# Embedding Polychrony into Synchrony

Jens Brandt, Mike Gemünde, Klaus Schneider, Sandeep Kumar Shukla, Jean-Pierre Talpin

# Embedding Polychrony into Synchrony

Jens Brandt, Mike Gemünde, Klaus Schneider, Sandeep K. Shukla, and Jean-Pierre Talpin

***Abstract*—This article presents an embedding of polychronous programs into synchronous ones. Due to this embedding, it is not only possible to deepen the understanding of these different models of computation but more importantly, it is possible to transfer compilation techniques that were developed for synchronous programs to polychronous programs. This transfer is nontrivial because the underlying paradigms differ more than their names suggest: since synchronous systems react deterministically to given inputs in discrete steps, they are typically used to describe reactive systems with a totally ordered notion of time. In contrast, polychronous system models entail a partially ordered notion of time, and are most suited to interface a system with an asynchronous environment by specifying input/output constraints from which a deterministic controller may eventually be refined and synthesized. As particular examples for the mentioned cross-fertilization, we show how a simulator and a verification backend for synchronous programs can be made available to polychronous specifications, which is a first step towards integrating heterogeneous models of computation.**

## I. Introduction

The *development of embedded systems* is a challenging task: For the hardware platforms, developers have to cope with tight resource constraints, heterogeneous and application-specific hardware architectures, and virtual prototypes, all of which lead to many problems. On the software side, several concurrent tasks are executed on the available hardware with or without the help of special operating systems, sometimes statically or dynamically scheduled to the available hardware platforms, and sometimes tightly coupled with the hardware platforms themselves (implementing memory barriers etc.). Finally, many non-functional aspects have to be considered as well such as the energy consumption, reliability and efficiency of the systems. As many embedded systems are real-time systems, it is not sufficient to perform the right computations; in addition, the outputs have to appear at the right points of time to achieve the desired functionality.

For these reasons, *model-based design flows* became popular where one starts with an abstract model of the embedded system. Many languages are discussed for such model-based approaches, but most of them are based on only a few models of computation. A *model of computation* [23], [29], [17] thereby defines when and how an action of the system is executed, taking into account the timeliness, the causality and the concurrency of the overall computations. Classic models of computation are data-flow process networks [28], where computations can take place as soon as sufficient input data are available, discrete-event based systems [12], where each process is sensitive to the occurrence of a set of given events, and synchronous systems [2], which are triggered by a timing signal or a clock. It is not surprising that all models of computation have their advantages and disadvantages.

Synchronous specifications such as Esterel [5], Quartz [39] or Lustre [19] are all based on the *synchronous abstraction of time*, which splits execution traces into typically infinitely many discrete reaction steps: in each reaction step, the system reads all its inputs, computes values for all of its outputs and determines its next state. Since all variables have unique values within one reaction step, it is often said that computation takes zero time. This abstraction from physical time to abstract (clocked) time guarantees many desirable properties for the development of safety-critical embedded systems: in particular, it enforces deterministic concurrency, which has many advantages in system design, e.g. avoiding Heisenbugs (i.e. bugs that disappear when one tries to simulate/test them), predictability of real-time behavior, as well as provably correct-by-construction software synthesis [42]. Furthermore, the concise formal semantics of synchronous specifications allows one to formally reason about program properties [41], compiler correctness [35], [37], [38] and worst-case execution time [30], [9].

Polychronous specifications, such as Signal [26] or UML MARTE's CCSL standard [33], or MRICDF [24] are based on a more elaborate time model than the synchronous specifications: executions are not scheduled to a totally ordered set of instants. Instead, instants of time are considered as a partially ordered set so that for certain pairs of instants it is not specified whether the one occurs before the other one or vice versa. This allows one to directly express asynchronous computations which possibly need to synchronize intermittently. For some specifications this partial order may turn out to be a total order. In such a case, the instants become totally ordered, and we say that a global clock has been found.

The lack of a global clock in general, for polychronous specifications, offers interesting features for the design of embedded system software. First, it allows one to model globally asynchronous locally synchronous (GALS) systems, where components based on different clock domains are integrated at the system level. Given the advent of multi-core embedded processors, formal models and automated synthesis of multi-threaded code from polychronous models is an attractive option for embedded system designers. Second, polychronous specifications avoid unnecessary synchronization in the model, thereby offering optimization opportunities without an expensive data-flow analysis that

J. Brandt, M. Gemünde and K. Schneider are with the Department of Computer Science, University of Kaiserslautern, Germany. S. Shukla is with the Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA. J.-P. Talpin is with INRIA Rennes-Bretagne-Atlantique, Rennes, France.

is needed to reconstruct such opportunities from a synchronized program. Although not supported by the current state-of-the-art tools, polychrony gives developers the possibility to refine the system in different ways, and compilers can choose from different schedules for computation, according to non-functional constraints. The implementation governed in a large part by non-functional constraints is quite common in embedded system design. On the other hand, polychronous specifications are often better suited than more general models. On one extreme, untimed models lack the notion of clocks or other forms of time at all, making it very difficult to recognize and reorganize interrelated fragments. On the other extreme, explicitly timed models directly bind clocks to physical time, which does not only make models unnecessarily concrete but generally poses the question whether those systems can be implemented on a given platform with the specified real-time requirements.

However, polychrony challenges its compiler, analysis and synthesis tools in several ways, and it is not straightforward to provide an appropriate tool support. On the one hand, there are issues common with synchronous specifications such as *causality problems*, which result from the synchronous abstraction of time (where trigger conditions can be modified by the action they trigger). On the other hand, there are specific issues which are e.g. handled by the *clock calculus*, which statically analyzes the different clocks of a polychronous specification in order to check its consistency and to generate code from it.

This article presents an embedding of polychronous specifications into synchronous programs, which is interesting for several reasons. First, the embedding is interesting on its own, since it allows us to better understand the relationship between synchrony and polychrony. Second, the embedding gives us access to the methods and tools that already exist for synchronous specifications, especially for simulation and verification. In contrast to state-of-the-art analysis of polychronous specifications, where simulation is done by execution of generated code and is therefore restricted to compilable specifications, our approach can also handle erroneous specifications to trace their erroneous behaviors back to the errors in the simulation. Standard polychrony compilers do not accept erroneous specifications for code generation but code generation is required for simulation. Also verification, which typically relies on sophisticated transformations accomplished by the compiler, can be directly done with our embedding using verification methods and tools available for synchronous programs.

The rest of this article is organized as follows: In Section II, we first review the foundations of synchrony and polychrony before Section III shows the first part of our contribution, namely the embedding of the polychronous language Signal in synchronous guarded actions. Section IV then explains its usage for simulation, while Section V considers verification. Finally, we conclude the article with a short summary in Section VI.

## II. FOUNDATIONS

### A. Abstraction of Time

This section presents the foundations of the two models of computation which are considered in this article, namely synchrony and polychrony. Both share the same abstraction of time, namely the introduction of logical time in terms of clocks which requires that computation and communication can be performed in zero time [18], [2]. Under this assumption, the executions of programs are divided into a set of instants $\mathcal{I}$. In each of these instants (also called reactions or macro steps [22]), the system reads its inputs and computes and writes its outputs. According to this abstraction, the actions (sometimes called micro steps) that take place within an instant (sometimes called macro step) are not explicitly ordered. Instead, micro steps are assumed to occur simultaneously, i.e. in the same variable environment. Hence, variables seem to be constant during the execution of the micro steps and only change synchronously for macro steps. From the semantic point of view, which postulates that a reaction is atomic, neither communication nor computation take time. In reality, all actions within an instant are executed according to their data dependencies (which establishes the illusion of zero-time computations).

Next, we list some formal definitions used in the article. We start with the definition of an order for the set of all possible events $\mathcal{E}$, which reflect the execution of micro steps of a program.

**Definition 1** (Event Order)**.** *Let $\preceq \subseteq \mathcal{E}\times\mathcal{E}$ be a preorder on $\mathcal{E}$ such that for any two events $e_1, e_2 \in \mathcal{E}$, we say $e_1 \preceq e_2$ iff $e_1$* occurs before *$e_2$, or if both of them occur together. Let $\approx$ be the equivalence relation induced by $\preceq$: thus $e_1 \approx e_2$ iff $e_1 \preceq e_2 \wedge e_2 \preceq e_1$. We also define a precedence relation $\prec \subseteq \mathcal{E}\times\mathcal{E}$ on events such that $e_1 \prec e_2$ iff $e_1 \preceq e_2 \wedge \neg(e_1 \approx e_2)$.*

The order gives rise to the macro steps or instants $\mathcal{I}$ of an execution.

