A Behavioral Type Inference System for Compositional System-on-Chip Design

Jean-Pierre Talpin, David Berner, Sandeep Shukla, Paul Le Guernic,
Abdoulaye Gamatié, R.K. Gupta

A behavioral type inference system for compositional system-on-chip design

Jean-Pierre Talpin[1] David Berner[1] Sandeep Kumar Shukla[2]
Paul Le Guernic[1] Abdoulaye Gamatié[1] Rajesh Gupta[3]
[1]*INRIA-IRISA* [2]*Virginia Tech* [3]*University of California at San Diego*

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 type system is conceptually minimal and supports a scalable notion and a flexible degree of abstraction. Our presentation targets the de facto standard SystemC, yet with a generic and language-independent method. Applications of our technique range from the detection of local design errors to the compositional assembly of modules.

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 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 [8], implemented in the Polychrony workbench, and on a layered Component Composition Environment, Balboa [2], 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), akin to a subset of Pnueli's synchronous transition systems [6]. It 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. In [9], it is equipped with a formal semantics defined in polychronous model of computation [4] and implemented by the Polychrony workbench (available from http : //www.irisa.fr/espresso/Polychrony).

2 Rationale

To allow for an easy grasp on the proposed behavioral type inference technique, we outline the analysis of a SystemC program (Figure 1, left), by depicting the construction

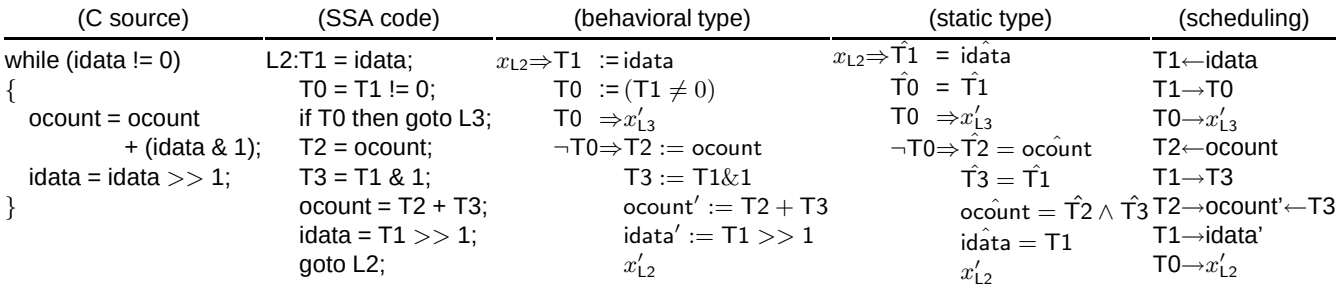

Figure 1. From a SystemC program to a behavioral types and its abstraction

of its behavioral type (Figure 1, middle) and the inference of its static abstraction (Figure 1, right). 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.

Static single assignment. Figure 1, left, depicts a simple C code fragment consisting of an iterative program that counts the number of bits set to one in the variable idata. While idata is not equal to zero, it adds its right-most bit to an output count variable ocount and shifts it right in order to process the next bit. In the static single-assignment (SSA) representation of the program (Figure 1, second column) all variables (idata and ocount) are read and written once per cycle. Label L2 is the entry point of the SSA block associated with the while loop. The first instruction loads the input variable idata into the register T1. The second instruction stores the result of its comparison with 0 in the register T0. If T0 is false, control is passed to block L3. Otherwise, the next instruction is executed: the variable ocount is loaded into T2, the last bit of T1 is loaded into T3, the sum of T2 and T3 assigned to ocount and the right-shift of T1 assigned to idata. The block terminates with an unconditional branch back to label L2.

Behavioral types. The SSA intermediate representation of an imperative program represents 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 1. The behavioral type of the block L2, middle, consists of the simultaneous composition of logical propositions (middle) that form a synchronous transition system. Each proposition is associated with one instruction: it specifies its *invariants*. In particular, it tells when the instruction is executed, what it computes, when it passes control to the next statement, when it branches to another block.

On line 1 for instance, we associate the instruction $\mathsf{T1} := \mathsf{idata}$ to the proposition $x_{\mathsf{L2}} \Rightarrow \mathsf{T1} := \mathsf{idata}$. In this proposition, the variable x_{L2} is a boolean that is true iff the block of label L2 is being executed. Hence, the proposition says that, if the label L2 is being executed, then T1 is equal to idata. Otherwise, another proposition may hold. In the present case, all propositions are conditioned by x_{L2} to mean that they hold when block L2 is being executed.

