Relational interprocedural analysis for concurrent programs

Bertrand Jeannet

INRIA Rhône-Alpes

December 5, 2008
Outline

Introduction
  Challenge: combining recursion and concurrency
  Existing approaches
  Our approach

Program model and semantics

Instrumenting the standard semantics

Concurrent stack abstraction
  Two sources of inspiration
  Concurrent stack abstraction

Combining stack and data abstraction

Evaluating the precision of stack abstraction
Analysis of concurrent and recursive programs

Why?

- Verification of SystemC/TLM model
- Viewed as programs with
  - concurrency (static multithreading)
  - interacting threads specified with procedures
Analysis of concurrent and recursive programs

Why?

- Verification of SystemC/TLM model
- Viewed as programs with
  - concurrency (static multithreading)
  - interacting threads specified with procedures

Why not?

- Strong undecidability result
Sequential versus concurrent program analysis

Analysis of concurrent programs without recursion

- Exact analysis computable for Boolean programs

Interprocedural analysis of sequential programs

- Well-understood [CC77, SP81, KS92]
- Exact analysis computable for Boolean programs
  - Not only w.r.t. properties/invariants
  - But also w.r.t. full call-stacks

Interprocedural analysis of concurrent programs?

- Active research, several and incomparable solutions
- Exact analysis not computable even for Boolean programs ([Ram00]: reduction to the Post problem)
Challenge: combination of recursion and concurrency

Sequential recursive programs

1. Computing the denotational semantics of procedures
   - block (procedure) $P_i \leftrightarrow$ predicate transformer $\phi_i : \mathcal{P} \rightarrow \mathcal{P}$
   - The $\phi_i$’s are expressed in function of each other
   - Fixpoint equation in the domain $\mathcal{P} \rightarrow \mathcal{P}$

2. Propagating reachability information
   - Propagation of input predicate using the $\phi_i$’s for procedure calls (in the domain $\mathcal{P}$)

The two steps can be combined in one single step (predicate transformers or summaries specialized on reachable input)

Concurrent single-procedure programs

- Reduces to a sequential program by interleaved product of control-flow graphs.
Challenge: combination of recursion and concurrency II

Requirement for the combination

1. Model accurately procedure call and return semantics in each thread
2. Take into account modifications of global variables made by the other threads.

Evaluation criteria

- static or dynamic thread creation
- communications between threads?
  - with `parbegin` construct, no communication *during* thread execution
  - with shared global variables, any synchronisation mechanism
- local variables in procedures?
  less easier than global variables, because of their temporary lifetime
Thread-modular model-checking [FQ03]

Principle

- To each thread $t$:
  - $R(t, g, l)$: true if global and local store is reachable;
  - $G(t, g, g')$: true if a step of $t$ can go from $g$ to $g'$.

- Inference rules

\[
\frac{R(t, g, l) \quad G(e, g, g')}{{t \neq e}} \quad R(t, g', l) \quad \frac{R(t, g, l) \quad T(t, g, l, g', l')}{{R(t, g', l') \quad G(t, g, g')}}
\]

Generalization to recursive programs: explicit stack for each thread

Pros/Cons

- Efficiency (the initial goal)
  - Termination non-guaranteed, unless further abstraction
  - Precision: abstracts the local stores, ignores the order of steps performed by the environment
Other approaches I

Identifying transactions [QRR04]

- Notion of transaction and transaction boundaries
- Boundaries may not match procedure calls and returns
- Subclass for which termination is guaranteed

Analysis under a context bound

- Considers only executions with a bounded number of context switches
  - unbounded number of steps possible between switches
- Allows reduction to sequential analysis
- More reminiscent of symbolic execution
  - discovers bugs, does not prove properties
Other approaches II

Alternative representations of the state-space

- Regular model-checking techniques (Bouajjani, Esparza, Touili)
  - Concurrent programs: set of PDA
  - Various abstractions applied to their stacks
- Rewriting techniques: SPADE tool (Sighireanu, Touili)
  - Represents states with terms and uses rewriting techniques (Timbuk tree automata library of Thomas Genet)
  - Handles dynamic thread creation

Not easily combined with infinite data abstraction, like convex polyhedra.
Our approach

State-space of 2-threads programs

\[ S = GEnv \times (K^1 \times LEnv^1)^+ \times (K^2 \times LEnv^2)^+ \]
- global store
- stack of thread 1
- stack of thread 2

We instrument the standard semantics
- properties on executions \( \implies \) properties on reachable states

\[ S_i = (K^1 \times Env^1)^+ \times (K^2 \times Env^2)^+ \]

