A Compositional Behavioral Modeling Framework for Embedded System Design and Conformance Checking

Jean-Pierre Talpin,[1,4] Paul Le Guernic,[1]
Sandeep Kumar Shukla,[2] and Rajesh Gupta[3]

We propose a framework based on a synchronous multi-clocked model of computation to support the inductive and compositional construction of scalable behavioral models of embedded systems engineered with *de facto* standard design and programming languages. Behavioral modeling is seen under the paradigm of type inference. The aim of the proposed type system is to capture the behavior of a system under design and to re-factor it by performing global optimizing and architecture-sensitive transformations on it. It allows to modularly express a wide spectrum of static and dynamic behavioral properties and 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 static single assignment intermediate representation.

KEY WORDS: Embedded system design; formal methods; models of computation; program transformation; verification.

1. INTRODUCTION

The popular slogan "*write once, run anywhere*" effectively renders the expressive capabilities of general purpose programming languages for

---

[1]INRIA-IRISA, Rennes, France. E-mail: talpin@irisa.fr
[2]Virginia Tech, Blacksburg, USA. E-mail: shukla@vt.edu
[3]University of California San Diego, La Jolla, USA. E-mail: rgupta@ucsd.edu
[4]To whom correspondence should be addressed.

developing, deploying, and reusing target-independent applications. Generality and simplicity has driven most attention of the compiler technology community to developing local and compositional compiler optimization techniques. When it comes to the implementation of embedded software, this approach is however far from satisfactory, especially in hard real-time system design (e.g. airborne systems, digital circuits) where conformance to real-time specifications is critical.

Domain-specific models and languages, such as these proposed under the synchronous programming paradigm, provides the necessary formal engineering models and design methodologies to allow for a program written once to be mapped on any distributed execution architecture by using global transformation and optimization techniques. Our aim is to relate this domain-specific model to embedded software development using general-purpose environments. To this end, we set the methodological framework of our synchronous model of computation within the general and reusable concept of a type system targeting the generic programming language setting of GCC's intermediate representations (three-address code and static single assignment). We give formal semantics to both our type system and the functional subset of SSA under consideration, define a type inference system and prove its correctness, before to depict the applications of our technique as developed in our project and presented in previous works.

A functional application domain. We consider embedded software implemented by resource-constrained[5] multi-threaded programs on a specific runtime sub-system (e.g., the real-time JVM, an RTOS, or simply hardware) which we call its execution architecture. Our technique consists of a type inference system that relates threads (imperative programs in intermediate form) to propositions expressed by synchronous transition systems that describe their behavior.

Figure 1 outlines the extent of our technique by depicting a test-case studied in Ref. 1. We consider modeling a real-time Java program consisting of three threads (right), a scheduler (top-left), and shared resources control (bottom-left). This decomposition is obtained by partitioning the executable program and its environment into:

- *The execution architecture*: A hardware platform, a middle-ware library, a real-time operating system, a virtual machine (e.g. in Java), a simulation kernel (e.g. in SystemC). The execution

---

[5]It is common sense to restrict ourselves to programs where all objects are first created and initialized to elaborate the application architecture. Then, threads implement reactions to inputs in the nominal phase of execution and do not allocate any new object (to comply with certification requirement in software design or simply with common sense in SoC design).
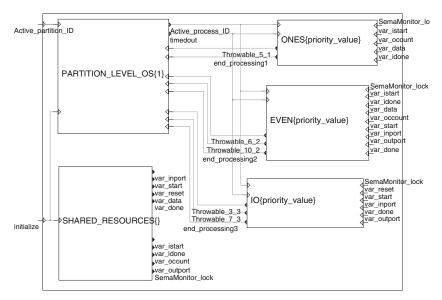
Fig. 1.   From a C-like program to its intermediate representation.

architecture describes an Application Programming interface (API) of generic process and communication management services.

– *The application architecture*: A program, starting from the `main()` procedure, which initializes and links objects to form a hierarchical structure of shared data and communicating threads. The application mapping constructively describes the architecture of the system.

– *The application functionalities*: A set of program threads which periodically or sporadically react to inputs from the environment by interacting with each other for the access to shared data.

Our methodology consists of considering the three elements of an embedded system (its execution and application architectures, its application functionalities) in specific ways.

– *Modeling*: The execution architecture, viewed through an API of generic services, is modeled by template propositions. For instance, the procedure for thread creation in an RTOS API corresponds to a template proposition in the RTOS model whose parameters are the number of threads supported by the application scheduler, the period and deadline of the thread (for a real-time thread), etc.

– *Analysis*: The application architecture, viewed as a hierarchical structure, is interpreted to elaborate a model by the instantiation of

generic API services to the parameters and initial values provided in the program (e.g. thread parameters).

– *Translation*: Each thread consists of a sequential program that describes a functionality to be periodically or sporadically executed by the scheduler and corresponds to a particular model.

This allows for a complete separation of the virtual (threading or functional) architecture of an application from its actual, real-time and resource-constrained implementation: it provides an implementation of the "*write once, run anywhere*" slogan in embedded system design.

Context, Our methodology arises from previous work on real-time operating systems modeling, embedded systems modeling and verification in the Polychrony workbench,[6] a tool-set for embedded system design based on a multi-clocked synchronous model of computation and implemented by the data-flow notation Signal.[2] In Ref. 3, the authors describe the implementation of a real-time operating system standard for avionics application: ARINC.[3] The commercial implementation of this library, RT-Builder from TNI-Valiosys, is used for industrial-scale embedded software engineering project in avionics.

In Ref. 1, this model is used to describe key services of the real-time Java virutal machine. It is applied to rethreading multi-threaded real-time Java programs by global optimization. In Ref. 4, the application of our methodology to system-level design is further developped by studying its application to checking behavioral conformance between embedded systems described in SpecC and at heterogeneous levels of abstraction. In Ref. 5, a generic translation scheme of SystemC programs to the Polychrony workbench is described by considering a static single assignement intermediate representation due to the GCC project.[6] It is applied to design checking (e.g. race and lock detection). In Ref. 7, it is applied to modular verification by model checking and component-wise model abstraction.

We set our methodological framework within the general paradigm of a behavioral type system that associates meaning to software functionalities. The type system is cast in the generic programming language-oriented context of the three-address code (TAC) and static single assignment (SSA) intermediate representations (IR) of GCC.

2. RATIONALE

To allow for an easy grasp on the type system proposed for modeling behaviors, we outline the analysis of an imperative program, Fig. 2,

---

[6]URL: `http://www.irisa.fr/espresso/Polychrony`

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

Fig. 2.   From a C-like program to its intermediate representation.

```
L2:T1 = idata;          L2⇒T1  := idata
   T0 = T1 == 0;           T0  := (T1 = 0)
   if T0 then goto L3;     T0  ⇒L3′
   T2 = ocount;           ¬T0⇒T2 := ocount
   T3 = T1 & 1;               T3 := T1&1
   ocount = T2 + T3;          ocount′ := T2 + T3
   idata = T1 >> 1;           idata′ := T1 >> 1
   goto L2;                   L2′
```

Fig. 3.   From a generic intermediate representation to propositions.

and depict the construction of its type, Fig. 3. Figure 2 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 IR of the program (Fig. 2, second column), all variables (idata and ocount) are read and written once per cycle.

This IR can equally be one of the TAC and SSA formats of GCC. Label L2 is the entry point of the 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.

A behavioral type system. The meaning of this C program fragment is given in a minimalist formalism akin to Pnueli's synchronous transition systems.[8] It not only describes a behavior of the program suitable for its formal verification but also allows for global model transformations to be performed on it. Let us zoom on the block L2 in the example of Fig. 3.

The behavioral type of the block L2, middle, consists of the simultaneous composition of logical propositions that form a synchronous

transition system. Each proposition is associated with one instruction: it specifies its *invariants*: 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 T1 := idata to the proposition L2 ⇒ T1 := idata. The variable 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. All propositions are conditioned by L2 to mean that they hold when block L2 is executed. The extent of a proposition is the duration of a reaction.

A reaction can be an arbitrarily long yet finite period of time provided 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 L3′ as the next value of the state variable L3, to indicate that it will be active during the next reaction. Hence, the proposition L2 ⇒ T0 ⇒ 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".

