Optimized Distribution of Synchronous Programs via a Polychronous Model

Ke Sun*, Loïc Besnard†, and Thierry Gautier*
*INRIA Rennes - Bretagne Atlantique, †CNRS/IRISA
*†Campus de Beaulieu, Rennes, France
Email: *{firstname}.{lastname}@inria.fr, †Loic.Besnard@irisa.fr

***Abstract*—This paper presents a distribution methodology for synchronous programs, applied in particular on programs written in the Quartz language. The given program is first transformed into an intermediate model of guarded actions. After user-specified partitioning, the generated sub-models are transformed into Signal processes. Using the multi-clock calculation model of the Signal language, an optimized data-flow network can be automatically constructed. The optimization includes reducing the communication quantity and the computation load, with no change to the interface behaviors.**

I. Introduction

Synchronous programming languages such as Esterel [1], Lustre [2] or Quartz [3] are all based on the *synchronous hypothesis* [4]. Under this assumption, system behaviors are projected onto a discrete sequence of logical instants. As the sequence is discrete, nothing occurs between two consecutive instants. Such temporal abstraction can greatly facilitate safety-critical reactive system design. It enforces deterministic concurrency of the system: Heisenbugs (i.e., bugs that disappear when one tries to simulate/test the system) are avoided; the system behaviors become predictable. It is also the key to a straightforward translation of synchronous programs to hardware circuits [5]. Furthermore, it guarantees semantic precision by using mathematical models, such as Mealy machine, as supporting foundations. These models enable a series of efficient optimization, compilation and verification techniques to be applied on synchronous programs.

Synchronous programs are modeled on centralized architectures with zero-time communications: any signal emitted by some component is instantaneously received by others. However, in real world, safety-critical reactive systems [6] [7] practically operate over distributed architectures with delayed communications. This mismatch results in a wide gap between centralized design model and distributed implementation. To get over this, *desynchronization* is introduced with the following requirement: how to distribute a synchronous program while preserving the same observable behaviors, i.e., the same interface behaviors. This issue has drawn a considerable attention both in theoretical challenges and in industrial relevance.

For Esterel and Lustre, a series of distribution methods have been proposed [8]. In [9], starting from a synchronous program, a network of communicating *codesign finite state machines* is constructed. The distribution of synchronous programs via the modeling as *finite deterministic automata* [10] is presented in [11]. Based on it, the extended method in [12] focused on the automatic deduction of distributed systems from centralized synchronous circuits. Furthermore, how to distribute synchronous programs to fulfill temporal constraints is introduced in [13]. The primary goal of these distribution methods does not include an optimization of the communication and the computation, which is in the focus of this paper.

A related distribution methodology is presented in [14]. Its procedure consists of three steps: 1) the given synchronous program is compiled into an intermediate AIF model [15], which is a common intermediate format for various synchronous languages [16] [17]; 2) the data dependencies within the model are analyzed and represented in a dependency graph, then the generated graph is subsequently partitioned into sub-graphs in which partitions can be made horizontal (for a pipelined execution) or vertical (for a parallel execution); 3) a synchronous elastic system [18] is synthesized by generating a distributed component for each sub-graph and establishing a communication infrastructure with synchronous elastic flow (SELF) protocol [18].

Owing to the analogy to synchronous hardware circuit, one can assume that the given synchronous program holds a *master clock* driving the whole program computation. We refer to such program as a *mono-clocked* program [19]. Furthermore, it follows the model of computation (MoC) *strict synchrony*: during each instant, each input channel always reads data and each output channel always produces data. This feature may cause unnecessary communications or computations. In [20], the communication quantity is reduced, according to an evaluation of communication necessity: a producer transmits a value only when it is really required for the calculation of a receiver. However, it may result in highly overestimated computations for evaluating communication necessity. To avoid this, the evaluation can be refined by constructing additional communications, while this operation may augment the communication. An optimization method on computations is presented in [21]: a condition is computed for every variable to determine whether its value is required for current or future computations; when the condition does not hold, the computation of the corresponding variable can be suppressed.

A new optimized distribution methodology is proposed, which is based on a more flexible MoC. In contrast to synchronous languages, the polychronous language Signal [22] [23] [24] is based on the MoC *polychrony*[1]. As its name suggests, a polychronous program makes use of multiple clocks to drive its execution, the system behaviors are defined on a partially ordered set of instants. One can consider that each component in the program holds its own master clock, and there is no longer a master clock for the whole program.

[1]From the Greek "poly chronos" to mean multiple clocks.

We refer to such program as a *multi-clocked* program. If there is no relation between the master clocks, the *multi-clocked* program does not follow a linear timeline, but a nonlinear one. Nonlinear instants t_1, t_2 are such that there is no order relation between them. The system behaviors over nonlinear instants occur asynchronously. Hence, the Signal language allows one to conveniently model the asynchronous communications between synchronous modules. This exactly accords with the requirement of distributed systems. In addition, nonlinear timeline implies the possibility to avoid unnecessary synchronization, thereby enabling potential optimization [25]. Due to these advantages, Signal is particularly suited as a coordination layer to finally build a data-flow network over desynchronized processing locations.

Based on the multi-clock calculation model of the Signal language, we propose a distribution methodology for the synchronous programs. First, the given synchronous program is compiled into AIF model. Second, according to the user specifications, the generated model is partitioned into sub-models. After the partitioning, the generated sub-models are transformed into equivalent Signal processes. Then, the unnecessary constraints are eliminated from the processes to avoid unnecessary synchronization. Finally, within the Signal framework, the minimal frequencies of communication and computation are computed via multi-clock calculation. This operation can efficiently reduce the communication quantity and the computation load. Along this way, an optimized data-flow network over desynchronized processing locations can be constructed. Note that both the methodology in [14] and ours process AIF model. Both methods are thereby independent of particular synchronous languages.

The rest of this paper is organized as follows. Section II briefly reviews the intermediate AIF model, which serves as the starting point for the distribution procedure, and introduces the Signal language and its associated polychronous semantic model. Section III presents the user-specified model partitioning and the transformation from partitioned models to Signal processes. In Section IV, how to achieve the optimization both on communications and on computations is presented in detail. Meanwhile, our methodology is illustrated through case studies in Section V. Finally, we conclude this paper and look forward the perspectives.

II. Foundations

A. AIF Model & Synchronous Guarded Actions

To process synchronous programs, it is quite natural to compile them into intermediate models at first. In this way, the whole processing can be modularly divided into several steps and the models can be reused for different purposes [8], such as validation, comparison, model transformation and code generation. Furthermore, the processing on the intermediate model is independent of particular synchronous languages. The distribution of synchronous programs converts to two steps: 1) compilation into intermediate models; 2) distribution of intermediate models. The intermediate model devoted to the Quartz language is the Averest Intermediate Format (AIF) model [15]. The compilation of Quartz programs into AIF models has been implemented in the Quartz/Averest framework[2] [26]. The essential part of AIF model is AIF model behaviors that are described by a set of synchronous guarded actions.

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

Guarded actions are designed in the spirit of traditional guarded commands [27], which are well-established intermediate code for the description of concurrent systems. The guarded actions are in the form of $\gamma \Rightarrow \mathcal{A}$, where the Boolean condition γ is called *guard* and $\mathcal{A}$ is called *action*. The guards represent the complex control structure of the synchronous program. The actions represent the assignments to variables. Due to the category of assignments, each guarded action has either the form $\gamma \Rightarrow x = \tau$ (called guarded immediate action) or $\gamma \Rightarrow next(x) = \tau$ (called guarded delayed action). Once the guard is evaluated to *true*, the corresponding action instantaneously starts. Both kinds of assignments evaluate the expression τ at the current instant. Then, the immediate assignment $x = \tau$ instantaneously transfers the value of τ to the variable x, whereas the assignment effect of the delayed one $next(x) = \tau$ takes place at the next instant. When the value of a variable x cannot be determined by any action, its value is determined by the *absence reaction*. It determines the value, according to the storage type of the assigned variable: a *non-memorized* variable is reset to the default value (denoted as def_value, the particular value depends on the data type); a *memorized* variable keeps its previous value, or takes its default value at the initial instant.