We abstract stacks into sets

\[ \varphi\left((K^1 \times Env^1)^+ \times (K^2 \times Env^2)^+\right) \]

\[ \varphi(K^1 \times K^2 \times Env^1 \times Env^2) \times \varphi(K^1 \times Env^1) \times \varphi(K^2 \times Env^2) \]
- pairs of stack tops
- stack tails 1
- stack tails 2

Defines the analysis method
 Outline

Introduction
  Challenge: combining recursion and concurrency
  Existing approaches
  Our approach

Program model and semantics

Instrumenting the standard semantics

Concurrent stack abstraction
  Two sources of inspiration
  Concurrent stack abstraction

Combining stack and data abstraction

Evaluating the precision of stack abstraction
Program example: synchronisation barriers

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<=5 do
        lgo0 = barrier(lgo0);
        p0 = p0 + 1;
    done;
end

thread T1:
var lgo1:bool;
begin
    p1 = 0; lgo1 = true;
    while p1<=10 do
        lgo1 = barrier(lgo1);
        p1 = p1 + 1;
    done;
    fail;
end
Program model I

Programs composed of

- global variables $g$
- procedures
  - $f_{pi}, f_{po}$: formal input/output parameters of $P$
  - $l$: local variables of $P$
- static threads communicating via global variables
  - $P_t^0$: main procedure of thread $t$
Program model II

Global control flow graph $G$

- control points $K$
- $s_j, e_j$: start and exit points of procedure $P_j$
- Edges labeled with instructions
Program semantics I

Program state

\[
\sigma \quad \text{Global environment} \quad \Gamma^1 \quad \text{Stack of thread 1} \quad \Gamma^2 \quad \text{Stack of thread 2}
\]

with \( \sigma \in \text{GEnv} = \text{GVar} \to \text{Value} \) : global environments

\( \epsilon \in \text{LEnv} = \text{LVar} \to \text{Value} \) : local environments
Program semantics II

Operational Semantics

\[ l(c, c') = \langle R \rangle \]
\[ R(\sigma, \epsilon, \sigma', \epsilon') \]
\[ \langle \sigma, \Gamma \cdot \langle c, \epsilon \rangle, \Gamma^2 \rangle \rightarrow \langle \sigma', \Gamma \cdot \langle c', \epsilon' \rangle, \Gamma^2 \rangle \]  
\hspace{3cm} (Intra)

\[ l(c, s_j) = \langle \text{call } y := P_j(x) \rangle \]
\[ R^+_{y:=P_j(x)}(\sigma, \epsilon, \epsilon_j) \]
\[ \langle \sigma, \Gamma \cdot \langle c, \epsilon \rangle, \Gamma^2 \rangle \rightarrow \langle \sigma, \Gamma \cdot \langle c, \epsilon \rangle \cdot \langle s_j, \epsilon_j \rangle, \Gamma^2 \rangle \]  
\hspace{3cm} (Call)

\[ l(e_j, c) = \langle \text{ret } y := P_j(x) \rangle \]
\[ R^-_{y:=P_j(x)}(\sigma, \epsilon, \epsilon_j, \sigma', \epsilon') \]
\[ \langle \sigma, \Gamma \cdot \langle \text{call}(c), \epsilon \rangle \cdot \langle e_j, \epsilon_j \rangle, \Gamma^2 \rangle \rightarrow \langle \sigma', \Gamma \cdot \langle c, \epsilon' \rangle, \Gamma^2 \rangle \]  
\hspace{3cm} (Ret)
Outline

Introduction
  Challenge: combining recursion and concurrency
  Existing approaches
  Our approach

Program model and semantics

Instrumenting the standard semantics

Concurrent stack abstraction
  Two sources of inspiration
  Concurrent stack abstraction

Combining stack and data abstraction

Evaluating the precision of stack abstraction
Instrumenting the semantics I

Principle

- Global variables pushed on stacks
- Formal input parameters get a frozen copy
- New thread environments $\epsilon(g_0, fpi_0, g, l)$
  - $g_0, fpi_0$: values of global and parameters at start point
  - $g, l$: values of global and local variables at current point
  - $g_0, fpi_0$ tags environments with (an abstraction of) the call-context

State-space $S_i = (K^1 \times Env^1)^+ \times (K^2 \times Env^2)^+$

Threads must agree on the current value of global variables
Instrumenting the semantics II