3. A BEHAVIORAL TYPE SYSTEM

The central element of the type system is a process. It consists of simultaneous propositions that manipulate signals. A signal is an infinite flow of values that is sampled by a discrete series of reactions. This series is called a clock. An event corresponds to the value carried by a signal during a reaction. The formal syntax of propositions in the behavioral type system is defined by the inductive grammar P. A proposition or process P manipulates boolean values noted $v \in \{\, \texttt{false}, \texttt{true} \,\}$ 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.

$$\text{(reference)} \quad r ::= x \mid v \qquad \text{(location)} \quad l ::= x^0 \mid x \mid x'.$$

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 $\hat{x} =^{\text{def}} (x = x)$, which stands for "$x$ is present"; the clock $x =^{\text{def}} (x = \texttt{true})$ for "x is true",

and the clock $\neg x \overset{\text{def}}{=} (x = \texttt{false})$ 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.

$$\text{(clock)} \quad e, f ::= 0 \,|\, x = r \,|\, e \wedge f \,|\, e \vee f \,|\, e \setminus f \,|\, 1$$

A process P consists of the simultaneous composition of elementary propositions. The 1 is the process that does nothing. The proposition $l = r$ means that "l holds the value r". 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.

$$\text{(process)} \quad P, Q ::= 1 \,|\, l = r \,|\, x \to l \,|\, e \Rightarrow P \,|\, (P \,|\, Q) \,|\, P/x.$$

An order of execution is imposed to a proposition by a scheduling constraint, noted $x \to l$, to mean that "l cannot happen before x". Consequently, a proposition, e.g. $x = y$, is seen as the abstraction of an assignment, written $x := y$, defined by $x = y \,|\, y \to x$.

3.1. A Synchronous Model of Computation

The meaning of our notation is given in the synchronous model of computation of Ref. 9. We consider a partially ordered set $(\mathscr{T}, \leqslant, 0)$ of tags. A tag $t \in \mathscr{T}$ denotes a symbolic period in time. The relation $\leqslant$ denotes a partial order and its minimum is noted 0. We note $C \in \mathscr{C}$ a *chain* of tags (a totally ordered subset of $\mathscr{T}$). We define an *event* $e \in \mathscr{T} \times \mathscr{V}$ by the pair of a value and a tag, a *signal* $s \in \mathscr{S} = \{C \to \mathscr{V} \,|\, C \in \mathscr{C}\}$ by a function from a *chain* of tags C to values, a *behavior* $b \in \mathscr{B} = \mathscr{X} \to \mathscr{S}$ by a finite map from signal names $\mathscr{X}$ to signals $\mathscr{S}$, a *process* $p \in \mathscr{P}$ by a set of behaviors of same domain. We write tags(s) for the tags of a signal s, $b|_X$ for the projection of a behavior b on $X \subset \mathscr{X}$ and $b/X = b|_{\text{vars}(b) \setminus X}$ for its complementary, vars(b) and vars(p) for the domains of b and p.

Example 1. Figure 4 depicts a behavior b over three signals named x, y and z. Two frames depict timing domains formalized by chains of tags. Signal x and y belong to the same timing domain: x is a down-sampling of y. Its events are synchronous to odd occurrences of events along y and share the same tags, e.g., t_1. Even tags of y, e.g., t_2, are ordered

along its chain, e.g., $t_1 < t_2$, but absent from x (we write $t < t'$ if $t \leqslant t'$ and $t' \nleqslant t$). Signal z belongs to a different timing domain. Its tags, e.g., t_3 are not ordered with respect to the chain of y, e.g., $t_1 \nleqslant t_3$ and $t_3 \nleqslant t_1$.

Scheduling structure. To schedule the occurrence of events during a period or an instant t, we consider the fact that the pair x_t of a time tag t and of a signal name x renders its very date d. The tag t represents the period during which the event takes place and the signal x its location. This consideration defines scheduling $\rightarrow$ by a pre-order relation between dates. Figure 5 depicts such a relation superimposed to the signals x and y of Fig. 4. The relation $y_{t_1} \rightarrow x_{t_1}$, for instance, requires y to be calculated before x at the period t_1. A scheduling relation naturally satisfies containment with respect to the timing partial order $\leqslant$ of every signal x in a behavior b, in that for all $t, t' \in \text{tags}(b(x))$, $t < t'$ naturally implies $x_t \rightarrow^b x_{t'}$ and, conversaly $x_t \rightarrow^b x_{t'}$ implies $t' \nless t$. A scheduling relation is implicitly transitive ($x_t \rightarrow^b y_{t'} \rightarrow^b z_{t''}$ implies $x_t \rightarrow^b z_{t''}$) and its closure for restriction b/X is defined by $x_t \rightarrow^{b/X} y_{t'}$ iff $x_t \rightarrow^b y_{t'}$ and $x, y \notin X$.

Synchronous composition is noted $p \,|\, q$ and defined by the union of all behaviors b (from p) and c (from q) which are synchronous. All signals x shared by b and c belong to $I = \text{vars}(p) \cap \text{vars}(q)$ and are equal, i.e., $b|_I = c|_I$: $p \,|\, q = \{b \cup c \,|\, (b, c) \in p \times q, I = \text{vars}(p) \cap \text{vars}(q), b|_I = c|_I \}$. (Fig. 6)

3.2. Meaning of Clocks

The denotation $[\![e]\!]_b$ of a clock expression e (Table 7) 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 Fig. 7, 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 $b(x)(t) =$

Fig. 4.   Behavior b over three signals x, y and z in two clock domains.

Fig. 5.   Scheduling relations between simultaneous events.

$$\begin{bmatrix} x: & \bullet^{t_1} & & \bullet \\ y: & \bullet^{t_1} & \bullet^{t_2} & \bullet \\ & & & \end{bmatrix} \;\Big|\; \begin{bmatrix} & & & \\ y: & \bullet^{t_1} & \bullet^{t_2} & \bullet \\ z: & & \bullet^{t_3} & \bullet \end{bmatrix} = \begin{bmatrix} x: & \bullet^{t_1} & & \bullet \\ y: & \bullet^{t_1} & \bullet^{t_2} & \bullet \\ z: & & \bullet^{t_3} & \bullet \end{bmatrix}$$

Fig. 6.   Synchronous composition of $b \in p$ and $c \in q$.

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

Fig. 7.   Denotational semantics of clocks.

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.

3.3. Meaning of Propositions

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). The meaning of an initialization $[\![x^0 = v]\!]^e$ consists of all behaviors defined on x, written $b \in \mathscr{B}|_x$ 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. We write $\mathscr{B}|_X$ 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 C and $\text{vars}(P)$ and $\text{vars}(e)$ for the sets of signals 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 $y_t \to^b x_t$. The propositions $x' = y$ and $y \to x'$ is 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

$$[\![x = y]\!]^e = \{b \in \mathscr{B}|_{\mathrm{vars}(e) \cup \{x,y\}} \,|\, \forall t \in [\![e]\!]_b,$$
$$t \in \mathrm{tags}(b(x)) \wedge t \in \mathrm{tags}(b(y)) \wedge b(x)(t) = b(y)(t)\}$$

$$[\![y \to x]\!]^e = \{b \in \mathscr{B}|_{\mathrm{vars}(e) \cup \{x,y\}} \,|\, \forall t \in [\![e]\!]_b,$$
$$t \in \mathrm{tags}(b(x)) \Rightarrow t \in \mathrm{tags}(b(y)) \wedge y_t \to^b x_t\}$$

$$[\![x' = y]\!]^e = \{b \in \mathscr{B}|_{\mathrm{vars}(e) \cup \{x,y\}} \,|\, \forall t \in [\![e]\!]_b,$$
$$t \in C = \mathrm{tags}(b(x)) \wedge t \in \mathrm{tags}(b(y)) \wedge b(x)(\mathrm{succ}_t(C)) = b(y)(t)\}$$

$$[\![y \to x']\!]^e = \{b \in \mathscr{B}|_{\mathrm{vars}(e) \cup \{x,y\}} \,|\, \forall t \in [\![e]\!]_b,$$
$$t \in C = \mathrm{tags}(b(x)) \Rightarrow t \in \mathrm{tags}(b(y)) \wedge y_t \to^b x_{\mathrm{succ}_t(C)}\}$$

$$[\![x^0 = v]\!]^e = \{b \in \mathscr{B}|_x \,|\, b(x)(\mathrm{min}(\mathrm{tags}(b(x)))) = v\}$$

$$[\![f \Rightarrow P]\!]^e = [\![P]\!]^{e \wedge f} \quad [\![P \,|\, Q]\!]^e = [\![P]\!]^e \,|\, [\![Q]\!]^e \quad [\![P/x]\!]^e = [\![P]\!]^e/x$$

Fig. 8.   Denotational semantics of propositions.

of P at the clock $e \wedge f$. The behavior of $P \,|\, Q$ consists the synchronous composition of the behaviors of P and Q (Fig. 8).

