocks, where $c_1 \leq c_2$ means that at each execution instant c_2 is true whenever c_1 is true. Given that the subclock operator is non-standard, we list here some of its properties: $\mathit{true}.c = c.\mathit{true} = c$, $\mathit{false}.c = c.\mathit{false} = \mathit{false}$; associativity: $(a.b).c = a.(b.c)$; left distributivity of the binary Boolean operators: $c.(c_1 \; op \; c_2) = c.c_1 \; op \; c.c_2$.

Determining clock inclusion or equality (a form of satisfiability modulo theories) is undecidable in the general case, because it involves the complexity of dealing with the functions of the dataflow nodes. However, various sufficient conditions (and associated decision algorithms, known as "clock calculi") have been proposed for clock inclusion and equality, ranging from simple syntactical ones (as in SynDex [11]), to BDD-based techniques like those of Kountouris and Wolinski [15]. All scheduling algorithms proposed later in this paper are designed to function with such sufficient conditions, instead of exact equality and inclusion tests.

3.2 Clocked graphs

A clocked graph is a pair $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ formed of the set of dataflow *nodes* $\mathcal{N}$ and the set of *arcs* $\mathcal{A}$. Each node $n \in \mathcal{N}$ has a set of *named input ports* $\mathcal{I}(n)$, a set of *named output ports* $\mathcal{O}(n)$, and a *clock* $\mathit{clk}(n)$. The name of a port p is denoted $\mathit{name}(p)$, and we assume that the ports of a node have all different names. The port of name name of node n shall be denoted with $n.\mathit{name}$. Each input and output port p is assigned a data type (a domain) $\mathcal{D}_p$.

Each dataflow arc $a \in \mathcal{A}$ connects one output port denoted $\mathit{src}(a)$ to one input port denoted $\mathit{dest}(a)$ upon a communication condition (a clock) denoted $\mathit{clk}(a)$. Therefore:

$$\mathcal{A} \subseteq \left(\bigcup_{n \in \mathcal{N}} \mathcal{O}(n) \right) \times \left(\bigcup_{n \in \mathcal{N}} \mathcal{I}(n) \right) \times \mathcal{C}$$

where $\mathcal{C}$ denotes the set of all clocks that can be defined using the previously-defined syntax. Each arc has a data domain $\mathcal{D}_a$ and we require that for all arc $a \in \mathcal{A}$ we have:

$$\mathcal{D}_{src(a)} = \mathcal{D}_{dest(a)} = \mathcal{D}_a$$

There are two types of dataflow nodes: computations and delays. Computation nodes represent atomic stateless computations (function calls) that are completed inside one execution cycle. The state of the system is maintained by the delays, which allow data to be passed from one execution cycle to the next (but otherwise perform no computation). We denote with $\mathcal{N}^C$ the set of computation nodes, and with $\mathcal{N}^\Delta$ the set of delays.

For each computation node $n \in \mathcal{N}^C$, we assume an implementation (a piece of code in a library) is provided that

[1] All dataflow constructs are defined in Section 3.2. We also explain there why only node outputs are considered.

[2] Note that our interpretation means that the set of clocks is a tag system, in the sense of Benveniste *et al.* [13].

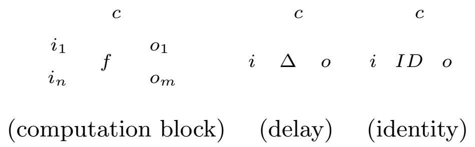

Figure 2: Basic dataflow nodes (identity is a special kind of computation block)

takes as input one value for each input port $i \in \mathcal{I}(n)$ and produces one value for each output port $o \in \mathcal{O}(n)$. The computation needs not be deterministic, to allow the modeling of sensors. It is assumed, however, that no hidden dependencies (e.g. side effects on unspecified variables) exist between implementations of different nodes, or between successive computations of a given computation node. The computation is assumed to be atomic, in the sense that all inputs are read before performing the computation and before all output is produced.

One particular type of computation node is the identity node that has one input port, named i, one output port, named o, and whose function is identity (copy the value from the input port to the output port). To an identity node n we associate its domain $\mathcal{D}_n$, which is the domain of its input and output ports.

A delay $\delta \in \mathcal{N}^{\Delta}$ of clock $clk(\delta)$ specifies the passing of values between the successive execution instants defined by clock δ. A delay δ has a data domain $\mathcal{D}_\delta$, one input port i of type $\mathcal{D}_\delta$, one output port o of type $\mathcal{D}_\delta$, and one initial value $\delta_0 \in \mathcal{D}_\delta$. In the first execution instant where $clk(\delta)$ is true, δ produces δ_0 through the output port, and reads a new value through the input port. In subsequent execution instants where $clk(\delta)$ is true, the output port produces the value previously input through i.

We use for dataflow nodes the simple graphical representations of Fig. 2. Delay nodes are labelled with Δ, identity nodes are labelled with ID, and other computation nodes (including non-deterministic ones) are labelled with their computing function.

This paper being focused on code generation and scheduling problems that are of a global nature, we do not give provisions for interconnecting dataflow specifications. Our formalism only allows the specification of complete systems, where acquisition of data and actuation is represented with computation nodes (non-deterministic for the sensing part). However, extending the formalism to allow composition can be easily done.

Derived dataflow blocks.