Instrumented semantics: thread in isolation

\[ I(c, c') = \langle R \rangle \]
\[ R(\epsilon, \epsilon') \land \epsilon(g_0, fpi_0) = \epsilon'(g_0, fpi_0) \]
\[ \Gamma \cdot \langle c, \epsilon \rangle \rightarrow^t_i \Gamma \cdot \langle c', \epsilon' \rangle \]  
\[(\text{IntraF})\]

\[ I(c, s_j) = \langle \text{call } y := P_j(x) \rangle \]
\[ R^+_{y:=P_j(x)}(\epsilon, \epsilon) \]
\[ \Gamma \cdot \langle c, \epsilon \rangle \rightarrow^t_i \Gamma \cdot \langle c, \epsilon \rangle \cdot \langle s_j, \epsilon_j \rangle \]  
\[(\text{CallF})\]

\[ I(e_j, c) = \langle \text{return } y := P_j(x) \rangle \]
\[ R^-_{y:=P_j(x)}(\epsilon, \epsilon_j, \epsilon') \]
\[ \Gamma \cdot \langle \text{call}(c), \epsilon \rangle \cdot \langle e_j, \epsilon_j \rangle \rightarrow^t_i \Gamma \cdot \langle c, \epsilon' \rangle \]  
\[(\text{RetF})\]
Instrumenting the semantics III

Instrumented semantics: full program
Notifying update of global variables to other threads

\[
\begin{align*}
\Gamma_1 & \rightarrow_i^1 \Gamma'_1 & \Gamma'_1 &= \Gamma''_1 \cdot \langle c'_1, \epsilon'_1 \rangle \\
\epsilon'_2 &= \epsilon_2[g \mapsto \epsilon'_1(g)] \\
\langle \Gamma_1, \Gamma_2 \cdot \langle c_2, \epsilon_2 \rangle \rangle & \rightarrow_i \langle \Gamma'_1, \Gamma_2 \cdot \langle c_2, \epsilon'_2 \rangle \rangle \quad \text{(Conc1F)} \\
\ldots & \quad \text{(Conc2F)}
\end{align*}
\]
Properties of reachable stacks in instrumented semantics

Definition
A stack \( \Gamma = \langle c_0, \epsilon_0 \rangle \ldots \langle c_n, \epsilon_n \rangle \in \text{Act}^+ \) is well-formed if:

(i) \( c_i \) is a call site for the procedure \( \text{Proc}(c_{i+1}) = P_j \)

(ii) equality between actual and formal input parameters holds:
\[
\epsilon_i(g, x) = \epsilon_{i+1}(g_0, fpi^j_0).
\]

A state \( \langle \Gamma^1, \Gamma^2 \rangle \in S_i \) is well-formed if \( \Gamma^1 \) and \( \Gamma^2 \) are well-formed

Theorem
Reachable states are well-formed

Strong condition for an activation record to lie below another activation record in stacks
Outline

Introduction
  Challenge: combining recursion and concurrency
  Existing approaches
  Our approach

Program model and semantics

Instrumenting the standard semantics

Concurrent stack abstraction
  Two sources of inspiration
  Concurrent stack abstraction

Combining stack and data abstraction

Evaluating the precision of stack abstraction
Relational interprocedural analysis of sequential prog. I
formalized as a stack abstraction on instrumented semantics [JS04]

Galois connection \( S_i = \wp(\text{Act}^+) \leftrightarrow \wp(\text{Act}) \times \wp(\text{Act}) \) with

\[
\alpha_f : \{ \Gamma = r_0 \ldots r_n \} \mapsto \langle \text{hd}(\Gamma), \text{tl}(\Gamma) \rangle = \langle \{ r_n \}, \{ r_i \mid 0 \leq i < n \} \rangle
\]

\[
\gamma_f : \langle Y_{\text{hd}}, Y_{\text{tl}} \rangle \mapsto \left\{ \Gamma = r_0 \ldots r_n \left| \begin{array}{l}
  r_n \in Y_{\text{hd}} \\
  \forall 0 \leq i < n : r_i \in Y_{\text{tl}} \\
  \Gamma \text{ is a well-formed stack}
\end{array} \right. \right\}
\]
Induced abstract semantics: for procedure return

\[ \text{instr}(e_j, \text{ret}(c)) = \langle \text{ret } y := P_j(x) \rangle \]
Relational interprocedural analysis of sequential prog. II

Induced abstract semantics: for procedure return

\[ instr(e_j, ret(c)) = \langle \text{ret } y := P_j(x) \rangle \]

\[ Y_{tl}[c](g_0, f\pi_0, g, l) \]  \hspace{1cm}  \[ Y_{hd}[e_j](g^j_0, f\pi^j_0, g', l') \]