**Definition 2** (Instants)**.** *The set of* instants *$\mathcal{I}$ is the quotient of $\mathcal{E}$ with respect to the equivalence relation $\approx$, i.e. $\mathcal{I} = \mathcal{E}/\approx$, and each instant $I \in \mathcal{I}$ groups simultaneous events. The relations $\preceq$ and $\prec$ can be naturally lifted to $\mathcal{I}$, i.e. $I_1 \prec I_2$ iff for all events in $e_1 \in I_1, e_2 \in I_2$ it holds that $e_1 \prec e_2$, and $I_1 \preceq I_2$ iff for all events in $e_1 \in I_1, e_2 \in I_2$ it holds that $e_1 \preceq e_2$.*

Thus, instants define a (partial) order of events. The events can be also grouped in a different way. All actions, which write to the same variable, describe a signal. Formally:

**Definition 3** (Signals and Clocks)**.** *A* signal *$x$ consists of a totally ordered sequence of events $(x_t)_{t\in\mathsf{NAT}}$, i.e. $x_i \prec x_{i+1}$. For each signal $x$, let $\mathsf{Instants}(x) \subseteq \mathcal{I}$ be the possibly infinite set of instants at which the signal has events. This set gives rise to the* clock *$\widehat{x}$ of a signal $x$, which holds in instant $I \in \mathcal{I}$ iff $I \in \mathsf{Instants}(x)$. The signal $x$ is* present *in an instant iff $\widehat{x}$ holds, otherwise the signal $x$ is* absent.

*Furthermore, two signals $x_1$ and $x_2$ are* synchronous to each other *iff* $\mathsf{Instants}(x_1) = \mathsf{Instants}(x_2)$.

These basic definitions help us to describe the foundations of the considered subclasses. In the following, we first describe single-clocked synchronous programs in Section II-B, before the second model, namely polychronous specifications will be introduced in Section II-C.

### B. Synchrony

The synchronous *model of computation* [18], [2] assumes that the executions of programs consist of totally ordered sequences of instants $\mathcal{I} = (I_t)_{t\in\mathsf{NAT}}$ (with $I_i \prec I_{i+1}$). Due to the analogy to synchronous hardware circuits, one often assumes a *dedicated clock signal that triggers the computation of a synchronous system* (even though the physical time required for the computations may differ from step to step). In each of these instants, the system reads its inputs and computes and writes its outputs. In the single-clocked case, which we will consider in the following, all signals have the same clock, i.e. for all signals $x_1$, $x_2$: $\mathsf{Instants}(x_1) = \mathsf{Instants}(x_2) = \mathcal{I}$ and thereby $\widehat{x_1} = \widehat{x_2}$.

The introduction of this logical time scale is not only a very convenient programming model, it is also the key to generate *deterministic* single-threaded code from multi-threaded synchronous programs. Thus, synchronous programs can be directly executed on simple micro-controllers without using complex operating systems. Another advantage is the straightforward translation of synchronous programs to hardware circuits [3], [34].

In this article, we make use of synchronous guarded actions [11]. This allows us to use the Averest system[1] for simulation and verification purposes. Synchronous guarded actions are designed in the spirit of classical *guarded commands* [13], [14], [25], which are a well-established formalism for the description of concurrent systems. However, note that in contrast to many other applications where guarded commands are used, the guarded actions considered here follow the *synchronous abstraction of time* as described above. The system is represented by a set of synchronous guarded actions of the form $\langle \gamma \Rightarrow \mathcal{A} \rangle$ defined over a set of variables $\mathcal{V}$. The Boolean condition $\gamma$ is called the guard and $\mathcal{A}$ is called the action of the guarded action. In this article, guarded actions are either

- $\gamma \Rightarrow x = \tau$ (immediate assignment),
- $\gamma \Rightarrow \texttt{next}(x) = \tau$ (delayed assignment),
- $\gamma \Rightarrow \texttt{assume}(\sigma)$ (assumption), or
- $\gamma \Rightarrow \texttt{assert}(\sigma)$ (assertion).

Both kinds of assignments evaluate the right-hand side expression $\tau$ in the current macro step. Immediate assignments $x = \tau$ write the obtained value of $\tau$ immediately to the variable $x$, whereas delayed ones $\texttt{next}(x) = \tau$ write the value in the following step. If there is no action which determines the current value of a variable $x$ (i.e. immediate ones of the current step and delayed ones of the previous step), the variable $x$ will be determined by the *default reaction*. The default reaction generally depends on the storage type of a variable: event variables (indicated by the modifier `event`) are reset to false/zero, while (ordinary) memorized variables keep the value from the previous step (and get the default value in the initial step).

[1] http://www.averest.org

```
module Example(
    nat ?i1, ?i2,
    nat !o1, event nat !o2)
{
  nat x = 0;

  i1 > 5 => o1 = i1 + x + 1;
  i1 < 5 => o1 = i1 + o2;
  o1 > 10 => next(x) = i1;
  o1 < 10 => next(x) = i2;
  i1 > 5 => o2 = i2 + o1;
  i1 < 5 => o2 = i2 + x + 1;

  true => assume(i1>0);
  true => assume(i2>0);
}
```

Fig. 1. Synchronous Guarded Actions

| | 1 | 2 | 3 | 4 | 5 | ... |
|---|---|---|---|---|---|---|
| i1 | 6 | 5 | 1 | 5 | 9 | ... |
| i2 | 2 | 4 | 6 | 8 | 10 | ... |
| x | 0 | 2 | 4 | 1 | 5 | ... |
| o1 | 7 | 7 | 12 | 12 | 15 | ... |
| o2 | 9 | 0 | 11 | 0 | 25 | ... |

Fig. 2. Trace of Example in Figure 1

Immediate assignments define a causal dependency within the instant from all the read variables (i.e. variables occuring in the guard $\gamma$ and on the right-hand side $\tau$) to the written variable $x$. The former ones must be known before the value of $x$ becomes known. In contrast, delayed assignments do not have causal dependencies within the instant since $x$ is written in a different instant. Assumptions $\texttt{assume}(\sigma)$ provide a condition $\sigma$ the developer guarantees, i.e. they restrict the set of states the user cares about. In contrast, an assertion $\texttt{assert}(\sigma)$ defines a verification goal $\sigma$, which has to be proved. Both of them do not impose any causal dependencies since they do not change values.

An example for a synchronous system which is described by guarded actions is given in Figure 1. The system has the inputs `i1`, `i2`, the outputs `o1`, `o2`, and uses the local variable `x`. The guarded actions are synchronously evaluated based on the given inputs. An example execution trace is given in Figure 2. The evaluation order of the guarded actions is based on the data dependencies.

In the first instant, the input `i1` has the value `6` and `i2` has value `2`. The local variable `x` is determined by its default value `0`, and output `o1` is determined by the first guarded action, while output `o2` is determined by its first guarded action which requires the value of `o1`. In the second instant, the outputs `o1` and `o2` are determined by their default values, which is the previous value in case of

o1, and the default value 0 in case of o2. In the third instant, input i1 is 1, and therefore the second guarded actions of o1 and o2 are enabled. Thus, one first determines the value of o2 and then the value of o1 using the current value of o1.

### C. Polychrony

In contrast to synchronous systems, polychronous specifications [16], [26] are based on a partially ordered model of time. Partially ordered time allows one to express asynchronous computations which possibly need to synchronize intermittently. As the name suggests, polychrony makes use of several clocks, which means that signals do not need to be present at all instants. Since the used clocks may not imply each other, polychronous models are not based on a linear model of time, so that the reactions of a polychronous system are only partially ordered. Two instants can be only compared on the time scale if both contain events of a shared signal $x$.

Another aspect of polychronous specifications is that they are *relational*, rather than *functional*. A polychronous behavior is not described in an operational way, but rather, it is constrained by relational clauses. Obviously, due to the relational approach, polychronous specifications are generally nondeterministic, when constraints do not sufficiently specify the suited functional behavior. Even in the presence of the same input values, various temporal alignments, which fulfill the constraints, may lead to different output values. In contrast, synchronous modules deterministically react to any possible input configuration. The primary concern of a polychronous system are the constraints to interface the system with possibly asynchronous inputs. This problem is solved by, first, providing the specification of (possibly non-deterministic) input/output constraints and, second, determining a solution by the automatic synthesis of a controller enforcing the specified input/output timing constraints. Hence, polychronous models may be seen as specifications, which describe a set of acceptable implementations. There are three different types of clauses, which restrict the overall behavior:

- *equations* define the *values* of signals in terms of each other,
- *clock constraints* define the presence and absence of signals in an instant, i.e. how signals are temporally aligned (according to Definition 3, we denote the clock of $x$ by $\widehat{x}$, which holds if and only if $x$ is present (i.e. $x$ has an event) in a given instant.), and
- *causal dependencies* describe the order in which the values of the signals are determined within an instant ($x \xrightarrow{\phi} y$ means that there is a dependency from $x$ to $y$ in all instants where $\phi$ holds.)

