Software Model Checking for Cooperative Threaded Programs

Alessandro Cimatti
j.w.w. Iman Narasamdya and Marco Roveri

Fondazione Bruno Kessler - Embedded System Unit, Italy

ATVA'13
October 2013
Hanoi, Vietnam

Motivations

- Multi-threaded software with cooperative scheduling (or cooperative threads) is adopted in many embedded system domains
  - SystemC, SpecC, FairThreads, OSEK/VDX, PLC, ...

Motivations

- Multi-threaded software with cooperative scheduling (or cooperative threads) is adopted in many embedded system domains
  - SystemC, SpecC, FairThreads, OSEK/VDX, PLC, ...

- Formal verification of cooperative threads is challenging:
  - Scheduling policy is complex, yet correctness depends on the details
  - Threads have inifinite state space

Motivations

- Multi-threaded software with cooperative scheduling (or cooperative threads) is adopted in many embedded system domains
  - SystemC, SpecC, FairThreads, OSEK/VDX, PLC, ...
- Formal verification of cooperative threads is challenging:
  - Scheduling policy is complex, yet correctness depends on the details
  - Threads have inifinite state space
- Existing formal verification approaches are limited:
  - Disregard significant semantics aspects
  - Perform under-approximations
  - Poor scalability

Outline

- Cooperative Threaded Programs (CTPs)
- Background
  - Safe Sequential Programs
  - Model Checking of Sequential Programs
    - Finite Model for Sequential Programs
    - Symbolic Model Checking of Sequential Programs
- Approaches to Model Checking of CTPs
  - Finite-Model for Cooperative Threaded Programs
  - Symbolic Model Checking of Sequential Software
  - Explicit Scheduler and Symbolic Threads (ESST)
- The Kratos Software Model Checker
- Experimental Results
- Related Work
- Conclusions

Outline

- Cooperative Threaded Programs (CTPs)
- Background
  - Safe Sequential Programs
  - Model Checking of Sequential Programs
    - Finite Model for Sequential Programs
    - Symbolic Model Checking of Sequential Programs
- Approaches to Model Checking of CTPs
  - Finite-Model for Cooperative Threaded Programs
  - Symbolic Model Checking of Sequential Software
  - Explicit Scheduler and Symbolic Threads (ESST)
- The Kratos Software Model Checker
- Experimental Results
- Related Work
- Conclusions

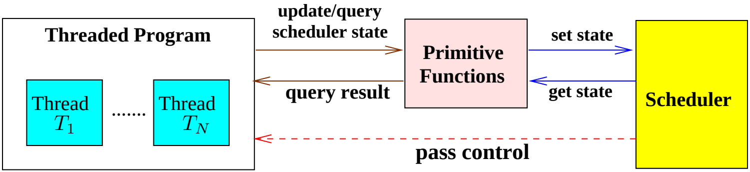
Cooperative Threaded Programs

- Threaded program: a set of sequential programs with shared variables

Cooperative Threaded Programs

- Threaded program: a set of sequential programs with shared variables
- Primitive functions (domain specific API):

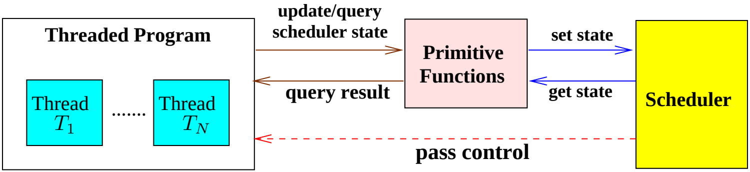
  `wait(EVENT_E), wait(100), notify(EVENT_J), ....`