(tail)  \hspace{1cm}  \text{(top)}
Relational interprocedural analysis of sequential prog. II

Induced abstract semantics: for procedure return

$$instr(e_j,\ ret(c)) = \langle ret\ y := P_j(x) \rangle$$

$$Y_{tl}[c](g_0, fpi_0, g, l)$$
(tail)

$$Y_{hd}[e_j](g^j_0, fpi^j_0, g', l')$$
(top)

$$(g, x) = (g^j_0, fpi^j_0)$$
(well-formedness condition)
Induced abstract semantics: for procedure return

\[ \text{instr}(e_j, \text{ret}(c)) = \langle \text{ret } y := P_j(x) \rangle \]

\[ Y_{\text{tl}}[c](g_0, fpi_0, g, l) \quad Y_{\text{hd}}[e_j](g^j_0, fpi^j_0, g', l') \]

(tail) \quad (top)

\((g, x) = (g^j_0, fpi^j_0)\)

(well-formedness condition)

\[ R^\text{y:=P}_j(x)(g, l, g', l', g'', l'') \]

(output parameter passing)
Induced abstract semantics: for procedure return

\[ \text{instr}(e_j, \text{ret}(c)) = \langle \text{ret} \ y := P_j(x) \rangle \]

\[ Y_{\text{tl}}[c](g_0, fpi_0, g, l) \] (tail)
\[ Y_{\text{hd}}[e_j](g^j_0, fpi^j_0, g', l') \] (top)

\[ (g, x) = (g^j_0, fpi^j_0) \]
(well-formedness condition)

\[ R_{y := P_j(x)}^-(g, l, g', l', g'', l'') \]
(output parameter passing)

\[ Y_{\text{hd}}[\text{ret}(c)](g^j_0, fpi^j_0, g'', l'') \]
Relational interprocedural analysis of sequential prog. II

Induced abstract semantics: for procedure return

\[ \text{instr}(e_j, ret(c)) = \langle \text{ret y := } P_j(x) \rangle \]

\[ Y_{\text{tl}}[c](g_0, fpi_0, g, l) \quad Y_{\text{hd}}[e_j](g^j_0, fpi^j_0, g', l') \]

(tail) \hspace{2cm} (top)

\[ (g, x) = (g^j_0, fpi^j_0) \]

(well-formedness condition)

\[ R^-_{y:=P_j(x)}(g, l, g', l', g'', l'') \]

(output parameter passing)

\[ Y_{\text{hd}}[\text{ret}(c)](g^j_0, fpi^j_0, g'', l'') \]

Theorem (Optimality)

The abstract semantics preserves stack tops (but not full stacks)
State-space: \( S = G\text{Env} \times (K^1 \times L\text{Env}^1) \times (K^2 \times L\text{Env}^2) \)

**Principle**

- One observes \( \wp(S) \simeq K^1 \times K^2 \rightarrow \wp(G\text{Env} \times L\text{Env}^1 \times L\text{Env}^2) \)
- If needed, one abstracts it with \( K^1 \times K^2 \rightarrow \text{Env}^\# \)
  - \( \text{Env}^\# \): abstract environments (octagons, convex polyhedra \ldots)

Ability to relate the local environments of different threads is fundamental!
Analysis of non-recursive concurrent prog. II

Example

- Two predicates \( Y^1(g, l^1) = (g = l^1) \) and \( Y^2(g, l^2) = (g = l^2 - 1) \)

- Thread 1 executes \( g := g + l^1 \)

- Effect on \( Y^1 \): \( (g = 2l^1) \)

- Effect on \( Y^2 \)?
  - build \( Y = Y^1 \land Y^2 = (g = l^1 \land l^1 = l^2 - 1) \)
  - compute the effect of the instruction on \( Y \)
  - forget the variable \( l^1 \)

  \[ \iff (g = 2l^2 - 2) \]

Need for relate at least temporarily local environments of different threads
Example

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

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

To establish that the thread 2 does not terminate we need to infer the invariant \( i = j \) just after the synchronisation.

