Behavioral type inference for compositional system design

Jean-Pierre Talpin, David Berner, Paul Le Guernic, Abdoulaye Gamatié, Rajesh Gupta, Sandeep Shukla

▶ **To cite this version:**

Jean-Pierre Talpin, David Berner, Paul Le Guernic, Abdoulaye Gamatié, Rajesh Gupta, et al.. Behavioral type inference for compositional system design. [Research Report] RR-5141, INRIA. 2004. inria-00071442

HAL Id: inria-00071442

https://inria.hal.science/inria-00071442

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Behavioral type inference for compositional system design

Jean-Pierre Talpin, David Berner, Sandeep Shukla, Paul Le Guernic, Abdoulaye Gamatié, Rajesh Gupta

N°5141

Mars 2004

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--5141--FR+ENG

Behavioral type inference for compositional system design

Jean-Pierre Talpin, David Berner, Sandeep Shukla*, Paul Le Guernic, Abdoulaye Gamatié, Rajesh Gupta†

Thème 1 — Réseaux et systèmes
Projet Espresso

Rapport de recherche nΣ5141 — Mars 2004 — 44 pages

Abstract: The design productivity gap has been recognized by the semiconductor industry as one of the major threats to the continued growth of system-on-chips and embedded systems. Ad-hoc system-level design methodologies, that lifts modeling to higher levels of abstraction, and the concept of intellectual property (IP), that promotes reuse of existing components, are essential steps to manage design complexity. However, the issue of compositional correctness arises with these steps. Given components from different manufacturers, designed with heterogeneous models, at different levels of abstraction, assembling them in a correct-by-construction manner is a difficult challenge. We address this challenge by proposing a process algebraic model to support system design with a formal model of computation and serve as a behavioral type system to capture the behavior of system components at the interface level. The proposed algebra is conceptually minimal, equipped with a formal semantics defined in a synchronous model of computation, and supports a scalable notion and a flexible degree of abstraction. We demonstrate its benefits by considering the type-based synthesis of latency-insensitive protocols. We show that the synthesis of component wrappers can be optimized by the behavioral information carried by interface type descriptions and yield minimized stalls and maximized throughput.

Key-words: polychronous model of computation, component-based engineering, embedded system design, SystemC

(Résumé : tsvp)

* Virginia Tech
† University of California at San Diego

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Inférence de types pour la conception de systèmes enfouis

Résumé : L'ingnierie d'applications embarques en avionique passe par plusieurs tapes de conception allant du prototypage sur station de travail, la ralisation de simulateurs puis le dploiement de composants logiciels sur des architectures embarques. La plate-forme Polychrony permet d'aider la conception d'application dans ce flot de conception en permettant la capture en amont de modlisation de haut niveau (par exemple en Java temps rel) et leur spcialisation, c'est--dire la production, correcte par construction, d'excutifs temps rel, en utilisant les techniques formelles mises en œuvre dans la plate-forme Polychrony.

Mots-clé : modèle de calcul polychrone, conception par composants, conception de systèmes enfouis, SystemC

1 Introduction

The design productivity gap has been recognized by the semiconductor industry as one of the major threats to the continued growth of complex system-chips and their applications. System level design methodology that moves from RTL-based design entry into design methods at higher-level of abstraction is essential to manage design complexity. A number of advances in high-level modeling and validation have been proposed over the past decade in an attempt to improve the level of abstraction in system design, most of these enable greater reuse of existing intellectual property (IP) blocks.

Design correctness is an important part of this move. Given the complexity of system-level designs, it is important that the composition of system-level IP blocks be guaranteed correct. However, *a posteriori* validation of component compositions is a difficult problem. Techniques are needed that ensure design correctness as a part of the design process itself. To address this issue, methodological precepts have been developed that separately focus on design reuse and correctness by construction.

Both reuse and elevation of abstraction at design entry critically depend on guaranteed design correctness. To enable design reuse and raise the level of design entry there is industry momentum towards standardization efforts such as the Virtual Socket Interface Alliance (VSIA) and Accellera. These efforts target standards in specification of the interfaces, test data and properties that should be verified in design of components for system-chips.

At the same time with the advent of high-level design specification languages from SystemC, SpecC to Superlog, a promise is held out for a successful move towards designing at the system level through adoption of a dominant high-level specification language standard. While useful, standardization alone - whether of design specification languages or of design properties - will not solve the fundamental problem of design correctness. Techniques are needed to ensure correctness through the composition process itself.

To improve the state of the art in component composition from existing IP libraries, we specifically seek to address the following issue: given a high level architectural description (e.g., a virtual architecture) and a library of implemented components, how can one automate the selection of implementation of virtual components from the library, and automatically ensure composability of data and behavior?

Our approach is based on a high-level modeling and specification methodology that ensures compositional correctness through a type theory capturing behavioral aspects of component interfaces. The proposed system builds upon previous work on scalable design exploration using refinement/abstraction-based design methodologies [26], implemented in the Polychrony workbench [27], and on a layered Component Composition Environment, Balboa [11], which allows specification of SystemC components with mixed levels of structural and behavioral details and high degree of concurrency and timing requirements.

Our behavioral type system consists of a minimalist formalism, called the iSTS (implicit synchronous transition systems), that is akin to Pnueli's synchronous transition systems (STS, [23]) and Dijkstra's guarded commands [9]. The iSTS is used as a type system to describe the behavior of system components and allow for global model transformations to be performed on the system based on behavioral type information. It is equipped with a formal semantics defined in a multi-clocked synchronous model of computation [16] and implemented by the Polychrony workbench [27].

1.1 Roadmap

We put principles of type theory to work for the definition of a behavioral type inference system targetting the high-level design language standard SystemC. Starting with an informal introduction to our behavioral type inference technique, we give a detailed and informal exposition of the model of computation and of the notation it relies on, Sections 2 and 3. This exposition yields a behavioral type system that is both conceptually minimal and equipped with a formal semantics defined in a synchronous model of computation [16].

After an introduction to the core of SystemC under consideration, Section 4, we present our behavioral type inference method, Section 4.2. Being defined for a static single-assignment (SSA) intermediate representation of programs, the proposed inference technique is generic and language-independent. It is generalized

to the structuring elements of the SystemC class system, Section 13. This extension yields the formulation of a SystemC design methodology that reduces compositional design correctness verification to the validation of synthesized proof obligations.

Applications of the technique to the compositional assembly of System modules and classes, and to the verification of design correctness properties are outlined, Section 4.4. Its implementation using the Polychrony workbench is described, Section 4.9.

1.2 Rationale

To allow for an easier grasp on the proposed behavioral type inference technique, we outline the analysis of a small fragment of a SystemC program, Figure 1, the construction of its dynamic behavioral type, Figure 2, and the inference of its static abstraction, Figure 3. Then we elaborate the notion of proof obligations synthesis by giving a brief outline of the design correctness issues which can be modeled and checked in the framework of our type system.

Figure 1, left, considers a simple C code fragment. It consists of the iterative program that counts the number of bits set to one in a data. While the processed input variable idata is not equal to zero, the program adds the right-most bit of the input data and adds it to the output count variable ocount and then shifts the input variable right in order to process the next bit.

In the static single-assignment representation (SSA) of this program, right, every variable, idata or ocount is read and written only once per iteration. Label L2 corresponds to the entry point of the SSA block that corresponds to the while loop. The first instruction consists of loading the input variable idata into the register T1 and the second of storing the result of its comparison with 0 in the register T0. If T0 is false then control is passed to the next block L3. If it is true then the execution of the block is continued. It proceeds by loading the current value of the variable ocount into T2 and the last bit of T1 into T3 before assigning their sum to ocount and the right-shift of T1 to idata. It terminates with an unconditional branch back to label L2.

(C source)	(SSA code)
while (idata != 0)	L2:T1 = idata;
{	T0 = T1 != 0;
ocount = ocount	if T0 then goto L3;
+ (idata & 1);	T2 = ocount;
idata = idata >> 1;	T3 = T1 & 1;
}	ocount = T2 + T3;
	idata = T1 >> 1;
	goto L2;

Figure 1: Translation of a source program into static single assignment form

Although particularly verbose, the SSA intermediate representation of an imperative program can present an otherwise arbitrarily obfuscating C program in a form that can be easily manipulated by an automatic program analyzer. Let us zoom on the block L2 in the example of Figure 2, left.

The behavioral type of the block L2 consists of the simultaneous composition of the logical proposition on the right. Each proposition is associated to one SSA instruction. It specifies the *invariants* of this instruction. In particular, it tells when it is executed, what it computes and when it passes control to the next statement or branches to another block.

In the first line, for instance, we associate the instruction T1 = idata of block label L2 to the proposition $x_{\mathsf{L2}} \Rightarrow \mathsf{T1} = \mathsf{idata}$. In this proposition, the new variable x_{L2} is a boolean that is true iff the label L2 is being executed. So, the proposition says that, if the label L2 is being executed, then T1 is always equal to idata.

If not, then another proposition may hold. In our case, all subsequent propositions are conditioned by x_{L2} meaning that they hold when L2 is being executed.

The extent of a proposition is for the duration of a reaction. A reaction can be an arbitrarily long period of time provided that it is finite and that every variable or register changes its value at most once during that period. For instance, consider the instruction if T0 then L3. It is likely that label L3 will, just as L2, perform some operation on the input idata. Therefore, its execution is delayed until after the current reaction. We refer to x'_{L3} as the next value of the state variable x_{L3}, to indicate that it will be active during the next reaction. Hence, the proposition $x_{\mathsf{L2}} \Rightarrow \mathsf{T0} \Rightarrow x'_{\mathsf{L3}}$ says that control will be passed to L3 at the next reaction when control is presently at L2 and when T0 is true. The instructions that follow this test are conditioned by the negative $\neg$T0, this means: "in the block L2 and not in its branch to L3".

(SSA code)	(behavioral type)
L2:T1 = idata;	$x_{\mathsf{L2}} \Rightarrow \mathsf{T1} := \mathsf{idata}$
T0 = T1 != 0;	$\mathsf{T0} := (\mathsf{T1} \neq 0)$
if T0 then goto L3;	$\mathsf{T0} \Rightarrow x'_{\mathsf{L3}}$
T2 = ocount;	$\neg\mathsf{T0} \Rightarrow \mathsf{T2} := \mathsf{ocount}$
T3 = T1 & 1;	$\mathsf{T3} := \mathsf{T1} \& 1$
ocount = T2 + T3;	$\mathsf{ocount}' := \mathsf{T2} + \mathsf{T3}$
idata = T1 >> 1;	$\mathsf{idata}' := \mathsf{T1} >> 1$
goto L2;	x'_{L2}

Figure 2: Behavioral type of the SSA program

We have seen that every instruction of the SSA program can be associated to a proposition that accurately renders its control and data flow behaviors. This representation provides a both formal and expressive way to model, analyze, optimize and verify the behavior of ordinary SystemC programs. To ease both optimization and verification of such programs based on that representation, we abstract it over its control flow, characterized by boolean relations between *clocks*, and its data flow, characterized by scheduling relations between *signals*.

Let us first define this terminology. A clock $\hat{x}$ is associated to a signal x. The signal x corresponds to the flow of the successive values of a variable, sampled by the discrete periods of time that we call reactions. The clock of $\hat{x}$ denotes that set of periods or instants.

Figure 3, all operations on integers and bits reported in the behavioral type on the left have been abstracted by boolean relations between clocks, middle, and by scheduling relations, right. We show, Section 3.4, that this is in fact sufficient information to reconstruct the entire control and data flow graphs of the program. All the abstracted information essentially consists of computations which can be used to decorate these graphs and regenerate the original program.

For instance, the instruction $\mathsf{T0} = (\mathsf{T1}! = 0)$ is abstracted by the type $x_{\mathsf{L2}} \Rightarrow \hat{\mathsf{T1}} = \hat{\mathsf{T0}}$. It means: "when the block L2 is executed, T0 is present iff T1 is present". The scheduling constraint $x_{\mathsf{L2}} \Rightarrow \mathsf{T0} \rightarrow x_{\mathsf{L3}}$ additionally says that "x'_{L3} cannot happen before T0 at L2". Indeed, one first needs to examine the status of T0 before branching to L3.

The type associated with the block L2 uses the clocks denoted by the booleans x_{L2}, T0 and $\neg$T0. Each clock denotes a branch in the control-flow graph of the block L2. The other clocks, e.g. $\hat{\mathsf{T1}}$, denote the presence of data. They are partially related to the "*label*" clocks x_{L2}, T0 and $\neg$T0.

Section 3.4 describes how the control and data flow graphs of block L2 can be entirely regenerated or transformed starting from this type information. Section 4.4 develops the use of the dynamic or static information provided by the behavioral type inference system to perform design correctness checks.

(dynamic type)	(static type)	(scheduling)
$x_{\mathsf{L2}} \Rightarrow \mathsf{T1} = \mathsf{idata}$	$x_{\mathsf{L2}} \Rightarrow \hat{\mathsf{T1}} = \hat{\mathsf{idata}}$	$\mathsf{T1} \leftarrow \mathsf{idata}$
$\mathsf{T0} = (\mathsf{T1} \neq 0)$	$\hat{\mathsf{T0}} = \hat{\mathsf{T1}}$	$\mathsf{T1} \rightarrow \mathsf{T0}$
$\mathsf{T0} \Rightarrow x'_{\mathsf{L3}}$	$\mathsf{T0} \Rightarrow x'_{\mathsf{L3}}$	$\mathsf{T0} \rightarrow x'_{\mathsf{L3}}$
$\neg \mathsf{T0} \Rightarrow \mathsf{T2} = \mathsf{ocount}$	$\neg \mathsf{T0} \Rightarrow \hat{\mathsf{T2}} = \hat{\mathsf{ocount}}$	$\mathsf{T2} \leftarrow \mathsf{ocount}$
$\mathsf{T3} = \mathsf{T1} \& 1$	$\hat{\mathsf{T3}} = \hat{\mathsf{T1}}$	$\mathsf{T1} \rightarrow \mathsf{T3}$
$\mathsf{ocount}' = \mathsf{T2} + \mathsf{T3}$	$\hat{\mathsf{ocount}} = \hat{\mathsf{T2}} \wedge \hat{\mathsf{T3}}$	$\mathsf{T2} \rightarrow \mathsf{ocount}' \leftarrow \mathsf{T3}$
$\mathsf{idata}' = \mathsf{T1} >> 1$	$\hat{\mathsf{idata}} = \mathsf{T1}$	$\mathsf{T1} \rightarrow \mathsf{idata}'$
x'_{L2}	x'_{L2}	$\mathsf{T0} \rightarrow x'_{\mathsf{L2}}$

Figure 3: Static abstraction of the behavioral type

The most salient feature of the behavioral type system is yet the capability to reduce compositional design correctness verification to the validation of synthesized proof obligations. It is presented in the context of the inference system proposed for the SystemC module system, Section 4.3.

As an example, consider a class whose virtual fields are two clocks x and y, and a procedure f. It defines an interface, named m_0, and will be used to type another class. Next, assume an explicit behavioral type declaration #TYPE(f, Q) which associates the procedure f with a description of its behavior: the proposition Q. Its aim is to associate the virtual class field f, a method, to the denotation of all possible implementations satisfying an expected functionality.

```
class m0 {
   virtual sc_clock x;
   virtual sc_clock y;
   virtual void f() {} #TYPE(f, Q)
};
```

Next, we associate the interface m_0 with the class parameter m_1 of a template class m_2. The interface m_0 now gives a behavioral type to the method f in the class parameter m_1 expected by the module m_2. Indeed, the template class m_2 uses the class parameter m_1, that implements m_0, to launch a thread $m_1.f$ sensitive to x. The behavioral type Q, which gives an assumption on the behavior of $m_1.f$, is required to provide a guarantee on the behavior of the module m_2, produced by the template class.

```
template ⟨class m1⟩ #TYPE(m1, m0)
   SC_MODULE(m2) {SC_CTOR(m2) {
      SC_THREAD(m1.f) sensitive ≪ x
   }
};
```

Let m_3 be a candidate parameter for the template class m_2. It structurally implements the interface m_0, because it provides the clocks x and y and defines the method f by the program *pgm*. Using the type inference technique previously outlined, the program *pgm* is associated with a proposition P, that describes its behavioral type, and the class m_3 be decorated with the corresponding type declaration #TYPE(f, P).

```
class m3 {
   sc_clock x; sc_clock y;
   void f() { pgm} #TYPE(f, P)
};
```

Finally, let m_4 be the class defined by the instantiation of the template m_2 with the actual parameter m_3. To check the compatibility of the actual parameter m_3 with the formal parameter m_0, we need to establish

the containment of the behaviors denoted by the proposition P, the behavioral type of the actual parameter, in the denotation of the proposition Q, the type abstraction declared in m_0.

$$m_2\langle m_3\rangle\, m_4 \text{ is type-safe iff } \models P \Rightarrow Q$$

This amounts to check that the proposition P implies Q. The validation of this proof obligation can either be implemented using model checking (if P and Q are *dynamic* interfaces) or using SAT checking, if Q is a *static* interface, by calculating the static abstraction $\hat{P}$ of P and by verifying that $\hat{P}$ implies Q.

2 A polychronous model of computation

To ground a formal modeling and design methodology for architecture design using SystemC on solid bases, we consider a mathematical framework that establishes a continuum from synchrony to asynchrony, a multi-clocked synchronous model: the polychronous model of computation [16]. Polychrony allows to capture design, transformation, verification issues within the same model and hence independently of spatial and temporal considerations implied by a local synchronous viewpoint and/or a global asynchronous viewpoint.

The definition of uniform methodologies for the formal design of GALS architectures has been the subject of recent and detailed studies in [16] and [26]. In the present article, we cast polychrony in the context of a behavioral type system to explore abstraction and refinement relations between system-level models in a way geared towards the aim of compositional system design.

The polychronous model of computation, proposed in [16], consists of a unique *domain* of traces, that does not differentiate synchrony from asynchrony, and semi-lattice structures, that render synchrony and asynchrony using specific timing equivalence relations.

2.1 Domain of polychrony

We consider a partially-ordered set $(\mathcal{T}, \leq, 0)$ of tags. A tag $t \in \mathcal{T}$ denotes a symbolic instant or a period in time. We note $C \in \mathcal{C}$ a *chain* of $\mathcal{T}$. Events, signals, behaviors, and processes are defined starting from tags as follows:

Definition 1 (polychrony)
- *An* event $e \in \mathcal{E} = \mathcal{T} \times \mathcal{V}$ *is the pair of a value and a tag.*
- *A signal* $s \in \mathcal{S} = C \to \mathcal{V}$ *is a function from a* chain *of tags to values.*
- *A* behavior $b \in \mathcal{B}$ *is a function from names* $x \in \mathcal{X}$ *to signals* $s \in \mathcal{S}$*.*
- *A* process $p \in \mathcal{P}$ *is a set of behaviors that have the same domain.*

Notations In the remainder, we write:
- $\text{tags}(s)$ for the tags of a signal s
- $\text{tags}(b) = \cup_{x \in \text{vars}(b)} \text{tags}(b(x))$ for the tags of a behavior b
- $b|_X$ for the projection of a behavior b on $X \subset \mathcal{X}$
- $b/X = b|_{\text{vars}(b) \setminus X}$ for its complementary
- $\text{vars}(b)$ and $\text{vars}(p)$ for the domains of b and p

The synchronous composition $p \,|\, q$ of two processes p and q is defined by the union of all behaviors b (from p) and c (from q) that are synchronous: all signals along the interface $I = \text{vars}(p) \cap \text{vars}(q)$ between p and q carry the same values at the same time tags.

$$p \,|\, q = \{ b \cup c \mid (b, c) \in p \times q, I = \text{vars}(p) \cap \text{vars}(q), b|_I = c|_I \}$$

Figure 4 depicts a behavior b in the polychronous domain $\mathcal{P}$. Tags t_1 and t_2 (top and middle green signals) are equal, meaning that the events they time are synchronous. Tag t_1 precedes t_3, written $t_1 < t_3$, to mean the scheduling relation that causally relates them in time. The blue signal at the bottom has no tag comparable to either of the other green ones, e.g. $t_4 \lesseqgtr\!\!\!\!/\; t_5$. It denotes a signal belonging to a different clock domain.