Each signal implicitly defines the dependency $\widehat{x} \xrightarrow{\widehat{x}} x$, i.e. the status (presence or absence) of a signal $x$ must be known before we can determine its value. In the same way, operators and equations also impose clock constraints *and* causal dependencies.

| expression | causal dependencies |
|---|---|
| $x$ (signal) | $\widehat{x} \xrightarrow{\widehat{x}} x$ |
| $y := \mathtt{f}(x_1, \ldots x_n)$ | $x_1 \xrightarrow{\widehat{y}} y, \ldots, x_n \xrightarrow{\widehat{y}} y$ |
| $y := x$ **$ init** $c$ | ——— |
| $y := x_1$ **when** $x_2$ | $x_1 \xrightarrow{\widehat{y}} y$ |
| $y := x_1$ **default** $x_2$ | $x_1 \xrightarrow{\widehat{x_1}} y, x_2 \xrightarrow{\widehat{x_2} \wedge \neg \widehat{x_1}} y$ |

Fig. 3. Causal Dependencies of Signal Statements

In the following, we use Signal programs as polychronous specifications, which generally consist of a composition of several nodes. Each node has an input interface consisting of input signals, an output interface consisting of output signals and several possible internal signals. Its body is given by the composition of other nodes and/or a set of basic equations, which can be built from one of the following four primitive operators:

*Function.* A general function $\langle y := \mathtt{f}(x_1, \ldots, x_n) \rangle$ can have an arbitrary number of inputs $x_1, \ldots, x_n$ and an arbitrary number of outputs $y = (y_1, \ldots, y_m)$. The output values are determined by applying the given function to the input values. This node requires that all inputs have the same clock, and it produces the outputs also at the same instant, i.e. $\widehat{x_1} = \ldots = \widehat{x_n} = \widehat{y_1} = \ldots = \widehat{y_m}$. Obviously, there are causal dependencies from the inputs to the output of the node (each time there are values), i.e. $x_1 \xrightarrow{\widehat{y_j}} y_j, \ldots, x_n \xrightarrow{\widehat{y_k}} y_k$.

*Delay.* The delay operator $\langle y := x$ **$ init** $c \rangle$ has exactly one input $x$ and one output $y$. Its behavior consists of two micro steps: Each time a new incoming value arrives, it outputs the previously stored value and stores the new value. For the initial value, the buffer simply returns the given value $c$. By definition, the input and the output have the same clock, i.e. $\widehat{x} = \widehat{y}$. Since the output never depends on the input of the same instant, this node does not impose any causal dependencies.

*When.* The downsampling operator $\langle y := x_1$ **when** $x_2 \rangle$ has two inputs, $x_1$ of arbitrary type and $x_2$ of Boolean type, and one output $y$. Each time a new $x_1$ arrives, it checks whether there is an input at $x_2$. If there is one and if it is true, a new output event with the value of $x_1$ is emitted for $y$. In all other cases, i.e. if $x_1$ or $x_2$ is absent or $x_2$ has the value false, no event will be produced. Thus, we obtain the following clock constraint $\widehat{y} = \widehat{x_1} \wedge \widehat{x_2} \wedge x_2$. As the input is immediately forwarded, there is a causal dependency from $x_1$ to $y$: $x_1 \xrightarrow{\widehat{y}} y$. Note that there is no dependency from the second input $x_2$ to the value of the output $y$ since it only influences its status.

*Default.* The merge operator $\langle y := x_1$ **default** $x_2 \rangle$ has two inputs $x_1$ and $x_2$ and a single output $y$. Each time an input arrives at $x_1$, it will be forwarded to $y$. If there are events present at both inputs in a particular instant, the value of $x_1$ will be forwarded, and the value of $x_2$ will be discarded. If $x_1$ is absent, and there is only a value for

```
process Counter =
(? integer n;
 ! integer o;)
(| c := o $ init 0
 | o := n default (c−1)
 | n ^= (when (c=0))
 |)
where
    integer c;
end;
```

n = [ 2, ⊡, ⊡, 1, ⊡ ]
c = [ 0, 2, 1, 0, 1 ]
o = [ 2, 1, 0, 1, 0 ]

*Trace 1*

n = [ 2, ⊡, 1, ⊡ ]
c = [ 0, 2, 1, 1 ]
o = [ 2, 1, 1, 0 ]

*Trace 2*

Fig. 4. Signal Example: `Counter`

$x_2$, $x_2$ will be forwarded. Hence, the operator always gives priority to its first input, i.e. we have the clock constraint $\widehat{y} = \widehat{x_1} \vee \widehat{x_2}$, and the dependencies $x_1 \xrightarrow{\widehat{x_1}} y$ and $x_2 \xrightarrow{\widehat{x_2} \wedge \neg \widehat{x_1}} y$.

In addition to these basic nodes, programs may contain additional clock constraints to restrict the behavior. For example, clocks can be declared to be equal $\widehat{x} = \widehat{y}$, mutually exclusive $\widehat{x} \oplus \widehat{y}$, or a clock can be declared to be a subclock of another one $\widehat{x} < \widehat{y}$. Figure 3 recapitulates the dependencies of all statements, which will be needed later in Section III-B.

The idea of Signal is illustrated with two examples. These examples are also used in the subsequent sections. The first example of a Signal specification is given in Figure 4. It implements a simple counter which has one input `n` and one output `o`. The intention of the process is that for each input value $n$, the output values $n, n-1, \ldots, 0$ are produced. To this end, the local signal `c` stores the last value of the produced output, whereas `o` is produced by subtraction of 1 from `c`. If a new value for the input `n` arrives, the output is updated by this value. The clock constraint `n ^= (when (c = 0))` ensures that new inputs are read when the local signal `c` reaches 0. Thus, the countdown is never aborted, and the initial value for the next countdown is guaranteed to be read in time.

On the right hand side of the figure, two sample traces for this example are shown. In the figures, ⊡ indicates the absence of a signal, i.e. it is not present in the instant. The first trace is a valid one and shows the desired behavior. First, 2 arrives as input and the output produces the sequence of the values $2, 1, 0$. After that, the local signal $c$ is 0 and a new input is read. The second trace is an invalid one, because the second input value of `n` arrives too early and thus, the clock constraint is not fulfilled by this execution. Note that without the given clock constraint both traces would be valid. However, the second one does not comply to the given constraint (`n ^= (when (c = 0))` is violated in the third step) so that only the first one is a valid trace of the example. This example shows also the relational character of polychronous specifications.

The second example is given in Figure 5. The example illustrates the relational aspect of Signal processes and shows that the clock flow does not necessarily follow the data flow. The data dependencies of this process are drawn in Figure 6. Consider the signal `y` which is produced by

```
process Causality =
(? integer i;
 ! integer o;)
(| y := (y $ init 0) + 1
 | x := y default i
 | o := x + y
 |)
where
  integer x, y;
end;
```

Fig. 5. Example: Causality

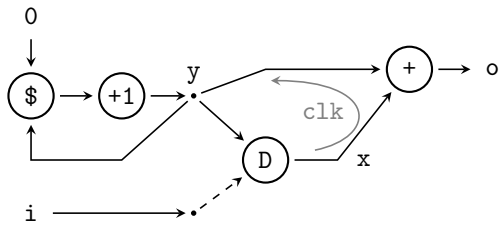



Fig. 6. Data Flow of Example `Causality`

the buffered loop that increments the previous value. The buffer is initialized with `0`. Thus, the signal `y` will hold the values $1, 2, 3, \ldots$, but it is not specified at which instants the values have to be present. The input `i` is fed into the **`default`** operator which is basically a merge node with priority given to its first input. Thus, if `i` is present, `x` is also present, but its value is still unknown, because if also `y` is present, its value is used instead of the value of `i`. However, since `x` is present, the addition node also requires its second input to be present. Therefore, also `y` must be present and the value of `x` will be the value of `y`. To summarize, the presence of `i` also forces `y` to be present which is against the data flow direction. The clock information follows a path which is not given by the data flow of the program. Even worse: we cannot build the clock flow in a bottom-up way as we do for the data-flow. This example illustrates the consequences of the relational nature of Signal specifications, and also the difficulty of describing it operationally. In a forthcoming article, we provide a constructive semantics for polychrony, which makes operational reasoning about Signal programs possible, which is however beyond the scope of this article.

## III. Embedding Polychrony

### A. Mapping Polychronous Instants

Synchronous and polychronous models both partition the execution of a program into logical instants. The computation within an instant follows the data dependencies in the synchronous model. An embedding of polychrony into a synchronous specification does not have to model the data dependencies explicitly since it also orders the actions within an instant implicitly. Instead, it lets the data flow percolate every time at least one signal has some data (i.e. in every instant).

However, a polychronous specification is based on a partial order of time, whereas the instants of a synchronous

execution are in a linear order. Therefore, a one-to-one mapping of logical instants is not possible. In order to capture all different linear executions of the partially ordered model, we introduce additional clock inputs, which choose between different executions. Thereby, the inherent nondeterminism (with respect to the asynchronous signal flows) is made explicit at the interface. By setting the additional clock inputs (according to clock constraints of the system) we choose among the possible embeddings.

The interesting part of this nondeterministic embedding of polychrony into synchrony is that now the synchronous steps have barriers at the end of each logical instant. In our embedding, no clock of $S$ can tick more than once within one logical synchronous instant. This gives us the power to simulate any nondeterministic path through a system and observe and interact with the system at these well defined barrier points.