The extent of a proposition is 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".

Static abstraction. We have seen that every instruction of an SSA program could be associated with a proposition to render its control-flow and data-flow behaviors. This representation provides a 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 some terminology. A clock $\hat{x}$ is associated with 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. On Figure 1, right, all operations on integers reported in the behavioral type are abstracted by boolean relations between clocks and by scheduling relations. This is in fact sufficient information to reconstruct the entire control and data flow graphs of the program. All information abstracted from this view essentially consists of computational instructions with which we could decorate this graph to 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 to branch 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.

Typed modules. Section 5 develops the use of the information provided by the behavioral type inference system to perform design correctness checks. 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.

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 $\#\mathsf{TYPE}(f, Q)$ which associates f with a description of its behavior: the proposition Q (a denotation of all possible implementations satisfying an expected functionality).

$$
\begin{array}{l}
\mathsf{class}\, m_0 \,\{\, \mathsf{virtual\ sc\_clock}\, x,\, y; \\
\qquad\quad \mathsf{virtual\ void}\ f()\ \{\}\ \#\mathsf{TYPE}(f, Q)\ \}; \\
\mathsf{template}\, \langle \mathsf{class}\, m_1 \rangle\ \#\mathsf{TYPE}(m_1, m_0) \\
\quad \mathsf{SC\_MODULE}(m_2)\ \{ \\
\qquad \mathsf{SC\_CTOR}(m_2)\ \{ \\
\qquad\quad \mathsf{SC\_THREAD}(m_1.f)\ \mathsf{sensitive} \ll x\ \}\ \}; \\
\mathsf{class}\, m_3 \,\{\, \mathsf{sc\_clock}\, x,\, y; \\
\qquad\quad \mathsf{void}\ f()\ \{\ pgm\}\ \#\mathsf{TYPE}(f, P)\ \}; \\
m_2\langle m_3 \rangle\ m_4;
\end{array}
$$

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.

Proof obligations. 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 $\#\mathsf{TYPE}(f, P)$. 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. This amounts to check that P implies Q. 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.

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 [6] and its syntax to Dijkstra's guarded command language, we call is the iSTS (i for implicit). 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.

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

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

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 $\neg s^0$ (or $s^0 = 0$). 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 left-most 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 right-most 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. 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

(component) $sys ::= [\mathsf{template}\ \langle \mathsf{class}\, m_1\rangle\ \#\mathsf{TYPE}(m_1, m_2)]\ \mathsf{class}\, m\, \{dec\} \mid \mathsf{SC\_MODULE}(m)\ \{dec; \mathsf{SC\_CTOR}(m)\ \{new\}\} \mid sys; sys$

(declaration) $dec ::= m\langle m^*\rangle\, x \mid \mathsf{void}\ f()\ \{pgm\} \mid dec; dec$ (constructor) $new ::= \mathsf{SC\_THREAD}(f)\, \mathsf{sensitive} \ll x^* \mid new; pgm$

(program) $pgm ::= L{:}blk \mid pgm; pgm$ (instruction) $stm ::= x = f(x^*) \mid \mathsf{wait}\, x \mid \mathsf{notify}\, x \mid \mathsf{if}\, x\, \mathsf{then}\, L$

(block) $blk ::= stm; blk \mid rtn$ (return) $rtn ::= \mathsf{goto}\, L \mid \mathsf{return} \mid \mathsf{throw}\, x; \mathsf{catch}\, x\, \mathsf{from}\, L\, \mathsf{to}\, L\, \mathsf{using}\, L$

Figure 2. Abstract syntax for SystemC

this end, one could for instance compose the counter P with the proposition $Q{=}^{\mathrm{def}}\ (y' = y + 1 \,|\, y^0 = 0)$.

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.

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 essentially geared towards verification by model-checking: one is choice $P \vee Q$, the other explicit absence $\perp$. Both can be modeled in the iSTS. An order of execution can be imposed to a proposition by its refinement with a scheduling constraint, noted $y \rightarrow x$. Then, $x = y$ is an abstraction of $x = y \,|\, y \rightarrow x$ 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. In the remainder, we adopt the syntax $x := y$ borrowed to Signal for an assignment of y to x: $(x = y \,|\, y \rightarrow x)$.