4. AN INTERMEDIATE REPRESENTATION

We are now equipped with the required mathematical framework to address the modeling of embedded systems described by communicating program threads. This model is described in terms of a type inference system and extended to the structuring elements of a generic 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.

Formal syntax. Imperative programs are represented in an intermediate form that is common to the TAC and SSA IRs of GCC which provides language-independence and local optimization. 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*.

Block instructions consist of native method invocations $x = f_{(1...n)}$, lock monitoring and branches if x then L. Blocks are returned from by either a goto L, a return or an exception throw x. The declaration catch x from L_1 to L_2 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 (Fig. 9).

$$
\begin{array}{llll}
\text{(program)} & pgm ::= L{:}blk \mid pgm; pgm \\
\text{(block)} & blk ::= stm; blk \mid rtn \\
\text{(instruction)} & stm ::= x = f(y_{1..n}) \\
& \qquad \mid \quad \text{if } x \text{ then } L \\
\text{(return)} & rtn ::= \text{goto } L \\
& \qquad \mid \quad \text{return } x \\
& \qquad \mid \quad \text{throw } x \\
& \qquad \mid \quad \text{catch } x \text{ from } L \text{ to } L \text{ using } L
\end{array}
$$

Fig. 9. Syntax for an intermediate representation of imperative programs.

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

Fig. 10. From three address code.

In the remainder, we only assume that a block always starts with a label and finishes with a return statement: $stm_1; L{:}stm_2$ is rewritten as $stm_1;$ goto $L;$ $L{:}stm_2$. A call $x = f(y)$ to a possibly blocking external method f, such as wait x in SystemC or Java, is always placed at the beginning of a block L. For instance, $stm_1;$ wait $x; stm_2$ is rewritten as $stm_1;$ goto $L;$ $L{:}$wait $v; stm_2$. By contrast, primitive operations $x = f(y, z)$ are assumed to take an insignificant amount of time and are executed with the normal control-flow of the block.

Example 2. To outline the construction of the intermediate representation of a program, let us reconsider the example of Section 2 and detail the function that counts the number of bits set to 1 in a bit-array data (Fig. 10).

It consists of three blocks. The block labeled L1 waits for the signal lock before initializing the local state variable idata to the value of the input signal 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 count and lock is unlocked before going back to L1.

The SSA form of the program differs in the function-wise guarantee that all variable be assigned once during an execution cycle. It consists of performing assignments to idata and ocount in blocks L1 and L2 to

```
L1:...idata1=data;  | L2:...ocount2 = T2 + T3;  | L4:idata=φ?idata1,idata2;
     ocount1=0;     |      idata2 = T1 >> 1;     |    ocount=φ?ocount1,ocount2;
     goto L4;       |      goto L4;              |    goto L2;
```

Fig. 11. ... to static single assignment.

$$\langle\!\langle x = f(x_{1..k})\rangle\!\rangle_{L_1 L_2}^E = E(f)(x_{1..k} x L_1 L_2)$$

$$\langle\!\langle \mathtt{if}\, x\, \mathtt{then}\, L_1\rangle\!\rangle_{L_2 L_3}^E = \{b \in \mathscr{B}|_{x L_{123}} \,|\, \forall t \in C = \mathrm{chain}_b(x L_{123}),$$
$$b(L_1)(t) \Rightarrow (b(L_3)(t) = \neg b(x)(t)$$
$$b(x)(t) \Rightarrow b(L_2)(\mathrm{succ}_t(C)))\}$$

$$\langle\!\langle \mathtt{return}\, x\rangle\!\rangle_L^E = \{b \in \mathscr{B}|_{Lxy}|E(\mathtt{return}) = y, \forall t \in C = \mathrm{chain}_b(Lxy),$$
$$b(L)(t) \Rightarrow b(y)(t) = b(x)(t)\}$$

$$\langle\!\langle \mathtt{goto}\, L_1\rangle\!\rangle_{L_2}^E = \{b \in \mathscr{B}|_{L_1 L_2} \,|\, \forall t \in C = \mathrm{chain}_b(L_1 L_2),$$
$$b(L_2)(t) \Rightarrow b(L_1)(\mathrm{succ}_t(C))\}$$

$$\langle\!\langle \mathtt{throw}\, x\rangle\!\rangle_L^E = \{b \in \mathscr{B}|_{Lx}|\forall t \in C = \mathrm{chain}_b(Lx),$$
$$b(L)(t) \Rightarrow b(x)(t)\}$$

$$\langle\!\langle stm; blk\rangle\!\rangle_{L_1}^E = (\langle\!\langle stm\rangle\!\rangle_{L_1 L_2}^E \,|\, \langle\!\langle blk\rangle\!\rangle_{L_2}^E)/L_2$$

$$\langle\!\langle L : blk; pgm\rangle\!\rangle^E = \langle\!\langle blk\rangle\!\rangle_L^E \,|\, \langle\!\langle pgm\rangle\!\rangle^E$$

$$\langle\!\langle m\, f(x_{1...k})\, \{pgm\}\rangle\!\rangle^E = E[f : \lambda x_{1..k} xyy^{exit}.(p/L_{1..j})] \,|$$
$$p = \langle\!\langle pgm\rangle\!\rangle^{E[\mathtt{return}:y]} \wedge \mathrm{labs}(pgm) = L_{1..j}$$

Fig. 12. Denotational semantics of instructions.

temporary variables and branch to a merge block L4 where the appropriate copy is assigned to the variable upon the value of a boolean condition ϕ (to mean from L1 or not) (Fig. 11).

Meaning of instructions. The denotation of instructions for programs which strictly adhere either of the TAC or SSA requirements (i.e., all variables are written at most once per block) is given in Fig. 12. To ligthen notations, we write $C = \mathrm{chain}_b(X)$ iff for all $x \in X$, $C = \mathrm{tags}(b(x))$ and write $b(x)(t)$ for $b(x)(t) = \mathtt{true}$ and $\neg b(x)(t)$ for $b(x)(t) = \mathtt{false}$. The denotation of a program $\langle\!\langle pgm\rangle\!\rangle^E$ takes an environment giving the meaning of external functions f using call-by-name λ-expressions and returns the set of behaviors b corresponding to the execution of pgm.

For an instruction stm, the function $\langle\!\langle stm\rangle\!\rangle_{L_1 L_2}^E$ takes two labels which represent the entry label L_1 of the statement and its continuation by the pseudo-label L_2. The denotation of a function call $x = f(x_{1..k})$ is that given by E for the variable names $x_{1...k} x$ and the entry and exit labels L_1 and L_2.

The meaning of an $\mathtt{if}\, x\, \mathtt{then}\, L_1$ instruction consists of all behaviors b defined on x, L_1, L_2 and L_3 which share the same chain of tags C and such that, if $b(L_1)(t)$ is true, then the continuation label L_3 is active iff x

if false, i.e., $b(L_3)(t) = \neg b(x)(t)$; and if x is true then L_2 is active next, i.e., $b(x)(t)$ true implies $b(L_2)(\text{succ}_t(C))$ true. For a return instruction rtn, the denotation function $\langle\langle rtn \rangle\rangle_L^E$ only takes one (entry) label L. The meaning of $\texttt{return}\,x$, $\texttt{goto}\,L$, and $\texttt{throw}\,x$ instructions are given using the same principle as for the $\texttt{if}\,x\,\texttt{then}\,L$.

Notice the introduction of a pseudo-label to handle a sequence of instructions. The meaning of a sequence $stm; blk$ starting at block L_1 is defined by using a local peudo-label L_2 to denote the continuation of stm by $\langle\langle stm \rangle\rangle_{L_1 L_2}^E$ and hence the entry point of blk by $\langle\langle blk \rangle\rangle_{L_2}^E$. The meaning of the sequence is finalized by synchronous composition and the scope of L_2 restricted to it. The meaning of a program $L : blk; pgm$ is similar yet simpler as there is no continuation between blocks. The meaning of a function declaration $m\ f(x_{1...k})\ \{pgm\}$ is listed just to show the order in which the argument, result, entry and exit label names are used to parameterize the meaning of the function body.

