2009 Seventh IEEE International Conference on Software Engineering and Formal Methods

Relational interprocedural verification of concurrent programs

Bertrand Jeannet
INRIA Grenoble – Rhône-Alpes, France
Email: bertrand.jeannet@inrialpes.fr

Abstract

We propose a general analysis method for recursive, concurrent programs that tracks effectively procedure calls and returns in a concurrent context, even in the presence of unbounded recursion and infinite-state variables like integers. This method generalizes the relational interprocedural analysis of sequential programs to the concurrent case. We implemented it for programs with scalar variables, and we experimented several classical synchronisation protocols in order to illustrate the precision of our technique, but also to analyze the approximations it performs.

1. Introduction

We consider in this paper the reachable-state analysis of concurrent programs with a fixed number of threads, recursive procedures, shared memory and interleaving semantics. Such an analysis has to model the procedure call and return semantics in each thread, and to take into account the modification of global variables made by the other threads during the execution of the procedure of the current thread.

It is precisely the combination of recursion and concurrency which is difficult to tackle: in the case where the other threads does not modify shared variables, classical interprocedural techniques apply and are well-understood [3], [18]; in the case where no thread performs procedure calls, one can reduce the concurrent program to a sequential one by considering the product of the control-flow-graphs (CFG) of all threads, as done in model-checking. But the combination of the two features makes the reachability problem undecidable, even when all data variables are finite [24].

Various approaches have been recently explored. A first approach is thread-modular analysis, in which one considers a thread interacting with a context that abstracts the possible steps of other threads [8]. Another option is to be less general on the class of considered program: [23] defines a notion of transactional procedures for which they succeed to summarize procedures. Another recently explored approach consists in focusing only on executions with a bounded number of context switches [19]. This restriction basically allows to reduce the concurrent program to a sequential one, but the inferred invariants are not sound for any execution: they allow to discover bugs but they cannot prove a property.

This work has been supported by the Conseil Général de l'Isère and the Région Rhône-Alpes as part of the OpenTLM project (pôle de compétitivité Minalogic).

We propose a method that analyzes all threads in parallel and tracks effectively procedure calls and returns in a concurrent context, even in the presence of unbounded recursion and infinite-state variables like integers. It is based on a generalisation of relational interprocedural analysis of sequential programs. Technically, the key idea of our method is to revisit relational interprocedural analysis as an abstraction of the operational semantics of sequential programs [17]. This abstraction consists mainly in collapsing call-stacks into sets, in order to get rid of the source of infinity due to unbounded stacks, but *only after having appropriately instrumented the original semantics*. We generalize this method to concurrent programs, in which each thread has it own call-stack. After a suitable instrumentation, which defines the call-context used to match procedure calls and returns, we apply to call-stacks an abstraction which collapses separately the stack tail of each thread, but which take the product of their stack tops, in order to relate the local environments of the different threads.

The gain of our method compared to the above-mentioned approaches is better precision w.r.t. thread-modular techniques [8], generality (termination is guaranteed on any program, unlike [23]), and completeness, unlike [19].

Besides the theoretical motivation, an important application we have in mind is the verification of SystemC/TLM (Transaction-Level-Modeling) models of Systems-on-Chips (SoCs) [9], which are multithreaded C++ programs using a cooperative scheduling policy.

Contributions. Our first contribution is to show that it is possible to analyze concurrent, recursive programs using relational techniques in the sense of [3], [18], and to efficiently tackle unbounded recursion, unlike most other techniques. Our second contribution is methodological: we use instrumentation to define how procedure calls and returns are matched, we then collapse unbounded stacks into sets to make the control finite, and we resort to data abstraction to deal with the remaining source of infinity. Our third contribution is experimental: we implemented our technique for programs with finite-type and numerical variables, and we experimented with several classical synchronisation protocols, that allows us to illustrate the precision of our technique, but also to illustrate the approximations it performs.

Outline. §2 defines the program model we consider and its semantics. In §3 we instrument the standard semantics with information that will be exploited in the stack abstraction. §4 motivates and defines our *concurrent stack abstraction*, describes the induced forward abstract semantics, and discusses optimality results. We discuss in §5 its practical implementation and its complexity. We eventually describe in §6 the experiments that we performed with our implementation. In §7 we discuss two improvements of our abstraction. §8 concludes and discusses related work. For space reason, we omitted the various proofs, available in an extended version [14].

2. Program model and standard semantics

We consider a simple concurrent imperative programming language in which a program is composed of a set of procedures with a value parameter passing policy, and a *fixed number* of threads that communicates using shared global variables. Figs. 7–10 give examples of such programs. The syntactic and semantic domains we use are summarised in Figs. 1 and 3.

Program Syntax. A *program* is defined by a set of global variables, a set of procedures, and a set of concurrent threads. A *thread* T^t is defined by its main procedure, denoted as P_0^t. Each *procedure* $P_i = \langle \mathbf{fp}_i, \mathbf{fr}_i, \mathbf{l}_i, G_i \rangle$ is defined by its vector of (formal) input parameters $\mathbf{fp}_i$, output parameters $\mathbf{fr}_i$, and local variables $\mathbf{l}_i$ (with $LVar_i \supseteq FP_i \cup FR_i$), and by its intraprocedural CFG (control flow graph) G_i.

The *intraprocedural CFG* of a procedure P is a graph $G = \langle K, I \rangle$ where K is the set of *control points* of P, containing unique entry and exit control points s and e; $I : K \times K \to \mathsf{Inst}$ labels edges of the graph with two kinds of instructions: intraprocedural instructions $\langle R \rangle$ and procedure calls $\langle \mathbf{y} := P_j(\mathbf{x}) \rangle$, where $\mathbf{x}$ and $\mathbf{y}$ are the vectors of actual input and output parameters. Intraprocedural instructions are specified as a relation $R \subseteq (GEnv \times LEnv)^2$ allowing to express both tests and assignments. We assume that there are no two procedure call edges from the same point in G (non-deterministic choices should just be made *before* the call-point). This allows us to define the functions *call* and *ret* recording matching call and return-site nodes: if $I(c, c') = \langle \mathbf{y} := P_j(\mathbf{x}) \rangle$, then $call(c) = c'$ and $ret(c') = c$.

The *global CFG* G of the program is constructed as the union of intraprocedural CFG G_i's, further modified by replacing edges labelled by procedure calls by a *call-to-start* edge (connecting the call-site to the entry point of the callee) and an *exit-to-return* edge (connecting the exit point of the callee to the return-site), see Fig. 1. Thus there are three kinds of instructions in global CFGs: intraprocedural instructions $\langle R \rangle$, procedure calls $\langle \texttt{call } \mathbf{y} := P_j(\mathbf{x}) \rangle$ and procedure returns $\langle \texttt{ret } \mathbf{y} := P_j(\mathbf{x}) \rangle$. $proc(c)$ denotes the (index of the) procedure that contains c.

Operational Semantics. For the sake of simplicity, from now on we assume a program with only two threads. The semantic domains are summarised in Fig. 3. A state $s = (\sigma, \Gamma_1, \Gamma_2)$ is defined by a global environment σ and the stacks Γ^t of *activation records* of the 2 threads. An activation record is a pair of a control point c and an local environment ϵ. $\langle c^t_{n_t}, \epsilon^t_{n_t} \rangle$ is the current or top activation record of the thread T^t. Environments map variables to values. They can be concatenated with the $\oplus$ operator, and updated with the notation $\sigma[x \mapsto v]$. If $\mathbf{v}, \mathbf{v}'$ are vectors of variables, $\epsilon(\mathbf{v})$ denotes the corresponding vector of values, and $\mathbf{v} \setminus \mathbf{v}'$ denotes the subvector of $\mathbf{v}$ that does not contain any variable in $\mathbf{v}'$.