Formal syntax of the type system. The formal syntax of propositions in the behavioral type system is defined by the inductive grammar P. 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 is either a value v or a signal x.

(reference) $r ::= x \mid v$ (location) $l ::= x^0 \mid x \mid x'$

(clock) $e, f ::= 0 \mid x = r \mid e \wedge f \mid e \vee f \mid e \setminus f \mid 1$

(process) $P, Q ::= 1 \mid l = r \mid x \rightarrow l \mid e \Rightarrow P \mid (P \,|\, Q) \mid P/x$

A clock expression e is a proposition on boolean values that, when true, 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 $\hat{x}{=}^{\mathrm{def}}(x = x)$, which stands for "$x$ is present"; the clock $x{=}^{\mathrm{def}}(x = 1)$ for "x is true", and the clock $\neg x{=}^{\mathrm{def}}(x = 0)$ for "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 $x \rightarrow l$ 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.

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.

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, Figure 2. 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 $\#\mathsf{TYPE}(m_1, m_2)$ to denote the type of m_1 with the virtual class m_2. A module $\mathsf{SC\_MODULE}(m)$ is a class that defines an architecture component. Its constructor $\mathsf{SC\_CTOR}(m)\ \{new; pgm\}$ allocates threads (e.g. $\mathsf{SC\_THREAD}(f)$) and executes an initialization

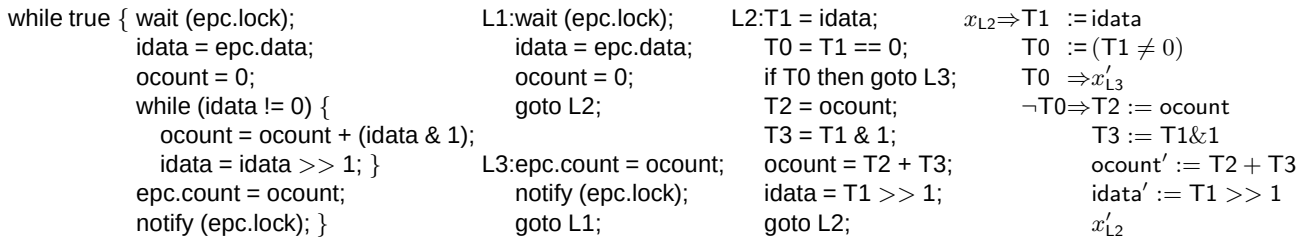

Figure 3. Translation of the SystemC thread of the EPC into SSA form

program pgm. While 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 C++ sub-typing (e.g., $\mathsf{bool} \leq \mathsf{num}$, $\mathsf{int} \leq \mathsf{num}$).

The formal grammar of SystemC programs, Figure 2, is represented in static single-assignment intermediate form akin to that of the Tree-SSA package of the GCC project [5]. 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 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 1 *To outline the construction of the intermediate representation of a SystemC program, let us reconsider the example of Section 2 and detail the method that counts the number of bits set to* 1 *in a bit-array* epc.data *(Figure 3). 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 to add its right-most bit to* ocount *until termination (condition* T0*). In the block* L3, ocount *is sent to the signal* epc.count *and* epc.lock *is unlocked before going back to* L1.

Behavioral type inference. The type inference function $\mathcal{I}[\![pgm]\!]$, Figure 4, 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 before 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 Figure 4). 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 2 *Let us zoom on the block* L2 *of the ones counter (Figure 3). On the first line, for instance, we associate the instruction* $\mathsf{T1} = \mathsf{idata}$ *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. Next, consider the instruction* $\mathsf{if}\, \mathsf{T0}\, \mathsf{then}\, \mathsf{L3}$. *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*

$$(1)\quad \mathcal{I}[\![L{:}blk; pgm]\!] = \mathcal{I}[\![blk]\!]_L^{x_L \vee x_L^{imm}} \,|\, \mathcal{I}[\![pgm]\!] \qquad (5)\ \mathcal{I}[\![\mathsf{notify}\ x]\!]_L^e = \langle e \Rightarrow (x' = \neg x)\rangle^e$$