5. BEHAVIORAL TYPE INFERENCE

The behavioral type inference system is defined by induction on the formal syntax of programs pgm. To define it, we assume that the finite set $\mathscr{L}$ of program labels L. To each block of label L, the inference system associates a boolean proposition L of the same name, called the *input clock*, and a boolean proposition L^{exit}, called its *output clock*. The proposition L is true iff the block L is active during a given transition. The proposition L^{exit} is true iff the execution of block L terminates during a given transition. The relation defined by the behavioral type system has the form:

$$\boxed{e_0, \mathscr{E} \vdash L : blk : \langle P, e_1 \rangle},$$

where e_0 denotes the input clock of the block of instructions blk, L is its label, P the proposition to denote its behavior, and e_1 its output or continuation clock. The type environment $\mathscr{E}$ gives the behavior of methods and functions defined in the context of the program. It associates a variable x to a type m (a class name), a class name m to a class type $\mathscr{T}$ (described in the next section), and a method f to a proposition P and an output clock e parameterized by the sequence $x_{1...n}$ formed of its input and output variables and input clock name (see rule (8) below).

$$\mathscr{E} ::= []\,|\,\mathscr{E}[x : m]\,|\,\mathscr{E}[m : \mathscr{T}]\,|\,\mathscr{E}[f : \lambda(_{1...n}).\langle P, e \rangle].$$

Rules (1–8) define the behavioral type inference system. Rules (1 and 2) are concerned with the iterative decomposition of a program pgm into

blocks *blk* and with the decomposition of a block into statements *stm* and return instruction *rtn*.

$$\frac{L, \mathscr{E} \vdash L : blk : P \quad \mathscr{E} \vdash pgm : Q}{\mathscr{E} \vdash L : blk; pgm : P \mid Q}. \tag{1}$$

Notice that, 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.

$$\frac{e_1, \mathscr{E} \vdash L : stm : P, e_2 \quad e_2, \mathscr{E} \vdash L : blk : Q}{e_1, \mathscr{E} \vdash L : stm; blk : P \mid Q}. \tag{2}$$

This is the case, rule (3), for instruction $\texttt{if } x \texttt{ then } L_1$. Let e be the input clock of the instruction. If x is false then control is passed to the continuation of this instruction in the block, at the output clock $e \wedge \neg x$. Otherwise, control is passed to block L_1, at the clock $e \wedge x$. Hence the type $(e \wedge x) \Rightarrow L_2'$ to mean that the next value of L_2 is true when e is active and when x is true.

$$e, \mathscr{E} \vdash L : \texttt{if } x \texttt{ then } L_1 : \langle (e \wedge x) \Rightarrow (L^{\text{exit}} \mid L_1'), e \wedge \neg x \rangle. \tag{3}$$

All return instructions, rules (4–7), define the output clock L^{exit} of the current block L by the input clock e. This is the right place to do that: e defines the condition upon which the block actually reaches its return statement. A $\texttt{goto } L_1$ instruction, rule (4), passes control to block L_1 unconditionally at the input clock e.

$$e, \mathscr{E} \vdash L : \texttt{goto } L_1 : e \Rightarrow (L^{\text{exit}} \mid L_1'). \tag{4}$$

A $\texttt{return}$ instruction, rule (5), fetches the variable y used as return variable for the current method or function and sets y^{exit} to true at clock e in order to notify the caller that the method terminates execution.

$$\frac{\mathscr{E}(\texttt{return}) = y}{e, \mathscr{E} \vdash L : \texttt{return } x : e \Rightarrow (L^{\text{exit}} \mid y^{\text{exit}} \mid y := x)}. \tag{5}$$

A $\texttt{throw } x$ instruction, rule (6), produces an event along the signal x at the input clock e by $e \Rightarrow \hat{x}$.

$$e, \mathscr{E} \vdash L : \texttt{throw } x : e \Rightarrow (L^{\text{exit}} \mid \hat{x}). \tag{6}$$

L2:...if T0 then goto L3;   L2⇒ ...T0 ⇒L3′
$\vdots$ ¬T0⇒...L2′
       goto L2;

Fig. 13.   Modeling control flow in an imperative program.

Example 3.   Let us zoom on the block L2 of Fig. 3. On the first line, for instance, we associate the instruction T1 = idata of block label L2 to the proposition L2 ⇒ T1 = idata. In this proposition, the variable L2 is a boolean that is true iff the block L2 is being executed. So, the proposition says that, if L2 is being executed, then T1 is always equal to idata. If it not, another proposition may hold. All subsequent propositions are conditioned by L2 to mean that they hold when L2 is executed. Next, consider the instruction if T0 then L3. Its invariant L2 ⇒ T0 ⇒ L3′ says that control passes to L3 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" (Fig. 13).

The catch statement $\mathtt{catch}\,x\,\mathtt{from}\,L\,\mathtt{to}\,L_1\,\mathtt{using}\,L_2$ matching rule (6), passes control in rule (7) to the exception handler L_2 and then to the block L_1 upon termination of L_2 notified by L_2^{exit}. This requires, first, to activate L_2 from L when x is present and then to pass the control to L_1 upon termination of the handler.

$$e, \mathscr{E} \vdash L : \mathtt{catch}\,x\,\mathtt{to}\,L_1\,\mathtt{using}\,L_2 : (\hat{x} \wedge L^{\mathrm{exit}}) \Rightarrow L_2' \mid L_2^{\mathrm{exit}} \Rightarrow L_1'. \quad (7)$$

Rule (8) is concerned with type assignement for native and external method invocations $x = f(x_{1...k})$. The generic type of f is taken from an environment $\mathscr{E}(f)$. It is given the name of the actual parameters $x_{1...k}$, of the result x and of the input clock e. $\mathscr{E}(f)(x_{1...k}xe)$ yields the corresponding behavioral type $\langle P, e_1 \rangle$.

$$e, \mathscr{E} \vdash L : x = f(x_{1...k}) : \mathscr{E}(f)(x_{1...k}, x, e). \quad (8)$$

Example 4.   As an example, the wait-notify protocol used in SystemC of Java to arbiter access to shared data is modeled using a boolean flip-flop variable x. The notify method defines the next value of the lock x by the negation of its current value at the input clock e. The wait method continues activates iff the value of the lock x has changed at the input clock L: $L \wedge (x \neq x')$. Otherwise, at the clock $L \wedge (x = x')$, the control

is passed to L by a delayed transition $e \setminus \hat{y} \Rightarrow L'$.

$$\mathcal{E}(\texttt{notify}) = \lambda x e.\langle e \Rightarrow (x' = \neg x), e \rangle,$$
$$\mathcal{E}(\texttt{wait}) = \lambda x L.\langle L \wedge (x = x') \Rightarrow L', L \wedge (x \neq x') \rangle.$$

Consider the wait-notify protocol at blocks L1 and L3, Fig. 14. The wait instruction continues if L1 receives control and if the lock is toggled (proposition $\mathsf{lock} \neq \mathsf{lock}'$). If so, the block is executed and control passes to the block L2 and, if not, to the block L1.

Completion: By definition, a proposition L holds the value `true` iff the block L is active during execution. Otherwise, L should be `false`. This default value requires a completion of the next-state logic for the type P of a given program *pgm*. We write $\overline{P}$ this completion. It is simply defined by considering the proposition $e_L \Rightarrow L'$ implied by the type P for all labels L of a given program *pgm*. The clock e_L is defined by the union (disjunction) of all clocks $e \Rightarrow L'$ present in P. The default rule is defined by $\hat{L} \setminus e_L \Rightarrow \neg L'$. The same holds for output clocks L^{exit}.

Correspondence: The correspondance between instructions and propositions defined through our type system $\mathcal{E} \vdash pgm : P$ can now be formally established by stating Property 1. We write $[\![\mathcal{E}]\!]$ for the interpretation of the environment $\mathcal{E}$ defined by

$$[\![\mathcal{E}[f : \lambda(x_{1\ldots k}xL).\langle P, x^{\text{exit}} \rangle]]\!] = [\![\mathcal{E}]\!][f : \lambda(x_{1\ldots k}xLx^{\text{exit}}).[\![P]\!]].$$

Property 1 established a classical soundness property by stating that whenever *pgm* has type P and the typing environment $\mathcal{E}$ has meaning E then b is a behavior of P (guarded by 1 to mean always) if and only if it is a behavior of *pgm* with the environment E. Notice that the top-level environment $\mathcal{E}$ defines the model of the runtime communication and processes management API for the application program *pgm*. The proof of Property 1 consists of showing that both implications $[\![P]\!] \subseteq \langle\!\langle pgm \rangle\!\rangle$ and $[\![P]\!] \supseteq \langle\!\langle pgm \rangle\!\rangle$ hold by induction on the structure of *pgm* ending up in a case analysis on the correspondence between each instruction.