The semantics of our format will be directly defined for the previously-defined "primitive" dataflow constructs. However, we allow the definition of derived dataflow constructs that serve as syntactic sugar at specification time, and which may be directly used (for efficiency purposes) by analysis and optimization algorithms. The definition of a derived dataflow block includes the syntax of the block, and its semantics under the form of an expansion over primitive dataflow. Fig. 3 provides one example of derived block: the classical memory cell that can be written and read at different rates (as opposed to the synchronizing message passing philosophy of delays).[3] Its definition by expansion is given in Fig. 3, and also shows the form of our dataflow specifications.

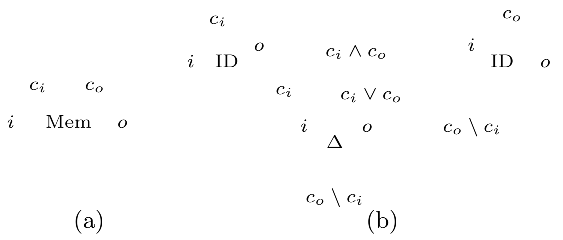

Figure 3: The memory cell written on clock c_i and read on clock c_o: graphical representation (a) and semantics by expansion (b).

3.3 Example

We give in Fig. 4 the intermediate representation corresponding to the simple SynDEx synchronous specification of Fig. 5. The specification represents a system with two switches (Boolean inputs) controlling its execution: high-speed (HS) vs. low-speed ($\neg HS$), and fail-safe (FS) vs. normal operation ($\neg FS$). In the low-speed mode, more operations can be executed, whereas in the fail-safe mode the operation that gets executed (N) does not use any of the inputs, because the sensors or treatment chain are assumed to be faulty (control is done using default values).

The behavior of our CG representation is: Nodes FS_IN and HS_IN have clock *true*, so they are executed at each execution cycle to read FS and HS. If $HS = false$ then clock $\neg HS$ is *true* for the instant, which triggers the execution of F1, followed by F2 and F3. Otherwise, execute G (on clock HS). Clock dependencies, such as clock HS depending on the output port $HS\_IN.HS$, are not explicit. Both F1 and G are computing through their output ports named ID the SynDEx-level output value ID of the hierarchical conditional node C1. The execution of M (on clock $\neg FS$) can start after ID is received from either $F1$ or G. The execution of N can start as soon as we can determine that FS is *true* for the instant.

Dataflow blocks having no dependency between them can be executed in parallel. For instance, if $FS = true$ then N can be executed as soon as FS is read, independently of the execution of $F1$, $F2$, $F3$, or G. On the contrary, the computation of M must wait until both FS and ID have arrived.

3.4 Clocked graph semantics

The previously-defined formalism allows the representation of programs written in languages such as Esterel [8], Scade/Lustre [9], Signal [4], or discrete Scicos [10]. In particular, it is flexible enough to allow the definition of semantics compatible to the semantics of the aforementioned languages.

[3] The functioning of the memory element is as follows: Whenever reading and writing occur in the same synchronous instant, the input value is directly copied to the output, hence the $c_i \wedge c_o$ clock on the arc between the two identity blocks. When reading occurs without writing (clock $c_o \setminus c_i$), the value is taken from the delay element. Each write updates the value of the delay (clock c_i). In instants where the cell is read, but not written (clock $c_o \setminus c_i$), we need to explicitly refresh its delay value (the loopback arc from the output to the input of the delay).

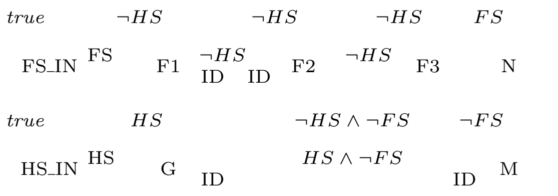

Figure 4: Example of specification in our formalism

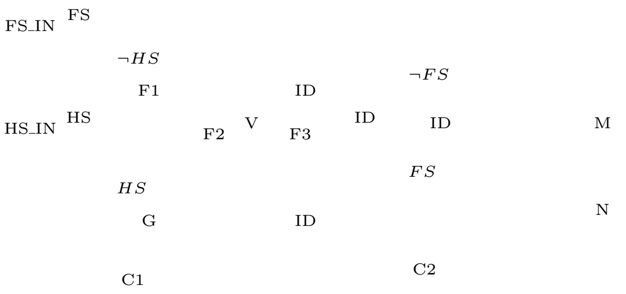

Figure 5: Corresponding SynDEx specification

However, our goal is the definition of efficient implementations, so we are not only interested in specification correctness. Indeed, the remainder of the paper is mostly dedicated to the definition *at the level of our intermediate representation* of:

- "Intrinsic" correctness properties of the specification that should be included in all the semantics assigned to the format. Classical correctness properties used in synchronous program analysis, such as causality and clock consistency, fall in this category.

- "External" correctness properties ensuring compatibility with the desired implementation architecture, including both hardware and software aspects. Such properties are:
  - The absence of reaction to signal absence [19], which ensures that a distributed implementation can be constructed that uses absence of messages to encode the signal absence of the synchronous model.
  - Endochrony [3], which ensures the existance of a static schedule, which can be computed offline.
  - Real-time schedulability [20], which ensures that a schedule exists satisfying the desired real-time constraints.

These properties remain compatible with the semantics of the high-level programming languages, in the sense that they identify (efficiently) implementable sub-classes of semantically correct programs. At the same time, however, the definition of these properties requires the refinement of the *CG* specification into a real-time implementation.