$$(2)\quad \mathcal{I}[\![stm; blk]\!]_L^e = \text{let } \langle P\rangle^{e_1} = \mathcal{I}[\![stm]\!]_L^e \text{ in } P \,|\, \mathcal{I}[\![blk]\!]_L^{e_1} \qquad (6)\quad \mathcal{I}[\![\mathsf{wait}\ x]\!]_L^e = \langle e \wedge (x \neq x') \Rightarrow \hat{y} \,|\, e \setminus \hat{y} \Rightarrow x'_L\rangle^{\hat{y}}$$

$$(3)\ \mathcal{I}[\![\mathsf{if}\ x\ \mathsf{then}\ L_1]\!]_L^e = \langle \mathcal{G}_L(L_1, e \wedge x)\rangle^{e \wedge \neg x} \qquad (7)\ \mathcal{I}[\![\mathsf{goto}\ L_1]\!]_L^e = (e \Rightarrow x_L^{exit} \,|\, \mathcal{G}_L(L_1, e))$$

$$(4)\quad \mathcal{I}[\![x = f(y^*)]\!]_L^e = \langle \mathcal{E}(f)(xy^*e)\rangle^e \qquad (8)\quad \mathcal{I}[\![\mathsf{return}]\!]_L^e = (e \Rightarrow (x_L^{exit} \,|\, x_f^{exit}))$$

$$(5)\quad \mathcal{I}[\![\mathsf{notify}\ x]\!]_L^e = \langle e \Rightarrow (x' = \neg x)\rangle^e \qquad (9)\ \mathcal{I}[\![\mathsf{throw}\ x]\!]_L^e = (e \Rightarrow (x_L^{exit} \,|\, \hat{x}))$$

where $\mathcal{G}_L(L_1, e) = \text{if } \mathcal{S}_L(L_1) \text{ then } e \Rightarrow x_{L_1}^{imm} \text{ else } \langle e \Rightarrow x'_{L_1}\rangle$ and $\mathcal{E}(f)(xyze) = e \Rightarrow (\hat{y} \wedge \hat{z} \Rightarrow (\hat{x} \,|\, y \to x \,|\, z \to x))$

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

Figure 4. Type inference

will be active during the next reaction. Hence, the proposition $x_{\mathsf{L2}} \Rightarrow \mathsf{T0} \Rightarrow x'_{\mathsf{L3}}$ *says that control passes 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*".*

Figure 4 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 after L in the control flow) and $\text{defs}(L_1) \cap \text{defs}(L) = \emptyset$ (the set of variables defined in L and L_1 are disjoint).

Example 3 *In Example 1,* $\text{defs}(\mathsf{L1}) = \text{defs}(\mathsf{L2}) = \{\mathsf{ocount}, \mathsf{idata}\}$ *and* $\text{defs}(\mathsf{L3}) = \{\mathsf{count}, \mathsf{lock}\}$*. 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*.*

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

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 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 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 catch x from L to L_1 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 the control to L_1 upon termination of the handler.

Completion of the state logic. The encoding of Figure 4 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, x_L 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. We write $x_f \setminus e_L \Rightarrow \neg x'_L$ where $e_L =^{\text{def}} \bigvee (e \,|\, P \models e \Rightarrow x'_L)$.

Extension to 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, as depicted next. 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 that this block 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.

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

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.

A behavioral module system. We define a module system starting from the behavioral type system of Section 4. The type $\mathcal{T}$ of a module m consists of an environment $\mathcal{E}$, that associates fields f and x with types, of a type P that denotes the behavior of its constructor, and of a proof obligation $\mathcal{C}$. The type $\mathcal{T}_1 \to \mathcal{T}_2$ 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 and by an actual class parameter, of type Q.

$$
\begin{array}{lrcl}
\text{(type)} & \mathcal{T} & ::= & \langle \mathcal{E}, P, \mathcal{C} \rangle \,|\, \mathcal{T} \to \mathcal{T} \\
\text{(context)} & \mathcal{E} & ::= & [] \,|\, \mathcal{E}[x : m] \,|\, \mathcal{E}[f : P] \,|\, \mathcal{E}[m : \mathcal{T}] \\
\text{(obligation)} & \mathcal{C} & ::= & 1 \,|\, P \Rightarrow Q \,|\, \mathcal{C} \wedge \mathcal{C}
\end{array}
$$

The type inference function for modules, $\mathcal{I}[\![sys]\!]_{\mathcal{E}}$, Figure 5, assumes a type environment $\mathcal{E}$ that associates names with 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.