Property 1.
If $\mathcal{E} \vdash pgm : P$ and $E = [\![\mathcal{E}]\!]$ then $b \in [\![P]\!]_1$ iff $b \in \langle\!\langle pgm \rangle\!\rangle^E$.

L1:wait (lock);	$L1 \wedge (\mathsf{lock} = \mathsf{lock}') \Rightarrow L1'$	L3:notify (lock);	$L3 \Rightarrow \mathsf{lock}' = \neg\mathsf{lock}$
$\vdots$		$\vdots$	
goto L2;	$L1 \wedge (\mathsf{lock} \neq \mathsf{lock}') \Rightarrow L2'$	goto L1;	L1'

Fig. 14. Modeling the access to locks.

From TAC to SSA. The type system and its semantics rely on the property of the TAC IR that every variable is defined at once within a block (this hypothesis is sound for a program in SSA form as well). As a consequence, each block delimits an atomic reaction in the type system and, therefore, transition from a block to another cannot be immediate (by saying L for "label L is active") but delayed (by saying L' for "label L will be active next time"). In SSA, this guarantee is provided for the whole "text" of the function. In particular, for a goto from a block L_1 to a block L_2 textually after L_1 (written $L_1 < L_2$), SSA guarantees that all variables defined in L_1 are different from those in L_2. This is of course not the case for a loop, in which case we have $L_1 \geqslant L_2$. To take advantage of this additional guarantee, our type inference system can be refined by considering the following rule to handle *goto*s (and similarly, *if-then*s and *throw-catch*s). It consists of activating the target block L_2 immediately.

$$(4b) \frac{L_1 < L_2}{e, \mathscr{E} \vdash L_1 : \mathtt{goto}\ L_2 : e \Rightarrow (L_1{}^{exit} \,|\, L_2)}.$$

The translation of the EPC in SSA form using rule (3b) outlines the benefits of this optimization. The resulting type has strictly fewer delayed transitions: one to L2 in L3 and another to L1 in L4. All other transitions are immediate and considered within the same reaction.

6. CONFORMANCE CHECKING

Just as the multi-clocked synchronous formalism Signal it is based upon, our type system allows for the refinement-based design methodologies considered in Ref. 7 to be easily implemented. Checking the correct refinement of an initial module, of type P, by its upgrade, of type Q, amounts to checking that the final guarantee Q satisfies the initial assumptions P. In Ref. 7, this is implemented by compositionally model checking that Q is finitely flow-equivalent to P (Fig. 15).

Figure 16 describes a typical case study of conformance checking. We consider the refinement of the C model of an even parity checker (EPC) from a high-level design abstraction, left, where communication is abstracted by shared variables and a lock, to an architecture-level design abstraction, right, where the communication medium is refined by the insertion of a channel implementing a double handshake protocol, Fig. 17.

Checking conformance of the architecture-level design with respect to its system-level abstraction amounts to checking that both designs are flow equivalent. The very notion of flow equivalence under consideration

```
L1:wait (lock);                      L1⇒lock=lock' ⇒ L1'
    idata1=data;                          lock≠lock' ⇒ idata1:=data
    ocount1=0;                                            ocount1:=0
    goto L3;                                              L3
L2:T1 = idata;                       L2⇒T1 := idata
    T0 = T1 == 0;                        T0 := T1 == 0
    if T0 then goto L3;                  T0 ⇒ L4
    T2 = ocount;                        ¬T0 ⇒ T2 := ocount
    T3 = T1 & 1;                               T3 := T1 & 1
    ocount2 = T2 + T3;                         ocount2 := T2 + T3
    idata2 = T1 >> 1;                          idata2 := T1 >> 1
    goto L3;                                   L3
L3:idata=φ?idata1,idata2;            L3⇒L1⇒idata:=idata1
    ocount=φ?ocount1,ocount2;                 ocount:=ocount1
    goto L2;                            L2⇒ idata:=idata2
L4:notify (lock);                             ocount:=ocount2
    count = ocount;                    L2'
    goto L1;                         L4⇒lock':=¬ lock
                                          count := ocount
                                          L1'
```

Fig. 15.   Model of the even-parity checker in SSA form.
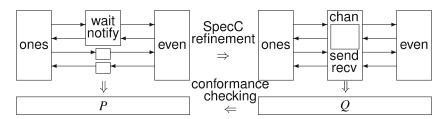

Fig. 16.   Conformance-checking the refinment of an EPC.
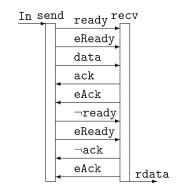

Fig. 17.   Refinement of locks with a double handshake protocol.

consists is defined in the asynchronous structure of our model of computation that is presented next.

6.1. Asynchronous Structure

The asynchronous structure of polychrony is modeled by weakening the clock-equivalence relation to allow for comparing behaviors whose successive values match regardless of time: two behaviors are flow-equivalent iff their signals hold the same values in the same order. The *relaxation* relation allows to individually stretch the signals of a behavior in a way preserving scheduling constraints. A behavior c is a *relaxation* of b, written $b \sqsubseteq c$, iff $\text{vars}(b) = \text{vars}(c)$ and, for all $x \in \text{vars}(b)$, $b|_{\{x\}} \leqslant c|_{\{x\}}$.

Relaxation is a partial-order relation which defines flow-equivalence: b and c are *flow-equivalent*, written $b \approx c$, iff there exists a behavior d s.t. $d \sqsubseteq b$ and $d \sqsubseteq c$. Figure 18 illustrates two asynchronously equivalent behaviors related by relaxation. The first event along x has been shifted (and its scheduling constraint with an initially synchronous event along y lost) as the effect of finitely delaying its transmission.

Asynchronous composition is noted $p \parallel q$ and defined using the partial-order structure induced by the relaxation relation. The composition of p and q consists of behaviors d that are relaxations of behaviors b and c from p and q along shared signals $I = \text{vars}(p) \cap \text{vars}(q)$, i.e., $b|_I \sqsubseteq d|_I \sqsupseteq c|_I$, and that are stretching of b and c along the independent signals of p and q, i.e., $b/I \leqslant d/I \geqslant c/I$.

Figure 19 illustrates the asynchronous composition of a behavior b and of a behavior c. Signals x and y are alternated in b, left, and synchronous in c, middle. Asynchronous composition allows x and y to be independently stretched in b and c in order to find a common flow in the asynchronous composition, right.

Fig. 18.   Relating asynchronous behaviors by relaxation.

Fig. 19.   Asynchronous composition.

Fig. 20.   Relation between events through a one place buffer along x.

6.2. Flow Preservation

To check the existence of a flow-preserving timing relation between the two systems outlined in the previous section, the refinement-based methodology similar of Ref. 4 shows that the types P and Q of Fig. 16 are finitely flow-equivalent. To this end, we formulate the timing deformation allowed by finite buffering protocols starting from the model of a one-place FIFO buffer which we will use to draw the spectrum of possible timing relations under consideration. Figure 20 depicts the timing deformation allowed along a signal x by a one place buffer.

Finite relaxation. Definition 1 formalizes this relation by considering the timing deformation between an initial behavior b and a final behavior c performed by a one-place FIFO buffer of internal signal m and behavior d. The behavior d is defined by stretching $b \leqslant d/m$ and c/x by d/mx. Let us write $\text{pred}_C(t)$ (resp. $\text{succ}_C(t)$) for the immediate predecessor (resp. successor) of the tag t in the chain C.