Guarded actions describe model behaviors via two parts: the *data-flow* part computes locals and outputs; the *control-flow* part computes *labels*, which denote the pause locations of control-flow. Guarded actions in the control-flow part are in the form of $\gamma \Rightarrow next(l) = true$, where label l is a non-memorized Boolean local denoting a pause location. If l holds at the current instant, it means that the control-flow reached the pause location of l at the end of the previous instant, then it resumes from this location at the beginning of the current instant. Note that more than one label can hold at the same instant. This enables the description of the parallelism feature [3] of synchronous programs.

B. The Signal Language

In Section II-B1, we introduce the polychronous model as the formal basis of Signal. Then, an overview of the language is given in Section II-B2.

1) Polychronous Model: We start with the following sets: $\mathbb{X}$ is a countable set of variables; $\mathcal{B} = \{ff, tt\}$ is a set of Boolean data values where *ff* and *tt* respectively denote *false* and *true*; $\mathcal{V}$ is a non-empty set of data values, $\mathcal{B} \subset \mathcal{V}$; and $\mathbb{T}$ is a dense set equipped with a partial order relation, denoted by $\leq$. The elements in $\mathbb{T}$ are called *tags*. We now introduce the notion of *time domain*.

Definition 1 (time domain)**.** *A time domain is a partially ordered set $(\mathcal{T}, \leq)$ where $\mathcal{T} \subset \mathbb{T}$ that satisfies: $\mathcal{T}$ is countable; $\mathcal{T}$ has a lower bound $0_\mathcal{T}$ for $\leq$, i.e., $\forall t \in \mathcal{T}, 0_\mathcal{T} \leq t$; $\leq$ over $\mathcal{T}$ is well-founded; the width of $(\mathcal{T}, \leq)$ is finite.*

$(\mathbb{T}, \leq)$ provides a continuous time dimension. $(\mathcal{T}, \leq)$ defines a discrete time dimension that corresponds to the logical instants [24], at which the presence/absence of data can be observed during the system execution. Thus, the mapping of $\mathcal{T}$ on $\mathbb{T}$ allows one to move from "abstract" descriptions to "concrete" descriptions [28].

A *chain* $(C, \leq) \subseteq (\mathcal{T}, \leq)$ is a totally ordered set of tags that admits a lower bound 0_C. The notation $t + 1$ means the immediate successor of a tag t in C, which satisfies $\forall t' \in C, t \leq t' \Rightarrow t + 1 \leq t'$. We denote the set of all chains in $\mathcal{T}$

by $\mathcal{C}_\mathcal{T}$.

Definition 2 (event). *An event on a given time domain $\mathcal{T}$ is a pair $(t, v) \in \mathcal{T} \times \mathcal{V}$, which associates a tag t with a data value v.*

All the events whose tags belong to the same chain, can constitute a data-flow. Formally,

Definition 3 (signal). *A signal $s : C \to \mathcal{V}$ is a function from a chain of tags to a non-empty set of data values, where $C \in \mathcal{C}_\mathcal{T}$. The domain of s is denoted by $tags(s)$.*

Two signals s_1 and s_2 are identical, denoted by $s_1 = s_2$, if and only if $tags(s_1) = tags(s_2)$ and $\forall t \in tags(s_1), s_1(t) = s_2(t)$. $\mathcal{S} = \cup_{C \in \mathcal{C}_\mathcal{T}}(s : C \to \mathcal{V})$ is the set of signals over the time domain $(\mathcal{T}, \leq)$.

Definition 4 (behavior). *Given a finite subset $\mathcal{X}$ of a countable set $\mathbb{X}$ of variables, a behavior over $\mathcal{X}$ is an injective function $b : \mathcal{X} \to \mathcal{S}$. The domain of b is denoted by $vars(b)$.*

The restriction of b over a set of variables X, denoted by $b|_X$, is the behavior defined by $vars(b|_X) = X \cap vars(b)$ and $\forall x \in vars(b|_X), (b|_X)(x) = b(x)$.

Two behaviors b_1 and b_2 are compatible, denoted by $b_1 \updownarrow b_2$, if and only if $b_1|_{vars(b_2)} = b_2|_{vars(b_1)}$. The composition of two compatible behaviors $b_1 : \mathcal{X}_1 \to \mathcal{S}$, $b_2 : \mathcal{X}_2 \to \mathcal{S}$ is a behavior $(b_1|b_2) : (\mathcal{X}_1 \cup \mathcal{X}_2) \to \mathcal{S}$ defined by $\forall x \in \mathcal{X}_1, (b_1|b_2)(x) = b_1(x)$ and $\forall x \in \mathcal{X}_2, (b_1|b_2)(x) = b_2(x)$.

Definition 5 (process). *A process p is a set of behaviors defined over the same domain. The domain is denoted by $vars(p)$, which satisfies $\forall b \in p, vars(b) = vars(p)$.*

Definition 6 (composition of processes). *The composition of two processes p_1 and p_2, denoted as $p_1|p_2$, is a process consisting of the compositions of any two compatible behaviors $b_1 \in p_1$ and $b_2 \in p_2$:*

$$p_1|p_2 = \{(b_1|b_2)|(b_1, b_2) \in p_1 \times p_2, b_1 \updownarrow b_2\}.$$

The notions presented in this section are sufficient to express the semantics of Signal elementary concepts within this polychronous model [24]. In the remainder of this paper, we also use them to formulate the optimization scheme.

2) An Overview of the Signal Language: Signal is a polychronous language processing unbounded series of typed values, called *signals*. At any tag t, a signal x (corresponding to $b(x)$ in the polychronous model, where b is a behavior) may be present or absent: when present (i.e., $t \in tags(b(x))$), it holds some value; when absent (i.e., $t \notin tags(b(x))$), it holds no value.

The presence status of x is denoted by its associated clock $\hat{x} : tags(b(x)) \to \{tt\}$. Furthermore, the Signal language supports clock calculation. The basic operations contain clock union $x_1 \hat{+} x_2$: $tags(b(x_1)) \cup tags(b(x_2)) \to \{tt\}$; clock intersection $x_1 \hat{*} x_2$: $tags(b(x_1)) \cap tags(b(x_2)) \to \{tt\}$; and clock difference $x_1 \hat{-} x_2$: $tags(b(x_1)) \backslash tags(b(x_2)) \to \{tt\}$. In order to enable reasoning on clock calculation, we define $\hat{0}$ for the empty clock (i.e., $\hat{0} : \emptyset \to \{tt\}$) and $[x]$ (resp. $[\neg x]$) for the clock at which tags a Boolean signal x holds the value *true* (resp. *false*).

a) Declarative Constraints

A Signal program declares constraints on the involved signals, that must be satisfied by both values and clocks. The constraints on clocks are referred to as *clock relations*, including synchronization relation $x_1 \hat{=} x_2$, i.e., $tags(b(x_1)) = tags(b(x_2))$; inclusion relation $x_1 \hat{\leq} x_2$, i.e., $tags(b(x_1)) \subseteq tags(b(x_2))$; and mutual exclusion relation $x_1 \hat{\#} x_2$, i.e., $tags(b(x_1)) \cap tags(b(x_2)) = \emptyset$.