Cooperative Threaded Programs

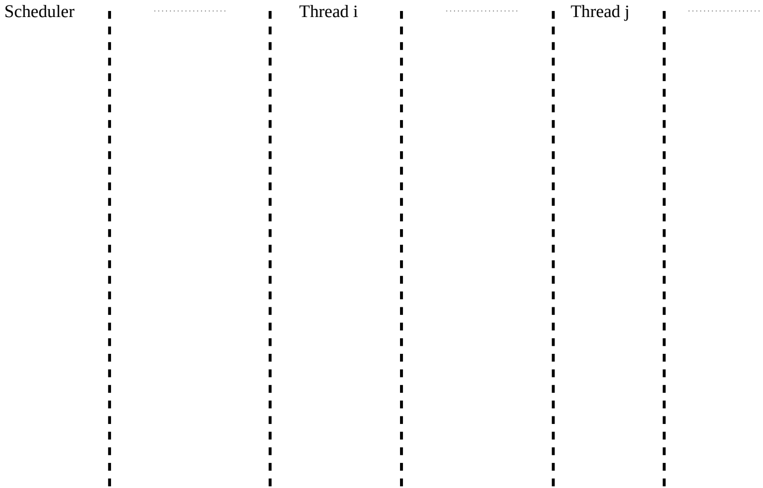
- Threaded program: a set of sequential programs with shared variables
- Primitive functions (domain specific API):

```
wait(EVENT_E), wait(100), notify(EVENT_J), ....
```

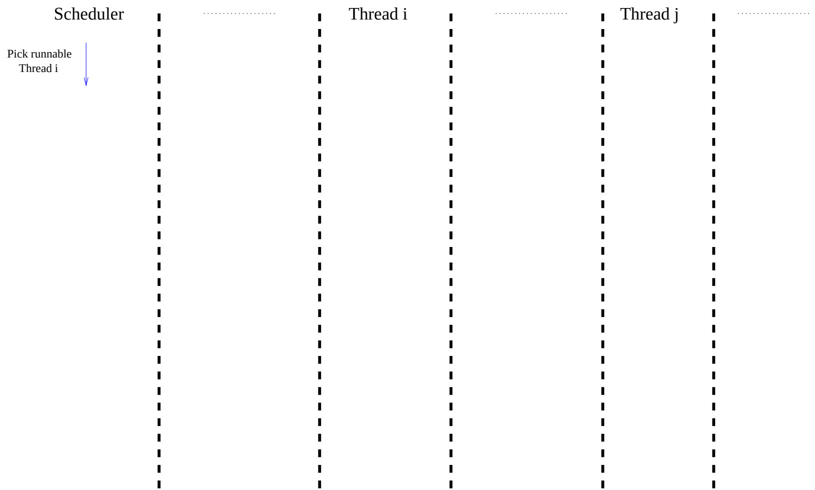


- Scheduler and primitive functions are left abstract, but exhibit cooperative scheduling with exclusive threads execution
  - Scheduler never preempts the running thread
  - At most one running thread at a time