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 merge 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 $(m_1 \cdot \mathcal{T}_1) \to \mathcal{T}_2$

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

$$(e) \qquad \mathcal{I}[\![sys_1; sys_2]\!]_{\mathcal{E}} = \text{let } \langle \mathcal{E}_1, P_1, \mathcal{C}_1\rangle = \mathcal{I}[\![sys_1]\!]_{\mathcal{E}} \text{ in } \langle \mathcal{E}_1, P_1, \mathcal{C}_1\rangle \uplus \mathcal{I}[\![sys_2]\!]_{\mathcal{E}\mathcal{E}_1}$$

$$(f)\ \mathcal{I}\left[\!\!\left[\begin{array}{l}\mathsf{SC\_MODULE}(m)\,\{\\ dec;\ \mathsf{SC\_CTOR}(m)\,\{new; pgm\}\}\end{array}\right]\!\!\right]_{\mathcal{E}} = \begin{array}{l}\text{let } \langle \mathcal{E}_1, P_1, \mathcal{C}_1\rangle = \mathcal{I}[\![dec]\!]_{\mathcal{E}} \text{ and } \mathcal{T}_2 = \mathcal{I}[\![new; pgm]\!]_{\mathcal{E}\mathcal{E}_1}\\ \text{in } m \cdot (\langle \mathcal{E}_1, P_1, \mathcal{C}_1\rangle \uplus \mathcal{T}_2)\end{array}$$

$$(g)\ \mathcal{I}\left[\!\!\left[\begin{array}{l}\mathsf{template}\,\langle \mathsf{class}\, m_1\rangle\\ \quad \#\mathsf{TYPE}(m_1, m)\,\mathsf{class}\, m_2\,\{dec\}\end{array}\right]\!\!\right]_{\mathcal{E}} = \begin{array}{l}\text{let } m \cdot \mathcal{T} = \mathcal{E}(m) \text{ and } \langle \mathcal{E}_1, \_, \_\rangle = m_1 \cdot \mathcal{T} \text{ and } \mathcal{T}_2 = \mathcal{I}[\![dec]\!]_{\mathcal{E}\mathcal{E}_1}\\ \text{in } \langle [m_2 : (m_1 \cdot \mathcal{T}_1) \to \mathcal{T}_2], \tau, 1\rangle\end{array}$$

$$(h) \qquad \mathcal{I}[\![m_2\langle m\rangle\, x]\!]_{\mathcal{E}} = \begin{array}{l}\text{let } (m_1 \cdot \mathcal{T}_1) \to \mathcal{T}_2 = \mathcal{E}(m_2) \text{ and } m \cdot \mathcal{T} = \mathcal{E}(m)\\ \text{in } x \cdot (\mathcal{T}_2[m_2.m/m_1]) \uplus \langle [], \tau, (\mathcal{R}(\mathcal{T}_1[m_2.m/m_1], \mathcal{T}[m_2.m/m]))\rangle)\end{array}$$

Figure 5. Type inference for declarations and modules

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 $\mathcal{T}_2[m_2.m/m_1]$.

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 $P_1 \Rightarrow P_2$. If P_2 is *stateless* (i.e. it defines no state transitions) then the problem reduces to checking satisfaction of the boolean proposition $\hat{P}_1 \Rightarrow P_2$ (where $\hat{P}_1$ stands for the stateless abstraction of P_1, e.g., the right-most columns of figure 6). If P_2 is *statefull* 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 in the Polychrony workbench by encoding behavioral types in Signal.

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

5 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.

Scalability. Just as the theory of interfaces automata [1], types allow for a scalable level of abstraction to be automatically obtained starting from the type inferred from a SystemC module using a simple formalism. 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. 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 $[\![P]\!] \subseteq [\![Q]\!]$ (i.e. the denotation of P is contained in the denotation of Q). If P is a *stateless interface*, this amounts to the satisfaction of boolean equations. Similarly, checking that a *statefull interface* Q is an abstraction of a process P amounts to verifying that Q simulates P by model-checking.

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.