Besides the explicit clock relations, constraints can be implicitly declared by the Signal equations. Each equation is a definition associating one defined signal with a Signal expression built on operators over signals. The operands of the operators can be signals or expressions.

There are two kinds of definitions:

- A *complete definition* ($:=$) is an equation in which the defined signal is assigned with the associated expression. For instance, $y := x$ is a complete definition of y that is assigned with x when x is present; y is synchronized with x:
$$\forall t \in tags(b(x)), b(y)(t) = b(x)(t) \text{ and } y \hat{=} x.$$
- A *partial definition* ($::=$) is an equation in which the defined signal is assigned with the associated expression when the expression is present. For instance, $y ::= x$ is a partial definition of y that is assigned with x when x is present; when x is not present, the value of y depends on other partial definitions:
$$\forall t \in tags(b(x)), b(y)(t) = b(x)(t).$$

According to the implied clock relations, the Signal expressions can be classified into two families: *synchronous expressions* (i.e., all the involved signals have the same clock) and *polychronous expressions* (i.e., the involved signals may have different clocks). The primitive expressions include

- *Instantaneous function*: $x := f(x_1, \ldots, x_n)$ defines a point-wise n-ary function on sequences of values, in which all the signals $x, x_1, \ldots, x_n$ are synchronous.
- *Delay*: $x := x'\$1$ **init** def_value defines that x and x' are synchronous; the current value of x is equal to the previous value of x' and equal to the default value def_value at the initial tag.
    - $x \hat{=} x'$
    - $b(x)(0_{\mathcal{C}_x}) = def_value$
    - $\forall t \in tags(b(x)), 0_{\mathcal{C}_x} < t+1 \Rightarrow b(x)(t+1) = b(x')(t)$

  where $\mathcal{C}_x = tags(b(x))$.
- *Downsampling*: $x := x'$ **when** b defines a downsampling of the signal x' that occurs only when both $\hat{x}'$ and the *downsampling condition* b hold *true*.
    - $x \hat{=} x' \hat{*} [b]$, where $[b] \hat{=}$**when** b
    - $\forall t \in tags(b(x)), b(x)(t) = b(x')(t)$
- *Deterministic merging*: $x := x_1$ **default** x_2 defines that the clock of x is the clock union of x_1 and x_2, its value is equal to x_1 when x_1 is present, otherwise equal to x_2 when x_1 is absent but x_2 is present.
    - $x \hat{=} x_1 \hat{+} x_2$
    - $\forall t \in tags(b(x_1)), b(x)(t) = b(x_1)(t)$
    - $\forall t \in tags(b(x_2)) \backslash tags(b(x_1)),$ $b(x)(t) = b(x_2)(t)$
- *Completion*: $x ::=$ **defaultvalue** x' completes the definition of x. Given the partial definitions $x ::= x_1, x ::= x_2$, where $x_1 \hat{\#} x_2$, x is then identical to x' when x_1 and x_2 are absent but x and x' are present.

Both explicit clock relations and equations are elementary processes. The composition of (elementary) processes defines a Signal program, which is a process as well. This composition leads to the conjunction of the involved constraints. Furthermore, the scope of the constraints on signals can be restricted by declaring the signals as local.

- *Composition of processes* defines a union of processes in which the involved constraints have to be simultaneously satisfied. The composition is in the form of $(|p_1|p_2|)$, where p_1 and p_2 are processes.
- *Restriction* defines a local signal declaration in a process. It is in the form of p **where x** or p/x, which means that the signal x is a local in the process p.

b) Conditional Dependencies

Besides constraints, the equations also implicitly express conditional dependencies between signals, which have to be strictly obeyed during the computations. A conditional dependency $x \xrightarrow{h} y$ specifies that the computation of signal y depends on signal x at the clock $h \hat{*} x \hat{*} y$. The dependencies are as follows:

$$\begin{array}{ll} x := f(x_1, \ldots, x_n) & x_1 \to x \cup \ldots \cup x_n \to x \\ x := x'\$1 \textbf{ init } def_value & \emptyset \\ x := x' \textbf{ when } b & x' \xrightarrow{[b]} x \\ x := x_1 \textbf{ default } x_2 & x_1 \to x \cup x_2 \xrightarrow{x_2 \hat{-} x_1} x \\ (|p_1|p_2|) & D_1 \cup D_2 \end{array}$$

Note that $x_1 \to x$ stands for $x_1 \xrightarrow{x_1 \hat{*} x} x$; D_1, D_2 are the dependencies respectively derived from processes p_1 and p_2.

To encode the derived dependencies, a bipartite *conditional dependency graph* (CDG) [29] is synthesized. It consists of vertices V representing signals and labeled edges representing the conditional dependencies. For instance, an edge labeled h from x to y represents $x \xrightarrow{h} y$.

c) Calculation Clock & Utility Clock

Recall that the clock of a signal denotes the presence status of the signal. This kind of clock is therefore also called *presence clock*. Furthermore, it is feasible to synthesize other kinds of clocks to exhibit various features of signals [30]. We introduce the notions of *calculation clock* and *utility clock* that will play a key role in the optimization process. Both of them are computed, based on the analysis of conditional dependencies.

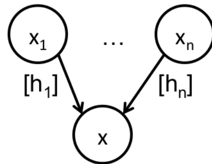

Fig. 1. Conditional Dependencies to x in Process p

The calculation clock of a signal x in a process p, denoted as $clk_{cal}(x)_p$, represents the tags at which the value of x is updated with a fresh value rather than with its previous value in p. The calculation clock can be computed by considering all the conditional dependencies from other signals to x. Given equations $x ::= x_1$ **when** $h_1, \ldots, x ::= x_n$ **when** h_n in process p, the conditional dependencies to x are synthesized as shown in Fig. 1. Its calculation clock is defined as follows:

$$clk_{cal}(x)_p \hat{=} \biguplus_{i=1}^{n} ([h_i] \hat{*} x_i).$$

where $\biguplus$ represents the accumulation of clock unions. Note that the calculation clock of an input is equal to its presence clock.

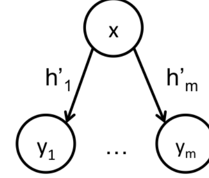

Fig. 2. Conditional Dependencies from x in Process p

The utility clock of the signal x in a process p, denoted as $clk_{uty}(x)_p$, represents the tags at which x is really useful for other signals computation in p. The utility clock can be computed by considering all the dependencies from x: x is useful when at least one target signal is useful and the corresponding condition clock holds. Fig. 2 illustrates the dependencies from x. Its utility clock is defined as follows:

$$clk_{uty}(x)_p \hat{=} \biguplus_{i=1}^{m} (h'_i \hat{*} clk_{uty}(y_i)_p).$$

Given a signal x in a process p, if its current value can be useful in the future in p, its utility clock is consequently equal to its presence clock. Note that the utility clock of an output in a process is equal to its presence clock in the process.

III. AIF Model to Signal Process

Starting from the generated intermediate model, the first step is to partition it into sub-models, according to user-specified topological annotations. After the partitioning, the generated models are transformed into Signal processes, in which the unnecessary constraints are then eliminated to avoid unnecessary synchronization.

A. Model Partitioning

We propose a partitioning method for AIF model, according to user-specified annotations. The given synchronous program describes the system behavior in a modular structure, the annotations can thereby specify a mapping of module instantiations onto the processing locations. For instance, the annotations

$$\begin{array}{rl} main & \{A\} \\ inst_1 : Module_x & \{B\} \\ inst_2 : Module_y & \{C\} \end{array}$$

specify that the main module (i.e., the entry of the program) is located on the location A, the instantiation $inst_1$ of $Module_x$ is located on B and the instantiation $inst_2$ of $Module_y$ is located on C. According to this mapping, the guarded actions of the AIF model are grouped into different classes.