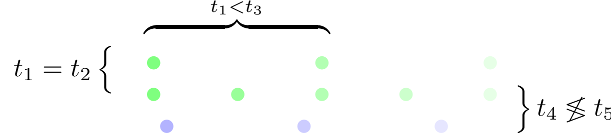

Figure 4: A behavior in the polychronous model of computation

2.2 Scheduling structure of polychrony

To render the scheduling relations between events occurring at the same time tag t, we refine the domain of polychrony with a scheduling relation, noted $t_x \rightarrow t'_y$, to mean that the event along the signal named y at t' may not happen before x at t.

The domain of dates $\mathcal{D} = \mathcal{T} \times \mathcal{X}$ of a given behavior b is subject to a pre-order relation $\rightarrow^b$ that denotes scheduling and contains causality $<$. When no ambiguity is possible on the identity of b in a scheduling constraint $x \rightarrow^b y$, we write it $x \rightarrow y$.

$$\begin{aligned} &\forall b \in \mathcal{B}, \forall x \in \text{vars}(b), \forall t, t' \in \text{tags}(b(x)), \\ &\qquad t < t' \Rightarrow t_x \rightarrow^b t'_x \wedge t_x \rightarrow^b t'_x \Rightarrow \neg(t' < t) \end{aligned}$$

Figure 5 depicts three scheduling relations superimposed to the signals x and y, Figure 4. The scheduling relation $t_x \rightarrow t_y$ denotes the observation that the event occurring along x at t precedes the event along y.

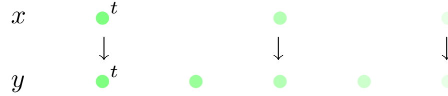

Figure 5: Scheduling relations between simultaneous events

The pair t_x of a time tag t and of a signal name x renders the very date d of an event along the signal x at the symbolic time t. The tag t itself represents the period during which multiple events take place to form a reaction: the tag t corresponds is the equivalence class of a synchronization relation between dates d, as in the synchronous structures [22].

2.3 Synchronous structure of polychrony

In the previous section, we gave a the structural definition of a domain of processes to allow for capturing the possible behavior of processes in the iSTS algebra. Building upon this domain, we define the semi-lattice structure which relationally denotes synchronous behaviors in this domain.

Figure 6 depicts the intuition behind this relation. It is to consider a signal as an elastic with ordered marks on it (tags). If the elastic is stretched, marks remain in the same relative and partial order but have more space (time) between each other.

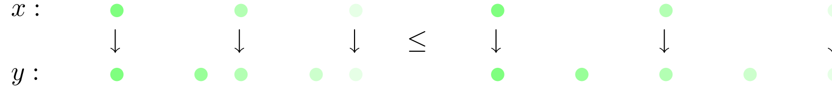

Figure 6: Relating synchronous behaviors by stretching.

The same holds for a set of elastics: a behavior. If elastics are equally stretched, the order between marks is unchanged. Figure 6, the time scale of x and y change but the partial timing and scheduling relations are preserved. Stretching is a partial-order relation that defines clock equivalence (Definition 2).

Definition 2 (clock equivalence)

A behavior c is a stretching *of b, written $b \leq c$, iff* $\text{vars}(b) = \text{vars}(c)$ *and there exists a bijection f on $\mathcal{T}$ that satisfies*

$$\left|\begin{array}{l} \forall t, t' \in \text{tags}(b), t \leq f(t) \wedge (t < t' \Leftrightarrow f(t) < f(t')) \\ \forall x, y \in \text{vars}(b), \forall t \in \text{tags}(b(x)), \forall t' \in \text{tags}(b(y)), t_x \rightarrow^b t'_y \Leftrightarrow f(t)_x \rightarrow^c f(t')_y \\ \forall x \in \text{vars}(b), \text{tags}(c(x)) = f(\text{tags}(b(x))) \wedge \forall t \in \text{tags}(b(x)), b(x)(t) = c(x)(f(t)) \end{array}\right.$$

b and c are clock-equivalent*, written $b \sim c$, iff there exists a behavior d s.t. $d \leq b$ and $d \leq c$.*

2.4 Asynchronous structure of polychrony

The asynchronous structure of polychrony is modeled by weakening the clock-equivalence relation to allow for comparing behaviors w.r.t. the sequences of values signals hold regardless of the time at which they hold these values.

The *relaxation* relation allows to individually stretch the signals of a behavior in a way preserving scheduling constraints. Relaxation is a partial-order relation that defines the flow-equivalence relation: two behaviors are flow-equivalent iff their signals hold the same values in the same order.

Figure 7 depicts two asynchronously equivalent behaviors related by relaxation. The first event along x has been shifted as the effect of delaying its transmission using e.g. a FIFO buffer.

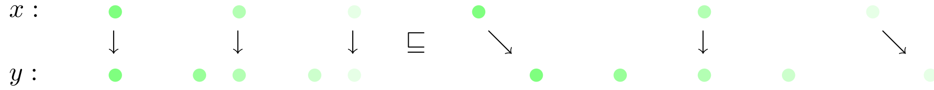

Figure 7: Relating asynchronous behaviors by relaxation.

Relaxation is a partial-order relation that defines flow-equivalence (Definition 3).

Definition 3 (flow equivalence)

A behavior c is a relaxation *of b, written $b \sqsubseteq c$, iff* $\text{vars}(b) = \text{vars}(c)$ *and, for all $x \in \text{vars}(b)$, there exists a bijection f_x on $\mathcal{T}$ that satisfies*

$$\left|\begin{array}{l} \forall t, t' \in \text{tags}(b(x)), t \leq f_x(t) \wedge t < t' \Leftrightarrow f_x(t) < f_x(t')) \\ \forall t \in \text{tags}(b(x)), \forall t' \in \text{tags}(b(y)), t_x \rightarrow^b t'_y \Leftrightarrow (f_x(t))_x \rightarrow^c (f_y(t'))_y \\ \text{tags}(c(x)) = f_x(\text{tags}(b(x))) \wedge \forall t \in \text{tags}(b(x)), b(x)(t) = c(x)(f_x(t)) \end{array}\right.$$

b and c are flow-equivalent*, written $b \approx c$, iff there exists a behavior d such that $d \sqsubseteq b$ and $d \sqsubseteq c$.*

Asynchronous composition $p \parallel q$ is defined by considering the partial-order structure induced by the relaxation relation . The parallel composition of p and q consists of behaviors d that are relaxations of behaviors b and c from p and q along shared signals $I = \text{vars}(p) \cap \text{vars}(q)$ and that are stretching of b and c along the independent signals of p and q.

$$p \parallel q = \{d \in \mathcal{B}|_{\text{vars}(p) \cup \text{vars}(q)} \mid \exists (b, c) \in p \times q, d/I \geq (b/I \,\|\, c/I) b|_I \sqsubseteq d|_I \sqsupseteq c|_I\}$$

2.5 From synchronous to asynchronous structures

The original definition of the polychronous model of computation is designed to render the synchronous hypothesis as implemented in the multi-clocked data-flow notation Signal and relate it to asynchronous architectures using communication with unbounded delay.

In an embedded architecture, however, the flow of a signal usually slides relatively to others as the result of introducing finite delays using, e.g., a finite FIFO buffer or a *relay station*.

To account for such a behavior, we refine the flow-equivalence by a series of (reflexive-anti-symmetric) relations $\sqsubseteq_N$ (for $N > 0$) which yields the (series of) reflexive-symmetric flow relations $\approx_N$ to identify processes of same flows up to a FIFO buffer of size N.

Definition 4 (finite relaxation)
A behavior c is a 0-relaxation *of b iff $b \leq c$.*
For all $N > 0$, c is a N-relaxation *of b, written $b \sqsubseteq_N c$, iff $b \sqsubseteq c$ and*
| *for all $x \in \text{vars}(b)$, $b(x) = (t_i, c_i)_{i \geq 0}$, $c(x) = (t'_i, c_i)_{i \geq 0}$*
| *for all $i \geq 0$, $t_i \leq t'_i < t_{i+N+1}$*

Notice that the limit $\lim_{N \geq 0} \sqsubseteq_N$ (or union $\cup_{N \geq 0} \sqsubseteq_N$) of the finite relaxation relations is the asynchronous partial-order of relaxation $\sqsubseteq$.

Property 1 (transitivity)
$b \sim b'$ *implies* $b \approx_0 b'$
$b \approx_M b' \approx_N b''$ *implies* $b \approx_{M+N} b''$
$b \approx_M b'$ *implies* $b \approx b'$

The series of relations $(\approx_N)_{N \geq 0}$ yields the largest equivalence classes of behaviors that can be modeled in the polychronous model of computation without asynchrony. It consists of behaviors equal up to finite-flow equivalence $\approx^*$ and corresponds to behaviors equal up to timing deformations performed by a finite FIFO buffer.

Definition 5 (finite flow-equivalence)
b and c are finitely flow-equivalent, written $b \approx^* c$, iff there exists $0 \leq N < \infty$ st. $b \sqsubseteq_N c$.

2.6 Polychronous design methodology

Having drawn the design spectrum of the polychronous model of computation, we characterize the formal properties of systems within this design space to define correct-by-construction methodologies for the specification and deployment of elementary components on a given architecture.

Endochrony The key condition we start from is the property of *endochrony*. A process is said endochronous (Definition 6) iff, given an external (asynchronous) stimulation of its inputs, it reconstructs a unique synchronous behavior (up to clock-equivalence). In other words, endochrony denotes the class of processes that are stallable or insensitive to internal and external propagation delays: patient processes, in the sense of [8].

Definition 6 (endochrony [16])
A process p of input signals $I \subset \text{vars}(p)$ is endochronous iff $\forall b, c \in p$, $(b|_I) \approx (c|_I) \Rightarrow b \sim c$.

Figure 8 depicts the behavior of an endochronous process p. The process accepts asynchronous yet flow-equivalent inputs x and y (left). Upon request from the process, inputs are fed from an input buffer in a clock equivalent manner (middle) so as to produce the same outputs in the same order at clock-equivalent rates (right).

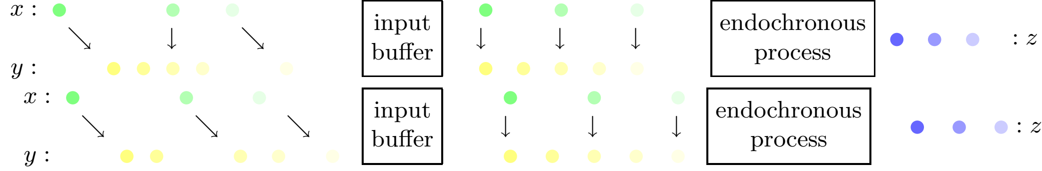

Figure 8: Endochrony: from flow-equivalent inputs to clock-equivalent outputs

Endo-isochrony In the Polychrony workbench [27], GALS architectures are modeled starting from the property of endochrony by considering the criterion of *endo-isochrony* [16]: two endochronous processes p and q of interface $I = \text{vars}(p) \cap \text{vars}(q)$ are endo-isochronous iff $(p|_I) \,|\, (q|_I)$ is endochronous.

This criterion ensures that the refinement of the specification $p \,|\, q$ by a distributed implementation $p \parallel q$ is semantics-preserving. Although restrictive, it is directly amenable to static verification by checking p, q and $(p|_I) \,|\, (q|_I)$ endochronous.

The property of *flow-preservation* [16] generalizes the notion of endo-isochrony to ensure that the refinement of an initial specification p by its implementation q (of inputs I) is *flow-preserving* iff for all $b \in p$, for all $c \in q$, $b|_I \approx c|_I$ implies $b \approx c$. Refinement-based design with the Polychrony workbench and using the criterion of flow-invariance has been studied in a number of related works on system design [16, 26, 19].

Finite flow-preservation In the present study, we focus on the definition of a refinement methodology and semantics preservation criterion that captures the spectrum of bounded protocol synthesis in system design. We therefore recast the notions of endochrony and of flow-invariance in the context of the largest congruence expressible in the polychronous model of computation: finite flow-equivalence $\approx^*$. We say that a process p is finitely flow-preserving iff, given finitely flow-equivalent inputs, it can only produce behaviors that are finitely flow equivalent.

Definition 7 (finite flow-preservation)
A process p of input signals $I \subset \text{vars}(p)$ is finitely flow-preserving *iff $\forall b, c \in p$, $(b|_I) \approx^* (c|_I) \Rightarrow b \approx^* c$.*

Example of finitely flow-preserving processes are endochronous processes. An endochronous process which receives finitely flow equivalent inputs produces clock-equivalent outputs. It hence forms a restricted sub-class of finitely-flow preserving processes. A refinement-based design methodology based on the property of finite flow-preservation consists of characterizing sufficient invariants for a given model transformation to preserve flows.

Definition 8 (finite flow-invariance)
The refinement of a process p by a process q of input signals $I \subset \text{vars}(p) = \text{vars}(q)$ is finitely flow-invariant *iff*

$$\forall b \in p, \forall c \in q, (b|_I) \approx^* (c|_I) \Rightarrow b \approx^* c$$

The property of finite flow-invariance is a very general methodological criterion. For instance, it can be applied to the characterization of correctness criteria for model transformations such as protocol insertion or desynchronization. For instance, it is stable to the introduction of flow-preserving protocols, such as finite FIFO buffers, or double hand-shakes, or relay stations [8], or loosely time-triggered architectures [5], exemplified in the next section.

3 A polychronous type system

We now give an informal outline of the type system that will support and materialize the polychronous model of computation and type and sign the behavior of SystemC programs and classes. The key principles put to work in this notation are essentially borrowed to Pnueli's STS [23] and Dijkstra's guarded command language [9]. We call this notation the iSTS (for implicit synchronous transition systems)

In the iSTS, a process consists of simultaneous propositions that manipulate signals. A signal is an infinite flow of values that is sampled by a discrete series of instants or reactions. An event corresponds to the value carried by a signal during a particular reaction or instant.

Figure 9 puts together the main features of the behavioral type system. It describes the behavior of a counter modulo 2, noted P ($\stackrel{\text{def}}{=}$ means "is defined by"), through a set of simultaneous *propositions*, labeled from (1) to (3) .

- Proposition (1) is an invariant. It says that the initial value of the signal s, denoted by s^0, is false. This is specified by the proposition $\neg s^0$ (equivalent to $s^0 = 0$).
- Proposition (2) is a guarded command. It says that if the signal x is present during a given reaction then the value of s is toggled.
  - The leftmost part of the sentence, the proposition $\hat{x}$, is a condition or a guard. It denotes the *clock* of x. It is true if the signal x is present during the current reaction.
  - The rightmost part of the sentence, the proposition $s' = \neg s$, is a transition. The term s' refers to the next value of the signal s. The proposition $s' = \neg s$ says that the next value of s is the negation of the present value of s.
- Proposition (3) is another guarded command. It says that if s is true then y is present.

$$P \stackrel{\text{def}}{=} \begin{array}{llll} & & \neg s^0 & (1) \\ \mid & \hat{x} \Rightarrow & s' = \neg s & (2) \\ \mid & s \Rightarrow & \hat{y} & (3) \end{array}$$

Figure 9: Specification of a counter modulo 2

Notice that, in proposition (3), the guard expects the signal s to hold the value 1 but that its action does not actually specify the value of the signal y: it simply requires it to be present. Proposition (3) is hence an *abstraction*: a proposition that partially describes the properties of the system under consideration without implying a particular implementation.

To implement a *function* or a *system*, this proposition needs to be compositionally *refined* by another one, saying what value y should hold at all time (i.e. when present). To this end, one could for instance compose the counter P, Figure 9, with the proposition Q:

$$Q \stackrel{\text{def}}{=} \left(y' = y + 1 \mid y^0 = 0\right)$$

Notice that the iteration specified by the proposition Q is not guarded. This means that it is an invariant that describes the successive values of the signal y in time but not the particular time samples at which the signal y should occur. The composition of Q with P has the effect of synchronizing the signal y in Q to the *clock* s in P: to the time samples during which the signal s holds the value true. This composition is called a refinement: the system obeys the specification denoted by the initial proposition P and is constrained to further satisfy the additional conditions implied by Q.

$$P \text{ is an abstraction of } P \mid Q \qquad P \mid Q \text{ is a refinement of } P$$

The notation introduced so far holds necessary and sufficient ingredients to specify the behavior of multi-clocked synchronous systems. Pnueli's original STS notation features two additional notions which, in retrospect, are essentially geared towards verification by model-checking.

- One feature is choice $P \vee Q$. For instance, in the STS, the proposition $a = 1 \vee b = 1$ allows to non-deterministically have a or b present with the value true at all time. Non-determinism can equally be modeled using guarded commands using a partially defined signal s whose scope is lexically restricted to the desired proposition. Instead, we write:

  $$R \stackrel{\text{def}}{=} (s \Rightarrow a \mid \neg s \Rightarrow b)/s$$

  to mean the non-deterministic proposition of choosing a or b upon the value of an internal signal s, whose calculation is left unspecified. Here, the notation P/s means that the scope of s is local to the process P. Again, notice that the proposition R may best be understood as the *abstraction* of an executable specification, e.g., one that specifies the actual value of s in time (for instance, the toggle s of the counting process P).

  $$R \text{ is an abstraction of } \left(s \Rightarrow a \mid \neg s \Rightarrow b \mid \left(\neg s^0 \mid s' = \neg s\right)\right)/s$$

- Another feature of the STS is explicit absence $\perp$. The proposition $x = \perp$ explicitly specifies that x does not hold a value in the context in which it is considered. For instance, the STS:

  $$(a = 1 \wedge x \neq \perp) \vee (x = \perp \wedge b = 1)$$

  means that either a is true and x is present or that b is true and x is absent. In the iSTS, this notion is implicit. It can for instance be specified by a refinement of R with the invariant "x is present iff b is true":

  $$R \mid b = \hat{x}$$

In the aim of moving back and forth from abstraction to refinement, one last essential feature is the notion of scheduling specification. A scheduling specification is designed to imply an order of execution to otherwise purely logical propositions. Whereas a transition, e.g. $s' = \neg s$, implicitly means that the next value of s is computed using the present value of s, a proposition, e.g. $y = x$, just means that x and y are equal. It does not specify any order of execution. An order of execution can be imposed to a proposition by its refinement with a scheduling constraint, noted $y \rightarrow x$. Then,

$$x = y \text{ is an abstraction of } x = y \mid y \rightarrow x$$

where $y \rightarrow x$ informally means that "x cannot happen before y". In doing so, we refine the time scale, from one in which x and y happen simultaneously, to a more precise one in which one observes that x cannot happen before y. We will adopt the following syntax, borrowed to the Signal language, and write $x := y$ for an assignment of y to x:

$$(x = y \mid y \rightarrow x) \text{ is an abstraction of } x := y$$

3.1 Formal syntax of the type system

The formal syntax of propositions in the behavioral type system is defined by the inductive grammar P in Table 1. A propositon or process P manipulates boolean values noted $v \in \{0,1\}$ and signals noted x, y, z. A location l refers to the initial value x^0, the present value x and the next value x' of a signal. A reference r stands for either a value v or a signal x.

A clock expression e is a proposition on boolean values. When true, a clock e defines a particular period in time. The clocks 0 and 1 denote events that never/always happen. The clock $x = r$ denotes the proposition: "x is present and holds the value r". Particular instances are:

- the clock noted $\hat{x}$ and defined by $x = x$ means that "x is present"
- the clock noted x and defined by $x = 1$ means that "x is true"
- the clock noted $\neg x$ and defined by $x = 0$ means that "x is false"

Clocks are propositions combined using the logical combinators of conjunction $e \wedge f$, to mean that both e and f hold, disjunction $e \vee f$, to mean that either e or f holds, and symmetric difference $e \setminus f$, to mean that e holds and not f.