In this section we only define some of the "intrinsic" correctness properties corresponding to the synchronous hypothesis. The next sections shall continue with the introduction of endochrony and scheduling properties.

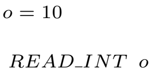

Figure 6: A globally non-causal example

To ensure compliance with the synchronous hypothesis and the atomicity assumption for node executions, the computation of the dataflow nodes must satisfy at each execution instant 3 correctness properties:

No uninitialized data: All the inputs of a node n are computed and transmitted in instants where n is executed. Formally, we require that:

- for all $n \in \mathcal{N}$ and for all $i \in \mathcal{I}(n)$ we have:

$$clk(n) \leq \bigvee_{a \in \mathcal{A}, dest(a)=i} clk(a)$$

- for all $a \in \mathcal{A}$ we have $clk(a) \leq clk(src(a))$.

Note that we allow data to be computed and transmitted in instants where it is not needed.

Single assignment: Each input of a node n is received from at most one source at each execution instant (no write conflict is possible). Formally, we require that for all $a_1, a_2 \in \mathcal{A}$ with $dest(a_1) = dest(a_2)$ we have $clk(a_1) \wedge clk(a_2) = false$.

No causality cycle: All cycles in the dataflow graph either contain a delay node, or have the conjunction of all conditions of all arcs equal to *false*. This condition, specific to the synchronous approach, ensures that the computation of each cycle can be performed in bounded time [1].

A good abstraction of the last condition is the absence of cycles that contain no delay. However, the acyclicity of the dataflow is not sufficient to ensure by itself the causal correctness of a specification. Indeed, the computation of the clocks may induce additional causal dependencies. For instance,we want to reject specifications such as the one in Fig. 6, where the output of a sensor (o) is used to compute its clock. We deal with these issues in the next section.

4. EFFICIENT IMPLEMENTATION: USING ENDOCHRONY

We explained in the introduction that endochrony [3] is a scheduling-independence property allowing us to encode signal absence with absence of messages. By consequence, endochrony allows us to perform no computation in parts of the system that are semantically inactive (something which is not possible in the general synchronous model). Moreover, endochrony ensures that an asynchronous simulation of the specification gives the same results as the synchronous one. Thus, endochrony identifies synchronous specifications that allow a deterministic and efficient implementation over asynchronous implementation architectures.

We introduce in this section a notion of *endochronous clocked graph (CGe)* that combines endochrony with dataflow acyclicity using a notion of *endochronous clock* which we define. The choice is natural, because (1) endochrony is a sufficient property ensuring the static schedulability of the computations in a synchronous specification, and (2) global acyclicity is a sufficient condition which can be checked using low-complexity algorithms and ensures the absence of causality cycles.

4.1 Clock with dependencies

The definition of endochronous clocked graphs is based on associating to each clock of the clocked graph a data *support* – the set of all the output ports used in its computation, along with the clocks defining the instants where these output ports are needed. In practice, this means that each dataflow element (node or arc) x is assigned not just a clock $clk(x)$, but also a support $supp(x)$. We call *clocks with dependencies* the pairs $\langle\, clk(x)\ ,\ supp(x)\,\rangle$ formed of a clock and its support.

The support of a clock with dependencies c is a set of pairs $o@c_o$, where o is an output port of some node n, and c_o is a clock defining the instants where the value of o is used in the computation of c. We call the pair $o@c_o$ the sampling of o on the clock c_o. Intuitively, the support of a clock gives sufficient data for some algorithm to compute the clock. For instance, a good support for the clock $c = (o_1 = 3) \wedge (o_2 = 5)$ is $\{o_1@true, o_2@(o_1 = 3)\}$ which corresponds to the following computation of c:

```
c = false ;
read(o1) ;
if(o1 == 3){
  read(o2) ;
  if(o2 == 5) c = true
}
```

Note that a given clock can have several supports. For instance, the clock c defined above also accepts $\{o_2@true, o_1@(o_2 = 5)\}$ as a support.

Of course, not all pairs formed of a clock and a support are meaningful. For instance, $\langle$ a=3 , $\{a@(01)^*\}$ $\rangle$ is not, because a is needed for the clock computation at all instants, not just once every two instants, as it is specified in the support. To identify meaningful clocks, we introduce the generation relation $s \vdash c$ stating that c can be computed from the output port samplings of s. This relation is inductively defined by the following rules:

1. $\emptyset \vdash w_i(w_p)^*$, for all $w_i, w_p \in \{0,1\}^*$
2. $\{o_1@true, \ldots, o_k@true\} \vdash B(o_1, \ldots, o_k)$, for all Boolean function B of arguments $o_1, \ldots, o_k$
3. if $s \vdash c$ then $s \vdash \neg c$
4. if $s_1 \vdash c_1$ and $s_2 \vdash c_2$ then $s_1 \cup s_2 \vdash c_1\ op\ c_2$ for all binary Boolean operator *op*
5. if $s_1 \vdash c_1$ and $s_2 \vdash c_2$, then $s_1 \cup \{o@c_1.c \mid o@c \in s_2\} \vdash c_1.c_2$
6. if $s \vdash c$ and $o@c_1 \in s$ and $c_2 \geq c_1$, then $(s \setminus \{o@c_1\}) \cup \{o@c_2\} \vdash c$
7. if $c_1 = c_2$ and $s \vdash c_1$ then $s \vdash c_2$.