From the guarded actions involved in each class, the I/O and local declarations are automatically generated, a sub-model is consequently synthesized. If the guarded actions computing the same variable x are grouped into multiple sub-models, this

means that the computation of x is distributed. For each sub-model that can compute x, a *completion input* x' is declared to receive the values of x computed by others.

In the generated sub-models, the variables are divided into three categories: *I/O* that connect a sub-model with the environment; *intermediates* that connect a sub-model with other sub-models; and *internals*, i.e., locals in a sub-model.

B. Guarded Actions to Declarative Constraints

After the partitioning, the transformation of generated models into Signal processes is performed. The *model name*, *interface* and *local declarations* of sub-models bijectively correspond to the homonymous parts in Signal processes. This section focuses on the transformation from guarded actions to Signal declarative constraints.

The transformation should preserve features of the source in the generated target, to guarantee the consistency between them. Furthermore, such features of the source can be checked within the framework of the target. The features of guarded actions are summarized as follows:

- Strict synchrony. All the variables included in the set of guarded actions are synchronous.
- No priority. No priority exists between the guarded actions that compute the same variable.
- Exclusivity. To avoid write conflicts [3], simultaneous assignment of different values to the same variable is forbidden. Given a pair of guarded actions assigning different values to the same variable, if both are immediate or delayed, they must be *simultaneously exclusive*; if one is immediate and another is delayed, then they must be *cross exclusive*, i.e., the activation of the delayed action implies that the immediate action must not be activated at the next tag.

To preserve these features, each guarded immediate action is transformed into a downsampling with synchronization relation:

$$\gamma \rightarrow x = \tau \quad \mapsto \quad \begin{array}{l}(|x ::= \tau \textbf{ when } \gamma \\ |x \hat{=} \tau \hat{=} \gamma|)\end{array}$$

The downsampling represents the behavior of the guarded action: when both $[\gamma]$ and $\hat{\tau}$ hold, τ is sampled; x is then immediately assigned with the evaluation result of τ.

For each guarded delayed action, its transformation should preserve the delay feature as well:

$$\begin{array}{ll} & \gamma \rightarrow next(x) = \tau \\ \mapsto & (|x ::= pre_x \textbf{ when } isDelayed_x \\ & |pre_x := nxt_x\$1 \textbf{ init } def_value \\ & |nxt_x ::= \tau \textbf{ when } \gamma \\ & |isDelayed_x := isNxt_x\$1 \textbf{ init false} \\ & |isNxt_x ::= \textbf{true when } \gamma \\ & |isNxt_x ::= \textbf{defaultvalue false} \\ & |x \hat{=} \tau \hat{=} \gamma \hat{=} pre_x \hat{=} nxt_x \hat{=} isNxt_x \hat{=} isDelayed_x \\ & |)/nxt_x, pre_x, isNxt_x, isDelayed_x \end{array}$$

The downsampling immediately occurs while the assignment delays for one tag. The Boolean flag $isNxt_x$ indicates if the downsampling occurs. The Boolean flag $isDelayed_x$ indicates if the assignment to x should take place. nxt_x and pre_x keep the values assigned to x.

By following this scheme, all the features of guarded actions are embodied in Signal declarative constraints: 1) the explicit synchronization relations stipulate the strict synchrony; 2) the partial definition "::=" suggests no priority among the assignments to the same signal; 3) the partial definitions must be checked to be exclusive as the guarded actions.

Besides the transformation of explicit actions, the transformation of absence reactions is also considered. The definition of a signal x has to be completed. If there is a completion input x', then the definition of x is completed by x':

$$(|x ::= \textbf{defaultvalue } x'|).$$

If not, the storage type determines that it is completed by its previous value or the default value of its data type.

C. Desynchronization of Downsampling Conditions

In guarded actions, each guard γ may be syntactically decomposed as the conjunction of one elementary sub-guard β consisting of labels, and one optional sub-guard λ that is derived from the conditional statements of synchronous program. β checks if the control-flow resumes from the desired location while λ checks if the conditions are satisfied. Thus, the corresponding downsampling condition γ is as follows:

$$\gamma ::= \beta \mid \lambda \textbf{ ''and'' } \beta$$

where λ consists of I/O, intermediate or internal signals (except label signals).

Labels are explicitly assigned only by guarded delayed actions. Correspondingly, β is always computed at the previous tag. This feature makes it feasible to avoid unnecessary synchronization between β and λ, with no change to γ. Each downsampling condition can be transformed by following the scheme:

$$\begin{array}{l}(|x ::= \tau \textbf{ when } \gamma \\ |\gamma := \lambda \textbf{ and } \beta \\ |x \hat{=} \tau \hat{=} \gamma|)\end{array} \quad \mapsto \quad \begin{array}{l}(|x ::= \tau \textbf{ when } \gamma \\ |\gamma := (\lambda \textbf{ when } \beta) \textbf{ default false} \\ |x \hat{=} \tau \hat{=} \gamma \hat{=} \beta \\ |\lambda \hat{=} \textbf{when } \beta|)\end{array}$$

After the transformation, the synchronization relation $\lambda \hat{=} \beta$ is weakened to $\gamma \hat{=} [\beta]$. The weakened synchronization relation makes the reduction of $\hat{\lambda}$ feasible (illustrated in Fig. 3).

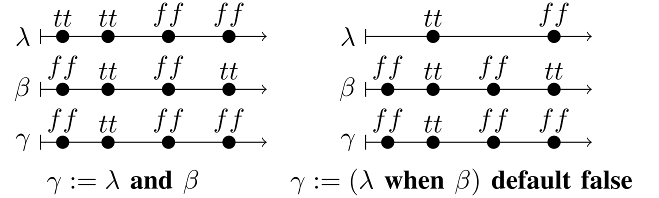

Fig. 3. Reduction of $\hat{\lambda}$ after desynchronizing λ **and** β

D. Desynchronization of Signals in Downsampling Equations

Synchronous expressions are built over synchronous operators, which impose synchronization relations between the operands. These imposed synchronization relations can cause unnecessary synchronization. For instance, in the downsampling equation $x := (o + y)$ **when** γ, $o + y$ imposes the synchronization relation $o \hat{=} y$, where y is a local and o is an output. Due to the preservation of interface behaviors, $\hat{o}$ is always equal to the master clock.

The downsampling of a synchronous expression can be transformed into the downsampling of each involved signal. Along this way, the imposed synchronization relations are weakened. For instance, after the transformation

$$x := (o + y) \textbf{ when } \gamma \quad \mapsto \quad x := (o \textbf{ when } \gamma) + (y \textbf{ when } \gamma)$$

the clock relation $o\hat{=}y$ is weakened to $o\hat{*}[\gamma]\hat{=}y\hat{*}[\gamma]$, i.e. y needs to be synchronized with o only when downsampling condition γ holds *true*. Then the reduction of $\hat{y}$ becomes feasible (illustrated in Fig. 4).