A process P consists of the simultaneous composition of elementary propositions. 1 is the process that does nothing. The proposition $l = r$ means that "l holds the value r" The proposition $x \to l$ means that "l cannot happen before x". The process $e \Rightarrow P$ is a guarded command. It means: "if e is present then P holds". Processes are combined using synchronous composition $P \,|\, Q$ to denote the simultaneity of the propositions P and Q. Restricting a signal name x to the lexical scope of a process P is written P/x. We refer to the *free variables* $\text{vars}(P)$ of a process P (Table 18, Appendix A) as the set of signal names that occur free in the lexical scope of P.

Table 1: Formal syntax of the type system

$$
\begin{array}{rrl}
\text{(reference)} & r & ::= x \,|\, v \\
\text{(location)} & l & ::= x^0 \,|\, x \,|\, x' \\
\text{(clock)} & e, f & ::= 0 \,|\, x = r \,|\, e \wedge f \,|\, e \vee f \,|\, e \setminus f \,|\, 1 \\
\text{(process)} & P, Q & ::= 1 \,|\, l = r \,|\, x \to l \,|\, e \Rightarrow P \,|\, (P \,|\, Q) \,|\, P/x
\end{array}
$$

Notational conventions In the formal presentation of the iSTS, we restrict ourself to a subset of the elementary propositions in the grammar of Table 1, which we call atoms a:

$$\text{(atoms)} \quad a, b ::= x^0 = v \,|\, l = y \,|\, x \to l \; s.t. \; l ::= x \,|\, x'$$

Other propositions as well as additional syntactic shortcuts, used in the examples, can be defined by using this restricted subset as follows.

$$
\begin{array}{rlrl}
l = v & \stackrel{\text{def}}{=} (l = x \,|\, x^0 = v \,|\, x' = x)/x \text{ iff } x \neq l \neq x' & \hat{x} & \stackrel{\text{def}}{=} (x = x) \\
l := x & \stackrel{\text{def}}{=} (l = x \,\|\, x \to l) & l & \stackrel{\text{def}}{=} (l = 1) \\
\hat{x} = \hat{y} & \stackrel{\text{def}}{=} (\hat{x} \Rightarrow \hat{y} \,\|\, \hat{y} \Rightarrow \hat{x}) & \neg l & \stackrel{\text{def}}{=} (l = 0)
\end{array}
$$

3.2 Denotational semantics of the type system

The detailed presentation and extension of the polychronous model of computation allows us to give a denotational model to the type system introduced, Section 3. This model consists of relating a proposition P to the set of behaviors p it denotes.

Meaning of clocks Let us start with the denotation of a clock expression e (Table 2). The meaning $[\![e]\!]_b$ of a clock e is defined relatively to a given behavior b and consists of the set of tags satisfied by the proposition e in the behavior b.

In Table 2, the meaning of the clock $x = v$ (resp. $x = y$) in b is the set of tags $t \in \text{tags}(b(x))$ (resp. $t \in \text{tags}(b(x)) \cap \text{tags}(b(y))$) such that $b(x)(t) = v$ (resp. $b(x)(t = b(y)(t))$, in particular, $[\![\hat{x}]\!]_b = \text{tags}(b(x))$. The meaning of a conjunction $e \wedge f$ (resp. disjunction $e \vee f$ and difference $e \setminus f$) is the intersection (resp. union and difference) of the meaning of e and f. Clock 0 has no tags.

Table 2: Denotational semantics of clock expressions

$$
\begin{aligned}
[\![x = y]\!]_b &= \{t \in \text{tags}(b(x)) \cap \text{tags}(b(y)) \mid b(x)(t) = b(y)(t)\} \\
[\![x = v]\!]_b &= \{t \in \text{tags}(b(x)) \mid b(x)(t) = v\} \\
[\![e \wedge f]\!]_b &= [\![e]\!]_b \cap [\![f]\!]_b \\
[\![e \vee f]\!]_b &= [\![e]\!]_b \cup [\![f]\!]_b \\
[\![e \setminus f]\!]_b &= b[\![e]\!]_b \setminus [\![f]\!]_b \\
[\![1]\!]_b &= \text{tags}(b) \\
[\![0]\!]_b &= \emptyset
\end{aligned}
$$

Meaning of propositions The denotation of a clock expression by a set of tags yields the denotational semantics of propositions P, written $[\![P]\!]$, Table 3. The meaning $[\![P]\!]^e$ of a proposition P is defined with respect to a clock expression e. Where this information is absent, we assume $[\![P]\!] = [\![P]\!]^1$ to mean that P is an invariant (and is hence independent of a particular clock).

Table 3: Denotational semantics of propositions

$$
\begin{aligned}
[\![x^0 = v]\!]^e &= \{b \in \mathcal{B}|_x \mid b(x)(\min(\text{tags}(b(x)))) = v\} \\
[\![x = y]\!]^e &= \{b \in \mathcal{B}|_{\text{vars}(e)\cup\{x,y\}} \mid \forall t \in [\![e]\!]_b, \\
&\qquad t \in \text{tags}(b(x)) \wedge t \in \text{tags}(b(y)) \wedge b(x)(t) = b(y)(t)\} \\
[\![y \rightarrow x]\!]^e &= \{b \in \mathcal{B}|_{\text{vars}(e)\cup\{x,y\}} \mid \forall t \in [\![e]\!]_b, \\
&\qquad t \in \text{tags}(b(x)) \Rightarrow t \in \text{tags}(b(y)) \wedge t_y \rightarrow^b t_x\} \\
[\![x' = y]\!]^e &= \{b \in \mathcal{B}|_{\text{vars}(e)\cup\{x,y\}} \mid \forall t \in [\![e]\!]_b, \\
&\qquad t \in C = \text{tags}(b(x)) \wedge t \in \text{tags}(b(y)) \wedge b(x)(\text{succ}_t(C)) = b(y)(t)\} \\
[\![y \rightarrow x']\!]^e &= \{b \in \mathcal{B}|_{\text{vars}(e)\cup\{x,y\}} \mid \forall t \in [\![e]\!]_b, \\
&\qquad t \in C = \text{tags}(b(x)) \Rightarrow t \in \text{tags}(b(y)) \wedge t_y \rightarrow^b (\text{succ}_t(C))_x\} \\
[\![f \Rightarrow P]\!]^e &= [\![P]\!]^{e \wedge f} \\
[\![P \mid Q]\!]^e &= [\![P]\!]^e \mid [\![Q]\!]^e \\
[\![P/x]\!]^e &= \{c \leq b/x \mid b \in [\![P]\!]^e\}
\end{aligned}
$$

The meaning of an initialization $x^0 = v$, written $[\![x^0 = v]\!]^e$, consists of all behaviors defined on x, written $b \in \mathcal{B}|_x$ such that the initial value of the signal $b(x)$ equals v. Notice that it is independent from the clock expression e provided by the context. In Table 3, we write:

- $\mathcal{B}|_X$ for the set of all behaviors of domain X
- $\min(C)$ for the minimum of the chain of tags C
- $\text{succ}_t(C)$ for the immediate successor of t in the chain C
- $\text{vars}(P)$ and $\text{vars}(e)$ for the set of free signal names of P and e.

The meaning of a proposition $x = y$ at the clock e consists of all behaviors b defined on $\text{vars}(e) \cup \{x, y\}$ such that all tags $t \in [\![e]\!]_b$ at the clock e belong to $b(x)$ and $b(y)$ and are associated with the same value. A scheduling specification $y \to x$ at the clock e denotes the set of behaviors b defined on $\text{vars}(e) \cup \{x, y\}$ which, for all tags $t \in [\![e]\!]_b$, requires x to preceed y: if t is in $b(x)$ then it is necessarily in $b(y)$ and satisfies $t_y \to^b t_x$. The propositions $x' = y$ and $y \to x'$ are interpreted similarly by considering the tag t' that is the successor of t in the chain C of x.

The behavior of a guarded command $f \Rightarrow P$ at the clock e is equal to the behavior of P at the clock $e \wedge f$. The meaning of a restriction P/x consists of the behaviors c of which a behavior b/x from P are a stretching of. The behavior of $P \,|\, Q$ consists the synchronous composition of the behaviors of P and Q.

3.3 Abstraction of behavioral types

In the terminology of [10], the process P specified in the grammar of Table 1 defines the *dynamic interface* of a given system: it defines a transition system that specifies the invariants of its evolution in time.

Just as the theory of interface automata [10], behavioral types allow for a scalable level of abstraction that can be automatically obtained starting from the inferred type of a given module (Table 4). The abstraction of a process P by its *static interface* $\hat{P}$ is defined by the type inference system $e \vdash P : Q$ of Table 4. To a process P in a context of clock e, it associates the clock and scheduling relations Q that form a static abstraction of P.

Table 4: Static abstraction of behavioral types

$$\vdash 1 : 1 \qquad \vdash x^0 = v : 1 \qquad e \vdash x \to y : e \Rightarrow x \to y$$

$$e \vdash x = y : e \Rightarrow x = y \qquad e \vdash x' = y : e \Rightarrow \hat{x} = \hat{y} \,|\, e \Rightarrow \hat{x} \,|\, e \Rightarrow \hat{y}$$

$$\frac{e \wedge f \vdash P : Q}{e \vdash f \Rightarrow P : Q} \qquad \frac{e \vdash P : Q}{e \vdash P/x : Q/x} \qquad \frac{e \vdash P_1 : Q_1 \quad e \vdash P_2 : Q_2}{e \vdash P_1 \,|\, P_2 : Q_1 \,|\, Q_2}$$

The *static* interface $\hat{P}$ of a process P consists of the abstraction of propositions P by a set of clock and scheduling relations Q, defined by the type inference relation $e \vdash P : Q$ of Table 4, and of the closure of scheduling relations by transitivity, i.e. $e \Rightarrow x \to y \,|\, f \Rightarrow y \to z$ implies $(e \wedge f) \Rightarrow x \to z$; and by distributivity, i.e. $e \Rightarrow x \to y \,|\, f \Rightarrow x \to y$ implies $(e \vee f) \Rightarrow x \to y$.

The correctness of the inference system is stated by a denotational containment relation and proved by induction on the structure of the term P.

Property 2 (soundness)
If $e \vdash P : Q$ then $[\![P]\!]^e \subseteq [\![Q]\!]^e$.

Example 1 *Let us for instance refine the specification of the counter modulo 2, Figure 9. The process P (left) constrains the event y to occur upon two successive inputs x. Its state s is initially false. It is synchronized to the input signal x. The next value of s is defined by the negation of its current value. When s is true then y is present. From the type inference relation of Table 4, we obtain the static interface of $\hat{P}$ (right).*

$$P \stackrel{\text{def}}{=} \begin{pmatrix} s^0 = 0 \\ \hat{x} = \hat{s} \\ s' := \neg s \\ s = \hat{y} \\ y := s \end{pmatrix} /s \qquad \hat{P} = \begin{pmatrix} \hat{x} = \hat{s} \\ s \to s' \\ s = \hat{y} \\ s = y \\ s \to y \end{pmatrix} /s$$

Let $Q = (\hat{x} > \hat{y})$ be the type declared in place of $\hat{P}$ to define a static interface type for some component implementing the proposition P. It just requires the clock $\hat{y}$ to be a sampling of $\hat{x}$. Checking that the declared type Q is compatible with the inferred type $\hat{P}$ amounts to checking that $\hat{P}$ implies Q to meet the required denotational containment relation: $[\![\hat{P}]\!] \subseteq [\![Q]\!]$. This is implemented by observing that $(\hat{x} = \hat{s} \mid \hat{y} = s)/s$ implies $\hat{x} > \hat{y}$.

3.4 Refinement of behavioral types

Behavioral type abstraction allows to associate a system of clock equations $\hat{P}$ with process P. In order to perform the dual type refinements:
- merging several processes into one component
- or, conversely, splitting one into several distributed components

we define type-based model transformations. On the path to such transformations, *hierarchization* [2] is the key concept put to work. Hierarchization consists of determining a canonical representation of the partial order between the signal clocks of a process P. This data-structure is the medium which we use to perform the verification, transformation and exploration of system design models.

Example 2 *As an example, consider the type P of the counter modulo 2 from Example 1 (Table 5, column process). The abstraction of P (Table 5, column type) consists of a set of clock and scheduling relations.*

Table 5: From behavioral types to sequential C code

(process)	(type)	(hierarchy)	(code)
$s^0 = 0$	$\hat{x} = \hat{s}$		bool s = false;
$\hat{x} = \hat{s}$	$s \to s'$	$\hat{x}, \hat{s}$	while true {
$s' := \neg s$	$s = \hat{y}$	/ \	if x then if s then { y = true;
$s = \hat{y}$	$y = s$	$\neg s$ $\hat{y}, s$	s = false; }
$y := s$	$s \to y$		else { s = true; } }

Hierarchization (Table 5, hierarchy), consists of building partial order relations between clocks starting from $\hat{P}$. The clock equivalence class at the top of the tree comprises $\hat{x}$ and $\hat{s}$. It is associated to the scheduling relation $\hat{x} \to s$. The left-most subtree corresponds to the clock $s = 0$ and the right-most sub-tree to the clocks $s = 1$ and $\hat{y}$.

The C program (Table 5, code) demonstrates the capability to build a control-flow grap starting from the system of clock equations and scheduling relations implied by a process P.

Definition 9 formalizes the algorithm introduced in [2] for the construction of a hierarchy starting from a set of boolean equations. From the example of Figure 5, we observe that both free and bound signals of a proposition P are involved by hierarchization. Therefore, Definition 9 considers the Skolemization $\overline{P}$ of P. By induction on P by:
- $\overline{a/X} = a/X$ and $\overline{P/x} = Q/X \cup \{x\}$ iff $\overline{P} = Q/X$
- $\overline{e \Rightarrow P} = (e \Rightarrow Q)/X$ iff $\overline{P} = Q/X$ and $\text{vars}(e) \cap X = \emptyset$
- $\overline{P_1 \,|\, Q_1} = (P_2 \,|\, Q_2)/XY$ iff $\overline{P}_1 = P_2/X$ and $\overline{Q}_1 = Q_2/Y$

Notice that $[\![\overline{P}]\!] = [\![P]\!]$. Definition 9 uses the static abstraction $\hat{Q}$ of $\overline{P} = Q/X$ to find clock equivalence classes and partially relate them. Rule 1 defines equivalence class for synchronous signals and constructs elementary partial order relations: the clock $(x = v)$ is smaller than $\hat{x}$. Rule 2 defines the insertion of a partial order of maximum z below a clock x. The insertion algorithm, introduced in [2], yields a canonical representation of the corresponding partial order by observing that there exists a unique minimum clock

below x such that rule 2 holds. In Definition 9, we write $P \models Q$, e.g. $\hat{Q} \models \hat{x} = \hat{y}$, iff the proposition P implies the proposition Q, e.g. the static interface $\hat{Q}$ implies the clock equation $\hat{x} = \hat{y}$.

Definition 9 (hierarchization)
The clock relation $\preceq_P$ *of a process P is the partial order relation between elementary clocks* $(x = r)$, *written h, and defined by application of rules* $(1-2)$ *with* $\overline{P} = Q/X$.

1. *if* $\hat{Q} \models \hat{x} = h$ *or* $\hat{Q} \models (x = v) = h$ *then* $\hat{x} \preceq h$.
2. *if* $\hat{x} \preceq h_1$, $\hat{x} \preceq h_2$ *and* $\hat{Q} \models h = h_1 \star h_2$ *for any* $\star \in \{\wedge, \vee, \setminus\}$ *then* $\hat{x} \preceq h$.

h_1 *and* h_2 *are clock-equivalent, written* $h_1 \prec\!\succ h_2$, *iff* $h_1 \preceq h_2$ *and* $h_2 \preceq h_1$.

A process whose clock relation is hierarchic, in the sense of Definition 10, meets a necessary criterion for being implemented by a sequential program.

Definition 10 (hierarchization)
A process P is hierarchic *iff* $\preceq_P$ *has a minimum class* $\min \preceq_P$

In order to lift a larger class of processes to the rank of hierarchical processes, the Polychrony workbench implements additional program transformations. These transformations consist of refinements that promote the clock equivalence classes of an initial process P until a minimum $\min \preceq_Q$ is found in the final process Q. One refinement consists of promoting a bound signal in an oversampling process. A process P/x is oversampling iff P is hierarchical but its minimun is $\hat{x} \in \min \preceq_P$. In that case, one may promote x by refining P as

$$Q \stackrel{\text{def}}{=} (P \,|\, \hat{y} = \hat{x})/x \;\; s.t. \; y \notin \text{vars}(P)$$

which, by definition of the initial process P, satisfies $\hat{y} \in \min \preceq_Q$ and is hence hierarchical.

3.5 Type-based design analysis

Starting from the algorithm specified in Definition 9 and the class of hierarchical processes isolated in Definition 10, we seek processes that satisfy the property of endochrony in the context of behavioral type inference by introducing the notion of controllability, Definition 14.

A correct definition of controllability requires a precise partition of the signals of a process P into input signals $\text{in}(P)$, output signals $\text{out}(P)$ and state variables $\text{def}(P)$. This partition is defined in Appendix A, Table 19.

A signal $x \in \text{out}(P)$ (resp. $x \in \text{def}(P)$) is an output (resp. state) of P iff, whenever its presence is implied by a clock e, then there exists another clock f implied by e whose action is $x := y$ (resp. $x' := y$) for some reference y. The disjoint sets $\text{def}(P)$ and $\text{out}(P)$ define the input signals $\text{in}(P)$ of a process by their complementary in $\text{vars}(P)$.

Definability A controllable process needs to correctly define its local and bound signals in order to meet the property of determinism, and hence endochrony. For instance, let us reconsider the example of the specification R in the introduction, Section 3.

$$R \stackrel{\text{def}}{=} (s \Rightarrow x \,|\, \neg s \Rightarrow y)/s$$

It may seem controllable by apparently having two output signals x and y defined upon the value of the boolean signal s. However, notice that s is left undefined. Hence the behavior of R is non-deterministic, since the choice of the value of s is free. Had s been associated to some external input z, as in Q,

$$Q \stackrel{\text{def}}{=} (s^0 = 1 \,|\, s' := z \,|\, s \Rightarrow x := s \,|\, \neg s \Rightarrow y := s)/s$$

one could have concluded that it is indeed controllable. In fact, notice that Q is a refinement of R: $[\![Q]\!] \subseteq [\![R]\!]$.

Definition 11 (definability)

A process P is well-defined *iff all bound variables of P are either output signals or state variables. By induction on the structure of P,*

- *an atomic proposition a is well-defined.*
- *a guarded command $e \Rightarrow P$ is well defined iff P is well-defined*
- *a composition $P \| Q$ is well-defined iff P and Q are well-defined*
- *a restriction P/x is well-defined iff so is P and $x \in \mathrm{def}(P) \cup \mathrm{out}(P)$.*

Causality Causality is another possible origin of non-determinism. Let us consider the example of a clock proposition $\hat{x} \stackrel{\text{def}}{=} (x = x)$. It does not define an output signal, it just says that the signal x is present. It can be regarded as an abstraction of the proposition $x := y \stackrel{\text{def}}{=} (x = y \| y \rightarrow x)$, which defines it as an output signal and assigns the value of the input y to it. What happens next, if we additionally write $y = x$ or $y := x$? In the first case, one can still regard the process $x := y \| y = x$ as a redundant yet deterministic specification: x takes the value of y. This is not the case in the latter specification, which tries to mutually define the output signals x and y: $x := y \| y := x$. It is a non-deterministic specification, as both true and false are possible values of x and y every time this is needed. Thanks to the definition of $x := y$ by $x = y \| y \rightarrow x$, the analysis of causal specifications relies on the information provided by the scheduling relation of a process P.

A cyclic definition of output signals yields a proposition $e \Rightarrow x \rightarrow x$ at some clock e in the transitive closure of the scheduling relation implied by $\hat{P}$, hence Definition 12.