Need for relate as long as possible local environments of different threads
Concurrent stack abstraction I

Galois connection
\[ \varphi(\text{Act}^+ \times \text{Act}^+) \leftrightarrow_{\gamma_c}^{\alpha_c} \varphi(\text{Act} \times \text{Act}) \times \varphi(\text{Act}) \times \varphi(\text{Act}) \]

with
\[ \alpha_c(\{\langle r_0^1 \ldots r_{n_1}^1, r_0^2 \ldots r_{n_2}^2 \rangle\}) = \left\langle \begin{array}{l}
\text{hd}(\Gamma_1, \Gamma_2), \\
\text{tl}(\Gamma_1), \\
\text{tl}(\Gamma_2)
\end{array} \right\rangle = \left\langle \begin{array}{l}
\{\langle r_{n_1}^1, r_{n_2}^2 \rangle\}, \\
\{r_{i_1}^1 \mid 0 \leq i_1 < n_1\}, \\
\{r_{i_2}^2 \mid 0 \leq i_2 < n_2\}
\end{array} \right\rangle \]
Concurrent stack abstraction I

Galois connection
\[ \phi(\text{Act}^+ \times \text{Act}^+) \overset{\gamma_c}{\leftrightarrow} \phi(\text{Act} \times \text{Act}) \times \phi(\text{Act}) \times \phi(\text{Act}) \]

\[ \alpha_c\left(\left\{ \langle r_0^1 \ldots r_{n_1}^1, r_0^2 \ldots r_{n_2}^2 \rangle \right\}\right) = \left\langle \text{hd}(\Gamma_1, \Gamma_2), \right. \right. \]
\[ \left. \left. \text{tl}(\Gamma_1), \right. \right. \]
\[ \left. \left. \text{tl}(\Gamma_2) \right\rangle = \left\langle \left\{ \langle r_{n_1}^1, r_{n_2}^2 \rangle \right\}, \right. \right. \]
\[ \left. \left. \left\{ r_{i_1}^1 \mid 0 \leq i_1 < n_1 \right\}, \right. \right. \]
\[ \left. \left. \left\{ r_{i_2}^2 \mid 0 \leq i_2 < n_2 \right\} \right\rangle \]

\[ \gamma_c(\langle Y_{\text{hd}}, Y_{\text{tl}}^1, Y_{\text{tl}}^2 \rangle) = \]
\[ \left\{ \langle r_0^1 \ldots r_{n_1}^1, r_0^2 \ldots r_{n_2}^2 \rangle \right\} \]
\[ \langle r_{n_1}^1, r_{n_2}^2 \rangle \in Y_{\text{hd}} \land \epsilon_{n_1}^1(g) = \epsilon_{n_2}^2(g) \]
\[ \forall 0 \leq i_1 < n_1 : r_{i_1}^1 \in Y_{\text{tl}}^1 \]
\[ \forall 0 \leq i_2 < n_2 : r_{i_2}^2 \in Y_{\text{tl}}^2 \]
\[ r_0^1 \ldots r_{n_1}^1 \text{ and } r_0^2 \ldots r_{n_2}^2 \text{ are well-formed stacks} \]
Concurrent stack abstraction I

Galois connection
\[ \varnothing(\text{Act}^+ \times \text{Act}^+) \leftrightarrow_{\gamma_c} \alpha_c \varnothing(\text{Act} \times \text{Act}) \times \varnothing(\text{Act}) \times \varnothing(\text{Act}) \]

\[ \alpha_c \left( \{ \langle r_0^1 \ldots r_{n_1}, r_0^2 \ldots r_{n_2} \rangle \} \right) = \left\langle \begin{array}{c} \text{hd}(\Gamma_1, \Gamma_2), \\ \text{tl}(\Gamma_1), \\ \text{tl}(\Gamma_2) \end{array} \right\rangle = \left\langle \begin{array}{c} \{ \langle r_{i_1}^1, r_{i_2}^2 \rangle \}, \\ \{ r_{i_1}^1 \mid 0 \leq i_1 < n_1 \}, \\ \{ r_{i_2}^2 \mid 0 \leq i_2 < n_2 \} \end{array} \right\rangle \]

\[ \gamma_c(\langle Y_{\text{hd}}, Y_{\text{tl}}^1, Y_{\text{tl}}^2 \rangle) = \left\{ \begin{array}{l} \langle r_{i_1}^1, r_{i_2}^2 \rangle \in Y_{\text{hd}} \land \epsilon_{n_1}^1(g) = \epsilon_{n_2}^2(g) \\ \forall 0 \leq i_1 < n_1 : r_{i_1}^1 \in Y_{\text{tl}}^1 \\ \forall 0 \leq i_2 < n_2 : r_{i_2}^2 \in Y_{\text{tl}}^2 \\ r_0^1 \ldots r_{n_1}^1 \text{ and } r_0^2 \ldots r_{n_2}^2 \text{ are well-formed stacks} \end{array} \right\} \]

Abstract semantics: induced mechanically by abstracting the instrumented semantics with the Galois connection
Concurrent stack abstraction II