Fig. 4 first defines (in SOS-style) the semantics of one thread in isolation (transition relation $\to^t$). The transition relation $\to \subseteq S \times S$ induced by the full program is then defined as a special asynchronous product of the two transition relations $\to^1$ and $\to^2$, in which the global environment is shared. We define the initial set of states as $S^0 = \{\langle \sigma, \langle s_0^1, \epsilon^1 \rangle, \langle s_0^2, \epsilon^2 \rangle \rangle \mid init(\sigma)\}$ where s_0^1 and s_0^2 denote the start point of the main procedure of each thread, and *init* an initial condition on global variables.

Analysis goal. Our analysis goal is reachable-state analysis. For $X \subseteq S$, we define the concrete postcondition operator $post(X) = \{s' \mid \exists s \in X : s \to s'\}$, which we decompose according to the interprocedural CFG in the sequel:

$$post(X) = \bigcup_{(c,c') \in K \times K} post(\underbrace{c \xrightarrow{I(c,c')} c'}_{\tau})(X)$$

where $post(\tau)(X)$ is defined by the rules of Fig. 4. We then define the forward transfer function $F[S^0](X) = S^0 \cup post(X)$ where $S^0 \subseteq S$ denotes the initial states. Since $F[S^0]$ is monotone and continuous, according to Kleene's theorem we have

$$reach(S^0) = lfp(F[S^0]) = \bigcup_{n \geq 0} (F[S^0])^n(\emptyset) \qquad (1)$$

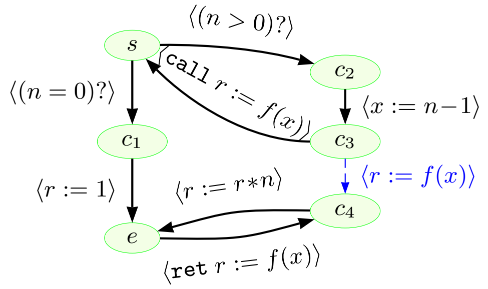
3. Instrumenting the standard semantics

We instrument now the operational semantics. The idea is to tag local environments of callee procedure with information about their call-context, and to use such tags to match procedure calls and returns. If we consider a procedure P in thread 1 of a two-threads program, its call-context is defined by

(1) The global variable and formal parameters at its start point in thread 1;
(2) The full call-stack of thread 2: during the execution of P in thread 1, thread 2 can perform several procedure returns and then calls again new procedures, with execution steps modifying global variables and

$T,\ P$:	Threads and procedures
P_0^t :	Main procedure of thread T^t
$GVar, \mathbf{g}$:	Global variables (set/vector)
$FP_i, \mathbf{fp}_i$:	Formal input parameters of P_i
$FR_i, \mathbf{fr}_i$:	Formal output parameters of P_i
$LVar_i, \mathbf{l}_i$:	Local variables of procedure P_i (including $\mathbf{fp}_i$ and $\mathbf{fr}_j$)
$\mathsf{s}_i, \mathsf{e}_i$:	Entry and exit points of P_i
$G = \langle K, I\rangle$:	Global flow graph

Figure 1. Syntactic domains.



$\langle (n>0)?\rangle$
s
c_2
$\langle \texttt{call}\ r := f(x)\rangle$
$\langle (n=0)?\rangle$
$\langle x := n-1\rangle$
c_1
c_3
$\langle r := f(x)\rangle$
$\langle r := 1\rangle$
$\langle r := r*n\rangle$
c_4
e
$\langle \texttt{ret}\ r := f(x)\rangle$

Figure 2. CFG for the Factorial (single-thread) program

v	$\in \mathit{Value}$	: values of expressions and variables
σ	$\in \mathit{GEnv} = \mathit{GVar} \to \mathit{Value}$	: global environments
ϵ	$\in \mathit{LEnv}_i = \mathit{LVar}_i \to \mathit{Value}$	: local environments for procedure P_i
ϵ	$\in \mathit{LEnv} = \bigcup_i \mathit{LEnv}_i$	: local environments for any procedure
$r = \langle c, \epsilon\rangle$	$\in \mathit{Act} = K \times \mathit{LEnv}$	: activation record (standard semantics)
Γ	$\in \mathit{Act}^+$	: stacks (sequences) of activation records
$\langle \sigma, \Gamma\rangle$	$\in S^t = \mathit{GEnv} \times \mathit{Act}^+$	: state of a thread in isolation
$\langle \sigma, \Gamma^1, \Gamma^2\rangle$	$\in S = \mathit{GEnv} \times \mathit{Act}^+ \times \mathit{Act}^+$	: full program states

Figure 3. Semantic domains

$$\frac{I(c,c') = \langle R\rangle \qquad R(\sigma,\epsilon,\sigma',\epsilon')}{\langle \sigma,\ \Gamma\cdot\langle c,\epsilon\rangle\rangle \to^t \langle \sigma',\ \Gamma\cdot\langle c',\epsilon'\rangle\rangle} \quad \text{(Intra)}$$

$$\frac{I(c,s_j) = \langle \texttt{call}\ \mathbf{y} := P_j(\mathbf{x})\rangle \qquad R^+_{\mathbf{y}:=P_j(\mathbf{x})}(\sigma,\epsilon,\epsilon_j)}{\langle \sigma,\ \Gamma\cdot\langle c,\epsilon\rangle\rangle \to^t \langle \sigma,\ \Gamma\cdot\langle c,\epsilon\rangle\cdot\langle s_j,\epsilon_j\rangle\rangle} \quad \text{(Call)}$$

$$\frac{I(e_j,c) = \langle \texttt{ret}\ \mathbf{y} := P_j(\mathbf{x})\rangle \qquad R^-_{\mathbf{y}:=P_j(\mathbf{x})}(\sigma,\epsilon,\epsilon_j,\sigma',\epsilon')}{\langle \sigma,\ \Gamma\cdot\langle call(c),\epsilon\rangle\cdot\langle e_j,\epsilon_j\rangle\rangle \to^t \langle \sigma',\ \Gamma\cdot\langle c,\epsilon'\rangle\rangle} \quad \text{(Ret)}$$

$$\frac{\langle \sigma,\Gamma_1\rangle \to^1 \langle \sigma',\Gamma_1'\rangle}{\langle \sigma,\Gamma_1,\Gamma_2\rangle \to \langle \sigma',\Gamma_1',\Gamma_2\rangle} \quad \text{(Conc1)} \qquad \frac{\langle \sigma,\Gamma_2\rangle \to^2 \langle \sigma',\Gamma_2'\rangle}{\langle \sigma,\Gamma_1,\Gamma_2\rangle \to \langle \sigma',\Gamma_1,\Gamma_2'\rangle} \quad \text{(Conc2)}$$

$$R^+_{\mathbf{y}:=P_j(\mathbf{x})}(\sigma,\epsilon,\epsilon_j) \overset{\text{def}}{=} \epsilon_j(\mathbf{fp}_j) = (\sigma\oplus\epsilon)(\mathbf{x}) \quad \text{(R+)}$$

$$R^-_{\mathbf{y}:=P_j(\mathbf{x})}(\sigma,\epsilon,\epsilon_j,\sigma',\epsilon') \overset{\text{def}}{=} \begin{cases} \sigma' = \sigma[\mathbf{y}^{(k)} \mapsto \epsilon_j(\mathbf{fr}_j^{(k)}) \mid \mathbf{y}^{(k)} \in \mathit{GVar}] \\ \epsilon' = \epsilon[\mathbf{y}^{(k)} \mapsto \epsilon_j(\mathbf{fr}_j^{(k)}) \mid \mathbf{y}^{(k)} \in \mathit{LVar}] \end{cases} \quad \text{(R-)}$$

Figure 4. Standard Operational Semantics: transition relation $\to^t$ of the thread T^t and transition relation $\to$ of the full program. The relations R^+ and R^- on environments define parameter passing mechanisms.

interacting with P. This dependency on the full call-stack of the other thread(s) is the intuitive reason why the combination of concurrency and recursion makes the reachability analysis undecidable even for Boolean programs.