The first 5 rules build a support for any clock by inductively following its syntax. Rule 6 states that having more information than is strictly necessary does not affect computability. Rule 7 is the most difficult. It states that if two syntaxes c_1 and c_2 represent the same clock in the clock algebra, then a support generating c_1 also generates c_2 (meaning that the algorithm used to compute c_1 can be used for c_2).

4.2 Endochronous clock

Note that the definition of a clock with dependencies may involve recursive computations of other clocks. To ensure that the recursive computation process is finite, we introduce the notions of endochronous support and endochronous clock.

We shall say of a support s that it is *endochronous* whenever we can associate to each $o@c \in s$ a subset $dep_s^{(\mathcal{N},\mathcal{A})}(o@c)$ of s such that (1) $dep_s^{(\mathcal{N},\mathcal{A})}(o@c) \vdash c$ and (2) the dependency sets induce a "dependency" partial order over the support set s (there are no cyclic dependencies). Intuitively, this means that there are some samplings in s whose clocks depend on no output ports. Once read, the corresponding output ports allow the computation of new clock samplings, and so on until all the clocks of the samplings have been computed and all corresponding output ports read. No cyclic computation is possible.

We say that a clock with dependencies $\langle$ c , s $\rangle$ is endochronous whenever s is endochronous and $s \vdash c$. Note that the first 5 rules of the previous section can be used to automatically transform any clock into an endochronous one, by assigning it a support.

4.3 Endochronous clocked graph

Consider now a clocked graph where all clocks are clocks with dependencies. We introduce a preorder $\preceq$ over the the ports and arcs of the dataflow graph, defined by the generators:

- $p \preceq a$ for each outgoing arc a of a port p
- $a \preceq p$ for each incoming arc a of a port p
- $o \preceq x$ for each output port o and port or arc x such that $supp(clk(x))$ contains $o@c$ for some c
- $i \preceq o$ for each input port i and output port o of a node n that is not a delay

With this definition, we shall say that the clocked graph is endochronous if, by definition:

- $\preceq$ is a partial order relation, and
- if o is an output of node n, then any sampling $o@c$ must have $c \leq clk(n)$

This criterion ensures a strong form of causal correctness, amounting to acyclicity (including the computation of the clocks), and ensuring the existence of a static schedule of all the operations (including the computation of the clocks, which implies that all clocks are endochronous).

5. SCHEDULING CLOCKED GRAPHS

In this section, we provide a method for bulding distributed real-time implementations of endochronous clocked graphs. The definition of the method is intended to show that our model supports a realistic real-time implementation technique. Therefore, it is restricted to the simple bus-based distributed architectures defined in the introduction. Extensions to cover more complex implementation architectures with more complex interconnect topologies and different bus types (time-triggered, unicast) are the object of ongoing work.

We follow the SynDEx approach [11] by computing a schedule for one execution cycle, the global scheduling being an

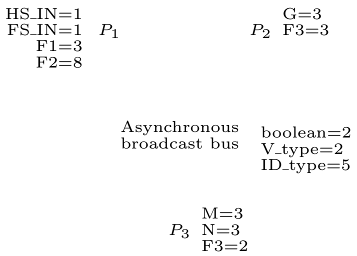

Figure 7: Hardware architecture

infinite repetition of the scheduling of a cycle. This corresponds to the case where the computations of the different cycles do not overlap in time in the real-time implementation.[4] Our scheduling method uses a greedy heuristic inspired from AAA/SynDEx, but deals with clocks in a finer way, potentially resulting in better real-time schedules.

The remainder of the section is structured in four main parts. We first define the timing information used as input for our scheduling technique. Then, we introduce a model of *scheduled clocked graph (CGsch)*, obtained by decorating endochronous clock graphs with spatial and temporal allocation information. Third, we define sufficient conditions ensuring that the allocation information of an *CGsch* is compatible with both the clocks of the initial endochronous clocked graph and the underlying hardware architecture. Finally, we introduce a simple scheduling technique generating schedules that satisfy the previously-defined correctness (compatibility) properties. The last part includes a brief comparison with the output of AAA/SynDEx.

5.1 Timing information

We explained in the introduction that we consider in this paper simple distributed architectures formed of a unique asynchronous or static TDMA message-passing reliable broadcast bus denoted $\mathcal{B}$ that connects a set of sequential processors $\mathcal{P} = \{P_i \mid i = 1..n\}$. Fig. 7 pictures an architecture formed of 3 processors connected to the central bus.

The architecture is decorated with timing information that specifes:

- The dataflow functions that can be executed by each processor, and their duration on that processor. Each processor has a list of timing information of the form "node_name=duration".
- The communication operations that can be executed by the bus (the data types that can be sent atomically), and the durations of the communications, assuming the bus is free. The bus has a list of timing information of the form "data_type=duration".

The durations provided for computations and communications must be upper bounds obtained by some worst-case execution/transmission time (WCET/WCTT) analysis. Timing information for one atomic node can be present on several processors, to denote the fact that the node can be executed by several processors. For instance, node F3 can be executed on P2 with duration 3, and on P3 with duration 2. We assume that writing a delay, reading a delay, and local communications (that do not use the bus) take no time. We shall denote with $d_P(n)$ the duration of computation node n on processor P. To represent the fact that a processor P cannot perform computation n, we set $d_P(n) = \infty$. We denote with $d_{\mathcal{B}}(\mathcal{D})$ the duration of a communication of a value of type $\mathcal{D}$ over the bus (in the absence of all interference).