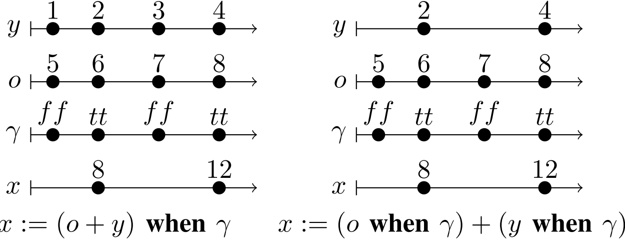

Fig. 4. Reduction of $\hat{y}$ after desynchronizing $(o + y)$ **when** γ

IV. Optimization using Multi-Clock Calculation

Within the Signal framework, a data-flow network can be automatically constructed by composing the generated processes. According to the user-specified pragmas, the generated processes will be mapped over different processing locations. As illustrated in Fig. 5, in the statement $p1 :: A\{\}$, $p1$ is declared as the delegate of the process A. The pragma *RunOn* specifies on which processing location the given process is located. For instance, $RunOn\ \{p1\}$ 1 specifies that the process delegated by $p1$ is mapped over the processing location 1. When two or more processes are mapped over the same processing location, these processes are transformed into a set of threads that are controlled by a local scheduler and share values via local memory. Hence, communications exist only between processing locations for value exchanges and are considered in the proposed optimization scheme.

```
process System =
  (? boolean i1, i2, i3;
   ! boolean o1, o2, o3;)
  pragmas
   Topology {i1, o1} 1
   Topology {i2, o2} 2
   Topology {i3, o3} 3
   Target MPI
   RunOn {p1} 1
   RunOn {p2} 2
   RunOn {p3} 3
  end pragmas
  (|   p1 :: A{}
   |   p2 :: B{}
   |   p3 :: C{}
   |)
```

Fig. 5. Signal Pragmas for Topological Specifications

Besides the process mappings, the topology of I/O and the implementation of the communications are also defined with pragmas [31]. The pragma *Topology* associates the I/O signals of the process with a processing location. For instance, $Topology\ \{i1, o1\}$ 1 tells that the input signal $i1$ and the output signal $o2$ are respectively received and produced on the processing location 1. The pragma *Target* specifies the API used to generate code for implementing the communications. For instance, $Target\ MPI$ tells that a MPI library is used for implementing the communications.

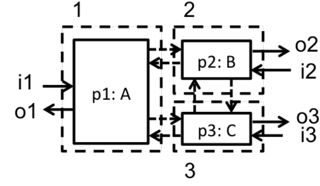

Fig. 6. Data-flow Network according to Pragmas in Fig. 5

According to the pragmas in Fig. 5, a data-flow network is constructed (see Fig. 6). The optimization of data-flow network focuses on reducing the inter-process communication quantity and the computation load, with no change to the interface behaviors between the network and the environment.

A. Formal Optimization Effects

The desired optimization effects are formulated, based on the polychronous model. We first state the formal definitions for the *reduction* and the *augmentation*.

Definition 7 (reduced/augmented signal)**.** *A signal s' is a reduced signal of signal s (or s is an augmented signal of s'), denoted as $s' \preceq s$, if and only if*

$$tags(s') \subseteq tags(s) \text{ and } \forall t \in tags(s'), s'(t) = s(t).$$

Definition 8 (reduced/augmented behavior)**.** *A behavior b' is a reduced behavior of behavior b (or b is an augmented behavior of b'), denoted as $b' \preceq b$, if and only if they own the same domain and given the same variable, the corresponding signal in b' is a reduced signal of the one in b, i.e.,*

$$vars(b') = vars(b) \text{ and } \forall x \in vars(b'), b'(x) \preceq b(x).$$

Definition 9 (set of reduced behaviors)**.** *A set of reduced behaviors for behavior b, denoted as $\mathfrak{B}^b$, satisfies that*

$$\forall b' \in \mathfrak{B}^b, b' \preceq b.$$

Definition 10 (set of augmented behaviors)**.** *A set of augmented behaviors for behavior b', denoted as $\mathfrak{B}_{b'}$, satisfies that*

$$\forall b \in \mathfrak{B}_{b'}, b' \preceq b.$$

A Signal process declares a set of constraints on signals. In the polychronous model, a process is a set of behaviors defined over the same domain. Furthermore, the behaviors included in the process must satisfy the declared constraints. A *process denotation*, denoted as $\langle p \rangle$, is thereby introduced to formally represent the largest set of behaviors that satisfy the constraints of the process p. In particular,

- $\langle x\hat{=}y \rangle$ expresses that in any behavior $b \in \langle x\hat{=}y \rangle$, the signals x and y should be synchronous, i.e.,

$$\{b \mid vars(b) = \{x, y\}, tags(b(x)) = tags(b(y))\}.$$

- $\langle [y]\hat{\leq}x\hat{\leq}y \rangle$ expresses that in any behavior $b \in \langle [y]\hat{\leq}x\hat{\leq}y \rangle$, the clock $\hat{x}$ has to be slower than $\hat{y}$ and faster than $[y]$, i.e.,

$$\begin{aligned}\{b \mid\ & vars(b) = \{x, y\},\\ & \mathfrak{T} = \{t \in tags(b(y)) | b(y)(t) = tt\},\\ & \mathfrak{T} \subseteq tags(b(x)) \subseteq tags(b(y))\}\end{aligned}$$

- $\langle o ::= x$ **when** $y\rangle$ expresses that in any behavior $b \in \langle o ::= x$ **when** $y\rangle$, the signal o is partially defined by the downsampling x **when** y, i.e.,

$$\{b \mid vars(b) = \{x, y, o\}, \mathfrak{T} = \{t' \in tags(b(x)) \cap tags(b(y)) \mid b(y)(t') = tt\}, \forall t \in \mathfrak{T}, b(o)(t) = b(x)(t)\}$$

The behaviors in $\langle o ::= x$ **when** $y\rangle$ can be further restricted by composing with explicit clock relations:

- $\langle o ::= x$ **when** $y \mid x \hat{=} y\rangle$ consists of the behaviors conforming to AIF model: $o ::= x$ **when** y corresponds to the guarded action $y \Rightarrow o = x$; $x \hat{=} y$ explicitly defines the synchronization relation that is implicit in AIF model.
- As the constraints in $(|x \hat{=} y|)$ is part of the ones in $(|[y] \hat{\leq} x \hat{\leq} y|)$, $\langle o ::= x$ **when** $y \mid x \hat{=} y\rangle$ is a subset of $\langle o ::= x$ **when** $y \mid [y] \hat{\leq} x \hat{\leq} y\rangle$.

Comparing the two compositions, we conclude the reduction/augmentation relations among their behaviors.

Proposition 1. *For each behavior* $b \in \langle o ::= x$ ***when*** $y \mid x \hat{=} y\rangle$*, there always exists a set of reduced behaviors* $\mathfrak{B}^b \subset \langle o ::= x$ ***when*** $y \mid [y] \hat{\leq} x \hat{\leq} y\rangle$*, s.t.,*

$$\forall b' \in \mathfrak{B}^b, b'(y) = b(y), b'(x) \preceq b(x), \forall t \in \mathfrak{T}, b'(o)(t) = b(o)(t),$$

where $\mathfrak{T} = \{t' \in tags(b(y)) \mid b(y)(t') = tt\}$.

This proposition states that the clock reduction of signal in downsampling equation does not change the downsampling result, as long as it is present when the condition holds. This is named *lossless reduction*. It formulates how to reduce the communication quantity and the computation load: if the given signal is really required by some computation (e.g., downsampling), it must be present; if not, its absence does not change the computation behavior.

Proposition 2. *For each behavior* $b' \in \langle o ::= x$ ***when*** $y \mid [y] \hat{\leq} x \hat{\leq} y\rangle$*, there always exists a set of augmented behaviors* $\mathfrak{B}_{b'} \subset \langle o ::= x$ ***when*** $y \mid x \hat{=} y\rangle$*, s.t.,*

$$\forall b \in \mathfrak{B}_{b'}, b'(y) = b(y), b'(x) \preceq b(x), \forall t \in \mathfrak{T}, b'(o)(t) = b(o)(t),$$

where $\mathfrak{T} = \{t' \in tags(b(y)) \mid b(y)(t') = tt\}$.