Definition 1 (finite relaxation). The behavior c is a 1-relaxation of x in b, written $b \sqsubseteq^x_1 c$ iff $\text{vars}(b) = \text{vars}(c)$ and there exists a signal m, a behavior d and a chain $C = \text{tags}(d(m)) = \text{tags}(d(x)) \cup \text{tags}(c(x))$ such that $d/m \geqslant b$, $d/mx = c/x$ and, for all $t \in C$,

(1) $t \in \text{tags}(d(x)) \Rightarrow d(x)(t) = d(m)(t) \wedge x_t \to_d m_t$;
(2) $t \notin \text{tags}(d(x)) \Rightarrow d(m)(t) = d(m)(\text{pred}_C(t))$;
(3) $t \in \text{tags}(c(x)) \Rightarrow c(x)(t) = d(m)(t) \wedge \forall y \in \text{vars}(d) \setminus m, \, y_t \to_d x_t$;
(4) $t \in \text{tags}(c(x)) \Rightarrow c(x)(t) = d(x)(t) \vee c(x)(\text{succ}_C(t)) = d(x)(t)$.

For all $t \in C$, rule (1) says that, when an input $d(x)$ is present at some time t, then $d(m)$ takes its value. If no input is present along x at t, rule (2), then $d(m)$ takes its previous value. Rule (3) says that, if the output $c(x)$ is present at t, then it is defined by $d(m)(t)$. Finally, rule (4) requires this value to either be the present or previous value of the input signal $d(x)$, binding the size of the buffer to one place (Fig. 21).

Definition 1 accounts for the behavior of bounded FIFOs in a way that preserves scheduling relations. It implies 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

$$d(x):$$
$$d(m):$$
$$c(x):$$

Fig. 21. Timing and scheduling relations through finite relaxation.

flow-preserving FIFO buffer of size n. We write $b \sqsubseteq_1 c$ iff $b \sqsubseteq_1^x c$ for all $x \in \text{vars}(b)$, and, for all $n > 0$, $b \sqsubseteq_{n+1} c$ iff there exists d such that $b \sqsubseteq_1 d \sqsubseteq_n c$. The largest equivalence relation modeled in the polychronous model of computation consists of behaviors equal up to a timing deformation performed by a finite FIFO protocol: b and c are *finitely flow-equivalent*, written $b \approx^* c$, iff there exists $n > 0$ and d s.t. $d \sqsubseteq_n b$ and $d \sqsubseteq_n c$.

6.3. A Compositional Methodology

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 2 (finite flow-preservation). P is finitely flow-preserving with $I \subset \text{in}(P)$ iff for all behaviors b, c of $[\![P]\!]$, if $(b|_I) \approx (c|_I)$ then $b/I \approx^* c/I$.

Example of finitely flow-preserving processes are endochronous processes Ref. 9. An endochronous process which receives flow equivalent inputs produces clock-equivalent outputs. It hence forms a restricted subclass of finitely-flow preserving processes. Furthermore, notice that flow-preservation is stable to the introduction of a wrapper of P consisting of a finite FIFO buffering protocol. 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 3 (finite flow-invariance). The transformation of P into Q such that $I \subset \text{in}(P) = \text{in}(Q)$ 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. Let P and Q be two finitely flow-preserving processes and

R a protocol to link P and Q, such as a finite FIFO buffer, or a double hand-shake protocol, or a relay station,[10] or a loosely time-triggered architecture[11]. In Definition 4, we write $b[x/y]$ for the behavior resulting of substitution of the signal name y by the signal name y in the domain of the behavior b and $[x_i/y_i]_{0<i\leqslant n}$ for the compposition of n substitutions.

Definition 4 (flow-preserving protocol). The process R is a flow-preserving protocol iff there exists $n > 0$ such that inputs $\text{in}(R) = \{x_{1...n}\}$ are finitely flow-equivalent to outputs $\text{out}(R) = \{y_{1...n}\}$, i.e., $\forall b \in [\![R]\!]$, $b|_{x_{1...n}} \approx^* (b|_{y_{1...n}}[x_i/y_i]_{0<i\leqslant n})$.

The wrapper $R\langle P\rangle$ of a process P with a protocol R is defined by redirecting the signals of P to R. In Definition 5, this redirection is modeled by substituting signal names: we write $P[x/y]$ for the process resulting of substituting y by x in P.

Definition 5 (wrapper). Let P be a process such that $\text{in}(P) = \{x_{1...m}\}$ and $\text{out}(P) = \{x_{m+1...n}\}$. Let R be a flow-preserving protocol such that $\text{in}(R) = \{y_{1...n}\}$ and $\text{out}(R) = \{z_{1...n}\}$. The wrapper of P with R is the template process noted $R\langle P\rangle$ and defined by:

$$R\langle P\rangle \stackrel{\text{def}}{=} \left(\begin{array}{c} \big(R[x_i/z_i]_{m<i\leqslant n}\big)[x_i/y_i]_{0<i\leqslant m} \\ | \ \big(P[y_i/x_i]_{m<i\leqslant n}\big)[z_i/x_i]_{0<i\leqslant m} \end{array} \right) /y_{1...n}z_{1...n}.$$

A sufficient condition for the insertion of a protocol between two synchronous processes P and Q to finitely preserve flow is to guaranty that $P|_I | Q|_I$ is finitely flow preserving for $I = \text{vars}(P) \cap \text{vars}(Q)$, 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 2 (protocol insertion).
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 ($I = \text{vars}(P) \cap \text{vars}(Q)$).

7. FURTHER APPLICATIONS

We have introduced a type system allowing to model the control and data flow graphs of a given imperative program in intermediate form. Applications of the proposed type system encompass optimization and verification issues encountered in system design.
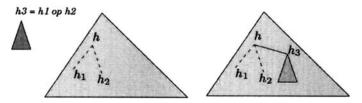
h3 = h1 op h2

Fig. 22. Combination of trees during hierarchization.

7.1. Rethreading

Because our type system entirely model the control and data-flow of application components and architecture functionalities, one can operate global optimization on the whole model of the application. Signal, in particular, implements the notation of our type system using data-flow equations and allows for the generation of sequential code by employing a global control-flow graph transformation called hierarchization.[12] Hierarchization consists of hooking elementary control flow graphs (in the form of if-then-else structures). For instance, Fig. 22, let $h3$ be a clock computed using $h1$ and $h2$ and h be the head of a tree in which $h1$ and $h2$ are computed. Then $h3$ can computed after $h1$ and $h2$ and placed under h.

Example 5. The implications of hierarchization for code generation can be outlined by considering the specification of one-place buffer. The process buffer has input x, output y and implements two functionalities.

$$\text{buffer}\langle x, y \rangle \stackrel{\text{def}}{=} \text{alternate}\langle x, y \rangle \,|\, \text{current} < x, y > .$$

One is the process alternate which desynchronizes the signals x and y by synchronizing them to the true and false values of an alternating boolean signal s.

$$\text{alternate}\langle x, y \rangle \stackrel{\text{def}}{=} \left(s^0 = \text{true} \,|\, \hat{x} = s \,|\, \hat{y} = \neg s \,|\, s' := \neg s \right).$$

The other functionality is the process current. It defines a cell in which values are stored at the input clock $\hat{x}$ and loaded at the output clock $\hat{y}$.

$$\text{current}\langle x, y, b \rangle \stackrel{\text{def}}{=} \left(r^0 = b \,|\, r' := x \,|\, \hat{x} \Rightarrow y := x \,|\, \hat{y} \setminus \hat{x} \Rightarrow y := r \right).$$

```
buffer_iterate () {
  s = !zs;
  cy = !s;
  if (s) { if (!r_buffer_i(&x)) return FALSE; }
  if (cy) { y = x; w_buffer_o(y); }
  zs = s;
  return TRUE; }
```

Fig. 23. C code generated for the one-place buffer specification.

We observe that s defines the master clock of **buffer**. There are two other synchronization classes, x and y, that corresponds to the true and false values of the boolean flip-flop variable s, respectively. This defines three nodes in the control-flow graph of the generated code (Fig. 23). At the master clock $\hat{s}$, the value of s is calculated from zs, its previous value. At the sub-clock $s = \hat{x}$, the input signal x is read. At the sub-clock $\neg s = \hat{y}$ the output signal y is written. Finally, the new value of zs is determined.

Operating this transformation on the model of a multi-threaded application results in merging all threads into a single control-flow graph whose scheduler foot-prints sequentially processes each elementary execution block upon a particular condition. In Ref. 1, we report a 300% average speedup resulting of applying this optimization to real-time Java programs compared to their execution using a commercial compiler.