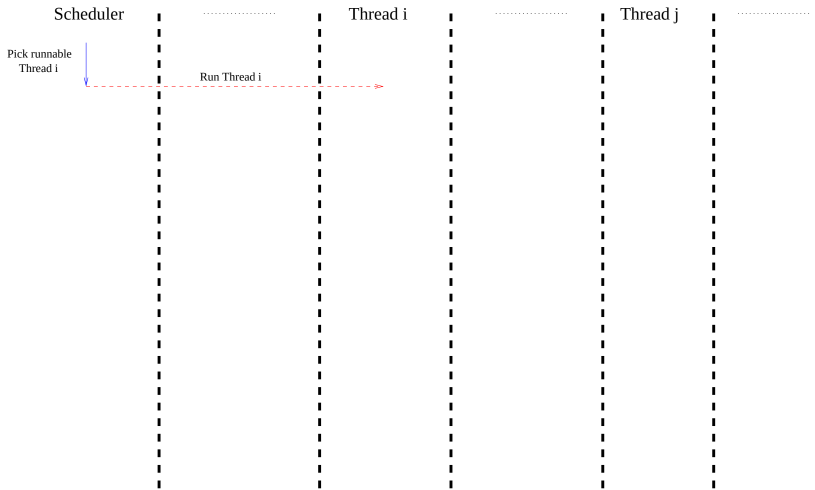
Dynamic View of Cooperative Threads



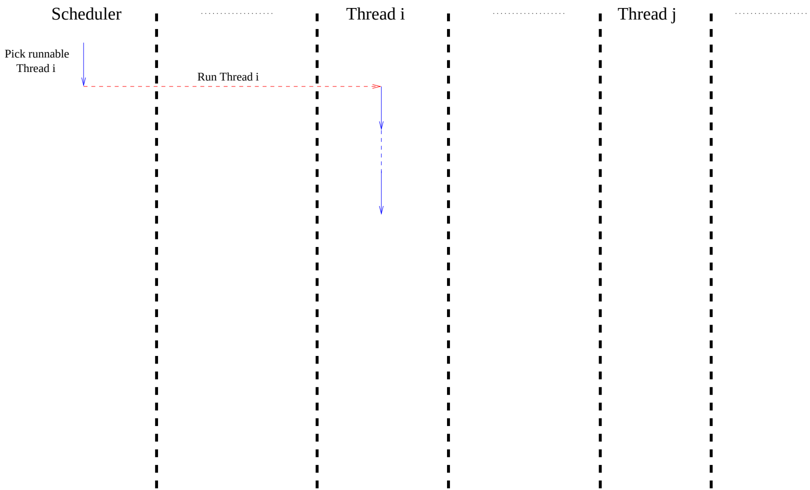
Dynamic View of Cooperative Threads



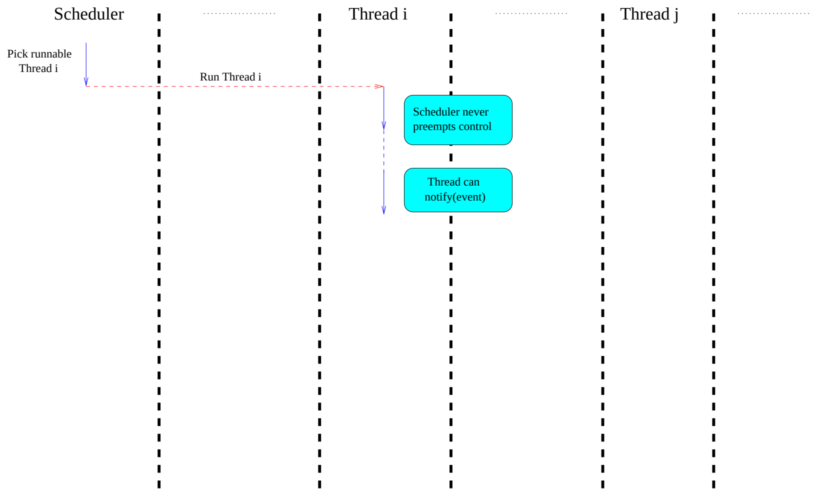
Dynamic View of Cooperative Threads



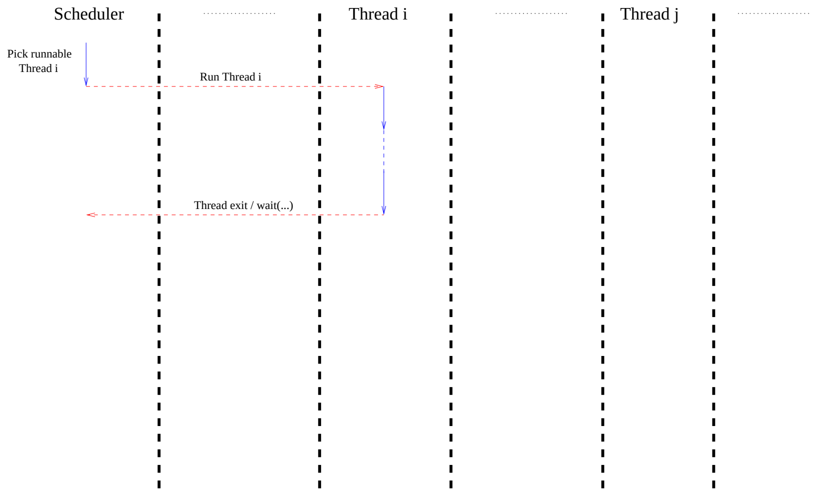
Dynamic View of Cooperative Threads



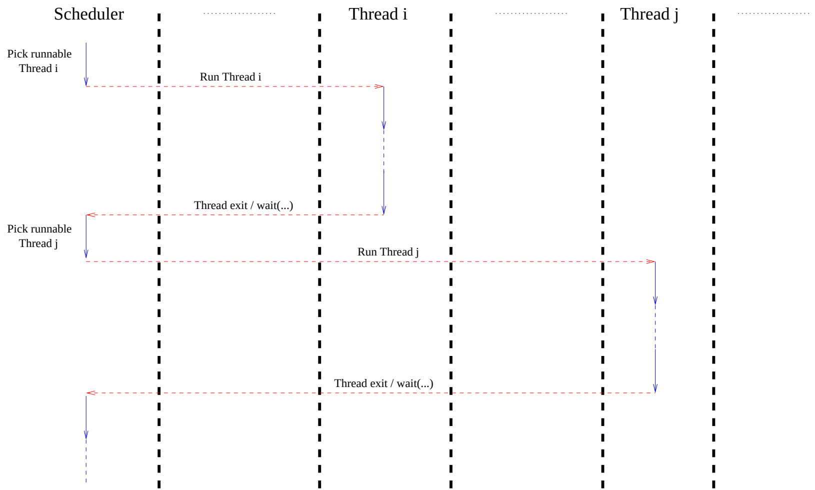
Dynamic View of Cooperative Threads



Dynamic View of Cooperative Threads



Dynamic View of Cooperative Threads



Sequential Program as CFG

Sequential program represented as a control-flow graph (CFG)

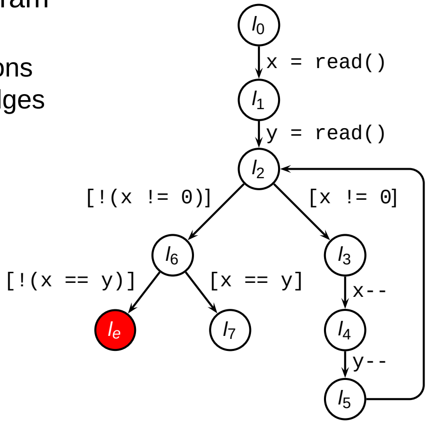
- A CFG for a sequential program P is a pair (L, G)
  - L: a set of program locations
  - $G \subseteq L \times Op \times L$: set of edges
  - l_0: unique entry location
  - l_e: error location