We *choose* here to take into account only part (1) of the call-context, and we delay alternative choices to §7. Because P may modify them during its execution, we will introduce in local environments copies $\mathbf{g}_0, \mathbf{fp}_0$ of global variables $\mathbf{g}$ and formal parameters $\mathbf{fp}$ that contain at any point of P the value of $\mathbf{g}$ and $\mathbf{fp}$ at its start point. The second (orthogonal) point of our instrumentation is to push global variables into call-stacks.

In the instrumented semantics, the new environments $\epsilon \in \mathit{Env}$ are thus defined on variables $\mathbf{g}_0, \mathbf{fp}_0, \mathbf{g}, \mathbf{l}$, where the values of $\mathbf{g}_0, \mathbf{fp}_0$ keep track of the call-context at start point of the current procedure. We now have $\mathit{Act}_i = K \times \mathit{Env}$ and $S_i = \mathit{Act}_i^+ \times \mathit{Act}_i^+$.

Fig. 5 defines the new semantic rules induced by the standard semantics and the interpretation of the auxiliary variables. Rules (Conc1F) and (Conc2F) propagates the update of global variables induced by one thread to the other thread, so that the top activation records of the concurrent stacks always agree on the current value of global variables.

$$\frac{\begin{array}{c} I(c,c') = \langle R \rangle \\ R(\epsilon,\epsilon') \wedge \epsilon(\mathbf{g}_0, \mathbf{fp}_0) = \epsilon'(\mathbf{g}_0, \mathbf{fp}_0) \end{array}}{\Gamma \cdot \langle c, \epsilon \rangle \to_i^t \Gamma \cdot \langle c', \epsilon' \rangle} \quad \text{(IntraF)}$$

$$\frac{\begin{array}{c} I(c,s_j) = \langle \mathtt{call}\ \mathbf{y} := P_j(\mathbf{x}) \rangle \\ R^{+}_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon, \epsilon_j) \end{array}}{\Gamma \cdot \langle c, \epsilon \rangle \to_i^t \Gamma \cdot \langle c, \epsilon \rangle \cdot \langle s_j, \epsilon_j \rangle} \quad \text{(CallF)} \qquad \frac{\begin{array}{c} I(e_j,c) = \langle \mathtt{ret}\ \mathbf{y} := P_j(\mathbf{x}) \rangle \\ R^{-}_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon, \epsilon_j, \epsilon') \end{array}}{\Gamma \cdot \langle call(c), \epsilon \rangle \cdot \langle e_j, \epsilon_j \rangle \to_i^t \Gamma \cdot \langle c, \epsilon' \rangle} \quad \text{(RetF)}$$

$$\frac{\begin{array}{c} \Gamma_1 \to_i^1 \Gamma'_1 \qquad \Gamma'_1 = \Gamma''_1 \cdot \langle c'_1, \epsilon'_1 \rangle \\ \epsilon'_2 = \epsilon_2[\mathbf{g} \mapsto \epsilon'_1(\mathbf{g})] \end{array}}{\big\langle \Gamma_1,\ \Gamma_2 \cdot \langle c_2, \epsilon_2 \rangle \big\rangle \to_i \big\langle \Gamma'_1,\ \Gamma_2 \cdot \langle c_2, \epsilon'_2 \rangle \big\rangle} \quad \text{(Conc1F)} \qquad \frac{\begin{array}{c} \Gamma_2 \to_i^2 \Gamma'_2 \qquad \Gamma'_2 = \Gamma''_2 \cdot \langle c'_2, \epsilon'_2 \rangle \\ \epsilon'_1 = \epsilon_1[\mathbf{g} \mapsto \epsilon'_2(\mathbf{g})] \end{array}}{\big\langle \Gamma_1 \cdot \langle c_1, \epsilon_1 \rangle,\ \Gamma_2 \big\rangle \to_i \big\langle \Gamma_1 \cdot \langle c_1, \epsilon'_1 \rangle,\ \Gamma'_2 \big\rangle} \quad \text{(Conc2F)}$$

$$R^{+}_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon, \epsilon_j) \overset{\text{def}}{=} \epsilon_j(\mathbf{g}_0, \mathbf{fp}^j_0, \mathbf{g}, \mathbf{fp}^j) = \epsilon(\mathbf{g}, \mathbf{x}, \mathbf{g}, \mathbf{x}) \quad \text{(R+)}$$

$$R^{-}_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon, \epsilon_j, \epsilon') \overset{\text{def}}{=} \begin{cases} \epsilon'(\mathbf{g}_0, \mathbf{fp}_0, \mathbf{l} \setminus \mathbf{y}) & = \quad \epsilon(\mathbf{g}_0, \mathbf{fp}_0, \mathbf{l} \setminus \mathbf{y}) \\ \epsilon'(\mathbf{g} \setminus \mathbf{y}, \mathbf{y}) & = \quad \epsilon_j(\mathbf{g} \setminus \mathbf{y}, \mathbf{fr}) \end{cases} \quad \text{(R-)}$$

Figure 5. Instrumented semantics: transition relation $\to_i^t$ of the thread T^t and transition relation $\to_i$ of the full program.

In this semantics, reachable call-stacks are necessarily *well-formed* in the following sense.

Definition 1 (Well-formed stacks and states): A stack $\Gamma = \langle c_0, \epsilon_0 \rangle \dots \langle c_n, \epsilon_n \rangle \in Act_i^+$ is *well-formed* if, for any $i < n$:

(i) c_i is a call site for the procedure P_j, with $j = proc(c_{i+1})$ and $I(c_i, s_j) = \langle \mathtt{call}\ \mathbf{y} := P_j(\mathbf{x}) \rangle$;

(ii) equality between actual and formal input parameters holds: $\epsilon_i(\mathbf{g}, \mathbf{x}) = \epsilon_{i+1}(\mathbf{g}_0, \mathbf{fp}^j_0)$.

A state $\langle \Gamma^1, \Gamma^2 \rangle \in S_i$ is *well-formed* if Γ^1 and Γ^2 are well-formed, and if top activation records agree on the current value of global variables.

Proposition 1: If $s \in S_i$ is a well-formed state, then any $s' \in S_i$ such that $s \to_i^* s'$ is a well-formed state.

Notice that without instrumentation, instead of (ii) we would only have $c_{i+1} = \mathsf{s_j} \Rightarrow \epsilon_i(\mathbf{g}, \mathbf{x}) = \epsilon_{i+1}(\mathbf{g}, \mathbf{fp}^j)$.

With this notion of well-formed state, we get a strong conditions for an activation record to lie below another activation record in reachable call-stacks.