Definition 12 (causality)

A process P is non-causal *iff $\hat{P} \models e \Rightarrow x \rightarrow x$ implies $\hat{P} \models e = 0$.*

Determinism The isolation of well-defined and non-causal processes allows to touch an important intermediate result. When a process defines its bound variables from the value of free variables and does not define its outputs in a cyclic manner, it is deterministic in the sense of Definition 13.

Definition 13 (determinism [16])

P is deterministic iff $\forall b, c \in [\![P]\!]$, $(b|_{\mathrm{in}(P)}) = (c|_{\mathrm{in}(P)}) \Rightarrow b = c$.

Property 3 (determinism)

If P is well-defined and non-causal then P is deterministic

Controllability If Definition 13 is compared to the pattern of behaviors depicted Figure 10, one easily notices that a deterministic process still misses some important capabilities to meet the requirements of endochrony.

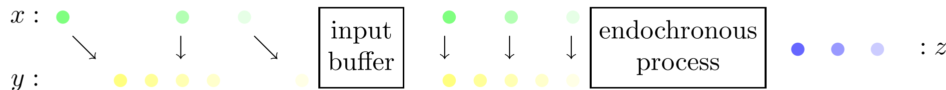

Figure 10: Endochrony as the reconstruction of clock-equivalence classes

A deterministic process must have a way to initiate a reaction upon the arrival of a pre-determined input signal (e.g. x, Figure 8), to check whether it needs further inputs (e.g. should it load y from the input buffer or not) to finally produce an output (e.g. produce z when the input y is absent).

The notion of hierarchy allows to fill this remaining gap. A hierarchical process P has a pre-determined input tick, the minimum $\min \preceq_P$ of its clock relation.

Upon the presence of the tick (e.g. x, Figure 10), it is able to decide whether an additional input is needed or not (e.g. $\hat{x} \prec \hat{y}$, Figure 10), it is able to decide whether it should produce outputs (e.g. $\hat{x} \prec \hat{z}$, Figure 10).

However, controllability requires this decision to be made starting from the input signals of a process (and not its output signals or state variables).

Definition 14 (controllability)
A process P is controllable *iff P is well-defined, non-causal, hierarchical and for all $y \in \mathrm{vars}(P)$, there exists $x \in \mathrm{in}(P)$ such that $x \preceq_P y$.*

The definition of controllability bridges the remaining gap from a deterministic specification to an endochronous program. It is stated by property 4. Property 4 generalizes the proposition of [16] to the context of partially defined signals of the behavioral type system.

Property 4 (controllability)
If P is controllable then P is endochronous.

3.6 Type-based design methodology

In the previous section, we identified the property of finite flow-invariance as a pivot criterion in a polychronous design methodology. We now show how it is used in the type system to characterize correctness criteria for model transformations such as the desynchronization incurred by the insertion of a protocol in the architecture under design.

Let us consider two finitely flow-preserving processes P and Q and let R be a protocol to link P and Q, such as a finite FIFO buffer fifo_N, or a double hand-shake protocol, or a relay station [8], or a loosely time-triggered architecture [5]. The wrapper $R\langle P\rangle$ of a process P with a protocol R is defined by redirecting the signals of P to R using substitutions.

Definition 15 (flow-preserving protocol)
The process R is a flow-preserving protocol iff there exists $n > 0$ such that inputs $\mathrm{in}(R) = \{x_{1..n}\}$ are finitely flow-equivalent to outputs $\mathrm{out}(R) = \{y_{1..n}\}$

$$\forall b \in [\![R]\!], b|_{x_{1..n}} \approx^* (b|_{y_{1..n}}[x_i/y_i]_{0<i\leq n})$$

Let P be a process such that $\mathrm{in}(P) = \{x_{1..m}\}$ and $\mathrm{out}(P) = \{x_{m+1..n}\}$. Let R be a flow-preserving protocol such that $\mathrm{in}(R) = \{y_{1..n}\}$ and $\mathrm{out}(R) = \{z_{1..n}\}$. The wrapper of P with R is the template process noted $R\langle P\rangle$ and defined by:

$$R\langle P\rangle \stackrel{\mathrm{def}}{=} \left(\begin{array}{ll} & (R[x_i/z_i]_{m<i\leq n})\,[x_i/y_i]_{0<i\leq m} \\ | & (P[y_i/x_i]_{m<i\leq n})\,[z_i/x_i]_{0<i\leq m} \end{array} \right) / y_{1..n} z_{1..n}$$

A sufficient condition that guarantees the insertion of a protocol between two synchronous processes P and Q finitely flow preserving is to have $P|_I \,|\, Q|_I$ finitely flow preserving for $I = \mathrm{vars}(P) \cap \mathrm{vars}(Q)$, meaning that all communications between P and Q via a shared signal $x \in I$ should be flow preserving and that P and Q may otherwise evolve independently.

Property 5 (protocol insertion)
If P is endochronous then P is finitely flow-preserving.

If R is a flow-preserving protocol and P is finitely flow-preserving then $R\langle P\rangle$ is finitely flow-preserving.

If R is a flow-preserving protocol and P, Q, $P|_I \,|\, Q|_I$ are finitely flow-preserving then $R\langle P\rangle \,|\, R\langle Q\rangle$ is finitely preserving ($I = \mathrm{vars}(P) \cap \mathrm{vars}(Q)$).

4 Behavioral types for SystemC modules

We are now equipped with the required mathematical framework and formal methodology to address the modeling of GALS architectures described using SystemC. This model is described in terms of a type inference system and extended to the structuring elements of SystemC in terms of a module system. This framework allows to give a behavioral signature of the component of the system, compositionally check the correct composition of such components to form architecture, to optimize the described software elements from the imposed hardware elements by, first, detaching the formal model from the functional architecture description and, second, using the model to regenerate an optimized software matching the requirements of the execution architecture. As a by-product, the association of types with SystemC programs provides a formal denotational semantics implied by the interpretation of types in the polychronous model of computation.

4.1 Formal syntax of SystemC core

We start with the definition of the core of the SystemC syntax relevant to the present study. A system consists of the composition of classes and modules *sys*, Table 6. A class declaration $\mathsf{class}\, m \, \{dec\}$ associates a class name m with a sequence of fields *dec*. It is optionally parameterized by a class with $\mathsf{template}\, \langle \mathsf{class}\, m_1 \rangle$. To enforce a strong typing policy, we annotate the class parameter m_1 with $\#\textsf{TYPE}(m_1, m_2)$ to denote the type of m_1 with the virtual class m_2. A module $\textsf{SC_MODULE}(m)$ is a class that defines an architecture component. Its constructor $\textsf{SC_CTOR}(m)\ \{new; pgm\}$ allocates threads (e.g. $\textsf{SC_THREAD}(f)$) and executes an initialization program *pgm*. While naturally declared sequentially in the program text, modules define threads whose execution is concurrent.

Declarations *dec* associate locations x with native classes or template class instances $m\langle m^* \rangle$ and procedures with a name f and a definition *pgm*. For instance, $\mathsf{int}\, x$ defines an integer variable x while $\mathsf{sc_signal}\langle \mathsf{bool} \rangle\ x$ defines a boolean signal x. We assume x to denote the name of a variable or signal and to be possibly prefixed as $m :: x$ by the name of the class it belongs to. We assume the relation $\leq$ to denote SystemC sub-typing, e.g., $\mathsf{bool} \leq \mathsf{num}$ or $\mathsf{int} \leq \mathsf{num}$.

Table 6: Abstract syntax for SystemC

(component)	sys	$::=$	$[\mathsf{template}\, \langle \mathsf{class}\, m_1 \rangle\ \#\textsf{TYPE}(m_1, m_2)]\ \mathsf{class}\, m\, \{dec\}$	(class)
			$\mid \textsf{SC_MODULE}(m)\ \{dec; \textsf{SC_CTOR}(m)\ \{new\}\}$	(module)
			$\mid sys; sys$	(sequence)
(declaration)	dec	$::=$	$m\langle m^* \rangle\, x$	(field)
			$\mid \mathsf{void}\ f()\ \{pgm\}$	(thread)
			$\mid dec; dec$	(sequence)
(constructor)	new	$::=$	$\textsf{SC_THREAD}(f)\ \mathsf{sensitive} \ll x^* \mid new; pgm$	

The formal grammar of SystemC programs, Table 7, is represented in static single-assignment intermediate form akin to that of the Tree-SSA package of the GCC project [20]. SSA provides a language-independent, locally optimized intermediate representation (Tree-SSA currently accepts C, C++, Fortran 95, and Java inputs) in which language-specific syntactic sugar is absent. SSA transforms a given programming unit (a function, a method or a thread) into a structure in which all variables are read and written once and all native operations are represented by 3-address instructions $x = f(y, z)$.

A program *pgm* consists of a sequence of labeled blocks $L{:}blk$. Each block consists of a label L and of a sequence of statements *stm* terminated by a return statement *rtn*. In the remainder, a block always starts

Table 7: Abstract syntax for SystemC programs in SSA form

(program)	$pgm ::= L{:}blk \mid pgm; pgm$			(block)	$blk ::= stm; blk \mid rtn$		
(instruction)	$stm ::=$	$x = f(x^*)$	(invoke)	(return)	$rtn ::=$	$\mathsf{goto}\, L$	(goto)
		$\mid \mathsf{wait}\, x$	(lock)			$\mid \mathsf{return}$	(return)
		$\mid \mathsf{notify}\, x$	(unlock)			$\mid \mathsf{throw}\, x;$	(throw)
		$\mid \mathsf{if}\, x\, \mathsf{then}\, L$	(test)			$\mathsf{catch}\, x\, \mathsf{from}\, L$ $\mathsf{to}\, L\, \mathsf{using}\, L$	(catch)

with a label and finishes with a return statement: $stm_1; L{:}stm_2$ is rewritten as $stm_1; \mathsf{goto}\, L; L{:}stm_2$. A wait is always placed at the beginning of a block: $stm_1; \mathsf{wait}\, v; stm_2$ is rewritten as $stm_1; \mathsf{goto}\, L; L{:}\mathsf{wait}\, v; stm_2$.

Block instructions consist of native method invocations $x = f(x^*)$, lock monitoring and branches $\mathsf{if}\, x\, \mathsf{then}\, L$. Blocks are returned from by either a $\mathsf{goto}\, L$, a return or an exception $\mathsf{throw}\, x$. The declaration $\mathsf{catch}\, x\, \mathsf{from}\, L_1\, \mathsf{to}\, L_2\, \mathsf{using}\, L_3$ that matches an exception x raised at block L_1 activates the exception handler L_3 and continues at block L_2.

Example 3 *To outline the construction of the intermediate representation of a SystemC program, let us reconsider the example of Section 1.2 and detail the method that counts the number of bits set to* 1 *in a bit-array* epc.data.

```
void epc::ones () { sc_int<16> idata = 0, ocount = 0;
  while true { wait (epc.lock);
               idata = epc.data;
               ocount = 0;
               while (idata != 0) { ocount = ocount + (idata & 1);
                                    idata = idata >> 1; }
               epc.count = ocount;
               notify (epc.lock); }}}
```

Figure 11: **default**

The method consists of three blocks. The block labeled L1 *waits for the lock* epc.lock *before initializing the local state variable* idata *to the value of the input signal* epc.data *and* ocount *to* 0*. Label* L2 *corresponds to a loop that shifts* idata *right and adds its right-most bit to* ocount *until termination (condition* T0*). Finally, in the block* L3*,* ocount *is sent to the signal* epc.count *and* epc.lock *is unlocked before going back to* L1*.*

```
L1:wait (epc.lock);      L2:T1 = idata;            L3:epc.count = ocount;
   idata = epc.data;        T0 = T1 == 0;             notify (epc.lock);
   ocount = 0;              if T0 then goto L3;       goto L1;
   goto L2;                 T2 = ocount;
                            T3 = T1 & 1;
                            ocount = T2 + T3;
                            idata = T1 >> 1;
                            goto L2;
```

4.2 Behavioral type inference

The type inference function $\mathcal{I}[\![pgm]\!]$, Table 8, is defined by induction on the formal syntax of *pgm*. To define it, we assume that the finite set $\mathcal{L}_f$ of program labels L defined in a given method f respects the order of appearance in the text: $L_1 < L_2$ means that L_1 occurs above L_2.

To each block of label $L \in \mathcal{L}_f$, the function $\mathcal{I}[\![pgm]\!]$ associates an *input clock* x_L, an *immediate clock* x_L^{imm} and an *output clock* x_L^{exit}. The clock x_L is true iff L has been activated in the previous transition (by posting the event x'_L). The clock x_L^{imm} is set to true to activate the block L immediately. The clock x_L^{exit} is set to true when the execution of block L terminates. The default activation condition of a block of label L is the clock $x_L \vee x_L^{imm}$ (equation (1) of Table 8). The block L is executed iff the proposition $x_L \vee x_L^{imm}$ holds, meaning that the program counter is at L. Otherwise, it is set to 0.

For a return instruction or a block, the type inference function returns a type P. For a block instruction *stm*, the type inference function $\mathcal{I}[\![stm]\!]_L^{e_1} = \langle P \rangle^{e_2}$ takes three arguments: an instruction *stm*, the label L of the block it belongs to, and an input clock e_1. It returns the type P of the instruction and its output clock e_2. The output clock of *stm* corresponds to the input clock of the instruction that immediately follows it in the sequence of the block.

Example 4 *Let us zoom on the block* L2 *of the ones counter, left. On the first line, for instance, we associate the instruction* T1 = idata *of block label* L2 *to the proposition* $x_{\mathsf{L2}} \Rightarrow \mathsf{T1} = \mathsf{idata}$. *In this proposition, the new variable* x_{L2} *is a boolean that is true iff the label* L2 *is being executed. So, the proposition says that, if the label* L2 *is being executed, then* T1 *is always equal to* idata. *If not, then another proposition may hold. In our case, all subsequent propositions are conditioned by* x_{L2} *meaning that they hold when* L2 *is being executed.*

(SSA code)	(behavioral type)
L2:T1 = idata;	$x_{\mathsf{L2}} \Rightarrow \mathsf{T1} := \mathsf{idata}$
T0 = T1 != 0;	$\mathsf{T0} := (\mathsf{T1} \neq 0)$
if T0 then goto L3;	$\mathsf{T0} \Rightarrow x'_{\mathsf{L3}}$
T2 = ocount;	$\neg\mathsf{T0} \Rightarrow \mathsf{T2} := \mathsf{ocount}$
T3 = T1 & 1;	$\mathsf{T3} := \mathsf{T1} \& 1$
ocount = T2 + T3;	$\mathsf{ocount}' := \mathsf{T2} + \mathsf{T3}$
idata = T1 >> 1;	$\mathsf{idata}' := \mathsf{T1} >> 1$
goto L2;	x'_{L2}

Next, consider the instruction if T0 then L3. *It is likely that label* L3 *will, just as* L2*, perform some operation on the input* idata. *Therefore, we delay its execution until after the current reaction and refer to* x'_{L3} *as the next value of the state variable* x_{L3}*, to indicate that it will be active during the next reaction. Hence, the proposition* $x_{\mathsf{L2}} \Rightarrow \mathsf{T0} \Rightarrow x'_{\mathsf{L3}}$ *says that control will be passed to* L3 *at the next reaction when control is presently at* L2 *and when* T0 *is true. The instructions that follow this test are conditioned by the negative* ¬T0*, this means: "in the block* L2 *and not in its branch to* L3*".*

Table 8 defines the behavioral type inference system. Rules $(1-2)$ are concerned with the iterative decomposition of a program *pgm* into blocks *blk* and with the decomposition of a block into *stm* and *rtn* instructions. In rule (2), the input clock e of the block *stm*; *blk* is passed to *stm*. The output clock e_1 of *stm* becomes the input clock of *blk*.

The input and output clocks of an instruction may differ. This is the case, rule (3), for an if x then L_1 instruction in a block L. Let e be the input clock of the instruction. When x is false then control is passed to the rest of the block, at the output clock $e \wedge \neg x$. Otherwise, control is passed to the block L_1, at the clock $e \wedge x$.

There are two ways of passing control from L to L_1 at a given clock e. They are defined by the function $\mathcal{G}_L(L_1, e)$: either immediately, by activating the immediate clock $x_{L_1}^{imm}$, i.e., $e \Rightarrow x_{L_1}^{imm}$; or by a delayed

transition to L_1 at e, i.e., $e \Rightarrow x'_{L_1}$. This choice is decided by the auxiliary function $\mathcal{S}_L(L_1)$. It checks whether the block L_1 can be executed immediately after the block L. By definition, $\mathcal{S}_L(L_1)$ holds iff $L_1 > L$ (L_1 is below L in the control flow) and $\mathcal{D}(L_1) \cap \mathcal{D}(L) = \emptyset$ (the variables defined in L and L_1 are disjoint).

Example 5 *In Example 3,* $\mathcal{D}(\mathsf{L1}) = \mathcal{D}(\mathsf{L2}) = \{\mathsf{ocount}, \mathsf{idata}\}$ *and* $\mathcal{D}(\mathsf{L3}) = \{\mathsf{count}, \mathsf{lock}\}$*. Hence, going from* L1 *to* L2 *requires a delayed transition because both* L1 *and* L2 *define* ocount *and* idata*. Conversely, going from* L2 *to* L3 *can be done immediately since* L3 *does not define* ocount *or* idata*.*

Table 8: Type inference

$$
\begin{array}{lrl}
(1) & \mathcal{I}[\![L{:}blk; pgm]\!] = & \mathcal{I}[\![blk]\!]_L^{x_L \vee x_L^{imm}} \mid \mathcal{I}[\![pgm]\!] \\
(2) & \mathcal{I}[\![stm; blk]\!]_L^e = & \text{let } \langle P \rangle^{e_1} = \mathcal{I}[\![stm]\!]_L^e \text{ in } P \mid \mathcal{I}[\![blk]\!]_L^{e_1} \\
(3) & \mathcal{I}[\![\mathsf{if}\ x\ \mathsf{then}\ L_1]\!]_L^e = & \langle \mathcal{G}_L(L_1, e \wedge x) \rangle^{e \wedge \neg x} \\
(4) & \mathcal{I}[\![x = f(y^*)]\!]_L^e = & \langle \mathcal{E}(f)(xy^*e) \rangle^e \\
(5) & \mathcal{I}[\![\mathsf{notify}\ x]\!]_L^e = & \langle e \Rightarrow (x' = \neg x) \rangle^e \\
(6) & \mathcal{I}[\![\mathsf{wait}\ x]\!]_L^e = & \langle e \wedge (x \neq x') \Rightarrow \hat{y} \mid e \setminus \hat{y} \Rightarrow x'_L \rangle^{\hat{y}} \\
(7) & \mathcal{I}[\![\mathsf{goto}\ L_1]\!]_L^e = & (e \Rightarrow x_L^{exit} \mid \mathcal{G}_L(L_1, e)) \\
(8) & \mathcal{I}[\![\mathsf{return}]\!]_L^e = & (e \Rightarrow (x_L^{exit} \mid x_f^{exit})) \\
(9) & \mathcal{I}[\![\mathsf{throw}\ x]\!]_L^e = & (e \Rightarrow (x_L^{exit} \mid \hat{x})) \\
\\
\text{where} & \mathcal{G}_L(L_1, e) = & \text{if } \mathcal{S}_L(L_1) \text{ then } e \Rightarrow x_{L_1}^{imm} \text{ else } \langle e \Rightarrow x'_{L_1} \rangle \\
& \mathcal{E}(f)(xyze) = & e \Rightarrow (\hat{y} \wedge \hat{z} \Rightarrow (\hat{x} \mid y \rightarrow x \mid z \rightarrow x)),\ \forall fxyze \\
\end{array}
$$

$$\mathcal{I}[\![\mathsf{catch}\ x\ \mathsf{from}\ L\ \mathsf{to}\ L_1\ \mathsf{using}\ L_2]\!]_L^e = \mathcal{G}_L(L_2, \hat{x} \wedge x_L^{exit}) \mid G_{L_2}(L_1, x_{L_2}^{exit})$$

Rule (4) is concerned with the typing of native and external method invocations $x = f(y^*)$. The generic type of f is taken from an environment $\mathcal{E}(f)$. It is given the name of the result x, of the actual parameters y^* and of the input clock e to obtain the type of $x = f(y^*)$. On the right, the generic type of 3-address instructions $x = f(y, z)$ at clock e is given by $\mathcal{E}(f)(xyze)$.