*Thus, a simulation of $S$ amounts to choosing a linear sequence of logical instants from the partial order of logical instants of $S$ by giving the system $S$ not only the values of the input signals but also the presence or absence of clocks of the signals (i.e. $\mathcal{C}$ ) according to the constraints on clocks in $S$.*

These signals $\mathcal{C}$ must be always given in order to make the polychronous specification work. As the later sections will show, there are basically two different ways to obtain them, either by (1) a dedicated procedure, usually known as the clock calculus, which will be used in our simulation in Section IV or (2) general controller synthesis, which is a generalization of the verification presented in Section V.

In contrast to simulation, for formal verification, we are interested in the entire partial order of all the possible instants of the system for verification. Due to modeling the clock signals $\mathcal{C}$ as additional inputs, they are left open. By automatically creating a transition system, where $\mathcal{C}$ is nondeterministically chosen so that they are consistent with the clock constraints, we can analyze all possible sequences of logical instances respecting the partial order $\preceq$. Thus, we can formally verify properties of the specification $S$ (invariants, causal loops etc.) as shown in Section V.

More formally, let $S$ be a Signal specification, and let $(\mathcal{I}, \preceq)$ describe all possible executions of the system specified by $S$. Now, for simulation, and verification purposes, we want to embed these logical instants $\mathcal{I}$ into a linearly ordered sequence of logical instants, say, $(\mathcal{R}, \leq)$ where $\leq$ is a total order on $\mathcal{R}$. In order to gain control over which instant $R \in \mathcal{R}$ is selected for an $I \in \mathcal{I}$, we add all the clocks in the system as extra inputs. So if $\mathcal{C}$ is a vector representing the clocks of all signals in the system, then one can envision the embed function as

$$\mathsf{embed} : (\mathcal{I}, \preceq) \times \mathcal{C} \rightarrow (\mathcal{R}, \leq)$$

Thus, our embedding maps each logical instant $I \in \mathcal{I}$ of the polychronous model to an instant $\mathsf{embed}(I, \mathcal{C}) \in \mathcal{R}$ on the synchronous one depending on the signals of the vector $\mathcal{C} = (C^1, \ldots, C^n)$. This embedding has to meet the following requirements:

1) The image instant of $I$ contains at least the same events as $I$: $\forall e \in I.\ e \in \mathsf{embed}(I)$.
2) The mapping preserves the partial order $(\mathcal{I}, \preceq)$, i.e. $\forall I_1, I_2 \in \mathcal{I}, I_1 \preceq I_2.\ \mathsf{embed}(I_1) \leq \mathsf{embed}(I_2)$.
3) The signals $(C^i_t)_{t \in \mathsf{NAT}}, i = 1, \ldots, n$ comply to the clock constraints in $S$.

Independent polychronous instances $I_1$ and $I_2$ may be mapped in any order ($\mathsf{embed}(I_1, C) \leq \mathsf{embed}(I_2, C)$, or $\mathsf{embed}(I_2, C) \leq \mathsf{embed}(I_1, C)$ ) or may be even collapsed to a single instant ($\mathsf{embed}(I_1, C) = \mathsf{embed}(I_2, C)$).

Our approach to map the polychronous instants is in the spirit of previous work. For example, Milner [31], [32] showed that asynchronous systems can be always simulated by stuttering and silent synchronous ones. Halbwachs et al. [20], [21] formalized the concept of sporadic activation and oracle-driven nondeterminism to simulate and verify (partially) asynchronous systems by synchronous ones. The oracle variables of their scheduler in [20], [21] correspond to our additional inputs for the clocks. Imperative synchronous specifications such as Esterel and Quartz also follow this approach to support the modeling of asynchronous concurrency [4], [7], [36].

### B. Embedding Operators

Having explained the overall idea of the embedding, we have to define the basic polychronous operators with the help of synchronous guarded actions. Each of them is implemented as a separate module with the same signals at its interface as the original operator. In addition, for each variable `x`, we add a clock variable `clk_x` to the interface. Our interpretation is that whenever `clk_x` holds, the corresponding variable `x` holds a valid value. Otherwise, `x` may have an arbitrary value, which is never read. Thus, the tuple $(\texttt{x}, \texttt{clk\_x})$ encodes the value and the status of the original Signal signal.

Our embedding has to mimic all three aspects of the Signal operators: in the synchronous implementation, values, clocks and dependencies must be implemented according to the semantics as described in Section II-C. If we can achieve a one-to-one correspondence, we can relate the definitions from both the worlds.

```
module Function (
  event bool ?clk_o, !o,
  ?clk_i1, ?i1, ..., ?clk_in, ?in,
) {
  clk_o => o = f(i1, ..., in);
  assume(clk_o == clk_i1);
  assume(clk_i1 == clk_i2);
    ...
  assume(clk_in-1 == clk_in);
}
```

*Function.* We describe the case of single output functions here, but this can be easily generalized for multiple output functions. Each function operator `f` is translated to a module as given above. For each input and output of the function, an additional clock variable is added as input. The value of output `o` is simply computed by applying the function to the inputs. It is only assigned if its clock `clk_o` holds. Thereby, we encode a causal dependency

from clk_o to o in the synchronous model according to the polychronous semantics. In addition, clock constraints are added with the help of **assume** statements so that all inputs and the output need to have the same clock. By using assumptions, we do not introduce causal dependencies between the inputs, which would break the correspondence to the original semantics.

```
module Delay (
  bool ?init
  event bool ?clk_o, !o,
  event bool ?clk_i, ?i
) {
  bool q = init;

  clk_i => next(q) = i;
  clk_o => o = q;
  assume(clk_o == clk_i);
}
```

*Delay.* The synchronous description of the delay operator is similarly wrapped into a synchronous module. Again, clock signals for the input and the output are added. The additional input init allows one to set an initial value for the buffer. The buffer stores the *last* value of the input stream, i.e. the last value of the stream i when clk_i held, in a local variable q that is initialized in the first step (which is indicated by the special expression start) with the given initial value. Then, on each occurrence of a new input value, i.e. when clk_i holds, the content of the local variable is copied to the output and the current value of the input stream is stored. The assumption forces the clocks of input and output streams to be the same.

```
module When (
  event bool ?clk_o,  !o,
  event bool ?clk_i1, ?i1,
  event bool ?clk_i2, ?i2
) {
  assume(clk_o == clk_i1 & clk_i2 & i2);
  clk_o => o = i1;
}
```

*When.* The embedding of the **when** operator copies the first input i1 to the output o if its clock clk_o holds. This clock variable should be true if both the first and the second inputs are present and the second input is **true**. By construction, we have the desired causal dependencies from the input clocks (clk_i1 and clk_i2) and the value i2 to the clock of the output clk_o.

```
module Default (
  event bool ?clk_o,  !o,
  event bool ?clk_i1, ?i1,
  event bool ?clk_i2, ?i2
) {
  assume(clk_o == clk_i1 | clk_i2);
  clk_o & clk_i1 => o = i1;
  clk_o & clk_i2 & !clk_i1 => o = i2;
}
```

*Default.* Finally, the **default** operator is described by the synchronous module as given above. The assumption guarantees that whenever an input is present, the output will be also present. When the clock of the first input holds, its value is copied to the output, otherwise the second input is copied to the output. Thus, the priority is preserved, and the guarded actions model the causal dependencies.

```
module Counter (
  event bool ?clk_n, nat ?n,
  event bool ?clk_o, nat !o,
  event bool ?clk_c
) {
  nat c;

  // c := o $ init 0
  nat q = 0;
  clk_o => next(q) = o;
  clk_c => c = q;
  assume(clk_c == clk_o);

  // o := n default (c-1)
  clk_o &   clk_n          => o = n;
  clk_o & ! clk_n & clk_c => o = c-1;
  assume(clk_o == (clk_n | clk_c));

  // n ^= (when (c=0))
  assume(clk_n == (clk_c & (c == 0)));
}
```

Fig. 7. Example Counter Translated to Guarded Actions

```
module Causality (
  event bool ?clk_i, nat ?i,
  event bool ?clk_o, nat !o,
  event bool ?clk_x, ?clk_y
) {
  nat x, y;

  // y := (y $ init 0) + 1
  nat q = 0;
  clk_y => next(q) = y;
  clk_y => y = q + 1;
  assume (clk_y == clk_y);

  // x := y default i
  clk_x & clk_y          => x = y;
  clk_x &!clk_y & clk_i => x = i;
  assume(clk_x == (clk_y | clk_i));

  // o := x + y
  clk_o => o = x + y;
  assume (clk_o == clk_x);
  assume (clk_x == clk_y);
}
```

Fig. 8. Example Causality Translated to Guarded Actions

A complete Signal specification can then be translated to synchronous guarded actions by instantiating the corresponding modules for the primitive operators and adding additional clock constraints. Additional clock constraints, which are either implicitly given by the polychronous operators or explicitly by the programmer, are modeled by assumptions, which constrain the valid behaviors of the program to the cases where the clocks are related according to the specification.