This proposition states that the signal in downsampling equation can be augmented by adding arbitrary values to be synchronized with the condition. This kind of augmentation operation is named *lossless augmentation*, since it does not change the downsampling result.

The optimization schemes proposed in the following section conform to both lossless reduction and lossless augmentation.

B. Optimization on Communications

1) General Idea: The Signal processes communicate via broadcast. Note that we consider the general case, i.e., each process can be a mixture of producer and receiver of a signal x. The processes can thus be capable of producing the data value of x or receiving the data value of x via the completion input x'. In the most general case, the input data is not always really required by the receiver. The broadcast can therefore lead to communication of redundant data, which has no effect to any computation in the receiver. More importantly, since the receiver has to read a data value from each input channel during each tag, reading redundant data can impair the operating efficiency of the receiver.

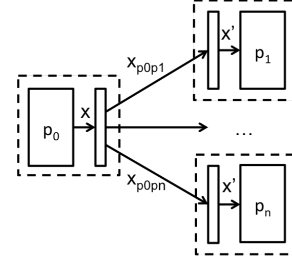

Fig. 7. Conditional Broadcast

To optimize the communications, the broadcast should evolve into conditional broadcast: the given data is only broadcasted to the receivers really requiring it. To achieve this evolution, each producer (e.g., p_0 in Fig. 7) is equipped with an adapter around it to selectively trigger the sending to the receivers (e.g., $p_1, \ldots, p_n$ in Fig. 7).

On the other hand, one signal can have multiple producers. A *merging* functionality is thus necessary for the receiver: it requires its adapter to transmit the data from the appropriate input channel. As illustrated in Fig. 8, when p_0 is a receiver of signal x, since the data of x can be produced by multiple producers $p_1, \ldots, p_n$, the adapter of p_0 should be capable of transmitting the data from the desired input channel to p_0.

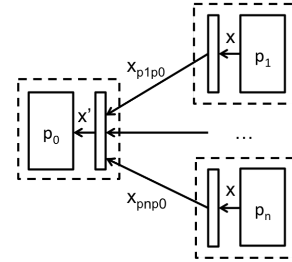

Fig. 8. Merging

The receiver's adapter should be also capable of resynchronizing the inputs [32]. Due to the asynchronous communications, the input data from different channels do not arrive simultaneously. The receiver's adapter must resynchronize the input data to meet the synchronization relations between the inputs. The crucial part of this functionality is to determine whether all the required input data values have arrived.

2) Constructing Adapters: To achieve these functionalities, the calculation clocks and the utility clocks of related signals have to be computed. Before presenting the clock calculation scheme, we first consider the general computation scheme for a given signal x. Assume that the computation of x is distributed over the processes $p_0, \ldots, p_n$, then its definition scheme in each process is as follows:

$$\begin{aligned}
&(|x ::= \tau_1 \textbf{ when } \gamma_1 \\
&\ \ldots \\
&|x ::= \tau_m \textbf{ when } \gamma_m \\
&|x ::= pre_x \textbf{ when } isDelayed_x \\
&|x ::= \textbf{defautlvalue } x' \\
&|x' \hat{=} x \hat{=} pre_x \hat{=} \gamma_1 \hat{=} \tau_1 \hat{=} \ldots \hat{=} \gamma_m \hat{=} \tau_m \hat{=} isDelayed_x|)
\end{aligned}$$

Based on the equations, a conditional dependency graph [33] is synthesized as shown in Fig. 9. According to the conditional

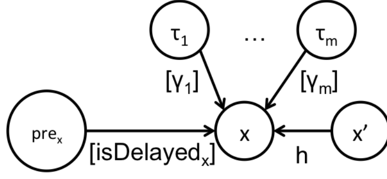

Fig. 9. Synthesized Conditional Dependency Graph

dependencies, the calculation clock of x in the process p_j ($0 \leq j \leq n$) is computed:

$$clk_{cal}(x)_{p_j} \hat{=} (\biguplus_{i=1}^{m}(\tau_i \hat{*} [\gamma_i])) \hat{+} (pre_x \hat{*} [isDelayed_x]).$$

It represents the tags at which the data of x is produced by p_j. Furthermore, the utility clock of the completion input x' in p_j is computed as follows:

$$\begin{aligned} clk_{uty}(x')_{p_j} &\hat{=} clk_{uty}(x)_{p_j} \hat{*} h \\ &\hat{=} clk_{uty}(x)_{p_j} \hat{*} (clk_{uty}(x)_{p_j} \hat{-} clk_{cal}(x)_{p_j}) \\ &\hat{=} clk_{uty}(x)_{p_j} \hat{-} clk_{cal}(x)_{p_j}. \end{aligned}$$

It represents the tags at which x' is required to complete x, i.e., x' is useful only when x is useful but not produced by p_j. Note that this clock calculation scheme covers all the potential topologies. For instance, if p_j is only a receiver of x, then the completion input x' is equivalent to x in the scheme: the calculation clock of x is empty (i.e., $clk_{cal}(x)_{p_j} \hat{=} \hat{0}$); the utility clock of x' is equal to the one of x in p_j.

Based on the computed clocks, the minimal frequency of the communication can be determined. The adapters at both ends of the communication follow the optimized communication scheme, which is adjusted according to the storage type of the transmitted data.

Recall that the non-memorized storage type means that the signal is completed by the default value of its data type when no assignment applies to it. Hence, the fresh value is never required by the receivers in the future. Assume that the data of x is transmitted from the producer p_0 to the receiver p_k ($1 \leq k \leq n$) via the channel $x_{p_0 p_k}$ (see Fig. 7). As one element in the *conditional broadcast* of x, $x_{p_0 p_k}$ transmits the fresh value only when the receiver really requires it.

$$\begin{aligned} (| x_{p_0 p_k} &\hat{=} clk_{cal}(x)_{p_0} \hat{*} clk_{uty}(x')_{p_k} \\ | x_{p_0 p_k} &:= x \textbf{ when } \hat{x}_{p_0 p_k} |) \end{aligned}$$

Assume that the adapter of p_0 receives the fresh values of x via the channels $x_{p_1 p_0}, \ldots, x_{p_n p_0}$ (see Fig. 8). Due to the exclusivity feature inherited from the guarded actions, at most one process can produce one fresh value of x during each tag. Their adapters thereby transmit the values at different tags, which is called *exclusive transmission*. Due to this feature, the adapter of the receiver can reorder the received values, according to the tagged order. When no fresh value of x is produced, the adapter of p_0 transmits the default value to p_0 via x'.

$$(| x' := x_{p_1 p_0} \textbf{ default } \ldots \textbf{ default } x_{p_n p_0} \textbf{ default } def_value |)$$

The memorized storage type means that the signal is completed by its previous value when no assignment applies to it. Hence, the fresh value can be required by the receivers in the future, even if it is not required at the current instant. The optimized communication must cover the potential future requirements. In a first optimization scheme, the adapter of the producer always transmits the fresh value to the receivers no matter whether they currently require it.

$$\begin{aligned} (| x_{p_0 p_k} &\hat{=} clk_{cal}(x)_{p_0} \\ | x_{p_0 p_k} &:= x \textbf{ when } \hat{x}_{p_0 p_k} |) \end{aligned}$$

The merging of data from multiple channels still follows the tagged order. When no value of x comes from the producers, the adapter transmits the previous value to p_0 via x'.

$$\begin{aligned} &(| x' := X \textbf{ when } \hat{x}' \\ &| X := x_{p_1 p_0} \textbf{ default } \ldots \textbf{ default } x_{p_n p_0} \\ &\qquad \textbf{default } X\$1 \textbf{ init } def_value \\ &| X \hat{=} x_{p_1 p_0} \hat{+} \ldots \hat{+} x_{p_n p_0} \hat{+} x' |)/X \end{aligned}$$

A second optimization scheme for transmitting memorized data, is to let the adapter of the producer transmit the value via the channel, only when the other end requires it.

$$\begin{aligned} (| x_{p_0 p_k} &\hat{=} clk_{uty}(x')_{p_k} \\ | x_{p_0 p_k} &:= x \textbf{ when } \hat{x}_{p_0 p_k} |) \end{aligned}$$

Note that this scheme is limited to the topology of *mono-producer of* x: if this scheme is applied to the topology of *multi-producer of* x, then the exclusive transmission feature is violated. For the topology of *mono-producer of* x, the merging functionality is not needed.