  - Op the set of operations

Sequential Program as CFG

Sequential program represented as a control-flow graph (CFG)

- A CFG for a sequential program P is a pair (L, G)
  - L: a set of program locations
  - $G \subseteq L \times Op \times L$: set of edges
  - l_0: unique entry location
  - l_e: error location
  - Op the set of operations

```
x = read();
y = read();
while (x != 0) {
   x--;
   y--;
}
assert( x == y );
```



Threads as CFGs

Each thread is represented as a control-flow-graph

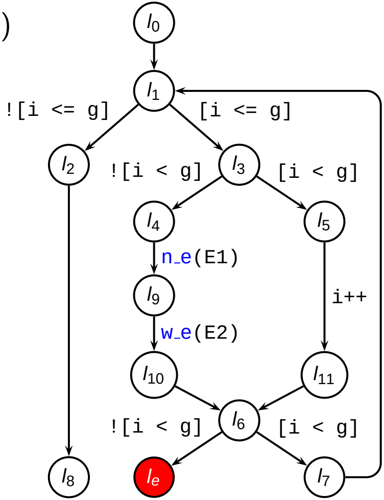
- A CFG for thread T is a pair (L, G)
  - L: a set of program locations
  - $G \subseteq L \times Op \times L$: set of edges
  - l_0: unique entry location
  - l_e: error location
  - Op set of operations, contains calls to primitive functions