Note that in the above discussion, we use operators with Boolean inputs and outputs, but they can be trivially generalized to any other data types.

The translation of the Signal example `Counter` from Figure 4 to synchronous guarded actions is shown in Figure 7. The interface of the module is extended by the clock variable `clk_c` of the internal signal. The translation of the second example `Causality` from Figure 5 is shown in Figure 8. Also this translation is straightforward according to the given guarded actions of each original operator.

## IV. Simulation

After embedding a Signal specification as presented in the previous section, we obtain a synchronous system based on synchronous guarded actions. As already mentioned in Section II-B, we use the AIF format as a concrete system representation. As this is the common intermediate format of the Averest system, we can use all its tools, in particular its simulator and verification procedures. In the following Section IV-A, we first sketch its general structure to understand its limitations and the adaptations made to leverage its application to polychronous systems, which is presented in Section IV-B.

### A. Synchronous Systems

The simulator for synchronous guarded actions takes an AIF system and simulates the instants, one by one: inputs are read, the internal state and the outputs are computed and the result is presented to the user. A detailed description is given in Figure 9. It formalizes the operational semantics briefly sketched in Section II-B. In the following description, we make use of the symbols:

- $\mathcal{V}$ is the set of all variables of the system, whereas $\mathcal{A}$ is the set of its guarded actions.
- $\mathcal{E}^{\mathsf{prv}}, \mathcal{E}^{\mathsf{cur}}, \mathcal{E}^{\mathsf{nxt}}$ are partial variable environments. They map each variable $x$ of domain $\mathsf{dom}(x)$ to an element of the set $\mathsf{dom}(x) \cup \{?, \lightning\}$[2], where ? means that the actual value of $x$ has not yet been determined and $\lightning$ means that $x$ cannot be given a value (due to runtime-failures like division by zero or write conflicts due to multiple assignments). $\mathcal{E}^{\mathsf{cur}}$ denotes the variable values for the current macro step, $\mathcal{E}^{\mathsf{prv}}$ for the previous one and $\mathcal{E}^{\mathsf{nxt}}$ collects the delayed assignments for the following one. Thereby, we use the following notations to access the values of the environment: we write $\mathcal{E}(x)$ to retrieve the current value of $x$ in environment $\mathcal{E}$, and similarly $[\![\tau]\!]_{\mathcal{E}}$ to evaluate expression $\tau$ with respect to the values in environment $\mathcal{E}$. To set a value $v$ for a variable $x$ in environment $\mathcal{E}$, we use the notation $\mathcal{E}(x) := v$ and we write $\emptyset$ for the *empty environment*, which assigns ? to all variables.

[2]In publications related to synchronous causality analysis (e.g. [40]), the symbols $\bot$ and $\top$ are often used for these purposes. However, since [16] $\bot$ and $\top$ represent the absence of presence of a signal in the polychronous community, we introduce new symbols.

**function** SimulateGuardedActions
  *(1) initialization*
  **forall** $x \in \mathcal{V}$ **do** $\mathcal{E}^{\mathsf{cur}}(x) := \mathsf{default(x)}$ **end**
  $\mathcal{E}^{\mathsf{nxt}} := \emptyset$

  **do**
    *(2) begin of macro-step*
    $\mathcal{E}^{\mathsf{prv}} := \mathcal{E}^{\mathsf{cur}}$
    $\mathcal{E}^{\mathsf{cur}} := \mathcal{E}^{\mathsf{nxt}} \bigsqcup \mathsf{ReadInputs}()$
    $\mathcal{E}^{\mathsf{nxt}} := \emptyset$

    *(3) immediate actions and reaction to absence*
    **do**
      $\mathcal{E}' := \mathcal{E}^{\mathsf{cur}}$;
      **forall** $\langle \gamma \Rightarrow x = \tau \rangle$ **do**
        **if** $[\![\gamma]\!]_{\mathcal{E}^{\mathsf{cur}}} = \mathsf{true} \wedge ([\![\tau]\!]_{\mathcal{E}^{\mathsf{cur}}} \neq ?)$ **then**
          $\mathcal{E}^{\mathsf{cur}}(x) := \mathcal{E}^{\mathsf{cur}}(x) \sqcup [\![\tau]\!]_{\mathcal{E}^{\mathsf{cur}}}$
      **forall** $x \in \mathcal{V}$ **do**
        **if** $\forall \langle \gamma \Rightarrow x = \tau \rangle . [\![\gamma]\!]_{\mathcal{E}^{\mathsf{cur}}} = \mathsf{false}$ **then**
          $\mathcal{E}^{\mathsf{cur}}(x) := \mathcal{E}^{\mathsf{prv}}(x)$
    **while** $\mathcal{E}^{\mathsf{cur}} \neq \mathcal{E}'$

    *(4) delayed actions*
    **forall** $\langle \gamma \Rightarrow \mathtt{next}(x) = \tau \rangle$ **do**
      **if** $[\![\gamma]\!]_{\mathcal{E}^{\mathsf{cur}}} = \mathsf{true} \wedge ([\![\tau]\!]_{\mathcal{E}^{\mathsf{cur}}} \neq ?)$ **then**
        $\mathcal{E}^{\mathsf{nxt}}(x) := [\![\tau]\!]_{\mathcal{E}^{\mathsf{cur}}}$

    *(5) end of macro-step*
    **if** $\exists x \in \mathcal{V}. \mathcal{E}^{\mathsf{cur}}(x) \in \{?, \lightning\}$ **then** Fail()
    **if** $\exists \langle \gamma \Rightarrow \mathtt{assume}(\sigma) \rangle . \gamma \wedge \neg\sigma$ **then** Fail()
    **if** $\exists \langle \gamma \Rightarrow \mathtt{assert}(\sigma) \rangle . \gamma \wedge \neg\sigma$ **then** Fail()
    WriteOutputs($\mathcal{E}^{\mathsf{cur}}$);
  **while** true
**end function**

Fig. 9. Interpreting synchronous guarded actions.

- To combine two values $v_1, v_2$ or to merge two environments $\mathcal{E}_1, \mathcal{E}_2$, we use the following notations:

$$v_1 \sqcup v_2 := \begin{cases} v_1 & \text{if } v_2 = ? \\ v_2 & \text{if } v_1 = ? \\ v_1 & \text{if } v_1 = v_2 \\ \lightning & \text{otherwise} \end{cases}$$

$$(\mathcal{E}_1 \bigsqcup \mathcal{E}_2)(x) := \mathcal{E}_1(x) \sqcup \mathcal{E}_2(x)$$

A synchronous system is *valid*, if it leads to a complete execution by the interpreter for all possible input traces. The particular steps of the interpreter are described in the following:

*Step (1):* The initialization consists of setting the default values for $\mathcal{E}^{\mathsf{cur}}$ and ? for $\mathcal{E}^{\mathsf{nxt}}$, since before the first macro step, no delayed action can be collected.

*Step (2):* The following loop is iterated for each macro step. At the beginning of a macro step, the environments are re-initialized, where $\mathcal{E}^{\mathsf{prv}}$ now obtains the values previously stored in $\mathcal{E}^{\mathsf{cur}}$ and $\mathcal{E}^{\mathsf{cur}}$ is set to the values collected with

delayed assignments of the previous step. Additionally, the inputs must be read and are also assigned to the current environment. $\mathcal{E}^{\mathsf{nxt}}$ is set to the empty environment, since no delayed assignments are collected yet for the new macro step.

*Step (3):* This loop evaluates the guarded actions and assigns values to the variables of the current environment $\mathcal{E}^{\mathsf{cur}}$. The actions are also evaluated with the current environment and a value is set, if an immediate action fires. If a write conflict occurs, i.e. two actions with different values fire for the same variable, the value is set to $\lightning$. If none of the actions of a variable can be fired, i.e. the guards of all actions possibly writing the variable are evaluated to false, then the value from the previous step is used. The loop determines a fixpoint of the environment, which must be obviously reached, since all assignments to $\mathcal{E}^{\mathsf{cur}}$ are monotonic[3] and only finitely many variables occur in a system, the fixpoint is reached after finitely many iterations. Note that this is different to the simulation of discrete-event languages such as VHDL or SystemC, where the inner loop (known as delta-cycles) may not terminate [10].

*Step (4):* The delayed actions are evaluated with the previously computed environment. This is done after the fixpoint iteration, since delayed assignments influence the following step and do not take part in the current one. The values evaluated for the next step are stored in $\mathcal{E}^{\mathsf{nxt}}$.

*Step (5):* At the end of a step, it is possible that some variables have either value ? or $\lightning$ (which indicates a causality problem or write conflict). If this is the case, the interpreter fails. It also fails if any activated assumption or assertion evaluates to false. If all checks succeed, the outputs are written to the environment and a new macro step is started.

The fixpoint iteration in Step 3 takes care about the simultaneous execution of guarded actions. Indeed, the actions are not really executed in parallel, but along their data dependencies. Therefore, the execution of an action does not influence any already executed action and the environment of the current macro step can be completed in this way.