7.2. Module Checking

In Ref. 5, we define a behavioral module checking algorithm based on similar principles as those exposed in the previous section. This system allows to give guarantees. As an example, consider a SystemC class m_0 whose virtual fields are the clocks x, y and a procedure f. Assume an explicit behavioral type declaration #TYPE(f, Q) which associates f with a description of its behavior: the proposition Q denotes its expected functionality. Let us 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. The assumption Q on the behavior of m_1. The f is required to provide a guarantee on the behavior of the module m_2 produced by the template class. Module m_3 is a candidate parameter for m_2. It structurally implements the interface m_0 and is annotated with the guarantee #TYPE(f, P), where P is the type of *pgm*. Now, let m_4 be the class defined by the instantiation of the template m_2 and the parameter m_3. To check the compatibility of the

actual parameter m_3 with the formal parameter m_0, we check the containment of the behaviors denoted by the proposition P (the type of the actual parameter) in the proposition Q (the type of the formal parameter). This amounts to check that P implies Q, either by model checking (if Q contains state transitions) or by static checking (if Q is a "stateless" property) (Fig. 24).

We consider a simple and minimalistic module system model for the purpose of exemplifying the scalability of our technique to structuring elements of general-purpose languages such as Java, C++, or System C. A component mod in an architecture is a class definition $\mathsf{class}\, m\, \{dec\}$, a template declaration $\mathsf{template}\, \langle \mathsf{class}\, x : m \rangle\, mod$ or a sequence of modules $mod; mod$. A class consists of a sequence of declarations. The keyword $\mathsf{use}\, m$ allows to use the members of class m within the current module (hence name elaboration is assumed to be explicit for simplification purposes). Declarations dec associate locations x with native classes m or template class instances $m\langle m_{1...k} \rangle$ and methods with a name f and a definition pgm. For instance, $\mathtt{integer}\, x$ defines an integer variable x (in Java or C) while $\mathsf{sc\_signal}\langle \mathtt{boolean} \rangle\, x$ defines a boolean signal x in SystemC. As we focus on typing program module behaviors we assume no sub-typing relation between data-types (Fig. 25).

We define our module system starting from the behavioral type system of Section 5. The type $\mathscr{T}$ of a module m consists of an environment $\mathscr{E}$ (that associates functions f with behaviors and variables x with data-types) and of a proof obligation $\mathscr{C}$. The type $\mathscr{T}_1 \rightarrow \mathscr{T}_2$ denotes a template class that produces a module of type $\mathscr{T}_2$ given a parameter of type $\mathscr{T}_1$.

$$\text{(type)} \quad \mathscr{T} ::= \mathscr{E}/\mathscr{C} \mid \Lambda x : \mathscr{T}_1.\mathscr{T}_2.$$

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

Fig. 24.   Type assumptions and guarantees in the SystemC module system.

$mod ::= \mathsf{class}\, m\, \{dec\} \mid \mathsf{template}\, \langle \mathsf{class}\, x : m \rangle\, mod \mid mod; mod$
$dec \; ::= m\langle n_{1..k} \rangle\, x \mid \mathsf{use}\, m \mid m\, f(x_{1..k})\, \{pgm\} \mid dec; dec$

Fig. 25.   Abstract syntax for declarations and modules.

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.

$$\text{(obligation)} \; \mathscr{C} ::= \texttt{true} \,|\, P \Rightarrow Q \,|\, \mathscr{C} \wedge \mathscr{C}.$$

The synthesis of proof obligations pertaining on the correctness of module composition is defined by the relation $\mathscr{E} \vdash mod : \mathscr{E}/\mathscr{C}$ and by induction on the syntax of modules and declarations. Rule (a) associates the location x with the type name m in the class-field type $[x : m]$. Rule (b) allows to use or open a module m.

$$\text{(a)} \; \mathscr{E} \vdash m \, x : [x : m] \qquad \text{(b)} \; \mathscr{E}[m : \mathscr{T}] \vdash \texttt{use} \, m : \mathscr{T}.$$

Rule (c) associates a method definition f with the class-field type $[f : \lambda x_{1...k} x L.\langle P, x^{\text{exit}}\rangle]$. Its side-condition $(*)$ is that $\mathscr{L} = \text{labs}(pgm)$ is the set of labels defined in pgm and that $L = \text{start}(pgm)$ is the entry point of pgm. It defines the proposition P and the continuation or output clock x^{exit} of the method f parameterized by its sequence $x_{1...k}$ of input variables, its result variable x, and the label L that defines its input clock. To process the function, we associate its reutrn value, denoted by $\texttt{return}$ to a signal x used to carry its value.

$$\text{(c)} \; \frac{L, \mathscr{E}[\texttt{return} : x] \vdash L : blk; pgm : P \;\; ^{(*)}}{\mathscr{E} \vdash m \; f(x_{1...k}) \; \{pgm\} : [f : \lambda x_{1...k} x L.\langle P/\mathscr{L}, x^{\text{exit}}\rangle]}.$$

Rule (d) sequentially processes the declarations dec in a module. The constraint $\texttt{true}$ is omitted in rules (a) and (c).

$$\text{(d)} \; \frac{\mathscr{E} \vdash dec_1 : \mathscr{E}_1/\mathscr{C}_1 \quad \mathscr{E} \uplus \mathscr{E}_1 \vdash dec_2 : \mathscr{E}_2/\mathscr{C}_2}{\mathscr{E} \vdash dec_1; dec_2 : \mathscr{E}_1 \uplus \mathscr{E}_2/\mathscr{C}_1 \wedge \mathscr{C}_2}.$$

Class-field declarations contribute to building the type $\mathscr{T}$ of a module. We write $\mathscr{E} \vdash m : \mathscr{T}$ iff $\mathscr{E}$ contains $[m : \mathscr{T}]$. An extension noted $\mathscr{E}_1 \uplus \mathscr{E}_2$ is defined by $\mathscr{E}_2$ and all class names and class-field names of $\mathscr{E}_1$ not overridden by a declaration in $\mathscr{E}_2$. Rule (α) defines the type $\mathscr{T}$ of a class by that of its field declarations.

$$\text{(α)} \; \frac{\mathscr{E} \vdash dec : \mathscr{T}}{\mathscr{E} \vdash \texttt{class} \, m \, \{dec\} : [m : \mathscr{T}]}.$$

Rule (β) defines the type of a template instance $m_1\langle m_2\rangle\, m$. Let $\Lambda(x : \mathcal{T}_1).\mathcal{T}$ be the type of the functor m_1. Let $\mathcal{T}_2$ be the type of the parameter m_2. If the subtyping relation $\mathcal{T}_1 \leqslant \mathcal{T}_2$ implies the proof obligation $\mathscr{C}$ then the type of m is $\mathcal{T}[m_2/x]$ (x is substituted by m_2).

$$(\beta) \quad \frac{\mathscr{E} \vdash m_1 : \Lambda(x : \mathcal{T}_1).\mathcal{T} \quad \mathscr{E} \vdash m_2 : \mathcal{T}_2 \quad \mathcal{T}_1 \leqslant \mathcal{T}_2 \Rightarrow \mathscr{C}}{\mathscr{E} \vdash m_1\langle m_2\rangle\, m : ([m : \mathcal{T}]/\mathscr{C})[m_2/x]}.$$

Rule (γ) defines the type of a template declaration $\mathsf{template}\,\langle\mathsf{class}\,m_1 : n_1\rangle\, mod$. Provided the *assumption* that the formal parameter m_1 of the template has the type $\mathcal{T}_1$ (that of the virtual class name n_1) the template *guarantees* that the module m_2 it defines has type $\mathcal{T}_2$. Hence the type $\Lambda(m_1 : \mathcal{T}_1).\mathcal{T}_2$ for module m_2.

$$(\gamma) \quad \frac{\mathscr{E} \vdash n_1 : \mathcal{T}_1 \quad \mathscr{E}[m_1 : \mathcal{T}_1] \vdash mod : [m_2 : \mathcal{T}_2]}{\mathscr{E} \vdash \mathsf{template}\,\langle\mathsf{class}\,m_1 : n_1\rangle\, mod : [m_2 : \Lambda(m_1 : \mathcal{T}_1).\mathcal{T}_2]}.$$

Rule (δ) processes module declarations in sequence.

$$(\delta) \quad \frac{\mathscr{E} \vdash mod_1 : \mathscr{E}_1/\mathscr{C}_1 \quad \mathscr{E} \uplus \mathscr{E}_1 \vdash mod_2 : \mathscr{E}_2/\mathscr{C}_2}{\mathscr{E} \vdash mod_1; mod_2 : \mathscr{E}_1 \uplus \mathscr{E}_2/\mathscr{C}_1 \wedge \mathscr{C}_2}.$$

Finally, the resolution of the behavioral sub-typing relation $\mathcal{T}_1 \leqslant \mathcal{T}_2$ is defined by structural induction. It reduces to the proof of a conjunction of propositions $P_1 \Rightarrow P_2$.