Threads as CFGs

Each thread is represented as a control-flow-graph

- A CFG for thread T is a pair (L, G)
  - L: a set of program locations
  - $G \subseteq L \times Op \times L$: set of edges
  - l_0: unique entry location
  - l_e: error location
  - Op set of operations, contains calls to primitive functions

```
while (i <= g) {
  if (i < g) i++;
  else { // n_e and w_e are
         // primitive functions
    n_e(E1);
    w_e(E2); }
  assert(i < g);
}
```



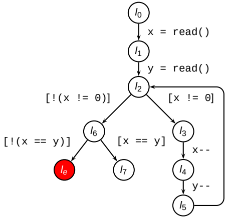
Outline

- Cooperative Threaded Programs (CTPs)
- Background
  - Safe Sequential Programs
  - Model Checking of Sequential Programs
    - Finite Model for Sequential Programs
    - Symbolic Model Checking of Sequential Programs
- Approaches to Model Checking of CTPs
  - Finite-Model for Cooperative Threaded Programs
  - Symbolic Model Checking of Sequential Software
  - Explicit Scheduler and Symbolic Threads (ESST)
- The Kratos Software Model Checker
- Experimental Results
- Related Work
- Conclusions

Safe Sequential Program

A sequential program is safe iff the error location is unreachable

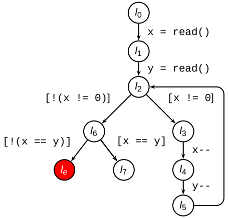
```
x = read();
y = read();
while (x != 0) {
   x--;
   y--;
}
assert( x == y );
```



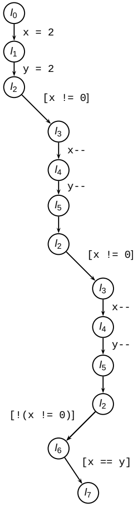
Safe Sequential Program

A sequential program is safe iff the error location is unreachable

```
x = read();
y = read();
while (x != 0) {
   x--;
   y--;
}
assert( x == y );
```



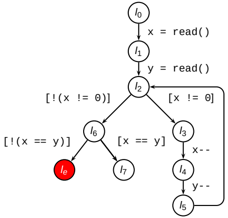
Safe: x = 2, y = 2



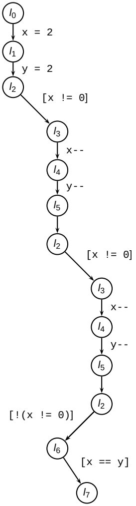
Safe Sequential Program

A sequential program is safe iff the error location is unreachable

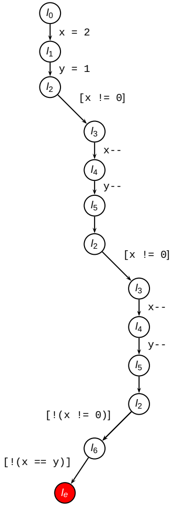
```
x = read();
y = read();
while (x != 0) {
   x--;
   y--;
}
assert( x == y );
```



Safe: x = 2, y = 2



Unsafe: x = 2, y = 1



Model Checking of Sequential Programs

- Finite Model for Sequential Programs
  - Explicit State Model Checking [Hol05]
- Symbolic Model Checking
  - Symbolic Bounded Model Checking of Software [CKL04]
  - Lazy Predicate Abstraction of Software [HJMS02]
  - Lazy Abstraction with Interpolants for Software [McM06]

Finite Model for Sequential Programs

Create a finite-model of the program:

- Decide inputs to be chosen over a finite range
- Fix bounds for memory and recursive calls

Perform verification with an explicit state model checker:

- The SPIN model checker [Hol05]
- The VERISOFT model checker [God05]
- …

Finite Model for Sequential Programs

Create a finite-model of the program:

- Decide inputs to be chosen over a finite range
- Fix bounds for memory and recursive calls

Perform verification with an explicit state model checker:

- The SPIN model checker [Hol05]
- The VERISOFT model checker [God05]
- ...