The wait-notify protocol (rules $(5-6)$) is modeled using a boolean flip-flop variable x. The notify method, rule (5), defines the next value of the lock x by the negation of its current value at the input clock e.

The wait method, rule (6), activates its output clock $\hat{y}$ iff the value of the lock x has changed at the input clock e: $e \wedge (x \neq x') \Rightarrow \hat{y}$. Otherwise, at the clock $e \setminus \hat{y}$, the control is passed to L by a delayed transition $e \setminus \hat{y} \Rightarrow x'_L$.

Example 6 *Consider the wait-notify protocol at the blocks labeled* L1 *and* L3 *in the* ones *counter. The type of the wait instruction defines the output clock* $\hat{y}$ *if* L1 *receives control at the clock* x_{L1}*, and if the value of lock has changed (proposition* $\mathsf{lock} \neq \mathsf{lock}'$*). If so, at the clock* $\hat{y}$*,* ocount *and* idata *are initialized and the control is passed to the block* L2 *by* $\mathcal{G}_{\mathsf{L1}}(\mathsf{L2}, \hat{\mathsf{y}})$*. Otherwise, at the clock* $x_{\mathsf{L1}} \setminus \hat{y}$*, a delayed transition to* L1 *is scheduled:* $x_{\mathsf{L1}} \setminus \hat{y} \Rightarrow x'_{\mathsf{L1}}$*.*

code	type	code	type
L1:wait (epc.lock);	$x_{\mathsf{L1}} \wedge (\mathsf{lock} \neq \mathsf{lock}') \Rightarrow \hat{y}$	L3:epc.count = ocount;	$\widehat{\mathsf{ocount}} \wedge x_{\mathsf{L3}} \Rightarrow \widehat{\mathsf{count}}$
⋮	$x_{\mathsf{L1}} \setminus \hat{y} \Rightarrow x'_{\mathsf{L1}}$	notify (epc.lock);	$x_{\mathsf{L3}} \Rightarrow \mathsf{lock}' = \neg\mathsf{lock}$
goto L2;	⋮	goto L1;	$x_{\mathsf{L3}} \Rightarrow x'_{\mathsf{L1}}$
	$\hat{y} \Rightarrow x'_{\mathsf{L2}}$		
⋮	⋮		

All return instructions, rules $(7-9)$, define the output clock x_L^{exit} of the current block L by their input clock e. This is the right place to do that: e defines the very condition upon which the block actually reaches its return statement.

A $\mathsf{goto}\, L_1$ instruction, rule (7), passes control to block L_1 unconditionally at the input clock e by $\mathcal{G}_L(L_1, e)$. A return instruction, rule (8), sets the exit clock x_f to true at clock e to inform the caller that f is terminated.

A $\mathsf{throw}\, x$ statement in block L, rule (9), triggers the exception signal x at the input clock e by $e \Rightarrow \hat{x}$. The matching catch statement, of the form $\mathsf{catch}\, x\, \mathsf{from}\, L\, \mathsf{to}\, L_1\, \mathsf{using}\, L_2$ passes the control to the handler L_2 and then to the block L_1 upon termination of the handler.

This requires, first, to activate L_2 from L when x is present, i.e., $\mathcal{G}_L(L_2, \hat{x} \wedge x_L^{exit})$, and then to pass control to L_1 upon termination of the handler.

Completion of the state logic The encoding of Table 8 requires all entry clocks x_L, x_L^{imm} and x_f to be simultaneously present when the f is being executed. Each signal x_L holds the value 1 iff the block L is active during a transition currently being executed. Otherwise, it is set to 0. This default setting of the entry clocks requires a completion of the next-state logic by considering, for all $L \in \mathcal{L}_f$, the proposition $e_L \Rightarrow x'_L$ implied by the inferred type $P = \mathcal{I}[\![pgm]\!]$ and define the default rule by $x_f \setminus e_L \Rightarrow \neg x'_L$. Completion is identical for the immediate and exit clocks x_L^{imm} and x_L^{exit} of the block L.

$$x_f \setminus e_L \Rightarrow \neg x'_L \text{ where } e_L \stackrel{\text{def}}{=} \bigvee (e \,|P \models e \Rightarrow x'_L\,)$$

Modular extension to external method calls The type inference scheme defined for wait, notify and operations, rules $(4-6)$ can be extended to handle externally defined method calls in a modular and compositional way, depicted in Table 9.

Table 9: Modular extension of the inference function to separately defined methods

$$\begin{array}{ll}
(a) & \mathcal{I}[\![m\, f(x_{1..m})\, \mathsf{raises}\, y\{\ pgm\ \}]\!]{=}\lambda x_{1..m} x_f\, x_f^{exit} y.\, \big(\mathcal{I}[\![pgm]\!] \,|\, x_f \Rightarrow x_{\min \mathcal{L}_f}\big)\, /\mathcal{L}_f \\
(b) & \mathcal{I}[\![L : x_0 = f(x_{1..m})]\!]_L^e{=}e \Rightarrow (\mathcal{E}(f)(x_{1..m} exy) \,|\, e \setminus (\hat{y} \vee \hat{x}) \Rightarrow x'_L)^{e \wedge \hat{x}} \\
(c) & \mathcal{I}[\![\mathsf{return}\, x]\!]_L^e{=}(e \Rightarrow (x_L^{exit} \,|\, x_f^{exit} := x))
\end{array}$$

Consider a method f with formal parameters $x_{1..m}$ (whose data-types are not displayed) and a result of type m, rule (a). Let y be an exception raised by the definition *pgm* of f and escaping from it. The type of f consists of a lambda abstraction whose arguments are the inputs $x_{1..m}$, the entry clock x_f, the exit clock x_f^{exit}, the return value y_f and the exception y. It is used to parameterize the proposition P, which corresponds to *pgm*, with respect to these arguments.

The lambda abstraction is instantiated in place of a method invocation $L : x_0 = f(x_{1..m})$, rule (b), which needs to be placed at the beginning of a block (assuming it can take several transitions before termination). To model the method call, one just needs to activate the entry clock x_f of the method at the input clock e.

The output signal x is used to carry the value of the result. Its clock determines when the method has reached the corresponding return statement (rule (c)). When the method terminates, the exit clock of the method call is defined by $e \wedge \hat{x}$. Otherwise, if the exception y is raised, a corresponding catch statement handles it. If f has not finished at the end of the transition (at the clock $e \setminus (\hat{y} \vee \hat{x})$), a delayed transition to L is performed $e \setminus (\hat{y} \vee \hat{x}) \Rightarrow x'_L$ in order to resume its execution at the next transition.

Static interface of SystemC modules The construction of a static abstraction from the behavioral type P of a program is automatic, thanks to the inference system, Figure 4.

Example 7 *For instance, the instruction* $\mathsf{T0} = (\mathsf{T1}! = 0)$ *is abstracted by the type* $x_{\mathsf{L2}} \Rightarrow \hat{\mathsf{T1}} = \hat{\mathsf{T0}}$ *. It means: "when the block* L2 *is executed,* T0 *is present iff* T1 *is present". The scheduling constraint* $x_{\mathsf{L2}} \Rightarrow \mathsf{T0} \rightarrow x_{\mathsf{L3}}$ *additionally says that "*x'_{L3} *cannot happen before* T0 *at* L2*". Indeed, one first needs to examine the status of* T0 *before branching to* L3*.*

Table 10: Abstraction of the behavioral type of block L2 by a static interface

(dynamic type)	(static type)	(scheduling)
$x_{\mathsf{L2}} \Rightarrow \mathsf{T1} = \mathsf{idata}$	$x_{\mathsf{L2}} \Rightarrow \hat{\mathsf{T1}} = \hat{\mathsf{idata}}$	$\mathsf{T1} \leftarrow \mathsf{idata}$
$\mathsf{T0} = (\mathsf{T1} \neq 0)$	$\hat{\mathsf{T0}} = \hat{\mathsf{T1}}$	$\mathsf{T1} \rightarrow \mathsf{T0}$
$\mathsf{T0} \Rightarrow \hat{x}_{\mathsf{L3}}$	$\mathsf{T0} \Rightarrow x'_{\mathsf{L3}}$	$\mathsf{T0} \rightarrow x'_{\mathsf{L3}}$
$\neg\mathsf{T0} \Rightarrow \mathsf{T2} = \mathsf{ocount}$	$\neg\mathsf{T0} \Rightarrow \hat{\mathsf{T2}} = \hat{\mathsf{ocount}}$	$\mathsf{T2} \leftarrow \mathsf{ocount}$
$\mathsf{T3} = \mathsf{T1}\&1$	$\hat{\mathsf{T3}} = \hat{\mathsf{T1}}$	$\mathsf{T1} \rightarrow \mathsf{T3}$
$\mathsf{ocount}' = \mathsf{T2} + \mathsf{T3}$	$\hat{\mathsf{ocount}} = \hat{\mathsf{T2}} \wedge \hat{\mathsf{T3}}$	$\mathsf{T2} \rightarrow \mathsf{ocount'} \leftarrow \mathsf{T3}$
$\mathsf{idata}' = \mathsf{T1} >> 1$	$\hat{\mathsf{idata}} = \mathsf{T1}$	$\mathsf{T1} \rightarrow \mathsf{idata'}$
x'_{L2}	x'_{L2}	$\mathsf{T0} \rightarrow x'_{\mathsf{L2}}$

The type associated with the block L2 *uses the clocks denoted by the booleans* x_{L2}*,* $\hat{\mathsf{T0}}$ *and* $\neg\mathsf{T0}$*. Each clock denotes a branch in the control-flow graph of the block* L2*. The other clocks, e.g.* $\hat{\mathsf{T1}}$*, denote the presence of data. They are partially related to the "label" clocks* x_{L2}*,* T0 *and* $\neg\mathsf{T0}$*.*

4.3 A behavioral module system

We define a module system starting from the behavioral type inference function of Section 4.2. The type $\mathcal{T}$ of a module m, Table 11, consists of an environment $\mathcal{E}$, that associates its methods f and fields x with types, a type P, that denotes the behavior of its constructor, and a proof obligation $\mathcal{C}$.

Table 11: Behavioral types for modules

(type)	$\mathcal{T}$	$::=$	$\langle \mathcal{E}, P, \mathcal{C} \rangle \mid \mathcal{T} \rightarrow \mathcal{T}$
(context)	$\mathcal{E}$	$::=$	$[] \mid \mathcal{E}[x : m] \mid \mathcal{E}[f : P] \mid \mathcal{E}[m : \mathcal{T}]$
(obligation)	$\mathcal{C}$	$::=$	$1 \mid P \Rightarrow Q \mid \mathcal{C} \wedge \mathcal{C}$

The type $\mathcal{T}_1 \rightarrow \mathcal{T}_2$ denotes a template class that produces a module of type $\mathcal{T}_2$ given a parameter of type $\mathcal{T}_1$. A proof obligation is a conjunction of propositions of the form $P \Rightarrow Q$. A proof obligation $P \Rightarrow Q$ is incurred by the instantiation of a template class, whose formal parameter has type P, by an actual class parameter, of type Q.

The type inference function for modules, $\mathcal{I}[\![sys]\!]_{\mathcal{E}}$, tables 12 and 13, assumes a type environment $\mathcal{E}$ that associates names to types. We write $\mathcal{E}(x)$ for the type of the location x and $\mathcal{E}(m.x) = \mathcal{F}(m)(x)$ for the path m to x iff $\mathcal{E}(m) = \langle \mathcal{F}, P, \mathcal{C} \rangle$.

Type inference for declarations Rule (a) sequentially processes the declarations *dec* in a module. Class field declarations contribute to building the type $\mathcal{T}$ of a module: rule (b) associates the location x with the type name m in the class-field $[x : m]$, rule (c) associates the procedure f with the class-field $[f : P]$. The type τ denotes a program that does nothing. It is neutral by composition.

In rule (d), the initialization of a thread $\mathsf{SC_THREAD}(f)\,\mathsf{sensitive} \ll x$ in the constructor is associated with the behavior $\mathcal{E}(f)$ of the method f it forks and with the type $\hat{x}_f \Leftarrow \hat{x}$, meaning that x triggers f.

Table 12: Type inference for declarations

$$
\begin{array}{ll}
(a) & \mathcal{I}[\![dec_1; dec_2]\!]_{\mathcal{E}} = \text{let } \langle \mathcal{E}_1, P_1, \mathcal{C}_1 \rangle = \mathcal{I}[\![dec_1]\!]_{\mathcal{E}} \text{ in } \langle \mathcal{E}_1, P_1, \mathcal{C}_1 \rangle \uplus \mathcal{I}[\![dec_2]\!]_{\mathcal{E}\mathcal{E}_1} \\
(b) & \mathcal{I}[\![m\,x]\!]_{\mathcal{E}} = \langle [x : m], \tau, 1 \rangle \\
(c) & \mathcal{I}[\![\mathsf{void}\ f()\ \{pgm\}]\!]_{\mathcal{E}} = \langle [f : \mathcal{I}[\![pgm]\!]_{\mathcal{E}}], \tau, 1 \rangle \\
(d) & \mathcal{I}[\![\mathsf{SC_THREAD}(f)\ \mathsf{sensitive} \ll x]\!]_{\mathcal{E}} = \langle [], \mathcal{E}(f) \,|\, (\hat{x}_f \Leftarrow \hat{x}), 1 \rangle
\end{array}
$$

Type inference for modules Rule (e) processes a sequence of module declarations $sys_1; sys_2$. We write $\langle \mathcal{E}_1, P_1, \mathcal{C}_1 \rangle \uplus \langle \mathcal{E}_2, P_2, \mathcal{C}_2 \rangle = \langle \mathcal{E}_1\mathcal{E}_2, P_1 \,|\, P_2, \mathcal{C}_1 \wedge \mathcal{C}_2 \rangle$ to union the types of sys_1 and sys_2. Whereas processing is sequential, the composition of the behavioral type $P_1 \,|\, P_2$ is synchronous.

Rule (f) first obtains the type $\mathcal{T}_1 = \langle \mathcal{E}_1, P_1, \mathcal{C}_1 \rangle$ of its class fields. Then, in the environment $\mathcal{E}$ extended with that of the class fields $\mathcal{E}_1$, the body $new; pgm$ of the constructor is processed to obtain its type $\mathcal{T}_2$. The type of the module becomes $m \cdot (\mathcal{T}_1 \uplus \mathcal{T}_2)$. The notation $m \cdot \langle \mathcal{E}, P, \mathcal{C} \rangle = \langle [m : \mathcal{E}], P, \mathcal{C} \rangle$ (resp. $m \cdot (\mathcal{T}_1 \to \mathcal{T}_2) = \mathcal{T}_1 \to (m \cdot \mathcal{T}_2)$) defines the type of the class m from the type $\langle \mathcal{E}, P, \mathcal{C} \rangle$ of its class fields.

Rule (g) determines the type of a template class m_2 whose formal parameter is a class m_1 that implements the virtual class m. The virtual class m provides the type, and hence the expected behavior, of the formal parameter name m_1. It is obtained from the environment $\mathcal{E}$ by $m \cdot \mathcal{T} = \mathcal{E}(m)$. The body of the template (i.e. the field declarations dec of the class m_2) is type-checked with the environment $\mathcal{E}$ extended with the association of m_1 to the type of the class fields $\mathcal{E}_1$ declared in m. This yields the type $m_2 \cdot \mathcal{T}_2$ of the class. The type of the template is defined by associating m_2 with the type $(m_1 \cdot \mathcal{T}_1) \to \mathcal{T}_2$ (and hence m_1 with the type $\mathcal{T}_1$).

Rule (h) performs the instantiation $m_2\langle m \rangle\, x$ of a template class m_2 with an actual parameter m to define the class name x. The type $(m_1 \cdot \mathcal{T}_1) \to \mathcal{T}_2$ of the template class m_2 and the type $m \cdot \mathcal{T}$ of the actual parameter m are obtained from the supplied environment $\mathcal{E}$. Type matching between $\mathcal{T}$ and $\mathcal{T}_1$ requires the resolution of a sub-typing between $\mathcal{T}_1[m_2.m/m_1]$ and $\mathcal{T}_2[m_2.m/m]$. The term $\mathcal{T}_1[m_2.m/m_1]$ stands for the substitution of the name m_1 by the concatenation $m_2.m$ in $\mathcal{T}_1$. The resolution of the type matching constraints reduces to the synthesis of the proof obligation $\mathcal{C}$ by the algorithm $\mathcal{R}$. If $\mathcal{C}$ is satisfied, then the type of the location x is $\mathcal{T}_2[m_2.m/m_1]$.

Table 13: Type inference for modules

$$
\begin{array}{lll}
(a) & \mathcal{I}[\![sys_1; sys_2]\!]_{\mathcal{E}} = & \text{let } \langle \mathcal{E}_1, P_1, \mathcal{C}_1 \rangle = \mathcal{I}[\![sys_1]\!]_{\mathcal{E}} \\
& & \text{in } \langle \mathcal{E}_1, P_1, \mathcal{C}_1 \rangle \uplus \mathcal{I}[\![sys_2]\!]_{\mathcal{E}\mathcal{E}_1} \\
(b) & \mathcal{I}\left[\!\!\left[\begin{array}{l}\mathsf{SC_MODULE}(m)\,\{\,dec; \\ \quad \mathsf{SC_CTOR}(m)\,\{new; pgm\}\}\end{array}\right]\!\!\right]_{\mathcal{E}} = & \text{let } \langle \mathcal{E}_1, P_1, \mathcal{C}_1 \rangle = \mathcal{I}[\![dec]\!]_{\mathcal{E}} \\
& & \text{and } \mathcal{T}_2 = \mathcal{I}[\![new; pgm]\!]_{\mathcal{E}\mathcal{E}_1} \\
& & \text{in } m \cdot (\langle \mathcal{E}_1, P_1, \mathcal{C}_1 \rangle \uplus \mathcal{T}_2) \\
(c) & \mathcal{I}\left[\!\!\left[\begin{array}{c}\mathsf{template}\,\langle \mathsf{class}\, m_1 \rangle \\ \#\mathsf{TYPE}(m_1, m) \\ \mathsf{class}\, m_2\,\{dec\}\end{array}\right]\!\!\right]_{\mathcal{E}} = & \text{let } m \cdot \mathcal{T} = \mathcal{E}(m) \\
& & \text{and } \langle \mathcal{E}_1, _, _ \rangle = m_1 \cdot \mathcal{T} \\
& & \text{and } \mathcal{T}_2 = \mathcal{I}[\![dec]\!]_{\mathcal{E}\mathcal{E}_1} \\
& & \text{in } \langle [m_2 : (m_1 \cdot \mathcal{T}_1) \to \mathcal{T}_2], \tau, 1 \rangle \\
(d) & \mathcal{I}[\![m_2\langle m \rangle\, x]\!]_{\mathcal{E}} = & \text{let } (m_1 \cdot \mathcal{T}_1) \to \mathcal{T}_2 = \mathcal{E}(m_2) \\
& & \text{and } m \cdot \mathcal{T} = \mathcal{E}(m) \\
& & \text{and } \mathcal{C} = \mathcal{R}(\mathcal{T}_1[m_2.m/m_1], \mathcal{T}[m_2.m/m]) \\
& & \text{in } x \cdot (\mathcal{T}_2[m_2.m/m_1]) \uplus \langle [], \tau, \mathcal{C} \rangle)
\end{array}
$$

Proof obligation synthesis The resolution $\mathcal{R}(\mathcal{T}_1, \mathcal{T}_2)$ of sub-typing constraints is defined by induction on the structure of the pair $(\mathcal{T}_1, \mathcal{T}_2)$. It reduces to the proof of a conjunction of propositions of the form $P_1 \Rightarrow P_2$ (of denotation $[\![P_1]\!] \subseteq [\![P_2]\!]$, Appendix 4.9).

$$\begin{aligned}
\mathcal{R}([],[]) &\Leftrightarrow 1 \\
\mathcal{R}(\mathcal{E}_1 \to \mathcal{T}_1, \mathcal{E}_2 \to \mathcal{T}_2) &\Leftrightarrow \mathcal{R}(\mathcal{E}_2, \mathcal{E}_1) \wedge \mathcal{R}(\mathcal{T}_1, \mathcal{T}_2) \\
\mathcal{R}(\mathcal{E}_1[x : t_1], \mathcal{E}_2[x : t_2]) &\Leftrightarrow \mathcal{R}(\mathcal{E}_1, \mathcal{E}_2) \wedge (t_2 \leq t_1) \\
\mathcal{R}(\mathcal{E}_1[f : P_1], \mathcal{E}_2[f : P_2]) &\Leftrightarrow \mathcal{R}(\mathcal{E}_1, \mathcal{E}_2) \wedge (P_2 \Rightarrow P_1) \\
\mathcal{R}(\mathcal{E}_1[m : \mathcal{T}_1], \mathcal{E}_2[m : \mathcal{T}_2]) &\Leftrightarrow \mathcal{R}(\mathcal{E}_1, \mathcal{E}_2) \wedge \mathcal{R}(\mathcal{T}_1, \mathcal{T}_2) \\
\mathcal{R}(\langle \mathcal{E}_1, P_1, \mathcal{C}_1 \rangle, \langle \mathcal{E}_2, P_2, \mathcal{C}_2 \rangle) &\Leftrightarrow \mathcal{R}(\mathcal{E}_1, \mathcal{E}_2) \wedge \mathcal{R}(P_1, P_2) \wedge \mathcal{R}(\mathcal{C}_1, \mathcal{C}_2)
\end{aligned}$$

If P_2 is *static* (i.e. $P_2 \Leftrightarrow \hat{P}_2$) then the problem reduces to checking satisfaction of the boolean proposition $\hat{P}_1 \Rightarrow \hat{P}_2$). If P_2 is *dynamic* then the problem reduces to verifying that $P_1 \Rightarrow P_2$ is an invariant of P, the type of the program, by using model-checking techniques. Both problems can be expressed and decided using the Polychrony workbench [27], as demonstrated in Appendix 4.9, where SystemC programs and their behavioral types are embedded in Signal.