7.3. Design Checking

Properties pertaining on common design errors can easily be expressed and checked using our type system. Whereas related 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, our type system provides a generic framework to perform verification via model checking of behavioral properties of embedded systems described using imperative programming languages.

Termination: A common design error found in embedded system design is the unexpected termination of a thread due to, e.g., an uncaught exception. Here, the termination of a thread f can simply be expressed by the accessibility of the property $f^{\text{exit}} = 1$. Unexpected termination can hence be avoided by checking that f satisfies $f^{\text{exit}} = 0$.

Deadlocks: Another common design error is a wait statement that does not match a notification and yields the thread to block. Let $L_{1...n}$ be the clocks of the blocks $L_{1...n}$ in which a lock x is notified. Waiting for x at a given label L eventually terminates if P satisfies $L \wedge \neg(\wedge_{i=1}^{n} L_i) = 0$.

Races: 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 satisfies $(e_i \wedge (\vee_{j \neq i} e_j)) = 0$ for all $0 < i \leqslant n$.

Larger case-studies reporting applications of our technique in system design and verification are the complete model of a finite input response (FIR) filter starting from the SystemC 2.0.1 distribution.[7] In this case study, we demonstrate the benefits of modularly associating each System module to a behavioral type interface to perform optimizations and verifications which are modular and yet sensitive to the architecture in which modules or components are placed as reflected by the architecture's behavioral type and by application of an assumption-guarantee reasoning principle. A more recent and larger experiment applies the principles presented in this article to co-modeling by considering predefined SystemC components and connecting them around a bus architecture by giving a synchronous data-flow model of the interconnection wrappers.

8. RELATED WORK

By contrast to traditional type systems, which focus on rendering data-structure abstractions, behavioral type systems[13,14] are concerned with the abstraction of control structure in concurrent programs.

A related direction of research is software model checking using popular tools like Bandera,[15] Mops,[16] Verisoft,[17] Modex,[18,19] Slam,[20] CBMC,[21] Magic,[22] Blast,[23] Pathfinder.[24] Most software model checking tools proceed by extracting temporal logic models of source programs (either Java or C but raraely both) and perform sophisticated and efficient abstractions to drastically accelerate property verification.

Our approach contrasts with the software model checking trend in that it is primarily aimed at *modeling* software and then perform either of global model transformations (desynchronization, rethreading, etc) and code generation,[1] conformance checking by finite-flow equivalence using model checking techniques[4] or modular state-less abstraction for efficient property verification.[7] As such, our approach most closely relates to that of Modex[19] in which temporal property models are extracted for later verification with Spin. We experienced that representing such models using executable specifications expressed in a multi-clocked synchronous model

offers the additional benefit of operating correctness-preserving model transformations such as protocol synthesis (desynchronization[4]) or static scheduling (rethreading[1]). Finally, and unlike most related approaches in SMC, which are geared towards a particular programming language, we focus on a language-independent intermediate representation of Gnu's GCC.

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.[25] 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.[9]

9. CONCLUSIONS

Our contribution contrasts from related studies by the capability to capture a complete behavioral model of the type-checked system as well as model abstractions expressed at a scalable degree of precision. 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 (just as in a traditional type system), down to properties expressed by boolean equations between clocks, allowing for a rapid static-checking of design correctness properties. 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.

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 *de facto* standard design and programming languages, 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. It yields to SAT and model checking verification tools for an efficient verification of expected design properties and an early discovery of design errors.

REFERENCES

1. J.-P. Talpin, A. Gamatié, B. Le Dez, D. Berner, and P. Le Guernic, Hard Real-Time Implementation of Embedded Software in JAVA, in *FIDJI'2003*, Lectures notes in computer science, Springer (2003).
2. A. Benveniste, P. Le Guernic, and C. Jacquemot, Synchronous Programming with Events and Relations: The Signal Language and its Semantics, in *Science of Computer Programming*, Vol. 16. Elsevier (1991).
3. A. Gamatié, and T. Gautier. Synchronous Modeling of Avionics Applications using the Signal Language, in *Real-time embedded technology and applications symposium*, IEEE Press (2002).
4. J.-P. Talpin, P. Le Guernic, S. Shukla, R. Gupta, and F. Doucet, Formal Refinement Checking in a System-Level Design Methodology, *Fundamenta Informaticae*, IOS Press (2004).
5. J.-P. Talpin, D. Berner, S. K. Shukla, P. Le Guernic, and R. Gupta, A Behavioral type Inference System for Compositional System Design, *Application of Concurrency to System Design*, IEEE Press (2004).
6. D. Novillo, Tree SSA, A New Optimization Infrastructure for GCC, GCC developers summit (2003).
7. D. Berner, J.-P. Talpin, P. Le Guernic, and S. K. Shukla, Modular Design through Component Abstraction, in *International conference on compilers, architectures and synthesis for embedded systems*, ACM Press and (2004).
8. A. Pnueli, N. Shankar, and E. Singerman, Fair Synchronous Transition Systems and their Liveness Proofs, in *Symposium on Formal Techniques in Real-time and Fault-tolerant Systems*, Lecture notes in computer science vol. 1468, Springer (1998).
9. P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, Polychrony for System Design, in *Journal of Circuits, Systems and Computers*. Special Issue on Application-Specific Hardware Design, World Scientific (2002).
10. L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, Latency-Insensitive Protocols, in *Proc. of the 11th International Conference on Computer-Aided Verification*, Lecture notes in computer science vol. 1633, Springer Verlag (1999).
11. A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis, A Protocol for Loosely Time-triggered Architectures, in *Embedded Software Conference*, Lecture notes in computer science, Springer Verlag 2002.
12. T. P. Amagbegnon, L. Besnard, and P. Le Guernic, Implementation of the Data-flow Synchronous Language SIGNAL, in Conference on Programming Language Design and Implementation, ACM Press (1995).
13. F. Nielson, and H. Nielson, Type and Effect Systems: Behaviours for Concurrency, IC Press (1999).
14. S. K. Rajamani and J. Rehof, A behavioral Module System for the π-calculus, *Static Analysis Symposium*, Lecture notes in computer science, Springer (2001).

15. J. Hatcliff, and M. Dwyer, Using the Bandera Tool Set to Model-check Properties of Concurrent Java Software, *Invited tutorial, conference on concurrency theory*, Lectures notes in computer science Vol. 2154, Springer (2001).
16. H. Chen, D. Dean, and D. Wagner, Model Checking One Million Lines of C Code, *Network and Distributed System Security*, ISOC (2004).
17. P. Godefroid, Software Model Checking: The VeriSoft Approach, *Technical Memorandum ITD-03-44189G*, Bell Labs (2003).
18. G. J. Holzmann, Software Model Checking, in *Journal of Circuits, Systems and Computers. Special Issue on Application-Specific Hardware Design*, World Scientific (2002).
19. G. Holzmann, and M. Smith, An Automated Verification Method for Distributed Systems Software based on Model Extraction, *IEEE Transactions on Software Engineering*, vol. 28, IEEE Press (2002).
20. T. Ball, B. Cook, S. Das, and S. Rajamani, Refining Approximations in Software Predicate Abstraction, *In Tools and Algorithms for the Construction and Analysis of Systems*, Lecture notes in computer science, Vol. 2988. Springer (2004).
21. D. Kroening, E. Clarke, and F. Lerda, A Tool for Checking ANSI-C Programs, *In Tools and Algorithms for the Construction and Analysis of Systems, Lecture notes in computer science*, Vol. 2988. Springer (2004).
22. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, Modular Verification of Software Components in C, *In Transactions on Software Engineering*, Vol 30, IEEE Press (2004).
23. D. Beyer, A. Chlipala, T. Henzinger, The Blast Query Language for SoftWare Verification, *International Static Analysis Symposium*, Lecture notes in computer science, Vol. 3148, Springer (2004).
24. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, Model Checking Programs, *Automated Software Eng. J*. Vol. 10 (2003).
25. L. De Alfaro, and T. A. Henzinger, Interface Theories for Component-based Design. *International Workshop on Embedded Software*, Lecture notes in computer science Vol. 2211, Springer-Verlag (2001).