Comments:

- It is an under-approximation
  - The ranges for the inputs may hide bugs
- State explosion problem

Bounded Checking of Software

SAT Based Bounded Model Checking [BCC+03] effective in finding bugs in hardware designs

- Build a first order formula that represents a counter-example of length k for the property φ to verify

$$I(X_0) \wedge \bigwedge_{0}^{k-1} R(X_i, X_{i+1}) \wedge \neg\varphi(X_k)$$

- If the formula is satisfiable, then a bug has been found
  - Exploits effectiveness of SAT and SMT solvers
- Otherwise there might be a longer counterexample

Extends to software "trivially"

Bounded Model Checking For Software

- Fix a bound to loop unwinding
- Rewrite the program into single static assignment (SSA)
- Build a first order formula that represents the execution of the resulting program
  - The property to verify is the reachability of the error location
- Check satisfiability of the formula
  - If satisfiable, then a bug has been found
  - Otherwise there might be a bug for a longer unwinding of the loops

Example

```
x = read(); y = read();
while (x != 0) {
   x--; y--;
}
assert( x == y );
```

Example

```
x = read(); y = read();
while (x != 0) {
   x--; y--;
}
assert( x == y );
```

```
x = read(); y = read();
if (x != 0) { // loop 1
  x--; y--;
  if (x != 0) { // loop 2
    x--; y--;
  }
}
assert( x == y );
```

Example

```
x = read(); y = read();
while (x != 0) {
   x--; y--;
}
assert( x == y );
```

```
x = read(); y = read();
if (x != 0) { // loop 1
  x--; y--;
  if (x != 0) { // loop 2
    x--; y--;
  }
}
assert( x == y );
```

```
x0 = read(); y0 = read();
if (x0 != 0) { // loop 1
  x1 = x0-1; y1 = y0-1;
  if (x1 != 0) { // loop 2
    x2 = x1-1; y2 = y1-1;
  }
  x3 = (x1 != 0) ? x2 : x1;
  y3 = (x1 != 0) ? y2 : y1;
}
x4 = (x0 != 0) ? x3 : x0;
y4 = (x0 != 0) ? y3 : y0;
assert( x4 == y4 );
```

Example

```
x = read(); y = read();
while (x != 0) {
   x--; y--;
}
assert( x == y );
```

```
x = read(); y = read();
if (x != 0) { // loop 1
  x--; y--;
  if (x != 0) { // loop 2
    x--; y--;
  }
}
assert( x == y );
```

```
x0 = read(); y0 = read();
if (x0 != 0) { // loop 1
  x1 = x0-1; y1 = y0-1;
  if (x1 != 0) { // loop 2
    x2 = x1-1; y2 = y1-1;
  }
  x3 = (x1 != 0) ? x2 : x1;
  y3 = (x1 != 0) ? y2 : y1;
}
x4 = (x0 != 0) ? x3 : x0;
y4 = (x0 != 0) ? y3 : y0;
assert( x4 == y4 );
```

$$
\begin{array}{l}
x_0 = read_X \wedge y_0 = read_Y \wedge \\
x_0 \neq 0 \rightarrow (\\
\quad x_1 = x_0 - 1 \wedge y_1 = y_0 - 1 \wedge \\
\quad x_1 \neq 0 \rightarrow (\\
\quad\quad x_2 = x_1 - 1 \wedge y_2 = y_1 - 1 \\
\quad) \wedge \\
\quad x_1 \neq 0 \rightarrow (x_3 = x_2 \wedge y_3 = y_2) \wedge \\
\quad x_1 = 0 \rightarrow (x_3 = x_1 \wedge y_3 = y_1) \\
) \wedge \\
x_0 \neq 0 \rightarrow (x_4 = x_3 \wedge y_4 = y_3) \wedge \\
x_0 = 0 \rightarrow (x_4 = x_0 \wedge y_4 = y_0) \wedge \\
x_4 = y4
\end{array}
$$

Bounded Model Checking For Software

There are many tools:

- CBMC [CKL04]
- LLBMC [FMS13]
- ESBMC [CFMS12]
- ...

Bounded Model Checking For Software

There are many tools:

- CBMC [CKL04]
- LLBMC [FMS13]
- ESBMC [CFMS12]
- . . .