Predicate version

\[ A_c \simeq (K^1 \times K^2 \rightarrow \varnothing(\text{Env}^1 \times \text{Env}^2)) \times (K^1 \rightarrow \varnothing(\text{Env}^1)) \times (K^2 \rightarrow \varnothing(\text{Env}^2)) \]

An abstract value: \( \langle Y_{\text{hd}}, Y_{\text{tl}}^1, Y_{\text{tl}}^2 \rangle \) with

\begin{itemize}
  \item \( Y_{\text{hd}}[c^1, c^2](g^1_0, fpi_0^1, g_0^2, fpi_0^2, g, l^1, l^2) \)
    Top environments directly related
  \item \( Y_{\text{tl}}^1[c^1](g_0^1, fpi_0^1, g, l^1) \)
  \item \( Y_{\text{tl}}^2[c^2](g_0^2, fpi_0^2, g, l^2) \)
    Tail environments indirectly related via global variables
\end{itemize}

Abstract postcondition relies on

1. conjunctions and disjunctions
2. equality constraints between variables/dimensions
3. existential quantification
4. relations induced by intraprocedural instructions \( \langle R \rangle \)
Concurrent stack abstraction III

Induced semantics for procedure return (in thread 1)

\[
instr(e_j, ret(c)) = \langle ret \ y := P_j(x) \rangle
\]
Concurrent stack abstraction III

Induced semantics for procedure return (in thread 1)

\[
\text{instr}(e_j, \text{ret}(c)) = \langle \text{ret } y := P_j(x) \rangle
\]

\[
Y_{ti}^{1}[c, c^2](g_0, f_{pi}^0, g, l) \quad Y_{hd}[e_j, c^2](g_j^0, f_{pi}^j, g_j^2, f_{pi}^2, g', l', l^2)
\]

(tail 1) (top)
Concurrent stack abstraction III

Induced semantics for procedure return (in thread 1)

\[ \text{instr}(e_j, \text{ret}(c)) = \langle \text{ret } y := P_j(x) \rangle \]

\[ Y_{ti}^1[c, c^2](g_0, \text{fpi}_0, g, l) \]  \hspace{1cm}  \[ Y_{hd}[e_j, c^2](g^j_0, \text{fpi}^j_0, g^2_0, \text{fpi}^2_0, g', l', l^2) \]

(tail 1) \hspace{2cm} (top)

\[ (g, x) = (g^j_0, \text{fpi}^j_0) \]

(well-formedness condition for stack 1)
Induced semantics for procedure return (in thread 1)

\[ \text{instr}(e_j, \text{ret}(c)) = \langle \text{ret } y := P_j(x) \rangle \]

\[ Y_{ti}^1[c, c^2](g_0, fpi_0, g, l) \]
\[ Y_{hd}[e_j, c^2](g^j_0, fpi^j_0, g^2_0, fpi^2_0, g', l', l^2) \]

(top)

(tail 1)

Well-formedness condition for stack 1

\[ (g, x) = (g^j_0, fpi^j_0) \]

Output parameter passing

\[ R_y^-=P_j(x)(g, l, g', l', g'', l'') \]
Induced semantics for procedure return (in thread 1)

\[ instr(e_j, ret(c)) = \langle ret \ y := P_j(x) \rangle \]

\[ Y_{ti}^1[c, c^2](g_0, fpi_0, g, l) \quad Y_{hd}[e_j, c^2](g_j^i, fpi_j^i, g_0^2, fpi_0^2, g', l', l^2) \]

(tail 1) (top)

\[ (g, x) = (g_0^i, fpi_0^i) \]

(well-formedness condition for stack 1)

\[ R_y^- = P_j(x)(g, l, g', l', g'', l'') \]

(output parameter passing)

\[ Y_{hd}[ret(c)](g_j^i, fpi_j^i, g_0^2, fpi_0^2, g'', l'', l^2) \]
Concurrent stack abstraction IV

Optimality result: concurrent stack abstraction reduces

1. to classical interprocedural analysis
   (one thread with recursion)

2. to classical analysis of concurrent programs
   (multiple threads without recursion)

In the other cases, it induces approximations.