In the proposed optimization schemes, utility clock in receiver can be used to compute transmission clock, i.e., $\hat{x}_{p_0 p_k}$, in the adapter of producer. Such kind of transmission clock calculation may introduce dependency cycles between the communication ends. Hence, after computing a transmission clock with the related utility clock, the resulting processes have to pass the verification that no dependency cycle exists among them. If dependency cycle exists, the utility clock in receiver cannot be used to compute this transmission clock. In the sequel, this transmission clock only depends on the calculation clock in producer, i.e., $x_{p_0 p_k} \hat{=} clk_{cal}(x)_{p_0}$.

When all the required input data values arrive, the adapter of the receiver transmits them to trigger the receiver's execution. The set of inputs really required by a receiver p_0, denoted as $I(t)_{p_0}$, dynamically changes along with the receiver's execution. It is determined by iteratively checking the utility clocks of inputs in the receiver:

$$I(t)_{p_0} := \{x' \in I_{p_0} | t \in clk_{uty}(x')_{p_0}\}$$

where I_{p_0} is the set of inputs in p_0.

3) Optimization Effects: Based on the proposed optimization schemes, the constructed adapters implement lossless reduction and lossless augmentation: the inter-process communications can be reduced and each wrapped process still produces the same output data-flow. This key property guarantees that the optimized data-flow network preserves the original interface behaviors.

Fig. 10 illustrates the interface behaviors after constructing adapters for processes p_0 and p_1. p_1 produces the non-memorized data of x, which is required for the computation of an output o in p_0. The data of x is down-sampled and transmitted via the path $p_1 : x \rightarrow x_{p_1 p_0} \rightarrow x' \rightarrow p_0 : x$. The clock of $x_{p_1 p_0}$ is computed by following the optimization

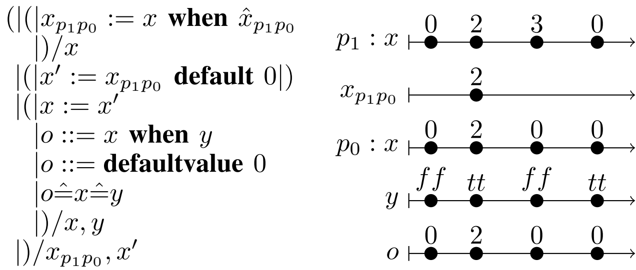

Fig. 10. Interface behaviors after the adapter construction

scheme:

$$x_{p_1p_0} \hat{=} clk_{cal}(x)_{p_1} \hat{*} clk_{uty}(x')_{p_0}.$$

As shown in the data-flow traces of Fig. 10, only when p_0 really requires x' (i.e., at the tags of $[y]$), the fresh values of x are transmitted to the adapter of p_0 via the channel x_{p1p0}. Then the adapter transmits the fresh values with the default value (0 for x) to p_0 via x'. After analyzing the optimized data-flow traces, we deduce that although the inter-process communication is reduced (e.g., only the fresh value 2 is transmitted to the adapter of p_0), the wrapped process p_0 still produces the same output data-flow.

Compared with the method presented in [20], the proposed optimization scheme is implemented using multi-clock calculation rather than iterative evaluation of communication necessity. It can thereby avoid overestimated computations that may occur when adopting the method of [20].

C. Optimization on Computations

The optimization on communications requires utility clocks and calculation clocks to determine the necessary communications. The optimization on computations requires utility clocks to determine the necessary computations. The utility clock of a given internal signal represents at which tags the internal is really required for the computation of some output (or intermediate). Then the clock of the internal can be reduced to its utility clock. In this way, its utility is always computed at first. Then if it is really required, its value computation is performed; if not, the computation is suppressed and its status is absent. This optimization can reduce the workload in computing internal signals. However, since utility clock is synthesized from conditional dependencies, the clock reduction to utility clock can introduce supplementary dependencies between signals. This optimization can consequently cause dependency cycles [30]. Hence, the optimized process must pass the verification that no dependency cycle exists in it.

The methodology presented in [21] works on AIF models, and follows a similar idea for avoiding unnecessary computations: computing required conditions for every local variable to determine whether its computation is really required for the computations of output variables. The main difference is that required conditions are computed as fixed-point equations, whose computational effort can become prohibitively high.

V. CASE STUDIES

The presented methodology has been implemented within the integrated framework *Quartz/Averest + Signal/Polychrony*. To illustrate and validate this methodology, a series of examples served as case studies. Each of them has been written in the Quartz language and distributed over different processing locations according to its modular structure.

The first example, *Traffic Control System* [3], is a control system designed for controlling the traffic order of a single tunnel between an island and the mainland. This system is distributed over five processing locations for different functionalities: two *traffic light controllers*, responsible for receiving the sensors' signals when cars enter or leave the tunnel and controlling the traffic lights at both ends of the tunnel; two *counters*, respectively counting the number of cars on the island or in the tunnel; and one *tunnel access controller*, responsible for managing access to the tunnel according to the numbers of cars and the sensors' signals. Using the presented methodology, an optimized data-flow network is finally established and can be simulated as optimized distributed C program. Its simulation result embodies the optimization effects on communications. Comparing with the simulation result of non-optimized distributed C program, we deduce that 1) the communication quantity on data (in particular, data of Boolean signals) is greatly reduced by 65%; 2) however, since the optimization requires additional control communications to retain system behavior, the global communication quantity is inversely increased by 9% (shown in Tab. I).

TABLE I. AMOUNT OF COMMUNICATION REDUCTION

Cast Study Name	Percentage of Communication Reduction
Traffic Control System	-9%
DFT-IDFT	30%
3D Transform	43%

This may appear as a negative result when there is a small amount of data that has to be communicated, even if this quantity is reduced, if, depending on the mapping, control communication has to be added to ensure a correct behavior. However, for a given mapping, the optimization on communication may be significant if the transmitted data are big data such as images represented as matrices. Consider, for instance, the previous application in which the Boolean data are replaced by 1024×1024 Boolean matrices. In that case, since the added control is the same, the global reduction of communication is almost 75%.

We have considered two other case studies: *DFT-IDFT* applies a Discrete Fourier Transform and an Inverse Discrete Fourier Transform in parallel to the incoming audio signals; *3D Transform* processes digital images for a camera in parallel, including calculating a transformation matrix and applying it to a sequence of vectors over different processing locations. Both systems focus on parallelism of big data processing. Using the presented optimization methodology, significant communication reductions (shown in Tab. I) are obtained for both of them.

VI. CONCLUSIONS

This paper presents a distribution methodology for synchronous programs. The key for the desynchronization contains the transformation from intermediate models into Signal processes, and the synthesis of optimized data-flow network using the multi-clock calculation. The optimization implements the lossless reduction and the lossless augmentation. It can therefore reduce the inter-process communication quantity and the process computation load, without changing the interface behaviors.