5.2 Scheduled clocked graph

In this section we introduce our model of scheduled clocked graph, and give sufficient conditions ensuring that it correctly implements the semantics of the given endochronous clocked graph.

By definition, a scheduled clocked graph is an endochronous clocked graph $(\mathcal{N}, \mathcal{A})$ along with a schedule $\mathcal{S}$ of its elements over the resources of the chosen architecture. Such a schedule is formed of:

- An allocation of the delays to processors determining on which processor the delay value is stored between execution cycles: $\mathcal{S}_\Delta : \mathcal{N}^\Delta \to \mathcal{P}$.
- A set of scheduled functions assigning a processor and a real-time date (an integer) to each computation of the dataflow: $\mathcal{S}_\mathcal{C} : \mathcal{N}^\mathcal{C} \to \mathcal{P} \times \mathbb{N}$.
- A set of scheduled communications assigning to each arc of the dataflow the emitter processor, a real-time date, and an effective communication clock:
$$\mathcal{S}_\mathcal{A} : \mathcal{A} \to \mathcal{P} \times \mathbb{N} \times \mathcal{C}$$
- For each element of the graph $x \in \mathcal{N} \cup \mathcal{A}$, a set of scheduled communications assigning to each sampled output of the support $\mathit{supp}(x)$ an emitter processor, a real-time date, and an effective communication clock:
$$\mathcal{S}_x : \mathit{supp}(x) \to \mathcal{P} \times \mathbb{N} \times \mathcal{C}$$

For convenience, we denote with:

- t_x the real-time date associated by $\mathcal{S}$ to any computation node, arc, or sampled output x of some support.
- $\mathit{Res}(x)$ the processor associated with each node, arc, or sampled output of some support (the execution processor for dataflow functions, the storage processor for delays, and the emitter processor for arcs and support elements).
- $\mathit{e\_clk}(x)$ the effective communication clock associated with an arc or sampled output.
- $d(n) = d_{\mathit{Res}(n)}(n)$ the duration of a scheduled computation node n.
- $d(x) = d_{\mathcal{B}}(\mathcal{D}_x)$ the duration of the scheduled communication of an arc or sampled output x.

We say that the schedule $\mathcal{S}$ is partial when either $\mathcal{S}_\Delta$ or $\mathcal{S}_\mathcal{C}$ is partial.[5] When $\mathcal{S}$ is a partial schedule of $(\mathcal{N}, \mathcal{A})$, then we allow its incremental completion using the following operator:

[4] The results of this paper can be extended to the case where cycles can overlap by considering modulo scheduling techniques.

[5] Communication arcs may remain unassigned, for instance when they represent local communications for which no code is necessary.

- If $\mathcal{S}_\Delta(\delta)$ is undefined, then we denote with $\mathcal{S}\cup\{\delta \mapsto P\}$ the partial schedule with $(\mathcal{S}\cup\{\delta\mapsto P\})_\Delta(\delta) = P$, and which equals $\mathcal{S}$ everywhere else.
- Similarly, we define $\mathcal{S}\cup\{n\mapsto(P,t)\}$ whenever $\mathcal{S}_\mathcal{C}(n)$ is undefined, $\mathcal{S}\cup\{a\mapsto(P,t,c)\}$ when $\mathcal{S}_\mathcal{A}(a)$ is undefined, and $\mathcal{S}\cup\{o@c_o\mapsto(P,t,c)\}$ when $\mathcal{S}_x(o@c_o)$ is not defined.

Our definition implies that computations are executed on the specification clock (the schedule defines no new clock). However, communications can be realized on a different clock, to avoid cases where a data is transmitted twice on the bus in the same execution instant. In this paper, we interpret the real-time date associated to communications and computations as the latest (worst-case) real-time date at which the execution of the communication or computation will start.

Note that the definition of $\mathcal{S}_\mathcal{A}$ and the $\mathcal{S}_x$ implicitly assumes that we make no use of specialized synchronization messages, nor data encoding. We also assume that each operation is scheduled exactly once in $\mathcal{S}$, meaning that no optimizing replication such as inlining is done. This hypothesis is common in real-time scheduling.

5.3 Consistency of an $CGsch$

The properties of this section formalize the compatibility between the dates and resource allocations of a scheduled clocked graph and the logical clocks of the underlying endochronous clocked graph. These properties depend on chosen target architectures, and extending the scheduling techniques to new architectures consists in defining new consistency properties.

5.3.1 Availability functions

Consider a schedule $\mathcal{S}$ of the endochronous clocked graph $(\mathcal{N},\mathcal{A})$. The execution condition defining the execution cycles where the value of an output port o is sent on the bus before date t is denoted $\mathit{clk}^{\mathcal{S}}(o,t,\mathcal{B})$, and is defined as the union of all the clocks $e\_\mathit{clk}(x)$, where e ranges over:

- the arcs $a\in\mathcal{A}$ with $\mathit{src}(a)=o$ that have been scheduled such that $t_a + d_\mathcal{B}(a)\leq t$
- the sampled ports $o@c\in\mathit{supp}(y)$ (for some arc or node y) that have been scheduled ($\mathcal{S}_x(o@c)$) such that $t_{o@c}+d_\mathcal{B}(o@c)\leq t$.

Obviously, $\mathit{clk}^{\mathcal{S}}(o,t,\mathcal{B})$ is the execution condition giving the cycles where o is available system-wide at all dates $t'\geq t$.