- Otherwise it would contradict the undecidability result
Outline

Introduction
  Challenge: combining recursion and concurrency
  Existing approaches
  Our approach

Program model and semantics

Instrumenting the standard semantics

Concurrent stack abstraction
  Two sources of inspiration
  Concurrent stack abstraction

**Combining stack and data abstraction**

Evaluating the precision of stack abstraction
Combining stack and data abstraction

We assume that we are given a Galois connection

\[ \mathcal{O}(\text{Env}) \xleftrightarrow{\gamma_e} \alpha_e \xrightarrow{\gamma_e} \text{Env}^\# \]

Then we can combine Galois connections:

\[ \mathcal{O} \left( (K^1 \times \text{Env})^+ \times (K^2 \times \text{Env})^+ \right) \]

\[ \xleftrightarrow{\gamma_c} \alpha_c \]

\[ \xrightarrow{A_c} \]

\[ \left( K^1 \times K^2 \rightarrow \mathcal{O}(\text{Env}) \right) \times \left( K^1 \rightarrow \mathcal{O}(\text{Env}) \right) \times \left( K^2 \rightarrow \mathcal{O}(\text{Env}) \right) \]

\[ \left( K^1 \times K^2 \rightarrow \text{Env}^\# \right) \times \left( K^1 \rightarrow \text{Env}^\# \right) \times \left( K^2 \rightarrow \text{Env}^\# \right) \]
Example of suitable data abstractions

- **Boolean programs**: $\text{Env} \sim \mathbb{B}^n$
  We have a finite lattice, no need for data abstraction

- **Programs with numerical variables**: $\text{Env} \sim \mathbb{R}^n$
  Relational abstraction applicable: octagons, convex polyhedra, linear congruences, ... 
  Approximations: due to $\mathbb{I}$ and to abstraction of single instructions

- **Programs with either Booleans or pointers to memory cells**
  Stores represented/abstracted with 3-valued logical structures [SRW02]:

  \[ \varphi(\text{Env}) \sim \varphi(2 - \text{STRUCT}) \iff \varphi(3 - \text{BSTRUCT}) \]

  Enable the extension of [JLRS04] to concurrent programs
# Complexity analysis

<table>
<thead>
<tr>
<th>Program</th>
<th>single-thread</th>
<th>concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>single-procedure recursion</td>
<td>$k \cdot \varphi(g + l)$</td>
<td>$k^n \cdot \varphi(g + nl)$</td>
</tr>
<tr>
<td></td>
<td>$2k \cdot \varphi(2g + l)$</td>
<td>$k^n \cdot \varphi(g + n(g + l))$</td>
</tr>
</tbody>
</table>

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

Assuming $\varphi(d) = \mathcal{O}(2^d)$, the **global complexity** is

- polynomial in the size $k$ of the CFGs,
- exponential in the number $n$ of threads,
- in $\mathcal{O}(\varphi(nd))$ if $d = g + l$ is the number of visible variables active in each thread.
Complexity analysis

<table>
<thead>
<tr>
<th>Program</th>
<th>single-thread</th>
<th>concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>single-procedure</td>
<td>$k \cdot \varphi(g + l)$</td>
<td>$k^n \cdot \varphi(g + nl)$</td>
</tr>
<tr>
<td>recursion</td>
<td>$2k \cdot \varphi(2g + l)$</td>
<td>$k^n \cdot \varphi(g + n(g + l))$</td>
</tr>
</tbody>
</table>

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

Assuming $\varphi(d) = \mathcal{O}(2^d)$, the global complexity is

- polynomial in the size $k$ of the CFGs,
- exponential in the number $n$ of threads,
- in $\mathcal{O}(\varphi(nd))$ if $d = g + l$ is the number of visible variables active in each thread.

Well-known techniques for reducing the complexity can be reused: partial order and symmetry reduction for concurrency, Cartesian product and/or variables packing for data abstraction, . . .
Outline

Introduction
  Challenge: combining recursion and concurrency
  Existing approaches
  Our approach

Program model and semantics

Instrumenting the standard semantics

Concurrent stack abstraction
  Two sources of inspiration
  Concurrent stack abstraction

Combining stack and data abstraction

Evaluating the precision of stack abstraction
Implementation

- **ConcurInterproc**: Interproc generalized with concurrency
- Programs with finite-state and/or numerical variables
  Data abstraction: \( \wp(\mathbb{B}^n \times \mathbb{R}^p) \leftrightarrow \mathbb{B}^n \rightarrow \text{Pol}(\mathbb{R}^p) \)
  implemented with MTBDDs (CUDD & APRON)
- Both forward and backward analysis implemented
- Choice between preemptive and cooperative scheduling

Online version available at
http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi
Experiments

Goal

- Illustrate the precision of our method
- Analyze some of the approximations it induces

Test programs

Synchronisation algorithms requiring a detailed analysis of interactions between threads