4. Stack abstraction and derived semantics

We will use the following functions on stacks: for any stack $\Gamma = r_0 \dots r_n \in Act^+$, $hd(\Gamma) = \{r_n\}$ and $tl(\Gamma) = \{r_i \mid 0 \leq i < n\}$. These functions are extended to sets of stacks. Also, a set $Y \in \wp(GEnv \times LEnv)$ will often be viewed as a predicate $Y(\mathbf{g}, \mathbf{l})$ on variables $\mathbf{g}$ and $\mathbf{l}$.

4.1. Two sources of inspiration

The abstract domain we propose for concurrent and recursive programs is inspired by two techniques.

The first one is the functional or relational approach described in [3], [18], in which one associates at each control point a relation between the input state and the current state of the enclosing procedure, so that at the exit point of a procedure P one obtains its input/output *summary* $R_P(x, x')$ capturing the effect of a call to P. [17] formalizes this approach by stack abstraction: starting from the instrumented semantics of §3, in which environments relate the input state of a procedure (variables $\mathbf{g}_0, \mathbf{fp}_0$) and their current state (variables $\mathbf{g}, \mathbf{l}$), it defines the Galois connection $\wp(Act_i^+) \underset{\alpha_f}{\overset{\gamma_f}{\leftrightarrows}} \wp(Act_i) \times \wp(Act_i)$ with

$$\alpha_f \ :\ \{\Gamma = r_0 \dots r_n\} \mapsto \left\langle \begin{array}{c} hd(\Gamma), \\ tl(\Gamma) \end{array} \right\rangle = \left\langle \begin{array}{c} \{r_n\}, \\ \{r_i \mid 0 \leq i < n\} \end{array} \right\rangle$$

$$\gamma_f : \langle Y_{hd}, Y_{tl} \rangle \mapsto \left\{ \Gamma = r_0 \dots r_n \left| \begin{array}{l} r_n \in Y_{hd} \\ \forall 0 \leq i < n : r_i \in Y_{tl} \\ \Gamma \text{ is a well-formed stack} \end{array} \right. \right\}$$

In the induced abstract semantics, when computing the effect of a procedure return, rule (RetF) of Fig. 5, the well-formedness condition in the definition of γ_f allows to match pairs of tail and top environments with the condition (ii) of Definition 1, so as to implement the relation composition of [18]. More precisely, if abstract values $(Y_{hd}, Y_{tl}) \in (\wp(K \times Env))^2 \simeq (K \to (Env \to \mathbb{B}))^2$ are viewed as pairs of functions from points to predicates, we get the following abstract procedure return:

$$\frac{\begin{array}{c} I(e_j, c) = \langle \mathtt{ret}\ \mathbf{y} := P_j(\mathbf{x}) \rangle \\ \underset{\text{(tail)}}{Y_{tl}(call(c))(\mathbf{g}_0, \mathbf{fp}_0,\ \mathbf{g}, \mathbf{l})} \quad \underset{\text{(top)}}{Y_{hd}(e_j)(\mathbf{g}^j_0, \mathbf{fp}^j_0,\ \mathbf{g}', \mathbf{l}')} \\ (\mathbf{g}, \mathbf{x}) = (\mathbf{g}^j_0, \mathbf{fp}^j_0) \\ \text{(well-formedness condition)} \\ R^{-}_{\mathbf{y}:=P_j(\mathbf{x})}(\mathbf{g}, \mathbf{l}, \mathbf{g}', \mathbf{l}',\ \mathbf{g}'', \mathbf{l}'') \\ \text{(output parameter passing)} \end{array}}{Y_{hd}(c)(\mathbf{g}_0, \mathbf{fp}_0,\ \mathbf{g}'', \mathbf{l}'') \qquad \text{(new top)}}$$

The second technique which inspired us is the classical method used for the analysis of non-recursive concurrent systems. Such systems have a state-space of the form $S = \mathit{GEnv} \times (K^1 \times \mathit{LEnv}^1) \times (K^2 \times \mathit{LEnv}^2)$. The usual technique for verifying such systems is to consider the product of their CFG by observing that

$$\wp(S) \simeq K^1 \times K^2 \to \wp(\mathit{GEnv} \times \mathit{LEnv}^1 \times \mathit{LEnv}^2) \qquad (2)$$

The ability to relate the local environments of concurrent threads is fundamental:

1) There is a technical reason. Assume that we maintain separate predicates $Y^1(g, l^1) = (g = l^1)$ and $Y^2(g, l^2) = (g = l^2 - 1)$ for two threads, and that the thread 1 executes an instruction $g := g + l^1$. It is easy to compute its effect on the predicate Y^1 (one obtains $(g = 2l^1)$), but less so on the predicate Y^2. The only way to perform this is actually to build $Y = Y^1 \wedge Y^2 = (g = l^1 \wedge l^1 = l^2 - 1)$, to compute the effect of the instruction on Y, and then to forget the variable l^1 (one obtains $(g = 2l^2 - 2)$).
   The conclusion is that one need to relate the local environments of different threads, at least temporarily, when a global variable is assigned in one thread.
2) There is also a precision reason. Consider the program below in which two threads synchronize their parallel execution by *rendez-vous* on a channel a (this can be implemented using global shared variables):

```
thread T1:            thread T2:
var i:int;            var j:int;
begin                 begin
  i = 0;                j = 0;
  while i<=10 do        while j<=11 do
    sync a;               sync a;
    i = i+1;              j = j+1;
  done                  done
end                   end
```

   In order to establish that the loop of the thread T2 does not terminate (the *rendez-vous* induces a deadlock when $j = 11$), we need to infer the invariant $i = j$ when each thread is at the control point just after the synchronisation instruction. If the possible environments of each thread are inferred separately, the non-termination of the thread T2 cannot be proved.

4.2. Concurrent stack abstraction

Intuition. The discussion of the previous section lead us to generalize the stack abstraction of [17] reminded above. Assuming two threads, an abstract value will be defined by three functions Y_{hd}, Y^1_{tl} and Y^2_{tl}, where

- $Y_{hd} : K^1 \times K^2 \to \wp(\mathit{Env}^1 \times \mathit{Env}^2)$ associates to pairs of control points (c^1, c^2) sets of (pairs of) environments $Y_{hd}(c^1, c^2)(\mathbf{g}^1_0, \mathbf{fp}^1_0,\ \mathbf{g}^2_0, \mathbf{fp}^2_0,\ \mathbf{g}, \mathbf{l}^1, \mathbf{l}^2)$ relating the global variables and the local variables of both threads (as in Eqn (2)), as well as auxiliary variables introduced by the instrumented semantics.[1]
- $Y^t_{tl} : K^t \to \wp(\mathit{Env}^t)$, $t = 1, 2$, associate to call-sites c the set of tail environments $Y^t_{tl}(\mathbf{g}^t_0, \mathbf{fp}^t_0,\ \mathbf{g}, \mathbf{l}^t)$ of thread t. Hence, the tail activation records of the different threads are not *directly* correlated, but *indirectly* they are linked by the means of global variables $\mathbf{g}$. This limits the loss of information due to the abstraction, as illustrated in §6.