4.4 Applications

We have introduced a type system allowing to model the control and data flow graphs of a given imperative program in SSA intermediate form and demonstrated that the expressive capability of the type system's semantics matched that of *de-facto* standard design languages (e.g. SystemC) and as well as that of related multi-clock synchronous formalisms (e.g. Signal). As such, applications of the proposed type system encompass optimization and verification issues encountered in related system design methodologies, yet with the following features that merit a highlight.

4.5 Scalability

Just as the theory of interfaces automata [10], types allow for a scalable level of abstraction to be automatically obtained starting from the type inferred from a SystemC module within the simple formalism of Table 1. Behavioral types share with interface automata the capability to define *static* interfaces (boolean relations) and *dynamic* interfaces (a transition system). Behavioral types allows to relate a given proposition P to a more abstract one, Q, in several ways:

- a transition $e \Rightarrow x' = y$ can be abstracted by a clock relation between e, $\hat{x}$ and $\hat{y}$;
- a bound signal x in P/x can be abstracted by any proposition Q not referencing it and containing P;
- a free signal x whose clock is not frequent (because it appears low in the hierarchy, see 3.4) can be abstracted by this clock in any Q containing P.

All these examples are instances of a more generic abstraction pattern. In general, checking a user-specified abstraction Q consistent with the type P inferred from a given program amounts to the satisfaction of the containment relation $[\![\hat{P}]\!] \subseteq [\![Q]\!]$ (i.e. the denotation of P is contained in the denotation of Q).

If P is a *static interface*, this amounts to the satisfaction of a boolean equation. Similarly, checking that a *dynamic interface* Q is an abstraction of a process P amounts to verifying that Q simulates P by model-checking.

Example 8 *To get a feel for hierarchical abstraction and highlight its benefits, let us just reconsider the naive modulo 2 counter of Example 2. The abstraction of the counter (left) consists of a bottom-up exploration of its hierarchy in the aim of eliminating state variables that are not directly involved with a property to verify.*

Here, clearly, we want to abstract the local state variable s and express the rest of the hierarchy with x and y. This yields the abstract hierarchy (right) that just say that y is a strict sampling of x.

$$P \stackrel{\text{def}}{=} \begin{array}{ccc} & \hat{x}, \hat{s} & \\ / & & \backslash \\ \neg s & & \hat{y}, s \end{array} \quad \Rightarrow \quad \begin{array}{ccc} & \hat{x} & \\ / & & \backslash \\ \hat{x} \setminus \hat{y} & & \hat{y} \end{array} \stackrel{\text{def}}{=} Q$$

The scalable abstraction of a dynamic interface type P by a type $Q = (\hat{x} > \hat{y})$ is obtained from the hierarchic structure of the partial-order between clocks. By contrast, the automata theory underlying Henzinger's interface types does not (directly) yield to such a representation.

4.6 Modularity

The main advantage of formulating a behavioral type system for SystemC is the formal foundation it offers to investigate modular and compositional design methodologies using separate compilation techniques. For instance, suppose that the declared type P of a SystemC class template provides sufficient information about its formal parameter for its body to be checked controllable and compiled.

One may then provide it with an actual class parameter, of type Q, satisfying $Q \Rightarrow P$, without having the burden of fully instantiating the template code and recompile its code for that given instance.

Example 9 *To exemplify the benefits of behavior type inference for SystemC modules, let us first consider the dynamic type P of a counter modulo 2. It generate an output event y upon two occurrences of the signal x. Its state s is initially false. The signal x triggers the calculation of the next value s' of s defined by the negation $\neg s$ of its current value. When s is true then y is triggered.*

$$P \stackrel{\text{def}}{=} \left(\neg s^0 \mid \hat{x} \Rightarrow (s' = \neg s) \mid s \Rightarrow \hat{y}\right)$$

Let $Q = (\hat{x} > \hat{y})$ be the static interface type of P declared for the virtual class of the introductory example, Section 1.2. It just requires the clock $\hat{y}$ to be a sampling of $\hat{x}$. Let us associate the type P and Q with the procedure f using the #TYPE annotation.

Let us now briefly recall the example of Section 1.2. First, an interface m_0, of virtual class fields x, y, and f is declared with the annotation #TYPE(f, Q). The template class m_2 uses a class parameter m_1 that implements m_0 to launch a thread $m_1.f$ sensitive to x. The class m_3 implements the interface m_0 and defines the method f by the program pgm of type P. Finally, class m_4 is defined by the instantiation of the template m_2 with the parameter m_3.

```
class m0 {
   virtual sc_clock x;
   virtual sc_clock y;
   virtual void f() {} #TYPE(f, Q)
};
template ⟨class m1⟩ #TYPE(m1, m0)
   SC_MODULE(m2) {SC_CTOR(m2) {
      SC_THREAD(m1.f) sensitive ≪ x
   }
};
class m3 {
   sc_clock x; sc_clock y;
   void f() { pgm} #TYPE(f, P)
};
m2⟨m3⟩ m4
```

Checking the guarantees of the actual parameter m_3 satisfy the assumptions of the formal parameter m_0 of the template m_2 amounts to verifying that P implies Q. This is done by calculating the static abstraction $\hat{P} = \exists s. (\hat{s} = \hat{x} \mid \hat{y} = s))$ of P and by checking that it implies $\hat{x} > \hat{y}$.

4.7 Design checking

The proposed type system allows to easily formulate properties pertaining on common design errors the analysis of which has been the subject of numerous related works. Most of these approaches consist of proposing an ad-hoc type system for analyzing a specific pattern of design errors: race conditions, deadlocks, threads termination; and in a given programming language: Java, C, SystemC.

By contrast, our behavioral type system provides a unified framework to perform both static verification via satisfaction checking or dynamic verification via model checking of behavioral properties of embedded systems described using imperative programming languages. The inference technique itself is language independent and the semantical peculiarities of language-specific runtime features and libraries can be modeled in the polychronous model of computation and its supportive type system.

Termination. One common design error found in embedded system design is the unexpected termination of a thread due to, e.g., an uncaught exception. In our behavioral model of SystemC, having the infinite loop of a thread f terminates can be represented by the property $x_f^{exit} = 1$. Unexpected termination can hence be avoided by model-checking the property that $x_f^{exit} = 0$ is an invariant of f:

$$P \models x_f^{exit} = 0$$

Deadlocks. Another common design error is a wait statement that does not match a notification and yields the thread to block. Let $x_{L_{1...n}}$ be the clocks of the blocks $L_{1...n}$ in which the lock x is notified. Waiting for x at a given label L eventually terminates if

$$P \models x_L \wedge \neg(\wedge_{i=1}^{n} x_{L_i}) = 0$$

Race conditions. Similarly, concurrent write accesses to a variable x shared by parallel threads can be checked exclusive by considering the input clocks $e_{1,..n}$ of all write statements $x = f(y, z)$ by verifying that

$$P \models (e_i \wedge (\vee_{j \neq i} e_j)) = 0, \forall i = 1, ..n$$

Example 10 *For instance, consider checking exclusion between the transitions of the* ones *counter. The type P of the counter implies the equations $x'_{\mathsf{L2}} = (\hat{y}_1 \vee \hat{y}_2)$ and $x'_{\mathsf{L1}} = x_{\mathsf{L3}}$. Verifying exclusion between them amounts to proving that $(\hat{y}_1 \vee \hat{y}_2) \wedge x_{\mathsf{L3}} = 0$ is an invariant of P. By construction of $\overline{P}$, we have: $\hat{y}_1 = x_{\mathsf{L1}} \wedge (\mathsf{lock} \neq \mathsf{lock}')$, $\hat{y}_2 = x_{\mathsf{L2}} \setminus \mathsf{T0}$ and $x_{\mathsf{L3}} = \mathsf{T0}$. The property follows by observing that $\hat{\mathsf{T0}} = x_{\mathsf{L2}}$.*

4.8 Design exploration

Just as the multi-clocked synchronous formalism Signal it is based upon, our type system allows for the refinement-based design methodologies considered in [26] to be easily implemented.

Checking the correctness of the refinement of an initial SystemC module, of type P, by its upgraded version, of type Q, amounts to verifying that the final type Q satisfies the assumptions made by the initial specification. In the spirit of the refinement-checking methodology proposed in [26], this can be implemented by checking the upgrade Q to be finitely flow preserving the initial design P.

In general, Q may differ from P by the insertion of a protocol between two components of P, by the adaptation of the services provided by P with a new functionality implemented in Q. Along the way, one may abstract from the type Q the signals and state variables introduced during the refinement in order to

accelerate verification. In most cases, such upgrades may be checked incrementally, by checking the static containment relation between the static abstractions of P and Q: $\hat{Q} \Rightarrow \hat{P}$.

4.9 Implementation in the Polychrony workbench

The Polychrony workbench [27] supports the synchronous multi-clocked data-flow programming language Signal in which the translation of the behavioral type system into Signal, Table 16, and the encoding of Signal by behavioral types, Table 15, is easily defined.

Signal belongs to the family of synchronous languages such as Esterel and Lustre. A Signal process P consists of simultaneous equations over *signals*. A signal $x \in \mathcal{X}$ describes an infinite flow of values $v \in \mathcal{V}$. An equation $x := f(y, r)$ denotes a relation between a the input signals (y, r) and the output signal x by a function or combinator f.

Signal has three primitive operators: the equation $x := y\$1\,\mathsf{init}\,v$ initially defines x by v and then by the previous value of y in time, the equation $x := y\,\mathsf{when}\,z$ defines x by y when z is true and the equation $x := y\,\mathsf{default}\,z$ defines x by y when y is present and by z otherwise. The synchronous composition $P \,|\, Q$ of two processes P and Q consists of the simultaneous solution of the system of equations P and Q.

Table 14: Signal syntax core

(process)	P	$::=$	$x := f(y, r) \mid (P \,|\, Q) \mid P/x$
(combinator)	f	$\in$	$\{\$1\,\mathsf{init}\,v \mid v \in \mathcal{V}\} \cup \{\mathsf{when}, \mathsf{default}, \ldots\}$

Table 15 associates Signal equations with behavioral types, showing their close relationship within the polychronous model of computation.

Table 15: From Signal to behavioral types $\lceil P \rceil$

$$\begin{array}{rlrl}
\lceil x := y\,\mathsf{when}\,z \rceil = & (z \Rightarrow x := y) & \lceil x := y\$1\,\mathsf{init}\,v \rceil = & (x^0 = v \mid x' := y) \\
\lceil x := y\,\mathsf{default}\,z \rceil = & (\hat{y} \Rightarrow x := y) & \lceil P \mid Q \rceil = & \lceil P \rceil \mid \lceil Q \rceil \\
 & \mid (\hat{z} \setminus \hat{y} \Rightarrow x := z) & \lceil P/x \rceil = & \lceil P \rceil / x
\end{array}$$

Table 16 completes the proof on the semantics equivalence between Signal and the behavioral type system. It defines an encoding of propositions by using partial equations $x ::= y\,\mathsf{when}\,z$, an additional feature of partial equation of the Polychrony workbench, whose meaning match that of propositions $z \Rightarrow x := y$.

Table 16: From behavioral types to Signal $\llbracket P \rrbracket$

$$\begin{array}{rlrl}
\llbracket x^0 = v \rrbracket = & x := x'\$1\,\mathsf{init}\,v & \llbracket v \rrbracket = & v \\
\llbracket l := r \rrbracket^e = & l ::= r\,\mathsf{when}\,\llbracket e \rrbracket & \llbracket \hat{x} \rrbracket = & \hat{}x \\
\llbracket x \to l \rrbracket^e = & x \to l\,\mathsf{when}\,\llbracket e \rrbracket & \llbracket l = r \rrbracket = & l = r \\
\llbracket P \mid Q \rrbracket^e = & \llbracket P \rrbracket^e \mid \llbracket Q \rrbracket^e & \llbracket e \wedge f \rrbracket = & \llbracket e \rrbracket \,\texttt{when}\, \llbracket f \rrbracket \\
\llbracket e \Rightarrow P \rrbracket^f = & \llbracket P \rrbracket^{e \wedge f} & \llbracket e \vee f \rrbracket = & \llbracket e \rrbracket \,\texttt{default}\, \llbracket f \rrbracket \\
\llbracket P/x \rrbracket^e = & \llbracket P \rrbracket^e / x & \llbracket e \setminus f \rrbracket = & \texttt{not}\, \llbracket f \rrbracket \,\texttt{default}\, \llbracket e \rrbracket
\end{array}$$

The behavioral type system supports a direct translation into the data-flow notation Signal of the Polychrony workbench [27]. This translation allows for the complete embedding of SystemC modules into Poly-

chrony, Table 17, to perform global design transformations, such as hierarchization or distribution and to perform a correct-by-construction design exploration towards the mapping of system functionalities on a target execution architecture.

Table 17: Embedding the intermediate representation in Signal

$$
\begin{aligned}
\mathcal{C}[\![L{:}blk;pgm]\!] &= \mathcal{C}[\![L{:}blk]\!]_L^{x_L} \mid \mathcal{C}[\![pgm]\!] \\
\mathcal{C}[\![stm;blk]\!]_L^e &= \text{let } \langle \mathtt{P}\rangle^{e_1} = \mathcal{C}[\![stm]\!]_L^e \text{ in } \mathtt{P} \mid \mathcal{C}[\![blk]\!]_L^{e_1} \\
\mathcal{C}[\![rtn]\!]_L^e &= \mathcal{T}[\![\mathcal{I}[\![rtn]\!]_L^e]\!] \\
\mathcal{C}[\![stm]\!]_L^e &= \text{if } (stm \neq \text{"}x = f(x_{1\ldots n})\text{"})) \\
& \quad \text{then let } \langle P\rangle^{e_1} = \mathcal{I}[\![stm]\!]_L^e \text{ in } \langle \mathcal{T}[\![P]\!]\rangle^{e_1} \\
& \quad \text{else let } \mathtt{e} = \mathcal{T}[\![e]\!] \text{ and } \mathtt{P} = \mathcal{T}[\![\mathcal{E}(f)(xx_{1\ldots n}e)]\!] \\
& \quad \text{in } \langle \mathtt{spec\ P\ pragmas\ CPP_CODE}\ \text{"}\mathtt{if\ e}\ \{\ \mathtt{x} = \mathtt{f}(\mathtt{x}_{1\ldots n})\ \}\text{"}\ \mathtt{end\ pragmas}\rangle^e
\end{aligned}
$$

The translation consists of representing guarded commands $e \Rightarrow (x' = v)$ and $e \Rightarrow \hat{x}$ by partial equations $\mathtt{x} ::= \mathtt{v}\ \mathtt{when}\ \mathtt{e}$ and $\hat{}\mathtt{x} ::= \mathtt{when}\ \mathtt{e}$, of identical meaning, and of embedding native method invocations $x = f(x_{1\ldots n})$ in wrappers (the `CPP_CODE` part) visible from Signal via a behavioral type (the `spec` part). Since the previous value of a signal x is noted $x\$1$ in Signal, we assume the default equation $x := x'\$1$ to define the current value x of a variable (of type bool or int in SystemC).

Example 11 *As an example, embedding the* ones *counter into Signal consists of emulating control by partial equations and of wrapping computations using typed* pragmas *statements. This embedding allows to operate global architectural transformations on the initial program, such as hierarchization or distributed protocol synthesis, using the Polychrony platform [27] or perform both static (SAT-checking) or dynamic (model-checking) verification of its design properties, whose spectrum is outlined next.*

(SSA code)	(Signal)
L1:wait (epc.lock);	$x_1 :=$ when x_{L1} when (lock $\neq$ lock$'$)
idata = epc.data;	$x'_{\mathsf{L1}} ::=$ when not x_1 default x_{L1}
ocount = 0;	idata::= data when x_1
goto L2;	ocount::= 0 when x_1
	$x'_{\mathsf{L2}} ::=$ when x_1
L2:T1 = idata;	T1:= idata\$1 when x_{L2}
T0 = T1 == 0;	T0:= (T1 = 0) when x_{L2}
if T0 then goto L3;	$x_{\mathsf{L3}} ::=$ when T0
T2 = ocount;	$x_2 :=$ when not x_{L3} default x_{L2}
T3 = T1 & 1;	T2:= ocount\$1 when x_2
ocount = T2 + T3;	T3:= T1 when x_2
idata = T1 >> 1;	ocount::= T2 + T3 when x_2
goto L2;	idata::= f(T1 when x_2)
	$x'_{\mathsf{L2}} ::=$ when x_2
L3:epc.count = ocount;	count::= ocount when x_{L3}
notify (epc.lock);	lock::= not lock when x_{L3}
goto L1;	$x'_{\mathsf{L1}} ::=$ when x_{L3}

The completion of the state-logic is implemented by the aggregation of partial state equations. Notice that L1 *and* L2 *are always activated by a delayed transition, whereas* L3 *is always immediate when* T0 *holds.*

Therefore:

$$
\begin{aligned}
&x_{\mathsf{L1}} := x'_{\mathsf{L1}}\$1\ \mathsf{init\ true}\\
&x_{\mathsf{L2}} := x'_{\mathsf{L2}}\$1\ \mathsf{init\ false}\\
&x'_{\mathsf{L1}} := \mathsf{true\ when}\ (\mathsf{not}\ x_1\ \mathtt{default}\ x_{\mathsf{L1}}\ \mathtt{default}\ x_{\mathsf{L3}})\ \mathtt{default}\ \mathsf{false}\\
&x'_{\mathsf{L2}} := \mathsf{true\ when}\ (x_1\ \mathtt{default}\ x_2)\ \mathtt{default}\ \mathsf{false}\\
&x_{\mathsf{L3}} := \mathsf{true\ when\ T0}\ \mathtt{default}\ \mathsf{false}\\
&x_f\hat{} = x_{\mathsf{L1}}\hat{} = x_{\mathsf{L2}}\hat{} = x_{\mathsf{L3}}
\end{aligned}
$$

We inlined native operations on integer-sized bit-arrays in the Signal code except the call to the native method >> that corresponds to an external function f defined by:

```
function f = ( ? i1 ! i2 )  spec (| ^i1 = ^i2 | i1→i2 |)
                            pragmas C_CODE "i2 = i1 >> 1" end pragmas;
```

Thanks to the polychronous model of computation, the SystemC scheduler, used to interleave parallel threads, does not have to be specified. Indeed, SystemC scheduling amounts to determining a fixed-point to inconsistently propagated signal values using the notion of δ-cycle, until a fixed-point is reached (Example 12). Fixed-point of δ-cycles relate to instants in the synchronous semantics using the notion of Kantor metrics of Lee et al. [15]. By contrast, the polychronous model of computation relies on the notion of synchrony to denote this fixed-point directly. Its implementation makes use of type-based scheduling information to determine a static scheduling of elementary instructions.