As usual, the simulation can be used to get an impression of the specified behavior. The inputs can be given interactively or at once by a testbench.

### B. Polychronous Systems

In Signal, two kinds of information flows must be distinguished: the values of the signals and their clocks. This separation already became apparent in the introduction of the Signal operators in Section II-C, and it is reflected in the synchronous embedding, which uses two variables for each signal $x$: its clock `clk_x` and its value `x`. Clocks may not be determined in the same way as the data flows, as it had already been illustrated by the example `Causality` in Figure 5. In particular, if we only consider a single function node, we do not know which signal will become present first and thereby triggers the presence of all other signals. This is exactly the reason why the synchronous modules in our embedding (see Section III-B) do not set clocks but only constrain them. Thereby, we do not fix an evaluation order but only determine the conditions which have to be fulfilled.

However, only the actions influence the values of the variables, the assumptions and assertions are just checked at the end of each instant by the synchronous simulator (as shown in the previous section). For a reasonable simulation, we need an alternative operational description, which complies to all of the given constraints. The usual way to obtain this description is to use the Signal clock calculus [27], [8], [15]. It primarily aims at constructively determining normalized definitions of the clocks, i.e. it tries to create a primitive Signal equation that *binds* the clock of a signal to other clocks and values[4]. Naturally, some clocks of the system cannot be bound and remain *free*, e.g. the clocks of completely unrelated input signals.

Although not used in the following examples, we can use the result of the clock calculus to build a wrapper around the system created by our embedding. All free clocks are simply forwarded by the wrapper, while it hides all bound clocks from the environment. They are converted to local variables, which are set by the definitions determined by the clock calculus. As a result, we obtain a synchronous system that only exposes the free clocks to the outside, while all other ones are automatically inferred by the system. Thereby, developers are given the maximum degree of freedom for the simulation without the burden to consistently set all clocks of the system in each step. For example, in process `Counter`, the clock of the input signal `i` can be computed accordingly to the clock constraint by the value of `n`. Hence, due to the clock constraints and due to the possible backward clock-flow, the computation is not directly given by the operators, but the system needs to be analyzed completely to determine the clocks. If a wrapper can be build which sets all clocks but one (the so-called *master clock* which triggers all the behavior), the whole process is said to be *endochronous* [1], [27].

The classical way to simulate a Signal specification is to synthesize code for it, i.e. generate a C program, which can be compiled and executed. This does not only require the clock calculus but also to schedule all the actions within an instant according to their dependencies so that they are executed in the right order by the C program. Programs which do not have such an order (since they have cyclic causal dependencies) cannot be translated to C and thus, they cannot be executed. Since our simulation is based on synchronous guarded actions, we do not need to statically schedule the actions. The simulator automatically performs this task at run-time. As a consequence, we can even simulate causally incorrect programs or programs, which were rejected due to the conservative checks (precise ones would be infeasible) of the C program generation. The

[3]We assume here the partial order where ? is less than the other values, $\lightning$ is greater than the other values, and all other values are incomparable.

[4]Notice that one may alternatively use controller synthesis to generate these clock definitions so as to enforce satisfaction of all clock assumptions inferred from the translation into guarded actions.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| n | 2 | 2 | 2 | 1 | 1 |
| clk_n | true | false | false | true | false |
| c | 2 | 1 | 0 | 1 | 0 |
| clk_c | true | true | true | true | true |
| o | 0 | 2 | 1 | 0 | 1 |
| clk_o | true | true | true | true | true |

Fig. 10. Simulation Trace for Example `Counter`

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| i | 3 | 3 | 7 | 7 | 1 |
| clk_i | true | false | true | false | true |
| x | 1 | 1 | 2 | 3 | 4 |
| clk_x | true | false | true | true | true |
| y | 1 | 1 | 2 | 3 | 4 |
| clk_y | true | false | true | true | true |
| o | 2 | 2 | 4 | 6 | 8 |
| clk_o | true | false | true | true | true |

Fig. 11. Simulation Trace for Example `Causality`

simulator will run until the causal problem is noticed (when Fail() is called in the first line of Step (5) in Figure 9) and will then present the current situation. This greatly improves the debugging phase since the developers can not only see the effect, e.g. which variables could not be determined due to cyclic dependencies, but they can also see the situation and the path that lead to the problem.

A simulation trace for the already shown example `Counter` in Figure 7 is shown in Figure 10. It illustrates the simulation of the Trace 1 of Figure 4 where the original Signal process has been introduced. Take care while reading this trace, because the signals are not grouped by inputs and outputs. The simulation of Trace 2 would lead to an error, because in the third step, the assumption which comes from the clock constraint of the signal process is not fulfilled. The simulation trace also shows the characteristics of the embedding. In the original trace, the signal `n` is not present in each instant, whereas the variable `n` is, because a synchronous system is considered. However, the tuple $(\texttt{n}, \texttt{clk\_n})$ encodes the value and the status of the original signal and in each instants where `clk_n` does not hold, the value is not of interest. Due to the definition of the embedding, in these instances, the value from the previous instance is kept, but it could also be any arbitrary value as well.

A simulation trace for the second example `Causality` in Figure 5 is shown in Figure 11. Note, that also the case where `i` is not present but `y` is present can be simulated.

## V. Verification

### A. Synchronous Systems

The concise formal semantics of synchronous specifications allows one to formally reason about program properties. Due to the underlying model of computation, which synchronizes concurrent computations and groups them into instants, the state space can be kept smaller compared to asynchronous models. This is the reason why synchronous systems are better suited for automatic verification techniques such as model checking, on which we will focus in the following.

For model checking, the system generally needs to be represented by a transition system. This basically consists of a triple $(\mathcal{S}, \mathcal{S}_0, \mathcal{T})$ with set of states $\mathcal{S}$, initial states $\mathcal{S}_0 \subseteq \mathcal{S}$ and a transition relation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$. Each state $s$ is a mapping from variables $\mathcal{V}$ to values, i.e. $s$ assigns to each variable a value of its domain. As we aim for a symbolic description, we do not enumerate $\mathcal{S}$, $\mathcal{S}_0$ and $\mathcal{T}$ explicitly but describe them by their characteristic functions $\Phi_{\mathcal{S}}$, $\Phi_{\mathcal{S}_0}$ and $\Phi_{\mathcal{T}}$ over the variables $\mathcal{V}$. This representation is used by symbolic model checking tools. [5]

In the following, we show how these formulas are generated for a given system represented by synchronous guarded actions. Thereby, assume that we group all the guarded actions according to the variable on the left-hand side $x$ so that we generally have for each variable $x$ the following sets of immediate and delayed actions:

$$\begin{array}{llll} (\gamma_1, \texttt{x} = \tau_1), & \ldots, & (\gamma_p, \texttt{x} = \tau_p) \\ (\chi_1, \texttt{next(x)} = \pi_1), & \ldots, & (\chi_q, \texttt{next(x)} = \pi_q) \end{array}$$

Furthermore, we have a set of assumptions constraining the system behavior:

$$(\delta_1, \texttt{assume}(\sigma_1)), \quad \ldots, \quad (\delta_r, \texttt{assume}(\sigma_r))$$

As already noted in Section II-B, there is a default reaction, which determines the value of a variable in the case no action explicitly determines it: ordinary memorized variables keep their previous value, while event variables are reset to false or zero. In order to simplify the presentation in the following, we assume that for each variable the default reaction is given by two terms $\mathsf{default}^0(x)$ and $\mathsf{default}^+(x)$ for the initial and all other steps. Thus, e.g. for a memorized Boolean variable $x$, we have $\mathsf{default}^0(x) = \mathsf{false}$ and $\mathsf{default}^+(x) = x$, and for a Boolean event variable $\mathsf{default}^0(x) = \mathsf{false}$ and $\mathsf{default}^+(x) = \mathsf{false}$.

Figure 12 now shows the translation of the immediate and delayed actions writing variable $x$ to clauses of the characteristic functions $\Phi_{\mathcal{S}}$, $\Phi_{\mathcal{S}_0}$ and $\Phi_{\mathcal{T}}$, which describe the symbolic transition system.