Assuming o is a port of node n, we also define the execution condition $\mathit{clk}^{\mathcal{S}}(o,t,P)$ defining the cycles where o is available on P at date t:

- If n is not allocated on P ($\mathit{Res}(n)\neq P$), then $\mathit{clk}^{\mathcal{S}}(o,t,P) = \mathit{clk}^{\mathcal{S}}(o,t,\mathcal{B})$
- If n is a delay node allocated on P ($\mathit{Res}(n)=P$), then $\mathit{clk}^{\mathcal{S}}(o,t,P)=\mathit{clk}(n)$, meaning that the value is available from the beginning of all execution instants of n.
- If n is a computation node allocated on P at date t_n, then $\mathit{clk}^{\mathcal{S}}(o,t,P)$ is $\mathit{clk}(n)$ if $t\geq t_n+d_P(n)$, and *false* if not.

If c is a clock with $c\leq \mathit{clk}(n)$, then we denote with $\mathit{ready\_date}(P,o,c)$ the minimum t such that $\mathit{clk}^{\mathcal{S}}(o,t,P)\geq c$, and with $\mathit{ready\_date}(\mathcal{B},o,c)$ the minimum date t such that $\mathit{clk}^{\mathcal{S}}(o,t,\mathcal{B})\geq c$.

Note that $\mathit{clk}^{\mathcal{S}}(o,\infty,R)$ is the clock giving the instants where o becomes available at some point on resource R.

5.3.2 Consistency properties

The following properties define the consistency of a static schedule $\mathcal{S}$ with the underlying endochronous clocked graph $(\mathcal{N},\mathcal{A})$ and the timing information that was provided. This concludes the definition of our model of real-time schedule, and allows us to reason about the correctness of the simple scheduling algorithm defined in the next section.

Exclusive resource use.

A resource (processor or bus) cannot be used by more than one operation at a time. Formally:

- *On processors:* If n_1 and n_2 are different scheduled computation nodes with $\mathit{clk}(n_1)\wedge\mathit{clk}(n_2)\neq\mathit{false}$, then either $t_{n_1}\geq(t_{n_2}+d_P(n_2))$ or $t_{n_2}\geq(t_{n_1}+d_P(n_1))$.
- *On the bus:* If x_1 and x_2 are different scheduled arcs or sampled outputs with $e\_\mathit{clk}(x_1)\wedge e\_\mathit{clk}(x_2)\neq false$, then either $t_{x_1}\geq(t_{x_2}+d_\mathcal{B}(x_2))$ or $t_{x_2}\geq(t_{x_1}+d_\mathcal{B}(x_1))$.

Causal correctness.

Intuitively, to ensure causal correctness our schedule must ensure in a static fashion that when a computation or communication is using the value of an output port o at time t on execution condition c, the port value has been computed or transmitted on the bus at a previous time and on a greater execution condition. Formally:

1. If $\mathcal{S}_\mathcal{C}(n)=(P,t)$ is defined for a node n, then:
   - $\mathit{clk}^{\mathcal{S}}(o,t,P)\geq c$ for all $o@c\in\mathit{supp}(n)$
   - $\mathit{clk}^{\mathcal{S}}(o,t,P)\geq c$ for all $o@c\in\mathit{supp}(a)$ if $\mathit{dest}(a)$ is an input port of n
   - $\mathit{clk}^{\mathcal{S}}(\mathit{src}(a),t,P)\geq\mathit{clk}(a)$ for all arc a with $\mathit{dest}(a)$ being an input port of n
2. To derive the rule ensuring that a delay has enough input at the end of an instant, we simply set in the previous rules the date to ∞. More precisely, if $\mathcal{S}_\Delta(n)=P$ is defined for a delay node δ, then:
   - $\mathit{clk}^{\mathcal{S}}(o,\infty,P)\geq c$ for all $o@c\in\mathit{supp}(\delta)$
   - $\mathit{clk}^{\mathcal{S}}(o,\infty,P)\geq c$ for all $o@c\in\mathit{supp}(a)$ if $\mathit{dest}(a)$ is an input port of δ
   - $\mathit{clk}^{\mathcal{S}}(\mathit{src}(a),\infty,P)\geq\mathit{clk}(a)$ for all arc a with $\mathit{dest}(a)$ being an input port of δ
3. If $\mathcal{S}_\mathcal{A}(a)=(P,t_a,c_a)$ is defined for an arc a with $c_a\neq\mathit{false}$ and if n_a is the source node of a, then:
   - $\mathit{clk}^{\mathcal{S}}(o,t,\mathcal{B})\geq c$ for all $o@c\in\mathit{supp}(a)$
   - $\mathcal{S}_\mathcal{C}(n_a)$ is defined and $\mathit{Res}(n_a)=P$ and $t_{n_a}+d_P(n_a)\leq t_a$
4. Assume that $x\in\mathcal{N}\cup\mathcal{A}$ and that $o@c\in\mathit{supp}(x)$. Then, if $\mathcal{S}_x(o@c)=(P,t,c_1)$ is defined with $c_1\neq\mathit{false}$ we have:

- c_1 is generated by the data that has already transited the bus before date t (the set of all $o'@clk^{\mathcal{S}}(o',t,\mathcal{B})$ where o' ranges over all the output ports of the system).
- $clk^{\mathcal{S}}(o',t,\mathcal{B}) \geq c'$ for all $o'@c' \in dep_{supp(x)}^{(\mathcal{N},\mathcal{A})}(o@c)$