Example 12 *Whereas the simulation semantics of SystemC operates by performing δ-delays to determine a scheduling of instructions into micro-steps, synchrony allows to directly capture the fixed-point of this iterative process. For instance, consider two methods simulating* and-or *gates connected via the signals a, b, c, d, and e.*

```
void andgate () { while true {d=a&&b;wait()}}
void orgate  () { while true {e=d||c;wait()}}
```

The synchronous semantics of the circuit has all signals available instantaneously, whereas its SystemC simulation requires knowledge on the and *gate to be sensitive to a, b and then on the* or *gate to be sensitive to c, d in order to schedule the* and *operation first and the* or *operation second. In the real circuit, as in the polychronous model of computation, no δ-cycles are needed: scheduling is resolved at compile-time.*

5 Case study of a Finite Impulse Response filter

We exemplify our approach with the design of a finite impulse response filter (FIR). It details the decomposition of a full featured SystemC specification into an SSA representation. We demonstrate the different analysis steps until the final typed Signal representation.

We start off with the SystemC model of an FIR (an example from the SystemC 2.0.1 distribution [28]) and translate it into SSA code. We show how this SSA code is analyzed and how clock and scheduling information can be extracted. In Section 5.4 we present the corresponding Signal type and how it can be obtained with the preceding information.

5.1 The SystemC model

The SystemC model of the FIR filter is taken from an example of the SystemC 2.0.1 distribution [28]. The Filter itself is one functional block, it is surrounded by a testbench consisting of a Stimulus that generates input values and a Display that receives the output and displays it on the screen (Figure 12).

The FIR unit is implemented as an SC_THREAD that is triggered on the positive clock edge. The other blocks are SC_METHODS. The left hand side of Table 18 displays the SystemC code of the entry function for the FIR block. The first 10 lines just handle the initialization of variables. Then there is an infinite *while* loop that contains the actual filter functionality. Roughly speaking, it waits until there is a valid input available, reads this input, processes it writes it to an output and then notifies its environment that the result is available. At the end of each *while* loop it suspends itself until the next positive clock.

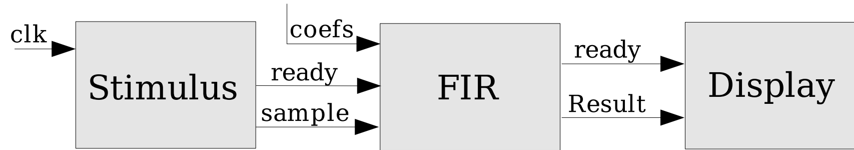

Figure 12: Structure of the FIR filter with testbench

The FIR result is the sum of the last 15 input values weighted with 15 coefficients. This is done in two *for* loops. The first one does the weighting and the second one is shifting the buffer array containing the last inputs. Communication with the environment is done via *enable* signals. The Stimulus indicates with the signal *input_valid* that a new value is available. In the same way, the Display is sensitive to the variable *output_data_ready* that is set when a new output value is available.

Table 18: SystemC code [28] and corresponding SSA code for the FIR

```
void fir::entry() {
  sc_int<8> sample_tmp;
  sc_int<17>pro;
  sc_int<19>acc;
  sc_int<8>shift[16];
  result.write(0);
  output_data_ready.write(false);
  for (int i=0; i<=15; i++)
    shift[i] = 0;
  wait();
  while(1) {
    output_data_ready.write(false);
    wait_until(input_valid.delayed() == true);
    sample_tmp = sample.read();
    acc = sample_tmp*coefs[0];
    for(int i=14; i>=0; i–) {
       pro = shift[i]*coefs[i+1];
       acc += pro;
    };
    for(int i=14; i>=0; i–)
       shift[i+1] = shift[i];
    shift[0] = sample_tmp;
    // write output values
    result.write((int)acc);
    output_data_ready.write(true);
    wait();
  };
}
```

```
void fir::entry() {
      int shift[16], i, acc, sample_tmp;
      i=0; goto L1;
      this → result = 0;
      this→output_data_ready = 0;
L0:   shift[i]= 0
      i = i + 1;
L1:   t_i = (i<=15)
      if (t_i) goto L0;
      else goto L1a;
L1a:wait (clk_pos);
L3:   this→output_data_ready = 0;
L3a:wait_until(input_valid == true);
L3b:sample_tmp = this→sample;
      acc = this→coefs[0] * sample_tmp;
      i = 14;
      goto L5;
L4:   acc = acc + shift[i]* this→coefs[i+1];
      i = i - 1;
L5:   if (i >= 0) goto L4;
      else goto L13;
L13:i = 14;
      goto L8;
L7:   shift[i + 1] = shift[i];
      i = i - 1;
L8:   if (i>=0) goto L7;
      else goto L9;
L9:   shift[0] = sample_tmp;
      this→result = acc;
      this→output_data_ready = 1;
L9a:wait (clk_pos);
      goto L3; }
```

5.2 Obtaining an SSA representation

The Single Static Assignment form (SSA) is an intermediate representation in which every variable is assigned exactly once. It is particularly used for high level compiler optimization and will be part of the upcoming version 3.5 of the Gnu Compiler Collection (GCC). For our approach, the usage of SSA as an intermediate step proves to be very useful: its simplicity makes the extraction of behavior straightforward.

The right hand side of Table 18 shows the SSA code that corresponds to the SystemC FIR. We use the tree-ssa [30] branch of the Gnu Compiler Collection [29] (the 3.5 version of GCC is not yet released).

For the generation of a clean SSA representation we follow three steps. First, preprocessing of the SystemC code, second, translation to SSA with GCC, and third, postprocessing of the generated SSA code. The direct generation of SSA from SystemC can be done, but it results in very large and hardly readable code. A closer look reveals that most of this bloating is due to the SystemC types and statements, which are implemented as macros and get translated as well. If we replace the SystemC types by corresponding C++ types, e.g. *sc_int* is changed to *int* or *unsigned*, in a simple preprocessing step, the size of the generated code shrinks drastically. More complex statements such as *wait(signal)*, however, still cause a considerable increase of the code size compared to the original SystemC code. We decide to simply comment these out in the SystemC source so they are ignored by the compiler and can later be taken care of separately in a postprocessing step.

Table 19 illustrates the different code sizes for different compiler options and preprocessing measures. Plain gimplifying of 39 lines of SystemC code results in over 2000 lines of gimple code. This can be reduced by using different compiler options for additional optimization steps. The option *-fdump-tree-optimized* results in almost 5 times less code, but still over 10 times the original SystemC code. Using no SystemC types and leaving the handling of the *wait* statements to the postprocessing results in an SSA code that is only slightly bigger than the original SystemC code.

Table 19: Different SSA code sizes for FIR entry function

original SystemC	39 lines
gcc -fdump-tree-gimple	2217 lines
gcc -fdump-tree-ssa	1310 lines
gcc -fdump-tree-optimized	446 lines
+ without SystemC types	142 lines
+ without wait()	55 lines

During post processing we replace the *wait(signal)* statements by corresponding SSA statements. Logically a *wait* statement is similar to an *if* branch. Depending on a condition something is executed, else something else. The condition is the signal that we are waiting for (e.g. *input_valid == true*. If there is no signal given, the process waits for the signal it is sensitive to (this is the positive slope of the clock in this example). In order to be able to execute the *wait* statement separately, we have to introduce a separate label for it. As we can see on the right hand side of Table 18, for *L1*, *L1a* is introduced since the wait statement is not at the beginning of the block, and for *L3*, there are two additional labels, *L3a* and *L3b* because this wait statement is in the middle of a block.

5.3 Extracting clock and scheduling information

Though slightly bigger in size, the SSA representation has several advantages in respect to automated analysis and conversion. On the one hand it consists of very simple, highly repetitive statements, and on the other hand it is separated into blocks of sequential execution that do not contain branches and where no variable

gets assigned twice. The extracted behavioral type information can be separated into two parts, control and data flow.

Table 15 displays this information separately for the FIR. On the left hand side there is the control flow in the form of clock relations, and on the right hand side the data flow information in the form of scheduling dependencies. In order to understand how these clock dependencies are obtained, we have to take a look at the SSA form in Table 18. $x_{\mathsf{L0}} \Rightarrow \hat{shift}$ means, that whenever block L0 is entered, the signal *shift* has to be present. Transitions from one block to another are represented like this: $x_{\mathsf{L4}} \Rightarrow x_{\mathsf{L5}}$. However, if in the following block a signal is assigned that has already been assigned in the current block, it cannot be executed in the same cycle. The time has to be advanced, this is expressed in $x_{\mathsf{fir}} \Rightarrow x'_{\mathsf{L1}}$, where the $'$ indicates the next value for this signal. For *if* statements - such as in block L1 - the value of a boolean signal decides which of the two targets is taken. The union ('$\wedge$') and the difference ('/') operator suffice to express all required action.

Figure 13 graphically details this control flow. There are several the small ones, such as the one between L1 and L2, represent the manipulation of an array of values. The big loop between L3 and L9 represents the actual program execution loop. Everything before that deals with initialization. After initialization the program waits for the next positive clock slope. At the beginning of the execution it is waiting for a valid input value. Then the calculations are executed and it subsequently waits for the next positive clock slope before resuming execution at block L3.

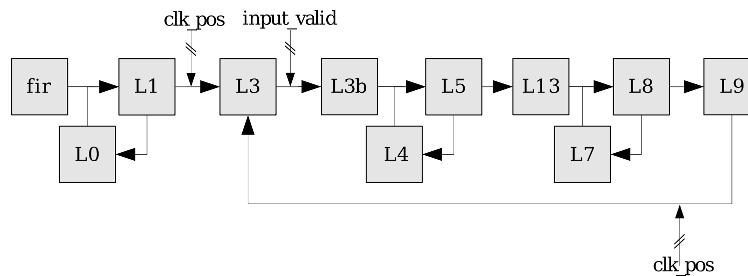

Figure 13: Control flow of the FIR filter

Data flow dependencies for the FIR are displayed on the right hand side of Table 15. The structure of these dependencies is very simple, the arrow (a→b) designing that *a* has to be present before *b* can be evaluated. Overall we see that for the FIR example, the control part largely overweights the data flow part.

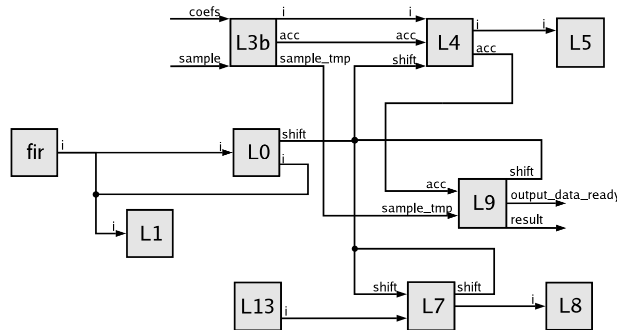

Figure 14: Data Flow of the FIR

Figure 14 illustrates the data flow of the example. We see, that for the execution loop, the major data activity takes place in blocks L3b, L4, L7, and L9. L3b reads the external inputs *coefs* and *sample*, L9 eventually produces the outputs *output_data_ready* and *result*.

$$x_{\mathsf{fir}} \Rightarrow \hat{i}$$
$$x_{\mathsf{fir}} \Rightarrow x'_{\mathsf{L1}}$$
$$x_{\mathsf{fir}} \Rightarrow \widehat{result}$$
$$x_{\mathsf{fir}} \Rightarrow \widehat{output_data_ready}; x_{\mathsf{L0}} \Rightarrow \widehat{shift}$$
$$x_{\mathsf{L0}} \Rightarrow \hat{i};\ x_{\mathsf{L0}} \Rightarrow x_{\mathsf{L1}} \qquad x_{\mathsf{L0}} \Rightarrow i \rightarrow shift$$
$$x_{\mathsf{L1}} \Rightarrow \widehat{t_i};\ x_{\mathsf{L1}} \Rightarrow \hat{i};\ x_{\mathsf{L1}} \wedge t_i \Rightarrow x'_{\mathsf{L0}} \qquad x_{\mathsf{L1}} \Rightarrow i \rightarrow t_i \rightarrow x'_{\mathsf{L0}}$$
$$x_{\mathsf{L1}}/x'_{\mathsf{L0}} \Rightarrow x_{\mathsf{L1a}}$$
$$x_{\mathsf{L1a}} \wedge (clk \neq clk') \Rightarrow \hat{y}$$
$$x_{\mathsf{L1a}} \wedge \hat{y} \Rightarrow x_{\mathsf{L3}};\ x_{\mathsf{L1a}}/\hat{y} \Rightarrow x'_{\mathsf{L1a}}$$
$$x_{\mathsf{L3}} \Rightarrow \widehat{output_data_ready};\ x_{\mathsf{L3}} \Rightarrow x_{\mathsf{L3a}}$$
$$x_{\mathsf{L3a}} \wedge input_valid \Rightarrow \hat{y}$$
$$x_{\mathsf{L3a}} \wedge \hat{y} \Rightarrow x_{\mathsf{L3b}};\ x_{\mathsf{L3a}}/\hat{y} \Rightarrow x'_{\mathsf{L3a}}$$
$$x_{\mathsf{L3b}} \Rightarrow \widehat{sample_tmp};\ x_{\mathsf{L3b}} \Rightarrow \widehat{sample};$$
$$x_{\mathsf{L3b}} \Rightarrow \widehat{acc};\ x_{\mathsf{L3b}} \Rightarrow \widehat{coefs}$$
$$x_{\mathsf{L3b}} \wedge \hat{i};\ x_{\mathsf{L3b}} \Rightarrow x_{\mathsf{L5}} \qquad x_{\mathsf{L3b}} \Rightarrow sample \rightarrow sample_tmp$$
$$x_{\mathsf{L4}} \Rightarrow \widehat{acc};\ x_{\mathsf{L4}} \Rightarrow \widehat{shift};\ x_{\mathsf{L4}} \Rightarrow \widehat{coefs} \qquad x_{\mathsf{L3b}} \Rightarrow coefs \rightarrow acc \leftarrow sample_tmp$$
$$x_{\mathsf{L4}} \Rightarrow \hat{i}$$
$$x_{\mathsf{L4}} \Rightarrow x_{\mathsf{L5}} \qquad x_{\mathsf{L4}} \Rightarrow i \rightarrow shift \rightarrow acc$$
$$x_{\mathsf{L5}} \wedge (i \mathbin{\hat{\geq}} 0) \Rightarrow x'_{\mathsf{L4}} \qquad x_{\mathsf{L4}} \Rightarrow i \rightarrow coefs \rightarrow acc$$
$$x_{\mathsf{L5}} \wedge (i \mathbin{\hat{<}} 0) \Rightarrow x'_{\mathsf{L13}}$$
$$x_{\mathsf{L13}} \Rightarrow \hat{i}$$
$$x_{\mathsf{L13}} \Rightarrow x_{\mathsf{L8}}$$
$$x_{\mathsf{L7}} \Rightarrow \widehat{shift};\ x_{\mathsf{L7}} \Rightarrow \hat{i};$$
$$x_{\mathsf{L7}} \Rightarrow x_{\mathsf{L8}}$$
$$x_{\mathsf{L8}} \Rightarrow \hat{i} \qquad x_{\mathsf{L7}} \Rightarrow i \rightarrow shift \rightarrow x_{\mathsf{L8}}$$
$$x_{\mathsf{L8}} \wedge (i \geq 0) \Rightarrow x'_{\mathsf{L7}}$$
$$x_{\mathsf{L8}}/x'_{\mathsf{L7}} \Rightarrow x_{\mathsf{L9}} \qquad x_{\mathsf{L8}} \Rightarrow i \rightarrow x'_{\mathsf{L7}}$$
$$x_{\mathsf{L9}} \Rightarrow \widehat{shift};\ x_{\mathsf{L9}} \Rightarrow \widehat{sample_tmp}$$
$$x_{\mathsf{L9}} \Rightarrow \widehat{result};\ x_{\mathsf{L9}} \Rightarrow \widehat{acc}$$
$$x_{\mathsf{L9}} \Rightarrow \widehat{output_data_ready};\ x_{\mathsf{L9}} \Rightarrow x_{\mathsf{L9a}} \qquad x_{\mathsf{L9}} \Rightarrow sample_tmp \rightarrow shift$$
$$x_{\mathsf{L9a}} \wedge (clk \neq clk') \Rightarrow \hat{y}; \qquad x_{\mathsf{L9}} \Rightarrow acc \rightarrow result \rightarrow output_data_ready$$
$$x_{\mathsf{L9a}} \wedge \hat{y} \Rightarrow x'_{\mathsf{L3}};\ x_{\mathsf{L9a}}/\hat{y} \Rightarrow x'_{\mathsf{L9a}}$$

Figure 15: Clock and Scheduling relations for the FIR

5.4 The equivalent Signal program

As described earlier, the combination of the control and dataflow can be expressed in the synchronous language Signal . In order to obtain this Signal code, it is helpful to have the clock and scheduling information particularized earlier, but it can also be obtained directly from the SSA representation. The signal type reflects all control information and basic dataflow. Figure 16 details the signal code for this FIR. At first, translation can be done by blocks, and mostly line by line. Whenever there is the need to advance time before the next block, i.e. the same variable is assigned in both, the execution of the next block is delayed with a statement such as *xL1 := xnL1$1 init false*. *xnL1* represents the next value and *xL1* the current value. The Signal language strictly prohibits the multiple assignment of a variable within one block and in the same instant. So if the time is not advanced between two assignments, the Signal compiler indicates

that there is an error. Obviously this is only needed, where we have delayed transitions, such as fir→L1, or loops such as in L0→L1→L0.