- The first part $\Phi_{\mathcal{S}}(x)$ constrains the set of states. Invariantly, whenever the guard $\gamma_j$ of an immediate action holds, the action leads to the equation $x = \tau_j$, which must hold in the state.
- From the set of states, the initial ones can be selected by simply adding the default reaction of the first step

[5] For example, SMV can be given the transition system by `INVAR` ($\Phi_{\mathcal{S}}$), `INIT` ($\Phi_{\mathcal{S}_0}$) and `TRANS` ($\Phi_{\mathcal{T}}$) formulas.

$$\Phi_{\mathcal{S}}(x) \quad :\equiv \quad \left(\bigwedge_{j=1}^{p}(\gamma_j \rightarrow x = \tau_j)\right)$$

$$\Phi_{\mathcal{S}_0}(x) \quad :\equiv \quad \left(\left(\bigwedge_{j=1}^{p}\neg\gamma_j\right) \rightarrow x = \mathsf{default}^0(x)\right)$$

$$\Phi_{\mathcal{T}}(x) \quad :\equiv \quad \left(\left(\bigwedge_{j=1}^{q}(\chi_j \rightarrow \mathsf{next}(x) = \pi_j)\right) \wedge \left(\mathsf{next}(\bigwedge_{j=1}^{p}\neg\gamma_j) \wedge \left(\bigwedge_{j=1}^{q}\neg\chi_j\right) \rightarrow \mathsf{next}(x) = \mathsf{default}^+(x)\right)\right)$$

Fig. 12. Transition System for Synchronous Guarded Actions

as done in $\Phi_{\mathcal{S}_0}(x)$: whenever there is no action writing $x$, its value is given by $\mathsf{default}^0(x)$.

- The transition relation describes how the system moves from one state to another. To this end, a symbolic description does not only use the variables $x \in \mathcal{V}$ but also variables $\mathsf{next}(x), x \in \mathcal{V}$, which refers to the following step. With its help, we can define the clause $\Phi_{\mathcal{T}}(x)$ for the transition relation. Whenever the trigger condition $\chi_j$ of a delayed assignment `next(x)` $= \pi_j$ holds, then $x$ must have the value $\pi_j$ at the next point of time (i.e. $\mathsf{next}(x)$). Note that $\pi_j$ is evaluated with the current variables to determine the value of $x$ for the next point of time. If neither a trigger condition $\gamma_j$ of an immediate assignment in the following step nor a trigger condition of a delayed assignment $\chi_j$ in the current step holds, then we take the value specified by the default reaction $\mathsf{default}^+(x)$.

Basically, this concludes the generation of the characteristic functions. We only need to collect the clauses for all writable variables $\mathcal{V}_W$ and the additional assumptions, which also constrain the sets of states. Thus, we obtain the following final result:

$$\begin{array}{rcl}\Phi_{\mathcal{S}} & = & \bigwedge_{x\in\mathcal{V}_W}\Phi_{\mathcal{S}}(x) \wedge \bigwedge_{j=1}^{r}(\delta_j \rightarrow \sigma_j)\\ \Phi_{\mathcal{S}_0} & = & \bigwedge_{x\in\mathcal{V}_W}\Phi_{\mathcal{S}_0}(x)\\ \Phi_{\mathcal{T}} & = & \bigwedge_{x\in\mathcal{V}_W}\Phi_{\mathcal{T}}(x)\end{array}$$

To illustrate the construction of the transition system, consider the synchronous guarded actions given in Figure 1. Applying the definitions above leads to the symbolically described transition system shown in Figure 13. The first four lines of $\Phi_{\mathcal{S}}$ come from the immediate actions of the system, while the last one stems from the assumptions. The behavior for $o_1$ and $o_2$ in the initial step is determined by $\Phi_{\mathcal{S}}$ if the immediate actions fire - otherwise, they are initialized by the default reaction as described by $\Phi_{\mathcal{S}_0}$. As $x$ is only set by delayed actions, it is always initialized to zero. The first two lines of the transition relation $\Phi_{\mathcal{T}}$ model the default reactions for $o_1$ and $o_2$. If there is no action in the following step setting the variable, $o_1$ is kept (memorized variable) while $o_2$ is reset to 0 (event variable). The last three lines of $\Phi_{\mathcal{T}}$ model the behavior of $x$: it is

$$\begin{array}{rcl}\Phi_{\mathcal{S}} & = & (i_1 > 5 \rightarrow o_1 = i_1 + x + 1) \wedge\\ & & (i_1 < 5 \rightarrow o_1 = i_1 + o_2) \wedge\\ & & (i_1 > 5 \rightarrow o_2 = i_2 + o_1) \wedge\\ & & (i_1 < 5 \rightarrow o_2 = i_2 + x + 1) \wedge\\ & & (i_1 > 0) \wedge (i_2 > 0)\\ \Phi_{\mathcal{S}_0} & = & (i_1 = 5 \rightarrow o_1 = 0) \wedge\\ & & (i_1 = 5 \rightarrow o_2 = 0) \wedge\\ & & (x = 0)\\ \Phi_{\mathcal{T}} & = & (\mathsf{next}(i_1 = 5) \rightarrow \mathsf{next}(o_1) = o_1) \wedge\\ & & (\mathsf{next}(i_1 = 5) \rightarrow \mathsf{next}(o_2) = 0) \wedge\\ & & (o_1 > 10 \rightarrow \mathsf{next}(x) = i_1) \wedge\\ & & (o_1 < 10 \rightarrow \mathsf{next}(x) = i_2) \wedge\\ & & (o_1 = 10 \rightarrow \mathsf{next}(x) = x)\end{array}$$

Fig. 13. Transition System for Example of Figure 1

either set by one of the two delayed actions given in the original system or by its default reaction.

When constructing the transition system as described above, one should be aware of (1) write conflicts and (2) cyclic causal dependencies in the original system.

Write conflicts (1) are due to several synchronous guarded actions firing in the same step and assigning a different value to the same variable, e.g. consider the following two guarded actions

```
i > 0 => x = 1;
i < 5 => x = 2;
```

which lead to a contradiction if $i$ is between 0 and 5. These states (and thereby all paths to them) are removed from the transition system if we apply the construction of the previous section. Cyclic causal dependencies (2) may also lead contradictions or to nondeterministic behavior , e.g.

```
true => x1 = x2;
true => x2 = x1;
```

which do not fix a concrete value for $x_1$ and $x_2$. Cyclic causal dependencies (2) may also show no problems during verification, e.g.

```
x1   => x1 = x2;
true => x2 = !x1;
```

which has a single consistent behavior. Although the behavior is well-defined in the transition system, this example

```
satisfies {
  observer (nat{16} last_o = 0)
  { // remember last value of o
    clk_o => next(last_o) = o;
  }
  S1:assert A G(clk_n  -> clk_o & n==o);
  S2:assert A G(clk_n  -> clk_o);
  S3:assert A G(!clk_n -> o>=last_o);
  S4:assert A G(clk_n  -> (last_o==0));
  // should be false
  S5:assert A G(clk_n  -> o==0);
}
```

Fig. 14. Specifications for Example `Counter`

can never be turned into efficient executable code due to the cyclic dependencies [6], [40].

Verification is still possible in the context of write conflicts and causal dependencies: however, these issues need always be resolved before final code generation. Hence, the functional behavior can be verified correctly when it is ensured that those issues do not occur.

### B. Polychronous Systems

As previously highlighted, the additional aspect we have to consider for polychronous specifications is the treatment of the clocks. In our original embedding of Section III-B, the clocks are addressed by assumptions. For the simulation, we needed to apply the clock calculus to obtain an operational description, because the simulation needs to *compute* the clocks. For the transition system, however, this is not needed, since it is declarative and is able to consider all possibilities. Thereby, it neglects the data dependencies: the immediate action `x=`$\tau$ and the assumption **`assume`**`(x==`$\tau$`)` are both mapped to the same clause $x = \tau$ in the transition system. Instead, it just collects all the clock constraints, and thereby considers all the paths that comply to them. Similar to the simulation, all free clock variables are additional inputs to the system. Thus, the model checker always verifies that a given property holds for all possible assignments.

The Averest tools can be used for verification of Signal processes with our translation. However, the specifications have to be given in the synchronous world by temporal logics. Figure 14 shows some specifications based on an observer, which is also given in guarded actions, for the already considered example `Counter` (Figure 7). The specifications have to be placed in the file of the module to be used. The temporal operator **`X`** considers the next instant, but there is no means to express the *next value* of a stream, i.e. the next value when its clock will hold. Therefore, the observer is used to store the last output value of the output `o`. With this value the specifications are formulated. As expected, the specifications `S1` to `S4` can be verified by using the Averest tools and SMV, whereas specification `S5` can be disproved.

Finally, consider the verification of the example `Causality`. Some specifications for this process are given

```
satisfies {
  S1:assert A G(clk_i -> clk_o);
  S2:assert A(
    G F (clk_i | clk_o | clk_x | clk_y)
      -> F (clk_o & (o == 12))
  );
  S3:assert A G (clk_o -> (o % 2 == 0));
}
```

Fig. 15. Specifications for Example `Causality`

in Figure 15. The specification `S1` verifies the clock dependency between `i` and `o` which ensures that an output is produced whenever an input is given. The second specification `S2` considers traces which do not end up in stuttering, i.e. traces where again and again at least one clock holds. The specification ensures that on these traces finally the value `12` is produced as output. Since the output `o` is produced by `y + y`, it cannot have an odd number, which is ensured with specification `S3`.

## VI. Summary

In this article, we presented an embedding of the polychronous language Signal into synchronous guarded actions. While instances of time are totally ordered in synchronous systems, they are only partially ordered in polychronous systems. For this reason, each variable is explicitly endowed by its clock to select one of the possible behaviors.