If we consider a procedure P_j called by the thread 1, the abstract procedure return of Eqn (4.1) becomes:

$$\frac{\begin{array}{c} I(e_j, c) = \langle \mathtt{ret}\ \mathbf{y} := P_j(\mathbf{x})\rangle \\ Y^1_{tl}(\mathit{call}(c))(\mathbf{g}^1_0, \mathbf{fp}^1_0,\ \mathbf{g}, \mathbf{l}) \qquad \text{(tail 1)} \\ Y_{hd}(e_j, c^2)(\mathbf{g}^j_0, \mathbf{fp}^j_0,\ \mathbf{g}^2_0, \mathbf{fp}^2_0,\ \mathbf{g}', \mathbf{l}', \mathbf{l}^2) \qquad \text{(top)} \\ (\mathbf{g}, \mathbf{x}) = (\mathbf{g}^j_0, \mathbf{fp}^j_0) \\ \text{(well-formedness condition for stack 1)} \\ R^-_{\mathbf{y}:=P_j(\mathbf{x})}(\mathbf{g}, \mathbf{l}, \mathbf{g}', \mathbf{l}',\ \mathbf{g}'', \mathbf{l}'') \\ \text{(output parameter passing)} \end{array}}{Y_{hd}(c, c^2)(\mathbf{g}^1_0, \mathbf{fp}^1_0,\ \mathbf{g}^2_0, \mathbf{fp}^2_0,\ \mathbf{g}'', \mathbf{l}'', \mathbf{l}^2) \quad \text{(new top)}} \qquad (3)$$

$Y_{hd}(e_j, c^2)(\mathbf{g}^j_0, \mathbf{fp}^j_0,\ \mathbf{g}^2_0, \mathbf{fp}^2_0,\ \mathbf{g}, \mathbf{l}^1, \mathbf{l}^2)$ can be seen as a relation between the input $(\mathbf{g}^j_0, \mathbf{fp}^j_0)$ and the output $(\mathbf{g}, \mathbf{l})$ of P_j in thread 1, which depends on the current state of thread 2. This relation takes into account the moves that thread 2 has performed since thread 1 started the execution of P_j.

Formal definition. We define the Galois connection

$$\wp(S_i) = \wp(\mathit{Act}^+_i \times \mathit{Act}^+_i) \underset{\alpha_c}{\overset{\gamma_c}{\leftrightarrows}} A_c = \wp(\mathit{Act}_i \times \mathit{Act}_i) \times \wp(\mathit{Act}_i) \times \wp(\mathit{Act}_i) \qquad (7)$$

with

$$\alpha_c\Big(\big\{\langle \overbrace{r^1_0 \ldots r^1_{n_1}}^{\Gamma^1}, \overbrace{r^2_0 \ldots r^2_{n_2}}^{\Gamma^2}\rangle\big\}\Big) = \left\langle \begin{array}{c} hd(\Gamma_1, \Gamma_2), \\ tl(\Gamma_1), \\ tl(\Gamma_2) \end{array} \right\rangle = \left\langle \begin{array}{c} \{\langle r^1_{n_1}, r^2_{n_2}\rangle\}, \\ \{r^1_{i_1} \mid 0 \le i_1 < n_1\}, \\ \{r^2_{i_2} \mid 0 \le i_2 < n_2\} \end{array} \right\rangle$$

$$\gamma_c\big(\langle Y_{hd}, Y^1_{tl}, Y^2_{tl}\rangle\big) = \left\{ \langle \overbrace{r^1_0 \ldots r^1_{n_1}}^{\Gamma^1}, \overbrace{r^2_0 \ldots r^2_{n_2}}^{\Gamma^2}\rangle \;\middle|\; \begin{array}{c} \langle r^1_{n_1}, r^2_{n_2}\rangle \in Y_{hd} \\ \forall 0 \le i_1 < n_1 \;:\; r^1_{i_1} \in Y^1_{tl} \\ \forall 0 \le i_2 < n_2 \;:\; r^2_{i_2} \in Y^2_{tl} \\ \langle \Gamma^1, \Gamma^2\rangle \\ \text{is a well-formed state} \end{array} \right\}$$

Observing that $\wp(\mathit{Act}_i) = \wp(K \times \mathit{Env}) \simeq K \to \wp(\mathit{Env})$, we have the isomorphism

$$A_c \simeq \begin{array}{c} \big(K^1 \times K^2 \to \wp(\mathit{Env}^1 \times \mathit{Env}^2)\big) \quad \times \\ \big(K^1 \to \wp(\mathit{Env}^1)\big) \times \big(K^2 \to \wp(\mathit{Env}^2)\big) \end{array} \qquad (8)$$

1. we implicitly merge the two copies of global variables $\mathbf{g}^1$ and $\mathbf{g}^2$, assuming that corresp. concrete states are well-formed, so that $\mathbf{g}^1 = \mathbf{g}^2$.

$$apost(c \xrightarrow{\langle R\rangle} c')(Y_{hd}, Y^1_{tl}, Y^2_{tl}) = (Z_{hd}, Z^1_{tl}, Z^2_{tl}) \text{ with}$$

$$Z_{hd} = \left\{ \langle c', \epsilon',\ c^2, (\epsilon^2)'\rangle \;\middle|\; \begin{array}{c} \langle c, \epsilon,\ c^2, \epsilon^2\rangle \in Y_{hd} \\ R(\epsilon, \epsilon') \wedge \epsilon(\mathbf{g}_0, \mathbf{fp}_0) = \epsilon'(\mathbf{g}_0, \mathbf{fp}_0) \\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array} \right\} \qquad Z^1_{tl} = Y^1_{tl} \qquad Z^2_{tl} = Y^2_{tl} \tag{4}$$

$$apost(c \xrightarrow{\langle \texttt{call } \mathbf{y}:=P_j(\mathbf{x})\rangle} s_j)(Y_{hd}, Y^1_{tl}, Y^2_{tl}) = (Z_{hd}, Z^1_{tl}, Z^2_{tl}) \text{ with}$$

$$Z_{hd} = \left\{ \langle s_j, \epsilon_j,\ c^2, \epsilon^2\rangle \;\middle|\; \begin{array}{c} \langle c, \epsilon,\ c^2, \epsilon^2\rangle \in Y_{hd} \\ R^+_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon, \epsilon_j) \end{array} \right\} \qquad Z^1_{tl} = Y^1_{tl} \cup \left\{ \langle c, \epsilon\rangle \mid \langle c, \epsilon,\ c^2, \epsilon^2\rangle \in Y_{hd} \right\} \qquad Z^2_{tl} = Y^2_{tl} \tag{5}$$

$$apost(e_j \xrightarrow{\langle \texttt{ret } \mathbf{y}:=P_j(\mathbf{x})\rangle} c)(Y_{hd}, Y^1_{tl}, Y^2_{tl}) = (Z_{hd}, Z^1_{tl}, Z^2_{tl}) \text{ with}$$

$$Z_{hd} = \left\{ \langle c', \epsilon',\ c^2, (\epsilon^2)'\rangle \;\middle|\; \begin{array}{c} \langle e_j, \epsilon_j,\ c^2, \epsilon^2\rangle \in Y_{hd} \wedge \langle call(c), \epsilon\rangle \in Y^1_{tl} \\ \epsilon(\mathbf{g}, \mathbf{x}) = \epsilon_j(\mathbf{g}_0, \mathbf{fp}^j_0) \\ R^-_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon, \epsilon_j, \epsilon') \\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array} \right\} \qquad Z^1_{tl} = Y^1_{tl} \qquad Z^2_{tl} = Y^2_{tl} \tag{6}$$

Figure 6. Abstract postcondition $apost_c : A_c \to A_c$. The relation R^+ and R^- are defined by Eqns. (R+) and (R-) of Fig. 5

used in Eqn. (3). The abstract domain A_c defines an abstract semantics that it is simpler than the concrete semantics (the control is now finite) and that can be seen as an analysis method that can be further combined with a data abstraction as shown in §5.