Variables that are assigned in several blocks can, in a first step, be defined with partial equations (e.g output_data_ready ::= false when xL3 | output_data_ready ::= true when xL9). Partial equations are a source of errors since it is difficult to make sure that there are no conflicts between them. This is why we subsequently combine these partial equations to complete equations (e.g output_data_ready := false when xL3 default true when xL9). In the code in Figure 16 six partial equations for variable i have been combined and it is easy to see that they are not in conflict. For array types, partial equations can currently not entirely avoided. This has to do with the fact that on different places in the program, different indexes are used (e.g. shift[i] when A— shift[j] when B). In addition, even if two array accesses are clearly separated (e.g. shift[0] := a—shift[1]:=b), the Signal compiler in the current revision V4.15 is not able to detect this.

```
process fir =
   (? boolean input_valid; integer sample
    ! integer result, boolean out_ready)
(| iold :=i$ init 0
 |coefs :=[16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
 | i := 0 when xfir
      default iold +1 when xL0
      default 14 when xL3b
      default iold -1 when xL4
      default 14 when xL13
      default iold -1 when xL7
 | result := 0 when xfir
      default acc when xL9
 | output_data_ready := false when xfir
      default false when xL3
      default true when xL9
 | xnL1 := when xfir
 | xL1 := when xL0
      default xnL1$1 init false
 | shift[i] ::= 0  when xL0
 | t_i :=  (i ¡= 15) when xL1
 | xnL0:= when t_i
 | xL0 := xnL0$1 init false
 | xL3 := when xL1a when not(clk = clk$)
      default when xL9a when not(clk = clk$)
 | xnL1a := when not t_i
      default when xL1a when not xL3
 | xL1a := xnL1a$1 init false
 | xL3a := xL3
      default xnL3a$1 init false
 | xL3b := when xL3a
      when input_valid==true
 | xnL3a := when xL3a when not xL3b
 | sample_tmp := sample when xL3b
 | acc := coefs[0] * sample_tmp
      when xL3b
      default acc + shift[i] * coefs[i]
      when xL4
 | xL5 := xL3b
 | xnL4:= when (i¿=0 when xL5)
 | xL4 := xnL4$1 init false
 | xnL13:= when (i¡0 when xL5)
 | xL13 := xnL13$1 init false
 | xL8 := xL13
 | shift[i+1] ::=  shift[i] when xL7
 | xnL7 :=  when (i¡=0 when xL8)
 | xL7 := xnL7$1 init false
 | xL9 :=  when (i¿0 when xL8)
 | shift[0] ::= sample_tmp when xL9
 | xnL9a := when xL9
      default when xL9a when not xL3
 | xL9a := xnL9a$1 init false
 | xfir ≙xL0 ≙xL1 ≙xL1a ≙xL3 ≙xL3a
   ≙xL3b ≙xL4 ≙xL5 ≙xL7 ≙xL8 ≙xL9
   ≙xL9a ≙xL13 ≙xnL0 ≙xnL1 ≙xnL1a
   ≙xnL3a ≙xnL4 ≙xnL13 ≙xnL7 ≙xnL9a|)
where
  boolean t_i;
  integer i, iold, acc, sample_tmp;
  [16] integer shift; [16] integer coefs;
  event xfir, xL0, xL1, xL1a, xL3,  xL3a,
  xL3b, xL4, xL5, xL7, xL8, xL9, xL9a, xL13,
  xnL0, xnL1, xnL1a, xnL3a, xnL4, xnL13,
  xnL7, xnL9a;
end;
```

Figure 16: Signal type for the FIR filter

In this example we do not have any complex data manipulations. Where these occur, it can be very complex to describe them in Signal, so for these cases they are wrapped into external functions, leaving them in the original code. This permits the handling of data flow intensive applications without much additional cost.

5.4.1 Abstracting the signal type

The type in Figure 16 implements the FIR (it is an exact Signal mirror of the original SystemC implementation). For many purposes, however, all functionality is not needed in order to evaluate the validity of a condition. An abstracted type, that does not contain data manipulations is much lighter and still can serve to check conditions such as deadlocks, termination, and race-conditions. Figure 17 depicts the code for a possible abstraction for the fir type. The light weight for this type allows for much faster verification of properties, and, therefore makes it possible to check for these properties on a higher level, possibly comprising the whole system. For detailed checks including correctness of data manipulations or range-checks, still the complete type can be used.

```
process fir =
   (? boolean input_valid; integer sample
    ! integer result, boolean out_ready)
(| result := 0 when xfir
      default 1 when xL9
 | output_data_ready := false when xfir
      default false when xL3
      default true when xL9
 | xL1a := when xfir
 | xL3 := when xL1a when not(clk = clk$)
      default when xL9a when not(clk = clk$)
 | xnL1a := when not t_i
      default when xL1a when not xL3
 | xL1a := xnL1a$1 init false
 | xL3a := xL3
      default xnL3a$1 init false
 | xL3b := when xL3a
      when input_valid==true
 | xnL3a := when xL3a when not xL3b
 | xL9 := when xL3b
 | xnL9a := when xL9
       default when xL9a when not xL3
 | xL9a := xnL9a$1 init false
 | xfir ≙ xL1a ≙xL3 ≙xL3a ≙xL3b
   ≙xL9 ≙xL9a ≙xnL1a ≙xnL3a ≙xnL9a|)
where
  event xfir, xL1a, xL3,  xL3a,
  xL3b, xL9, xL9a,  xnL1a, xnL3a,xnL9a;
end;
```

Figure 17: Abstracted Signal type

5.5 Status

To get this process smoothly to work, there are some obstacles. As we have seen in Section 5.2, it is not obvious to have a clean SSA code. Therefore substantial effort has to be put into the generation of clean and reasonably short SSA code. This can be done with compiler optimization on the one hand and pre- / postprocessing on the other.

When generating the clock dependencies and the Signal type respectively, the crucial point is the advancement of time. It has to be made sure that if blocks assign the same variable, there has to be an advancement of time in between. As we have different branches, this problem breaks down to graph coloring and is not trivial.

The presented approach illustrates its applicability for C++ and the same applies for Java . However, for SystemC, there are some additional issues to consider. As for now, we treated only one entry function of a SystemC program. In order to type a whole SystemC application consisting of several modules in the same way, multiple entry functions would have to be treated as well as the architecture and connectivity between them. This is ineffective to do in SSA because the change to the lower level will obstruct the higher level hierarchy structures. Consequently the preprocessing step has to be designed to be more sophisticated in order to handle the structure correctly. Also adequate Signal equivalence for certain SystemC constructs such as *sc_fifo* or *sc_semaphore* have to be defined.

The approach we presented shows how to obtain a behavioral Signal type from SystemC components. The passage through the SSA form allows for a relatively straightforward translation to the formal synchronous description. When handling plain C++ or Java code the pre-and-post-processing steps can be omitted, significantly simplifying the transformation process. In the current state of the project, the transformations

are done manually, but we are working on automating this process, which should be relatively straightforward due to the genericity of the translation steps.

6 Related works

On behavioral abstraction. Our approach is based on a high-level specification methodology that ensures compositional correctness through a type theory capturing behavioral aspects of component interfaces. The proposed system builds upon previous work on scalable design exploration using refinement and abstraction-based design methodologies [26], implemented in the Polychrony workbench [27], and on a layered Component Composition Environment, Balboa [11], which allows the specification of SystemC [28] components with mixed levels of structural and behavioral details, and a high degree of concurrency and strict timing requirements. The capture of the behavior of a system through a type theoretical framework relates our technique to the work of Rajamani et al. [24], and many others, on abstracting high-level and concurrent specifications, e.g. the π-calculus, by using a formalism, e.g. Milner's CCS, in which, primarily, checking type equivalence, e.g. bisimulation, is decidable.

Our contribution contrasts from related studies by the capability to capture scalable abstractions of the type-checked system. In our type system, scalability ranges from the capability to express the exact meaning of the program, in order to make structural transformations and optimizations on it (as demonstrated Example 2), down to properties expressed by boolean equations between clocks, allowing for a rapid static-checking of design correctness properties (as demonstrated in the examples of Section 4.4, especially Example 10). Our system further allows for a wide spectrum of correct-by-construction design abstractions and refinement patterns to be applied on a model, e.g. abstraction of states by clocks, abstraction of existentially quantified clocks, hierarchic abstraction (as demonstrated Example 7), in the aim of choosing an optimal degree of abstraction for a faster verification.

We share the aim of a scalable and correct-by-construction exploration of abstraction/refinement of system behaviors with the work of Henzinger et al. on interface automata [10]. Our approach primarily differs from interface automata in the data-structure used in the Polychrony workbench: clock equations, boolean propositions and state variable transitions express the multi-clocked synchronous behavior of a system. Compared to an automata-based approach, our declarative approach allows to hierarchically explore abstraction capabilities and to cover design exploration with the methodological notion of refinement along the whole design cycle of the system, ranging from the early requirements specification to the latest sequential and distributed code-generation [26, 16].

On synchrony and asynchrony. Synchronous programming is a computational model that is popular in hardware design, and desynchronization is a technique to convert that computational model into a more general, globally asynchronous and locally synchronous computational model, suitable for system-on-chip design. Therefore, one may hence naturally consider investigating further the links between these two models understood as Ptolemy domains [7] and study the refinement-based design of GALS architectures starting from polychronous specifications captured from heterogeneous elementary components. The aim of capturing both synchrony and asynchrony in a unifying model of computation is shared by several approaches: interaction categories of Abramsky et al. in [1], communicating sequential processes of Hoare [12], Kahn networks [14], latency insensitive protocols of Carloni et al. [8], the heterogeneous systems of Benveniste et al. [4]. These models partition systems into synchronous islands (i.e. predefined *pearls* or IPs) and asynchronous networks. For instance, the heterogeneous model of [4] defines a tag-less model of asynchrony and a densely tagged model of synchrony (where stuttering equivalence is mathematically identical to the clock equivalence relation of the polychronous model of computation) Synchrony and asynchrony are not partitioned in the present model. Both are captured within the same partially ordered trace structure, and related by the clock and finite-flow equivalence relations $\sim$ and $\approx^*$. The iSTS algebra carries over this

unified model to capture modeling, transformation and verification of embedded systems from the highest levels of requirements specification down to its clock-accurate implementation as a Kahn network or a GALS architecture.

7 Conclusions and future works

The main novelty in our approach is the use of a multi-clocked synchronous formalism to support the construction of a scalable behavioral type inference system for the *de facto* standard system-design language SystemC, and the materialization of a companion refinement-based design methodology imposed through the strong typing policy of a module system, that reduces compositional design correctness verification to the validation of synthesized proof obligations. The proposed type system allows to capture the behavior of an entire system-level design and to re-factor it, allowing to generate an optimized scheduling using hierarchization, allowing to modularly express a wide spectrum of static and dynamic behavioral properties, and to automatically or manually scale the desired degree of abstraction of these properties for efficient verification. The type system is presented using a generic and language-independent intermediate representation. It operates transformations implemented in the platform Polychrony, to perform refinement-based design exploration and directly yields to verification tools using SAT checking and model checking allowing for an efficient verification of expected design properties and an early discovery of design errors.

References

[1] Abramsky, S., Gay, S. J., Nagarajan, R. Interaction categories and the foundations of typed concurrent programming. In *Deductive Program Design: Proceedings of the 1994 Marktoberdorf International Summer School*. Nato Asi Series F, Springer-Verlag, 1996.

[2] Amagbegnon, T. P., Besnard, L., Le Guernic, P. “Implementation of the data-flow synchronous language Signal”. In *Conference on Programming Language Design and Implementation*. ACM Press, 1995.

[3] Hoe, J., Arvind. ”Synthesis of Operation-Centric Hardware Descriptions”. *Proceedings of International Conference on Computer Aided Design*. IEEE Press, November 2000.

[4] Benveniste, A., Caspi, P., Carloni, L. P., Sangiovanni-Vincentelli, A. L. “Heterogeneous Reactive Systems Modeling and Correct-by-Construction Deployment”. In *Embedded Software Conference*. Lecture Notes in Computer Science, Springer Verlag, October 2003.

[5] Benveniste, A., Caspi, P., Le Guernic, P., Marchand, H., Talpin, J.-P., Tripakis, S. “A protocol for loosely time-triggered architectures”. In *Embedded Software Conference*. Lecture Notes in Computer Science, Springer Verlag, October 2002.

[6] Berry, G., Gonthier, G. “The Esterel synchronous programming language: design, semantics, implementation”. In *Science of Computer Programming*, v. 19, 1992.

[7] J.T. Buck, S. Ha, E.A. Lee and D.G. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. In *International Journal of Computer Simulation, special issue on ”Simulation Software Development”*v. 4, pp. 155-182. Ablex, April 1994.

[8] Carloni, L. P., McMillan, K. L., Sangiovanni-Vincentelli, A. L. “Latency-Insensitive Protocols”. In *Proceedings of the 11th. International Conference on Computer-Aided Verification*. Lecture notes in computer science v. 1633. Springer Verlag, July 1999.

[9] E. Dijkstra “A Discipline of Programming”. Prentice Hall, 1976.

[10] De Alfaro, L., Henzinger, T. A. “Interface theories for component-based design”. *International Workshop on Embedded Software*. Lecture Notes in Computer Science v. 2211. Springer-Verlag, 2001.

[11] F. Doucet, S. Shukla, and R. Gupta. “Typing abstractions and management in a component framework”. *Asia and South Pacific Design Automation Conference*, January 2003.

[12] Hoare, C. *Communicating sequential processes*. Prentice Hall, 1985.

[13] Jeffords, R. and Heitmeyer, C. ”A Strategy for Efficiently Verifying Requirements Specifications Using Composition and Invariants”. *Symposium on the Foundations of Software Engineering*. ACM Press, September 2003

[14] Kahn, G. The semantics of a simple language for parallel programming. In Ifip *Congress*. North Holland, 1974.

[15] Lee, E. A., Sangiovanni-Vincentelli, A. “A framework for comparing models of computation”. In *IEEE transactions on computer-aided design*, v. 17, n. 12. IEEE Press, December 1998.

[16] Le Guernic, P., Talpin, J.-P., Le Lann, J.-L. Polychrony for system design. In *Journal of Circuits, Systems and Computers. Special Issue on Application-Specific Hardware Design*. World Scientific, 2002.

[17] Marchand, H., Bournai, P., Le Borgne, M., Le Guernic, P. Synthesis of Discrete-Event Controllers based on the Signal Environment. In *Discrete Event Dynamic System: Theory and Applications*, v. 10(4), pp. 325–346, 2000.

[18] H. Marchand, E. Rutten, M. Le Borgne, M. Samaan. Formal Verification of SIGNAL programs: Application to a Power Transformer Station Controller. *Science of Computer Programming*, v. 41(1), pp. 85–104, 2001.

[19] Mousavi, M., R., Le Guernic, P., Talpin, J.-P., Shukla, S., Basten, T. Modeling and validation of asynchronous systems in synchronous frameworks. In *Digital Automation and Test Europe*. IEEE Press, February 2004.

[20] Novillo, D. "Tree SSA, a new optimization infrastructure for GCC". GCC developers summit, 2003.

[21] Nowak, D., Beauvais, J.-R., Talpin, J.-P. "Co-inductive axiomatization of a synchronous language". In *International Conference on Theorem Proving in Higher-Order Logics*. Lecture Notes in Computer Science, Springer Verlag, October 1998.

[22] Nowak, D., Talpin, J.-P., Le Guernic, P. "Synchronous structures". In *International Conference on Concurrency Theory*. Lecture Notes in Computer Science, Springer Verlag, August 1999.

[23] Pnueli, A., Shankar, N., Singerman, E. Fair synchronous transition systems and their liveness proofs. *International School and Symposium on Formal Techniques in Real-time and Fault-tolerant Systems*. Lecture Notes in Computer Science v. 1468. Springer Verlag, 1998.

[24] S. K. Rajamani and J. Rehof, "A behavioral module system for the π-calculus". *Static Analysis Symposium*. Lecture Notes in Computer Science. Springer Verlag, July 2001.

[25] Talpin, J.-P., Gamatié, A., Le Dez, B., Berner, D., Le Guernic, P. "Hard real-time implementation of embedded systems in JAVA". *International Workshop on Scientific Engineering of Distributed JAVA Applications*. Lecture Notes in Computer Science. Springer Verlag, November 2003.

[26] J.-P. Talpin, P. Le Guernic, S. K. Shukla, R. Gupta, and F. Doucet. "Polychrony for formal refinement-checking in a system-level design methodology". *Application of Concurrency to System Design*. IEEE Press, June 2003.

[27] The Polychrony website. http://www.irisa.fr/espresso/Polychrony, 2004.

[28] The OSCI SystemC website. http://www.SystemC.org, 2004.

[29] The Gnu Compiler Collection (GCC). http://http://gcc.gnu.org, March 2004.

[30] The GCC Tree-SSA Branch. http://gcc.gnu.org/projects/tree-ssa, March 2004.

A Naming conventions

We refer to the *free variables* vars(P) of a process P, Table 18, as the set of signal names that occur free in the lexical scope of P.

$$\begin{array}{c}
\mathrm{vars}(0) = \mathrm{vars}(1) = \emptyset \qquad \mathrm{vars}(x^0 = v) = \mathrm{vars}(x = v) = \mathrm{vars}(x' = v) = \{x\} \\
\mathrm{vars}(x = y) = \mathrm{vars}(x' = y) = \mathrm{vars}(x \to y) = \mathrm{vars}(x \to y') = \{x, y\} \\
\mathrm{vars}(P/x) = \mathrm{vars}(P) \setminus \{x\} \quad \mathrm{vars}(e \wedge f) = \mathrm{vars}(e \vee f) = \mathrm{vars}(e \setminus f) = \mathrm{vars}(e) \cup \mathrm{vars}(f) \\
\mathrm{vars}(e \Rightarrow P) = \mathrm{vars}(e) \cup \mathrm{vars}(P) \qquad \mathrm{vars}(P \,|\, Q) = \mathrm{vars}(P) \cup \mathrm{vars}(Q)
\end{array}$$

Figure 18: Free variables vars(P) of a process P

The states and output signals def(P) and out(P) of a process P are defined by the relation of Figure 19 starting from the disjunctive form $\mathcal{D}_P$ of P (Figure 20). A signal $x \in \mathrm{out}(P)$ (resp. $x \in \mathrm{def}(P)$) is an output (resp. state) of P iff, whenever its presence is implied by a guard g, then there exists a guard h implied by g whose action is $x := r$ (resp. $x' := r$) for some r. In the definition of the generic function locs(P), we overload $\hat{x}$ to $\hat{l}$ and assume that $\hat{x'} \stackrel{\mathrm{def}}{=} \hat{x}$.

$$\begin{array}{rcl}
x \in \mathrm{def}(P) & \Leftrightarrow & x' \in \mathrm{locs}(\mathcal{D}_P) \wedge \exists 0 < i \leq n, g_i \stackrel{\mathrm{def}}{=} 1 \wedge a_i \stackrel{\mathrm{def}}{=} x^0 = v \\
x \in \mathrm{out}(P) & \Leftrightarrow & x \in \mathrm{locs}(\mathcal{D}_P) \wedge x \notin \mathrm{def}(P) \\
\mathrm{in}(P) & = & (\mathrm{vars}(P) \setminus \mathrm{def}(P)) \setminus \mathrm{out}(P)
\end{array}$$

$$\begin{array}{l}
l \in \mathrm{locs}\left(\left(\prod_{i=1}^{n} g_i \Rightarrow a_i\right)/x_{1..m}\right) \Leftrightarrow \\
\qquad \forall 0 < i \leq n \, s.t. \, \hat{P} \models g_i \Rightarrow \hat{l}, \exists\, 0 < j \leq n, \hat{P} \models g_i \Rightarrow g_j \wedge a_j \stackrel{\mathrm{def}}{=} l := r
\end{array}$$

Figure 19: Input and output signals of a process

The disjunctive form $\mathcal{D}_P$ of the process P is the composition of elementary guarded commands $g \Rightarrow a$ consisting of a conjunctive clock proposition g and of an atomic action a.

$$D ::= G \,|\, D/x \qquad G ::= g \Rightarrow a \,|\, (G \,|\, G) \qquad g ::= 0 \,|\, 1 \,|\, (x = r) \,|\, g \wedge g \,|\, g \setminus g$$

$$\begin{array}{rclrcl}
\mathcal{D}[\![l = r]\!]^g & = & g \Rightarrow l = r & \mathcal{D}[\![f \Rightarrow P]\!]^e & = & \mathcal{D}[\![P]\!]^{e \wedge f} \\
\mathcal{D}[\![x \to l]\!]^g & = & g \wedge \hat{x} \Rightarrow x \to l & \mathcal{D}[\![P/x]\!]^e & = & (\mathcal{D}[\![P]\!]^e)/x \\
\mathcal{D}[\![a]\!]^{e \vee g} & = & \mathcal{D}[\![a]\!]^e \,|\, \mathcal{D}[\![a]\!]^g & \mathcal{D}[\![P \,|\, Q]\!]^e & = & \mathcal{D}[\![P]\!]^e \,|\, \mathcal{D}[\![Q]\!]^e
\end{array}$$

Figure 20: Disjunctive form $\mathcal{D}_P$ of a process P

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399