The mapping of instants in our embedding helps to get some insights about the relationship between synchronous and polychronous models. It clearly shows the asynchronous aspects of the polychronous specifications. Furthermore, by separating the temporal part, the relational aspects of polychronous specifications become very clear.

In addition to these theoretical insights, we also obtain significant practical benefits. With the help of the embedding, we are now able to use simulators and model checkers of the Averest framework, which has been originally implemented for synchronous systems. Thereby, we improve the tool support for the analysis and verification of polychronous systems.

### A. Acknowledgement

We thank the INRIA Associated project POLYCORE for travel support that enabled the authors to get together to work on this project, and an US Air Force Contract (FA8750-11-1-0042) for partial support of one of the author's work.

## References

[1] 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.

[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*, 91(1):64–83, 2003.

[3] G. Berry. A hardware implementation of pure Esterel. In *Formal Methods in VLSI Design*, Miami, Florida, USA, 1991.

[4] G. Berry. Preemption and concurrent systems. In R.K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 761 of *LNCS*, pages 72–93, Bombay, India, 1993. Springer.

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

[6] G. Berry. The constructive semantics of pure Esterel, July 1999.

[7] G. Berry and E. Sentovich. Multiclock Esterel. In T. Margaria and T.F. Melham, editors, *Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, pages 110–125, Livingston, Scotland, UK, 2001. Springer.

[8] L. Besnard, T. Gautier, P. Le Guernic, and J.-P. Talpin. Compilation of polychronous data flow equations. In S.K. Shukla and J.-P. Talpin, editors, *Synthesis of Embedded Software – Frameworks and Methodologies for Correctness by Construction*. Springer, 2010.

[9] M. Boldt, C. Traulsen, and R. von Hanxleden. Compilation and worst-case reaction time analysis for multithreaded Esterel processing. *EURASIP Journal on Embedded Systems*, 2008. Article ID 594129.

[10] J. Brandt and K. Schneider. How different are Esterel and SystemC? In *Forum on Specification and Design Languages (FDL)*, pages 98–103, Barcelona, Spain, 2007. Electronic Chips and Systems Design Initiative (ECSI).

[11] J. Brandt and K. Schneider. Separate translation of synchronous programs to guarded actions. Internal Report 382/11, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, March 2011.

[12] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2 edition, 2008.

[13] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Austin, Texas, USA, May 1989.

[14] D.L. Dill. The Murphi verification system. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 390–393, New Brunswick, New Jersey, USA, 1996. Springer.

[15] A. Gamatie. *Designing Embedded Systems with the SIGNAL Programming Language*. Springer, 2010.

[16] A. Gamatié, T. Gautier, P. Le Guernic, and J.P. Talpin. Polychronous design of embedded real-time applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2), April 2007.

[17] A. Girault, B. Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 18(6):742–760, June 1999.

[18] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[19] N. Halbwachs. A synchronous language at work: the story of Lustre. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 3–11, Verona, Italy, 2005. IEEE Computer Society.

[20] N. Halbwachs and S. Baghdadi. Synchronous modelling of asynchronous systems. In A.L. Sangiovanni-Vincentelli and J. Sifakis, editors, *Embedded Software (EMSOFT)*, volume 2491 of *LNCS*, pages 240–251, Grenoble, France, 2002. Springer.

[21] N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *Application of Concurrency to System Design (ACSD)*, pages 3–14, Turku, Finland, 2006. IEEE Computer Society.

[22] D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[23] A. Jantsch. *Modeling Embedded Systems and SoCs*. Morgan Kaufmann, 2004.

[24] Bijoy A. Jose and Sandeep K. Shukla. An alternative polychronous model and synthesis methodology for model-driven embedded software. In *ASP-DAC* [24], pages 13–18.

[25] H. Järvinen and R. Kurki-Suonio. The DisCo language and temporal logic of actions. Technical Report 11, Tampere University of Technology, Software Systems Laboratory, 1990.

[26] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[27] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers (JCSC)*, 12(3):261–304, June 2003.

[28] E.A. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[29] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 17(12):1217–1229, December 1998.

[30] G. Logothetis and K. Schneider. Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In *Design, Automation and Test in Europe (DATE)*, pages 10196–10203, Munich, Germany, 2003. IEEE Computer Society.

[31] R. Milner. On relating synchrony and asynchrony. Technical Report CSR- 75-80, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, UK, 1981.

[32] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science (TCS)*, 25(3):267–310, 1983.

[33] Object Management Group OMG. Modeling and analysis of real-time and embedded systems (MARTE). http://www.omg.org/omgmarte/Documents/Specifications, June 2008.

[34] F. Rocheteau and N. Halbwachs. Implementing reactive programs on circuits: A hardware implementation of LUSTRE. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 195–208, Mook, The Netherlands, 1992. Springer.

[35] K. Schneider. A verified hardware synthesis for Esterel. In F.J. Rammig, editor, *Distributed and Parallel Embedded Systems (DIPES)*, pages 205–214, Schloß Ehringerfeld, Germany, 2000. Kluwer.

[36] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *Application of Concurrency to System Design (ACSD)*, pages 143–154, Newcastle Upon Tyne, England, UK, 2001. IEEE Computer Society.

[37] K. Schneider. Improving automata generation for linear temporal logic by considering the automata hierarchy. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2250 of *LNCS*, pages 39–54, Havana, Cuba, 2001. Springer.

[38] K. Schneider. Proving the equivalence of microstep and macrostep semantics. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 2410 of *LNCS*, pages 314–331, Hampton, Virginia, USA, 2002. Springer.

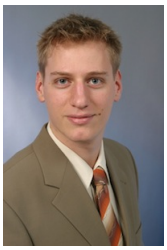
[39] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

[40] K. Schneider, J. Brandt, and T. Schuele. Causality analysis of synchronous programs with delayed actions. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 179–189, Washington, District of Columbia, USA, 2004. ACM.

[41] K. Schneider, J. Brandt, and T. Schuele. A verified compiler for synchronous programs with local declarations. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 153(4):71–97, 2006.

[42] B.L. Titzer and J. Palsberg. Nonintrusive precision instrumentation of microcontroller software. In Y. Paek and R. Gupta, editors, *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 59–68, Chicago, Illinois, USA, 2005. ACM.

**Jens Brandt** Jens Brandt is a research associate in the Embedded Systems Group at the University of Kaiserslautern. He studied applied computer science at the same university from 1998 to 2003, where he also received his Ph.D. from the Department of Computer Science.



**Mike Gemünde** Mike Gemünde received his diploma in computer science from the University of Kaiserslautern in 2008. He is now working towards his Ph.D. in the Embedded Systems Group at the University of Kaiserslautern.

**Sandeep K. Shukla** Sandeep K. Shukla (SM'03) received the Ph.D. degree from the State University of New York at Albany in 1997, M.S from the same university in 1995 and B.E from Jadavpur University, Kolkata, India in 1991. He is a Professor in the Department of Electrical and Computer Engineering at Virginia Polytechnic and State University, Blacksburg. He is also the director of the Center for Embedded Systems for Critical Applications (CESCA). Prof. Shukla is Associate Editor of IEEE Transactions on Computers and IEEE Embedded Systems Letters. He was associate editors for IEEE Design and Test of Computers (2003–2011), and IEEE Transctions on Industrial Informatics (2005–2006). He was awarded the PECASE in 2004, Virginia Tech Engineering Faculty Fellow award in 2005, SUNY Albany Distinguished Alumni Award in 2007, and Humboldt Foundation's Bessel award in 2008. He is an IEEE Computer Society Distinguished Visitor, and an ACM Distinguished Speaker.

**Klaus Schneider** Klaus Schneider studied computer science from 1987 to 1992 at the University of Karlsruhe in Germany. Between 1992 and 2002, he worked as a research associate at the Institute of Computer Design and Fault Tolerance at the University of Karlsruhe. He received his Ph.D. for his research in hardware verification from the department of computer science at the University of Karlsruhe in 1997, and wrote a habilitation thesis in 2001 about the verification of reactive systems. In April 2002, he became a full professor at the University of Kaiserslautern and since then, he is heading the Embedded Systems Group there. His main research interests are innovative design methods for the model-based design of embedded systems including their formal verification. To this end, he studies various system level design languages and their models of computation, in particular, synchronous specifications.

**Jean-Pierre Talpin** Jean-Pierre Talpin is Senior Researcher with INRIA and leads the project-team who develops the open-source Polychrony environment. He received his PhD from Université Paris VI Pierre et Marie Curie in 1993. He then was a research associate with the European Computer-Industry Research Centre in Munich before to join INRIA in 1995. Jean-Pierre edited two books with Elsevier and Springer, guest-edited more than ten special issues of ACM and IEEE scientific journals, and authored more than 20 journal articles and book chapters and 60 conference papers. He received the 2004 ACM Award for the most influential POPL paper, for his 2nd conference paper with Mads Tofte, and the 2012 LICS Test of Time Award, for his 1st conference paper with Pierre Jouvelot.