Function 1 SchedulingDriver

```
Input: $(\mathcal{N},\mathcal{A})$: endochronous clocked graph
  timing information
Output: $\mathcal{S}$: full schedule
  $\mathcal{S} \leftarrow \emptyset$
  while exists $n$ with $\mathcal{S}_C(n)$ undefined do
    choose such an $n$ minimal in the sense of $\preceq$
    for all $P$ processor with $d_P(n) \neq \infty$ do
      $(\mathcal{S}_P, t_P) \leftarrow ScheduleOneNode(\mathcal{S}, n, P)$
    Assign to $\mathcal{S}$ the $\mathcal{S}_P$ that minimizes $t_P$
  return
```

5.4 Scheduling algorithm

Our scheduling algorithm is a simple heuristic inspired from the one of SynDEx, and which works on the primitive subset of our format. It works by scheduling one dataflow node at a time on the processor that minimizes its completion date. The main scheduling routine is Function 1. Scheduling a node n on a given processor p is realized by Function 2, and consists in: (1) scheduling the communications allowing the computation of $clk(n)$, (2) scheduling the communications allowing the acquisition of all inputs, and (3) scheduling the node at the earliest possible date. The auxiliary function $FirstAvailable(\mathcal{S}, R, t, c, d)$ returns the first slot of duration d available on resource R after date t and on the condition c.

Function 2 ScheduleOneNode

```
Input: $\mathcal{S}$: partial schedule, $P$: target processor
  $n$: yet unscheduled computation node (the clocked graph
  and timing information are assumed global)
Output: $\mathcal{S}$: schedule (partial or not), $t$: date on $P$ where
  $n$ completes its execution
  $(\mathcal{S}, t) \leftarrow ScheduleEndoSupport(\mathcal{S}, P, supp(n))$
  for all incoming arc of $n$ do
    $(\mathcal{S}, t') \leftarrow$
      $ScheduleEndoSupport(\mathcal{S}, P, supp(a) \cup \{src(a)@clk(a)\})$
    $t \leftarrow max(t, t')$
  $t \leftarrow FirstAvailable(\mathcal{S}, P, t, c, d_P(n))$
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{n \mapsto (P, t)\}$ ; $t \leftarrow t + d_P(n)$
  return
```

The real complexity of the scheduling algorithm is hidden in Function 3 which schedules the communications which ensure that a given endochronous support is available on a given processor P. If the clock operations are not taken into consideration, the complexity of the algorithm is low-exponent polynomial over the size of the graph.

As explained in the introduction, our algorithm can handle asynchronous or static TDMA buses. The only difference between the two cases is the *FirstAvailable* function which gives a slot where some operation can be allocated. Our algorithm, however, is better adapted to asynchronous architectures, because the generated schedule does not take advantage of the time triggers of a TDMA architecture.

Function 3 ScheduleEndoSupport

```
Input: $\mathcal{S}$: partial schedule, $P$: target processor
  $s = \{o_1@c_1, \ldots, o_n@c_n\}$: endochronous support set such
  that for all $k$ $\{o_1@c_1, \ldots, o_k@c_k\} \vdash c_{k+1}$
  (the clocked graph and timing information are assumed
  global)
Output: $\mathcal{S}$: schedule (partial or not), $t$: date on $P$ where
  the data in the support set become available
  $t \leftarrow 0$
  $s_1 \leftarrow \{o@c \in s \mid o$ output of a delay $\delta$ not yet allocated $\}$
  for all $o@c \in s_1$ do
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{\delta \mapsto P\}$
  $s_2 \leftarrow \{o@c \in s \setminus s_1 \mid o$ produced on some $P' \neq P\}$
  $s_3 \leftarrow$ minimal (ordered) subset of $s$ with $s_3 \vdash s_2$
  for all $\overline{o}@\overline{c} \in s_3$, in order do
    if $clk^{\mathcal{S}}(\overline{o}, \infty, \mathcal{B}) < \overline{c}$ then
      $t_0 \leftarrow$
        $max(ready\_date(\mathcal{B}, o, c) \mid o@c \in dep_s^{(\mathcal{N},\mathcal{A})}(\overline{o}@\overline{c}))$
      Assign to $c''$ the union of all effective clocks $c'$ of bus
      communications of $\overline{o}$ such that $c' \wedge \overline{c} \neq false$, and to
      $t''$ the greatest date of these communications.
      $t_0 \leftarrow max(t_0, t'')$
      $t_0 \leftarrow FirstAvailable(\mathcal{S}, \mathcal{B}, t_0, \overline{c} \setminus c'', d_{\mathcal{B}}(\overline{o}))$
      Let $\overline{n}$ be the source processor of $\overline{o}$
      $\mathcal{S} \leftarrow \mathcal{S} \cup \{\overline{o}@\overline{c} \mapsto (\overline{n}, t_0, \overline{c} \setminus c'')\}$
    $t \leftarrow max(t, ready\_date(\mathcal{B}, \overline{o}, \overline{c}))$
  for all $o@c \in s \setminus (s_1 \cup s_3)$ do
    $t \leftarrow max(t, ready\_date(P, o, c))$
  return
```

The output of the scheduling for the example in Fig. 4 and an asynchronous bus is given in Fig. 8. We simplified the notations for space reasons. The figure also gives the result of SynDEx for the same example (which is worse, because of the coarser clock manipulations).[6] The width of an operation (its support) inside its lane intuitively represents its execution condition (the larger it is, the greater its execution condition). Much like in Venn diagrams, the logical relations between execution conditions of various operations are given by the horizontal overlapping of supports.