4.3. Forward abstract semantics

Fig. 6 defines an abstract postcondition operator $apost_c : A_c \to A_c$ induced by the concrete postcondition $post : S_i \to S_i$. We decompose $apost_c$ according to the global CFG, and we detail only the steps performed by the thread 1.

The case of intraprocedural instruction (Eqn (4)) is simple: the top environment of thread 1 is modified according to the relation R, and the top environment of thread 2 is modified to reflect the new values of global variables. The sets of tail activation records are not modified. For procedure call, Eqn (5), the new top environment of thread 1 is initialized using the relation R^+ defined in Fig. 5. The set of tail activation records of thread 1 is extended by projecting the former top environment on thread 1. The case of procedure return, Eqn (6), is the most complex (see also Eqn (3)). We select a global top activation record in $r_{hd} \in Y_{hd}$ and a tail activation record $r^1_{tl} \in Y^1_{tl}$ for thread 1, so that actual parameters in r^1_{tl} match frozen copy of formal parameters in r_{hd}. The new top activation record is then obtained using the relation R^- defined in Fig. 5.

Proposition 2($apost_c$ is a correct approximation of post): For any set $X \subseteq S_i$ of well-formed states, $apost_c \circ \alpha(X) \sqsupseteq \alpha_c \circ post(X)$. More precisely:

(i) if τ is an intraprocedural or a call instruction, $apost_c(\tau) \circ \alpha(X) = \alpha \circ post(\tau)(X)$;
(ii) if τ is a return instruction, $apost_c(\tau) \circ \alpha(X) \sqsupseteq \alpha \circ post(\tau)(X)$

As a corollary, for any set $X_0 \in S_i$ of well-formed states, $lfp(F_c[X_0]) \sqsupseteq \alpha(lfp(F[X_0]))$, where $F_c[X_0](Y) = \alpha_c(X_0) \sqcup apost_c(Y)$ is the abstract transfer function.

Not surprisingly, the abstract semantics is less precise for return instructions, because they implicitly need to rebuild the stacks. We delay a deeper analysis of the loss of precision to §6.

Completeness results. The first trivial result we have is that in the case of multithreaded programs without procedure calls, our technique is exact, as the abstraction function becomes the identity. Observing now that the stack abstraction falls back to the *functional* abstraction defined in [17] for single-thread program, we inherit from the following theorem:

Theorem 1 ([17]): For single-thread programs, and for initial sets of states X_0 composed only of one-element stacks, the abstract reachability analysis is optimal: $areach_c(X_0) = \alpha_c(reach(X_0))$

This implies that the set of *top* activation records of reachable stacks is computed exactly, so that the invariants inferred at each control point are the exact ones.

5. Combining stack and data abstractions

Any abstraction for environments $\wp(Env) \leftrightarrows Env^\sharp$ can be applied to the stack abstraction domain A_c of Eqn (8) in order to obtain an implementable domain

$$A^\sharp_c = (K^1 \times K^2 \to Env^\sharp) \times (K^1 \to Env^\sharp) \times (K^2 \to Env^\sharp)$$

Provided that the lattice $Env^\sharp$ is equipped with meet and join operators, an abstract equality constraint between variables/dimensions, an abstract existential quantification, and an abstract operator $R^\sharp$ for intraprocedural instructions R,

Program	single-thread	concurrent	concurrent (var. 1)	concurrent (var. 2)
single-procedure	$k\cdot\varphi(g+l)$	$k^n\cdot\varphi(g+nl)$ (model-checking)		
recursion	$2k\cdot\varphi(2g+l)$ (relational interproc. analysis)	$k^n\cdot\varphi\big((n{+}1)g+nl\big)$ $+nk\cdot\varphi(2g+l)$	$nk^n\cdot\varphi(2g+nl)$ $+nk\cdot\varphi(2g+l)$	$nk^{2n-1}\cdot\varphi\big(2g+(2n{-}1)l\big)$ $+nk^{2n-1}\cdot\varphi\big(2g+(2n{-}1)l\big)$

n : number of threads $\quad$ g : number of global variables
k : number of control points $\quad$ l : number of local variables (max. per thread)
$\varphi(d)$: complexity of d-dimensional environments

Table 1. Complexity comparison. The meaning of the two last columns is given on §7.

the predicate formulation of $apost$ given in [14] can be implemented.

Literature offers several example of suitable abstractions for environments.

- When all variables are Booleans, $Env \simeq \mathbb{B}^n$, A_c is finite, and properties can be represented exactly with BDDs [2];
- When all variables are of numerical types, $Env \simeq \mathbb{R}^n$, and properties in $\wp(Env)$ can be abstracted by octagons [21], convex polyhedra [4], *etc...* In this case only intraprocedural instructions R and logical disjunction will induce a further approximation in $apost_c^\sharp : A_c^\sharp \to A_c^\sharp$ w.r.t. $apost_c : A_c \to A_c$.
- When variables are either Boolean or pointers to memory cells, [25] proposes an abstraction in which Boolean variables, pointers and memory configurations are represented and abstracted using 3-valued logical structures: $\wp(Env) \simeq \wp(\mathit{2-STRUCT}) \leftrightarrows \wp(\mathit{3-BSTRUCT})$. All the needed operations can be implemented, as shown in [15] in the context of interprocedural shape analysis of *sequential* programs.

Complexity analysis. We analyze here the space (resp. computational) worst-case complexity of the domain $A_c^\sharp$, by considering the size of the representation of abstract values (resp. the cost of operations on them). We assume that $\varphi(d)$ denotes the space/computational (worst-case) complexity of abstract environments $Env^\sharp$ of dimension d. For instance the space complexity $\varphi(d)$ is 2^d for BDDs and d^3 for octagons.

Table 1 (left part) gives complexity results in function of recursion and concurrency features. If we assume that $\varphi(d)$ is bounded by 2^d (case of Boolean programs without data abstraction) the complexity is

(1) polynomial in the size k of the CFGs,
(2) exponential in the number n of threads,
(3) in $\mathcal{O}(\varphi(nd))$ if $d = g + l$ is the number of visible variables in each thread: we inherit the complexity of the data abstraction modulo a factor n.

(1) and (2) corresponds to the complexity of model-checking, which is not surprising as our technique reduces to it in the single-procedure case. (3) shows the complexity of our method is higher than for the concurrent, non-recursive case due to the (expensive) duplication of variables performed by the instrumentation of §3.

It is important to note that most techniques aimed at reducing the practical complexity can be reused, as partial order reduction for concurrency [11], Cartesian product and/or variables packing for the number of variables [12].

6. Implementation and experiments

We implemented our analysis for programs manipulating finite-type and numerical variables. The applied data abstraction abstracts $\wp(Env) = \wp(\mathbb{B}^n \times \mathbb{R}^p)$ with functions $\mathbb{B}^n \to \mathsf{Pol}(\mathbb{R}^p)$ associating to Boolean variables convex polyhedra. These functions are implemented as Mtbdds [2], using the APRON numerical abstract domain library [16].

Our CONCURINTERPROC analyzer [13] takes as input a concurrent program, performs forward and/or backward analysis by solving Eqn. (1) on the above-described abstract domain using Kleene iteration and possibly widening, and then displays the results using various options. During the fixpoint analysis, the global equation system is actually built dynamically from the product of initial control points, using the CFG of each thread.

We experimented a number of synchronisation algorithms to illustrate the precision of our method, but also in order to analyze some of the approximations it induces. The experimented programs can be analyzed online [13].