Comments

- This is an under-approximation: bound on loops
  - Checks whether loop-unwinding is enough can make the approach complete

Bounded Model Checking For Software

There are many tools:

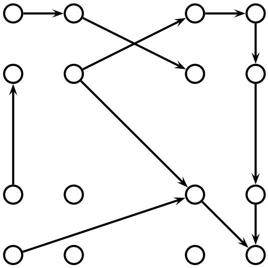
- CBMC [CKL04]
- LLBMC [FMS13]
- ESBMC [CFMS12]
- ...

Comments

- This is an under-approximation: bound on loops
  - Checks whether loop-unwinding is enough can make the approach complete
- State explosion
  - For some programs, the required unwinding is too large to be handled by state-of-the-art SAT/SMT solvers

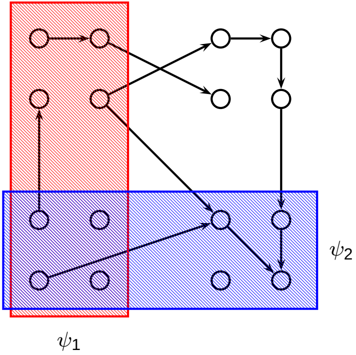
Predicate Abstraction

- A concrete program P over states S



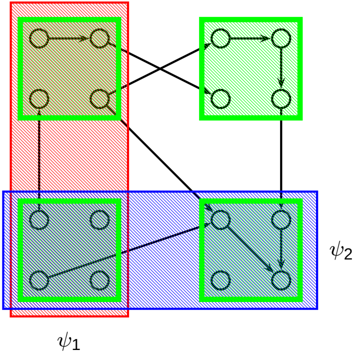
Predicate Abstraction

- A concrete program P over states S
- Predicates ψ_i induce partition over S



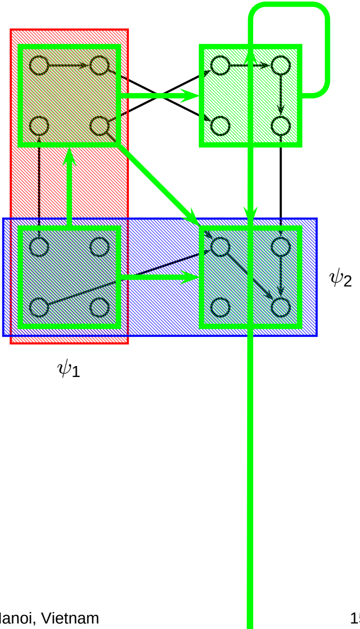
Predicate Abstraction

- A concrete program P over states S
- Predicates ψ_i induce partition over S
- Each partition is a state of the abstract program



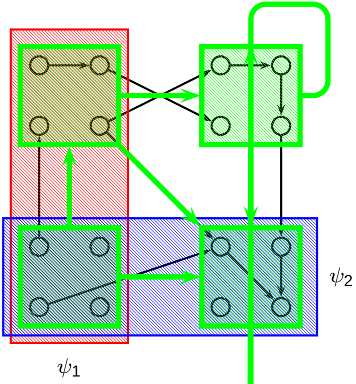
Predicate Abstraction

- A concrete program P over states S
- Predicates ψ_i induce partition over S
- Each partition is a state of the abstract program
- Transition in abstract space:
  - from as_0 to as_1 iff there is a transition from cs_0 to cs_1 with $cs_0 \in as_0$ and $cs_1 \in as_1$