- Mutual exclusion algorithms: Peterson, Kessel
- Barrier synchronisation: a protocol using counters
Mutual exclusion: Peterson

```
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 true do
    acquire(tid); /* C */
    release(tid);
  done;
end

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

thread T1:
  var tid: bool;
  begin
    tid = true; main(tid);
  end
```
Non-terminating example of [QRR04]

var .., g:uint[3], x,y:bool, ...
initial .. and
    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);
        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);
    acquire(tid);
    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
Synchronisation barriers with counters

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<=5 do
        lgo0 = barrier(lgo0);
        p0 = p0 + 1;
    done;
end

thread T1:
var lgo1:bool;
begin
    p1 = 0; lgo1 = true;
    while p1<=10 do
        lgo1 = barrier(lgo1);
        p1 = p1 + 1;
    done;
    fail;
end

Success if p1,p2 global, failure if they are local
Synchronisation barrier with counters: half-working

var go : bool, counter: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 /* E0 */

thread T0:
var lgo0:bool;
begin
  lgo0 = true; /* A0 */
  lgo0 = barrier(lgo0); /* A1 */ /* A2 */ /* A3 */
end /* E0 */

thread T1:
var lgo1:bool;
begin
  lgo1 = true; /* B0 */
  lgo1 = barrier(lgo1); /* B1 */
  lgo1 = barrier(lgo1); /* B2 */
  lgo1 = barrier(lgo1); /* B3 */
  fail;
end
Conclusion

- Unifies methods for recursive and concurrent programs
Conclusion

- Unifies methods for recursive and concurrent programs
- Technically, rather simple:
  - Classical instrumentation of the standard semantics
    - Less classical for backward semantics
  - Stack abstraction
    - collapses stacks into sets
  - Abstract semantics derived mechanically
    - Most technical aspect
Conclusion

- Unifies methods for recursive and concurrent programs
- Technically, rather simple:
  - Classical instrumentation of the standard semantics
    - Less classical for backward semantics
  - Stack abstraction
    - Collapses stacks into sets
  - Abstract semantics derived mechanically
    - Most technical aspect
- Separates control and data abstraction
  - Can extend any relational interprocedural analysis to concurrent programs
Conclusion

- Unifies methods for recursive and concurrent programs
- Technically, rather simple:
  - Classical instrumentation of the standard semantics
    - Less classical for backward semantics
  - Stack abstraction
    - Collapses stacks into sets
  - Abstract semantics derived mechanically
    - Most technical aspect
- Separates control and data abstraction
  - Can extend any relational interprocedural analysis to concurrent programs
- Experimental evaluation of its precision
  - Local variables handled less precisely than global variables
  - Hence, procedure inlining may improve precision
Perspective

Stack abstraction

- Better precision/Less modularity/Worse complexity: adding more information in the call-context for each thread?
- Further abstraction to derive thread-modular analysis of [FQ03]?
- Adapting iteration and widening techniques to a concurrent context

Application to TLM models

- Exploiting cooperative scheduling
- Encoding of TLM synchronisation concepts (events, time)
- Built-in, higher-level synchronisation primitives?
Patrick Cousot and Radhia Cousot.  
Static determination of dynamic properties of recursive procedures.  
In *IFIP Conf. on Formal Description of Programming Concepts*, 1977.

M. Sharir and A. Pnueli.  
Semantic foundations of program analysis.  

J. Knoop and B. Steffen.  
The interprocedural coincidence theorem.  
Tom Reps, Susan Horwitz, and Mooly Sagiv.
Precise interprocedural dataflow analysis via graph reachability

Javier Esparza and Jens Knoop.
An automata-theoretic approach to interprocedural data-flow analysis.

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

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

Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof.
Summarizing procedures in concurrent programs.
In *Principles of programming languages, POPL’04. ACM, 2004.*
Patrick Cousot and Radhia Cousot.
Static determination of dynamic properties of recursive procedures.
In *IFIP Conf. on Formal Description of Programming Concepts, 1977*.

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

A relational approach to interprocedural shape analysis.
In *Static Analysis Symposium, SAS’04, volume 3148 of LNCS, 2004*.

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*. 
J. Knoop and B. Steffen.  
The interprocedural coincidence theorem.  

Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof.  
Summarizing procedures in concurrent programs.  

G. Ramalingam.  
Context-sensitive synchronization-sensitive analysis is undecidable.  

M. Sharir and A. Pnueli.  
Semantic foundations of program analysis.  

M. Sagiv, T. Reps, and R. Wilhelm.
Parametric shape analysis via 3-valued logic.