6. CONCLUSION

We have introduced a new method for the distributed real-time implementation of synchronous specifications. Our method is built around a new *clocked graph* intermediate representation, and works as a seamless series of transformations that add causality, time refinement or allocation information on the clocked graph nodes and arcs. Endochrony, a scheduling-independence property related to the Kahn principle and the notion of confluence, is used as a criterion ensuring efficient implementability. Endochronous clocked graphs are transformed into scheduled graphs by assigning real-time dates and resources to their elements. We defined the correctness of such a schedule graph, and provided an algorithm for the scheduling of endochronous clocked graphs on distributed architectures built around a single asynchronous bus.

We are currently focusing on the extension of the for-

[6] But of course, examples can be built where the finer analysis can lead to worse scheduling results, as is generally the case in scheduling theory.

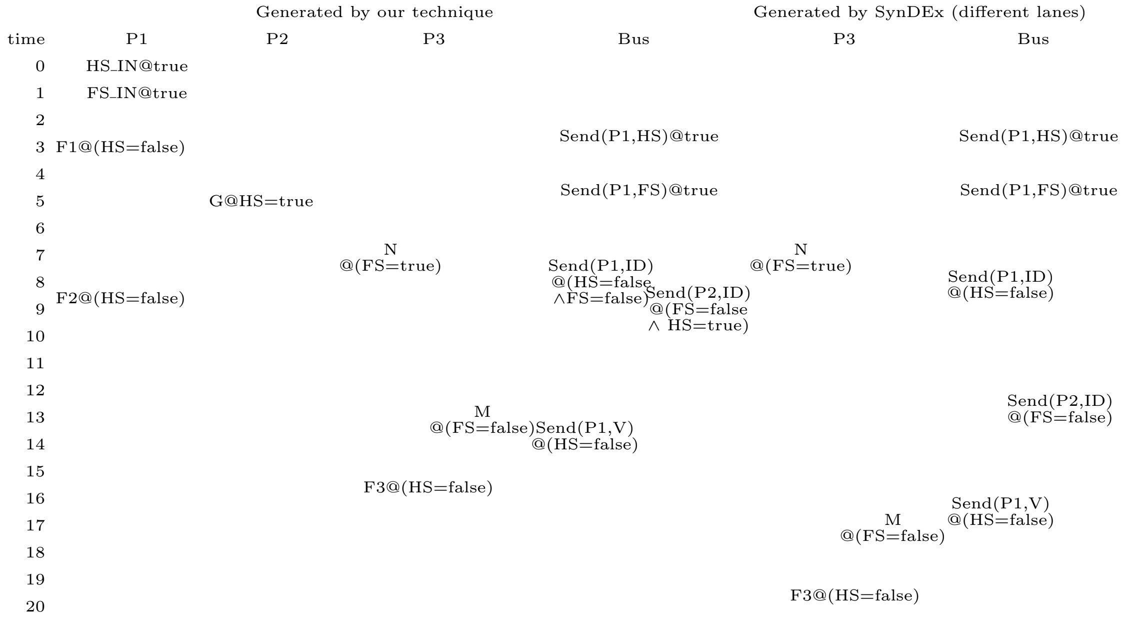

Figure 8: The real-time schedules generated by our algorithm and by SynDEx. We only figure for the SynDEx schedule the lanes that differ from ours. Time flows from top to bottom. We give here the schedule for one execution cycle. An execution of the system is an infinite repetition of this pattern. The width of an operation inside its lane intuitively represents clock inclusion and exclusion properties.

malism and scheduling techniques to cover (1) multi-period implementations where all the computations and communications are not bound to a single global clock, and (2) larger classes of implementation architectures, such as time-triggered ones.

7. REFERENCES

[1] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer academic Publishers, 1993.

[2] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[3] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation*, 163:125 – 171, 2000.

[4] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, April 2003. Special Issue on Application Specific Hardware Design.

[5] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, March 2006.

[6] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[7] G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Information Processing '74*, pages 471–475. North Holland, 1974.

[8] D. Potop-Butucaru, S. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[10] S. Campbell, J.-P. Chancelier, and R. Nikoukhah. *Modeling and Simulation in Scilab/Scicos*. Springer, 2006.

[11] T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives. In *Proceedings MEMOCODE*, 2003.

[12] R. Obermeisser. *Event-Triggered and Time-Triggered Control Paradigms*. Springer, 2005.

[13] A. Benveniste, B. Caillaud, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli. Composing heterogeneous reactive systems. *ACM TECS*, 7(4), 2008.

[14] P. Amagbégnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language signal. In *Proceedings PLDI*, 1995.

[15] A. Kountouris and C. Wolinski. Efficient scheduling of conditional behaviors for high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 7(3):380–412, July 2002.

[16] Z. Gu, X. He, and M. Yuan. Optimization of static task and bus access schedules for time-triggered distributed embedded systems with model-checking. In *Proceedings DAC*, 2007.

[17] W. Zheng, J. Chong, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. Extensible and scalable time triggered scheduling. In *Proceedings ACSD*, 2005.

[18] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to scade/lustre to tta: a layered approach for distributed embedded applications. In *Proceedings LCTES*, 2003.

[19] D. Potop-Butucaru, R. de Simone, and Y. Sorel. Necessary and sufficient conditions for deterministic desynchronization. In *Proceedings EMSOFT'07*, Salzburg, Austria, October 2007.

[20] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.