Mutual exclusion algorithms. We first analyzed a few mutual exclusion algorithms, in which code to acquire and to release the critical section is delegated to two procedures `acquire` and `release`, as done for the Peterson algorithm depicted on Fig. 7. A forward analysis (3.5s on a 2GHz Pentium M laptop) succeeds to show that at most one thread can be in a critical section C1 or C2. Notice that this simple example already contains unbounded recursion (without correlation between threads), and several return-sites for most procedures. We also tried the program of Fig. 9, on which the analysis of [23] does not terminate, whereas ours terminates (in 8s) and proves that the mutual exclusion is ensured at the two sites and that the `fail` instruction is not reachable in any thread.

Notice that these two small examples are demanding, in the sense that synchronisation algorithms are very subtle and

```
var b0,b1,turn:bool;
initial not b0 and not b1;
proc acquire(tid:bool) returns ()
begin
  if not tid then
    b0 = true; turn = tid;
    assume (b1==false or turn==not tid);
  else
    b1 = true; turn = tid;
    assume (b0==false or turn==not tid);
  endif;
end
proc release(tid:bool) returns ()
begin
  if not tid then b0 = false;
             else b1 = false; endif;
end
proc main(tid:bool) returns ()
begin
  while random do
    acquire(tid); /* C1 */ release(tid);
  done;
  if random then main(tid); endif;
  acquire(tid); /* C2 */ release(tid);
end

thread T0:
var tid:bool;
begin tid = false; main(tid); end
thread T1:
var tid:bool;
begin tid = true; main(tid); end
```

Figure 7. The Peterson algorithm

```
var go : bool, counter,p0,p1:int;
initial counter==0 and go;
proc barrier(lgo:bool) returns (nlgo:bool)
begin
  lgo = not lgo; counter = counter+1;
  if counter==2
  then counter=0; go = lgo;
  else assume(lgo==go); endif;
  nlgo = lgo;
end
thread T0:
var lgo0:bool;
begin
  p0 = 0; lgo0 = true;
  while p0<=500 do
    lgo0 = barrier(lgo0); p0 = p0 + 1;
  done;
  lgo0 = barrier(lgo0);
end
thread T1:
var lgo1:bool;
begin
  p1 = 0; lgo1 = true;
  while p1<=502 do
    lgo1 = barrier(lgo1); p1 = p1 + 1;
  done;
  fail;
end
```

Figure 8. A synchronisation barrier algorithm with a counter, with calls inside counting loops

```
var g:uint[3],
    x,y:bool;
initial g==uint[3](0) and not x and not y;

proc foo(tid:bool,q:bool) returns ()
begin
  if not q then
    x=true; y=true; foo(tid,q);
  else
    acquire(tid); /* C1 */
    g = g + uint[3](1);
    release(tid);
  endif;
end
proc main(tid:bool) returns ()
var q:bool;
begin
  q = random;
  foo(tid,q);
  acquire(tid); /* C2 */
  if g==uint[3](0) then fail; endif;
  release(tid);
end

thread T0:
var tid:bool;
begin tid = false; main(tid); end
thread T1:
var tid:bool;
begin tid = true; main(tid); end
```

Figure 9. The example of [23], on which our analysis terminates

```
var T0,T1: int;
initial T0==0 and T0==T1;
proc
wait(pid:uint[1], time:int) returns ()
begin
  if pid == uint[1]0
  then T0 = T0 + time; yield; assume(T0 <= T1);
  else T1 = T1 + time; yield; assume(T1 <= T0);
  endif;
end
thread reader:
var p0,time0:int, pid:uint[1];
begin
  pid = uint[1]0; time0 = 10; p0 = 0;
  while (p0 < 100) do
    wait(pid,time0); p0 = p0 + 1;
  done;
  yield;
end
thread writer:
var p1,time1:int, pid:uint[1];
begin
  pid = uint[1]1; time1 = 20; p1 = 0;
  while (p1 < 100) do
    wait(pid,time1); p1 = p1 + 1;
  done;
  yield;
end
```

Figure 10. Producer and consumer with wrong time synchronisation, analyzed with a cooperative scheduling policy (use of `yield` instructions)

ask for precise analysis. Concerning running times, the size of the reachable part of the equation graph remains quite high: $(217, 486)$ and $(438, 800)$ for the two examples (in terms of nb. of locations and transitions).

Barrier synchronisation algorithms. We now experiment a synchronisation barrier algorithm from [26], Fig. 8. Our method proves (in 4s) that thread T1 cannot reach the `fail` instruction, provided we use the guided iteration technique of [10] to prevent loss of information due to the widening on convex polyhedra.

But if we make the counters `p0`, `p1` local to the main procedure of each thread, our method fails to infer that $p_0 = p_1$ when the control is at the head of the two loops, because they become uncorrelated when both threads are in the procedure `barrier`: in this case neither the tail environments of thread t nor the top environments contains both counters: the relation between these counters is lost and cannot be recovered on procedure return. This is a typical case where the call-context taken into account, as discussed in §3, is not sufficient: one should add the stack top of the other thread.

This phenomenon can be limited if local variables are related to global variables. In the example of Fig. 10, which is the skeleton of a timed SystemC/TLM model with cooperative scheduling, the counters `p0` and `p1` are local, but remains correlated by the two global clocks `T0` and `T1`. Thus, we can prove that the writer cannot terminate its loop (it is too slow). This example also illustrate the usefulness of reduction techniques. Here, because context switches can occur only in the `wait` procedure, only 32 locations are explored (in 0.4s).

7. Variations around the stack abstraction

Reducing the complexity by projection. In the abstract domain A_c defined by Eqn. (7), an abstract value is a triplet $\langle Y_{hd}, Y^1_{tl}, Y^2_{tl}\rangle$, the complexity of which is dominated by Y_{hd}, which can be viewed as a predicate $Y_{hd}(c^1,c^2)(\mathbf{g}^1_0,\mathbf{fp}^1_0,\ \mathbf{g}^2_0,\mathbf{fp}^2_0,\ \mathbf{g},\mathbf{l}^1,\mathbf{l}^2)$. Now, for the analysis to be relational, it is necessary:

1) in term of concurrency, to relate the variables $\mathbf{g},\mathbf{l}^1,\mathbf{l}^2$, as discussed in §4.1;
2) in term of procedure call/return, to relate the call-context $\mathbf{g}_0,\mathbf{fp}_0$ to the current value of variables, in order to perform the relation composition of Eqn. (3).

However, there is no strong intuition between correlating the variables $\mathbf{g}^1_0,\mathbf{fp}^1_0$ and $\mathbf{g}^2_0,\mathbf{fp}^2_0$. We could thus approximate Y_{hd} with the conjunction

$$\underbrace{Y^1_{hd}(c^1,c^2)(\mathbf{g}^1_0,\mathbf{fp}^1_0,\ \mathbf{g},\mathbf{l}^1,\mathbf{l}^2)}_{=\exists(\mathbf{g}^2_0,\mathbf{fp}^2_0)\ Y_{hd}(c^1,c^2)} \wedge \underbrace{Y^2_{hd}(c^1,c^2)(\mathbf{g}^2_0,\mathbf{fp}^2_0,\ \mathbf{g},\mathbf{l}^1,\mathbf{l}^2)}_{=\exists(\mathbf{g}^1_0,\mathbf{fp}^1_0)\ Y_{hd}(c^1,c^2)}$$

The new complexity of abstract values, which is given on Table 1, col. "var. 1", is lower due to the reduction of the number of (global) variables to be related in the same predicate. It is all the more interesting that the global store is likely to be more complex than the local store (for instance when it includes a model of the memory as in shape analysis [15]). We did not implement yet this technique, but we conjecture that the negative impact on precision should be very minor in practice.