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

$x_1 := \mathsf{when}\ x_{\mathsf{L1}}\ \mathsf{when}\ (\mathsf{lock} \neq \mathsf{lock}')$
$x'_{\mathsf{L1}} ::= \mathsf{when}\ \mathsf{not}\ x_1\ \mathtt{default}\ x_{\mathsf{L1}}$
$\mathsf{idata} ::= \mathsf{data}\ \mathsf{when}\ x_1$
$\mathsf{ocount} ::= 0\ \mathsf{when}\ x_1$
$x'_{\mathsf{L2}} ::= \mathsf{when}\ x_1$
$\mathsf{T1} := \mathsf{idata}\$1\ \mathsf{when}\ x_{\mathsf{L2}}$
$\mathsf{T0} := (\mathsf{T1} = 0)\ \mathsf{when}\ x_{\mathsf{L2}}$
$x_{\mathsf{L3}} ::= \mathsf{when}\ \mathsf{T0}$
$x_2 := \mathsf{when}\ \mathsf{not}\ x_{\mathsf{L3}}\ \mathtt{default}\ x_{\mathsf{L2}}$
$\mathsf{T2} := \mathsf{ocount}\$1\ \mathsf{when}\ x_2$
$\mathsf{T3} := \mathsf{T1}\ \mathsf{when}\ x_2$
$\mathsf{ocount} ::= \mathsf{T2} + \mathsf{T3}\ \mathsf{when}\ x_2$
$\mathsf{idata} ::= (\mathsf{T1} \gg 1)\ \mathsf{when}\ x_2$
$x'_{\mathsf{L2}} ::= \mathsf{when}\ x_2$
$\mathsf{count} ::= \mathsf{ocount}\ \mathsf{when}\ x_{\mathsf{L3}}$
$\mathsf{lock} ::= \mathsf{not}\ \mathsf{lock}\ \mathsf{when}\ x_{\mathsf{L3}}$
$x'_{\mathsf{L1}} ::= \mathsf{when}\ x_{\mathsf{L3}}$

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

Figure 6. Model of the EPC in Signal

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. The termination of the infinite loop of a thread f can be expressed by the property $x_f^{exit} = 1$. Unexpected termination can hence be avoided by model-checking the property that the opposite 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 $x_{L_{1\ldots n}}$ be the clocks of the blocks $L_{1\ldots n}$ in which a 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 5 *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 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 $\hat{\mathsf{T0}} = x_{\mathsf{L2}}$.*

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 [8] 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 [8], this can be implemented by checking the refinement Q to be finitely flow preserving with respect to 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 statically checking the containment relation between the stateless abstractions of P and Q.

Implementation in the Polychrony workbench. The Polychrony workbench supports the synchronous multi-clocked data-flow programming language Signal in which the translation of the behavioral type system into Signal is easily defined.

Example 6 *As an example, embedding the* ones *counter into Signal, Figure 6, 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 or perform both static (SAT-checking) or dynamic (model-checking) verification of its design properties, whose spectrum is outlined next. 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.*

Thanks to the polychronous model of computation, the SystemC scheduler, used to interleave the execution of

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. Fixed-point of δ-cycles relate to instants in the synchronous semantics using the notion of Kantor metrics of Lee et al. [3]. 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.

6 Related works and conclusions

The capture of the behavior of a system through a type theoretical framework relates our technique to the work of Rajamani et al. [7], 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, 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 5, especially Example 5). Our system allows for a wide spectrum of design abstraction and refinement patterns to be applied on a model, e.g. abstraction of states by clocks, abstraction of existentially quantified clocks, hierarchic abstraction, in the aim of choosing a better degree of abstraction for 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 [1]. 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 [8, 4].

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 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] 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.

[2] F. DOUCET, S. SHUKLA, AND R. GUPTA. "Typing abstractions and management in a component framework". *Asia and South Pacific Design Automation Conference*, January 2003.

[3] 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.

[4] LE GUERNIC, P., TALPIN, J.-P., LE LANN, J.-C. Polychrony for system design. In *Journal of Circuits, Systems and Computers. Special Issue on Application-Specific Hardware Design*. World Scientific, 2002.

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

[6] 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.

[7] 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.

[8] 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.

[9] J.-P. TALPIN, BERNER, D., SHUKLA, S. K., LE GUERNIC, P., GAMATIÉ, A., GUPTA, R. "Behavioral type inference for compositional system design". Technical report n. 5141. INRIA, March 2004.