The proposed methodology utilizes the polychronous model for the distribution of synchronous programs. One perspective for future work is to coordinate synchronous components (in particular, Quartz modules) with Signal processes to finally construct an optimized data-flow network. This integrated methodology would fully exploit the advantages of both synchrony and polychrony.

REFERENCES

[1] G. Berry, "The Esterel v5 Language Primer," http://www-sop.inria.fr/esterel.org/, July 2000.

[2] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Dataflow Programming Language Lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, September 1991.

[3] K. Schneider, "The Synchronous Programming Language Quartz," Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, Internal Report 375, December 2009.

[4] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The Synchronous Languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[5] F. Rocheteau and N. Halbwachs, "Pollux: a Lustre based Hardware Design Environment," in *Proceedings of the international workshop on Algorithms and parallel VLSI architectures II*. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., 1992, pp. 335–346.

[6] J. Almeida, M. Frade, J. Pinto, and S. Melo de Sousa, "An Overview of Formal Methods Tools and Techniques," in *Rigorous Software Development*, ser. Undergraduate Topics in Computer Science. Springer London, 2011, pp. 15–44.

[7] D. Harel and A. Pnueli, *Logics and Models of Concurrent Systems*, ser. NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems. Springer-Verlag Berlin Heidelberg, 1985, vol. F13, ch. On the development of reactive systems, pp. 477–498.

[8] A. Girault, "A Survey of Automatic Distribution Method for Synchronous Programs," in *International Workshop on Synchronous Languages, Applications and Programs, SLAP'05*, ser. ENTCS, F. Maraninchi, M. Pouzet, and V. Roy, Eds. Edinburgh, UK: Elsevier Science, New-York, Apr. 2005.

[9] G. Berry and E. Sentovich, "An Implementation of Constructive Synchronous Programs in POLIS," *Formal Methods in System Design*, vol. 17, no. 2, pp. 135–161, 2000.

[10] J.-P. Paris, G. Berry, F. Mignard, P. Couronné, P. Caspi, N. Halbwachs, Y. Sorel, A. Benveniste, T. Gautier, P. Le Guernic, F. Dupont, and C. Le Maire, "Projet Synchrone : Les Formats Communs des Langages Synchrones," INRIA, Rapport de recherche RT-0157, 1993.

[11] P. Caspi, A. Girault, and D. Pilaud, "Distributing Reactive Systems," in *7th International Conference on Parallel and Distributed Computing Systems, PDCS'94*. Las Vegas (NV), USA: ISCA, Oct. 1994.

[12] A. Girault and C. Ménier, "Automatic Production of Globally Asynchronous Locally Synchronous Systems," in *International Workshop on Embedded Software, EMSOFT'02*, ser. LNCS, A. Sangiovanni-Vincentelli and J. Sifakis, Eds., vol. 2491. Grenoble, France: Springer-Verlag, Oct. 2002, pp. 266–281.

[13] A. Girault, X. Nicollin, and M. Pouzet, "Automatic Rate Desynchronization of Embedded Reactive Programs," *ACM Trans. Embedd. Comput. Syst.*, vol. 5, no. 3, pp. 687–717, Aug. 2006.

[14] D. Baudisch, J. Brandt, and K. Schneider, "Dependency-Driven Distribution of Synchronous Programs," in *Distributed and Parallel Embedded Systems (DIPES)*, M. Hinchey, B. Kleinjohann, L. Kleinjohann, P. Lindsay, F. Rammig, J. Timmis, and M. Wolf, Eds. Brisbane, Queensland, Australia: International Federation for Information Processing (IFIP), 2010, pp. 169–180.

[15] J. Brandt, M. Gemünde, K. Schneider, S. Shukla, and J.-P. Talpin, "Integrating System Descriptions by Clocked Guarded Actions," in *Forum on Specification and Design Languages (FDL)*, A. Morawiec, J. Hinderscheit, and O. Ghenassia, Eds. Oldenburg, Germany: IEEE Computer Society, 2011, pp. 1–8.

[16] Z. Yang, J.-P. Bodeveix, M. Filali, K. Hu, and D. Ma, "A Verified Transformation: From Polychronous Programs to a Variant of Clocked Guarded Actions," in *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '14. New York, NY, USA: ACM, 2014, pp. 128–137.

[17] J. Brandt, M. Gemünde, K. Schneider, S. Shukla, and J.-P. Talpin, "Embedding Polychrony into Synchrony," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 7, pp. 917–929, July 2013.

[18] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, "Elastic Circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 10, pp. 1437–1455, 2009.

[19] J. Brandt, M. Gemünde, and K. Schneider, "Desynchronizing Synchronous Programs by Modes," in *Application of Concurrency to System Design (ACSD)*, S. Edwards, R. Lorenz, and W. Vogler, Eds. Augsburg, Germany: IEEE Computer Society, 2009, pp. 32–41.

[20] D. Baudisch, Y. Bai, and K. Schneider, "Reducing the Communication of Message-Passing Systems Synthesized from Synchronous Programs," in *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. Turin, Italy: IEEE Computer Society, 2014.

[21] J. Brandt, K. Schneider, and Y. Bai, "Passive Code in Synchronous Programs," *Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, p. 67, January 2014.

[22] P. Le Guernic and T. Gautier, "Data-flow to von Neumann: the Signal Approach," in *Advanced topics in data-flow computing*, J.-L. Gaudiot and L. Bic Eds., Eds. Prentice Hall, 1990, pp. 413–438.

[23] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming Real-Time Applications with Signal," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1321–1336, 1991.

[24] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "Polychrony for System Design," *Journal of Circuits, Systems, and Computers*, vol. 12, no. 3, pp. 261–304, 2003.

[25] J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, and S. Shukla, "Constructive Polychronous Systems," in *Logical Foundations of Computer Science (LFCS)*, ser. LNCS, S. Artemov and A. Nerode, Eds., vol. 7734. San Diego, California, USA: Springer, 2013, pp. 335–349.

[26] J. Brandt and K. Schneider, "Separate Translation of Synchronous Programs to Guarded Actions," Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, Internal Report 382/11, March 2011.

[27] L. Lamport, "The Temporal Logic of Actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 872–923, May 1994.

[28] A. Gamatié and T. Gautier, "The Signal Synchronous Multi-Clock Approach to the Design of Distributed Embedded Systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 21, no. 5, pp. 641 –657, may 2010.

[29] O. Maffeïs and P. Le Guernic, "Combining Dependability with Architectural Adaptability by means of the Signal Language," in *Static Analysis*, ser. Lecture Notes in Computer Science, P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, Eds. Springer Berlin Heidelberg, 1993, vol. 724, pp. 99–110.

[30] B. Chéron, "Transformations Syntaxiques de Programmes Signal," Ph.D. dissertation, Université de Rennes I, September.

[31] L. Besnard, T. Gautier, P. Le Guernic, and J.-P. Talpin, "Compilation of Polychronous Data Flow Equations," in *Synthesis of Embedded Software*, S. Shukla and J.-P. Talpin, Eds. Springer US, 2010, pp. 1–40.

[32] Y. Bai, J. Brandt, and K. Schneider, "Monitoring Distributed Reactive Systems," in *High Level Design Validation and Test Workshop (HLDVT)*. Huntington Beach, California, California: IEEE Computer Society, 2012, pp. 84–91.

[33] O. Maffeïs and P. Le Guernic, "Distributed Implementation of Signal: Scheduling & Graph Clustering," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, ser. Lecture Notes in Computer Science, H. Langmaack, W.-P. Roever, and J. Vytopil, Eds. Springer Berlin Heidelberg, 1994, vol. 863, pp. 547–566.