Improving the precision by extending the call-context. In §3 we explained that the call-context of a procedure in a thread includes the full call-stacks of the other threads, and we made the explicit choice to abstract away this aspect in the analysis. A refinement would be to consider the top activation records of the other threads, which is a less rough abstraction of their call-stacks. Combined with the previous technique, for the thread 1 we would have tail and head activation records of the form:

$$\begin{tikzcd}
Y^1_{tl}(c^1,\boxed{c^2_0}, & \boxed{c^2})(\mathbf{g}^1_0,\mathbf{fp}^1_0, \arrow[d, leftrightarrow] & \mathbf{g}, \arrow[d, leftrightarrow, dotted] & \mathbf{l}^1, \boxed{\mathbf{l}^2_0}, \arrow[d, leftrightarrow, dotted] & \boxed{\mathbf{l}^2}) \arrow[d, leftrightarrow] \\
Y^1_{hd}(c^1,c^2, & \boxed{c^2_0})(& \mathbf{g}^1_0, & \mathbf{fp}^1_0,\mathbf{g},\mathbf{l}^1,\mathbf{l}^2, & \boxed{\mathbf{l}^2_0})
\end{tikzcd}$$

where the framed variables are the additional auxiliary variables, and the (solid) arrows indicate the additional matching performed when unifying tail and head activation records during procedure returns. The complexity of the resulting abstract values is given on Table 1, column "var. 2", is of course higher. Intuitively, extending the call-context makes mechanically the analysis less modular and more precise.

In particular this solution solves the precision problems raised by the example of Fig. 8 when the counters `p0` and `p1` are local variables.

8. Related work and conclusion

Our first contribution is an existence proof that it is possible to analyze concurrent, recursive programs using relational techniques. Our approach unifies the relational approach to interprocedural analysis of sequential programs, and the analysis technique for concurrent, non-recursive systems based on the product of their CFGs.

We also think that our method is conceptually elegant, based on a simple instrumentation of the concrete semantics, followed by a control abstraction that collapses stacks into sets and from which we derive mechanically an abstract semantics. §7 shows that the approach is general enough to define various alternatives to the abstraction of §4.

We showed that this method can be implemented using a non-trivial combination of Bdds and convex polyhedra, which allowed us to experiment small (but demanding) examples combining concurrency, unbounded recursion and infinite-state variables, and to illustrate its practical relevance. More experimental results are available at [13].

We did not address here the well-known efficiency problem raised by concurrency and interleaving semantics. However most techniques attacking this problem, like identification of atomic blocks [6] and partial order reduction [11] are fully applicable in our context and can be very efficient. Moreover, as mentioned in introduction, our target application is the analysis of SystemC/TLM models of SoCs [9], which follows a cooperative scheduling policy, thus making this problem less severe.

Our plan for the future is to apply our CONCURINTERPROC tool to SystemC/TLM models, and also to analyze concurrent data-structure algorithms using a suitable shape abstraction.

Related work. We focus on general techniques dealing with combination of recursion and concurrency. The SPADE tool [22] analyzes concurrent programs with dynamic threads and recursion by representing the program state by terms and by using rewriting techniques on sets of terms. Their running times are much higher than ours. [5] was a first step in this direction, but considers only unsynchronized concurrency. Works like [1] exploits the principles of regular model-checking, with each thread being represented with a pushdown system communicating by *rendez-vous*. Compared to our method, those techniques cannot be combined easily with infinite data-abstractions such as convex polyhedra, but most of them can handle dynamic thread creation.

Thread-modular techniques like [8] are more efficient but inherently less precise than our method w.r.t. concurrency: they never relate the local store of the different threads and they do not track the order of the updates of the global store performed by the environment of a thread (i.e., the other threads). [20] shows in the non-recursive case that such a thread-modular approach is an abstraction of the interleaving semantics. [8] uses explicit stacks and cannot tackle unbounded recursion (they can thus be more precise than us w.r.t. recursion). The gain of this approach is of course efficiency, and the ability to handle dynamic thread creation as in [7].

[23] is close to us in the ambition of extending relational analysis to concurrent programs. However their method is based on the notion of transactional procedures, and requires access to global variables to be protected by mutex, which makes it less general than ours. It is also guaranteed to terminate only for an identified class of programs, but in this case it seems that the analysis is exact, which is the good side of this approach.

According to our first experiments and our intuition, our approach should be especially efficient for the cases where the local environments of the different threads must be related (as in timed TLM models), because our analysis effectively relates them, but where synchronization mechanisms do not involve several *local* stores at different recursion depth, because this would require to put in the call-context of procedures several stack elements of the concurrent threads.

References

[1] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Concurrency Theory, CONCUR'05*, volume 3653 of *LNCS*, 2005.

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8), 1986.

[3] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP Conf. on Formal Description of Programming Concepts*, 1977.

[4] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Prog. Languages, POPL'78*. ACM, 1978.

[5] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *Principles of Prog. Languages, POPL'00*. ACM, 2000.

[6] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4), 2008.

[7] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3), 2005.

[8] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN'03: Workshop on Model Checking Software*, volume 2648 of *LNCS*, 2003.

[9] F. Ghenassia, editor. *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems*. Springer, 2005.

[10] D. Gopan and T. W. Reps. Guided static analysis. In *Static Analysis Symposium, SAS'07*, volume 4634 of *LNCS*, Aug. 2007.

[11] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian partial-order reduction. In *SPIN'07: Model Checking Software*, volume 4595 of *LNCS*, 2007.

[12] N. Halbwachs, D. Merchat, and L. Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design*, 29(1), 2006.

[13] B. Jeannet. The CONCURINTERPROC interprocedural analyzer for concurrent programs. http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi.

[14] B. Jeannet. Relational interprocedural analysis of concurrent programs. Technical Report 6671, INRIA, Oct. 2008.

[15] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *Static Analysis Symposium, SAS'04*, volume 3148 of *LNCS*, 2004.

[16] B. Jeannet and A. Miné. APRON: A library of numerical abstract domains for static analysis. In *Computer Aided Verification, CAV'2009*, LNCS, 2009. http://apron.cri.ensmp.fr/library/.

[17] B. Jeannet and W. Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *Int. Conf. on Algebraic Methodology and Software Technology, AMAST'04*, volume 3116 of *LNCS*, 2004.

[18] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Compiler Construction, CC'92*, volume 641 of *LNCS*, 1992.

[19] A. Lal, T. Touili, N. Kidd, and T. W. Reps. Interprocedural analysis of concurrent programs under a context bound. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08*, LNCS, 2008.

[20] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification is cartesian abstract interpretation. In *Int. Colloquium on Theoretical Aspects of Computing (ICTAC'06)*, volume 4281 of *LNCS*, 2006.

[21] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1), 2006.

[22] G. Patin, M. Sighireanu, and T. Touili. Spade: Verification of multithreaded dynamic and recursive programs. In *Computer Aided Verification, CAV'07*, volume 4590 of *LNCS*, 2007.

[23] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Principles of programming languages, POPL'04*. ACM, 2004.

[24] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. on Programing Language and Systems*, 22(2), 2000.

[25] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Prog. Languages and Systems*, 24(3), 2002.

[26] G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Prentice Hall, 2006.