Predicate Abstraction

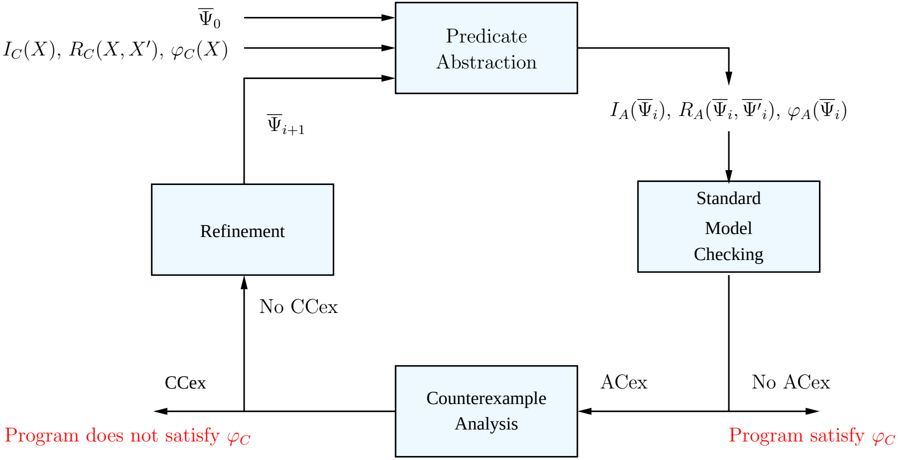
- A concrete program P over states S
- Predicates ψ_i induce partition over S
- Each partition is a state of the abstract program
- Transition in abstract space:
  - from as_0 to as_1 iff there is a transition from cs_0 to cs_1 with $cs_0 \in as_0$ and $cs_1 \in as_1$



$$R_A(\overline{\Psi}, \overline{\Psi}') = \exists X, X'.(R_C(X, X') \wedge \bigwedge_i (\overline{\psi_i} \leftrightarrow \psi(X) \wedge \overline{\psi_i}' \leftrightarrow \psi(X')))$$

Counter-Example Guided Abstraction Refinement

The Counter-Example Guided Abstraction Refinement (CEGAR) Loop



Lazy Predicate Abstraction of Software

On-the-fly construction of an abstract reachability tree ART with counterexample-guided abstraction refinement

- A node of an ART is a pair (q, φ)
  - q is a location of the CFG
  - φ is the reachable region representing a set of states
- Node expansion from $q \xrightarrow{op} q'$:
  - $(q, \varphi) \rightarrow (q', \varphi')$
  - $\varphi' = SP^{\pi}_{op}(\varphi)$
    - the strongest post-condition for operation *op* w.r.t. set of predicates π
- Node (q, φ) is covered by internal node (q, φ') iff $\varphi \Rightarrow \varphi'$
- ART is safe iff
  - error location l_e is not reachable
  - all the leaves are covered
- If the ART is safe then, the program is safe

Lazy Predicate Abstraction of Software

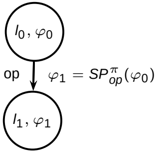
On-the-fly construction of an ART with CEGAR

1. Pick an ART node

Lazy Predicate Abstraction of Software

On-the-fly construction of an ART with CEGAR

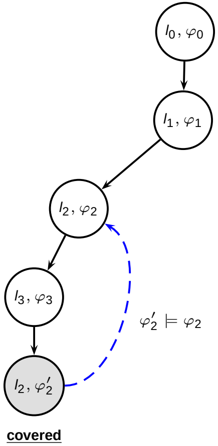
1. Pick an ART node
2. Compute abstract successors



Lazy Predicate Abstraction of Software

On-the-fly construction of an ART with CEGAR

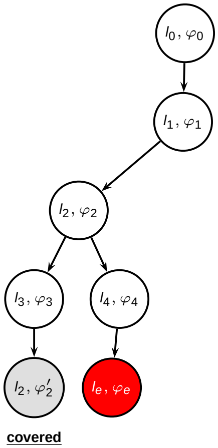
1. Pick an ART node
2. Compute abstract successors



Lazy Predicate Abstraction of Software

On-the-fly construction of an ART with CEGAR

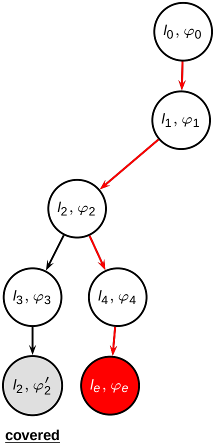
1. Pick an ART node
2. Compute abstract successors
3. If reach the error location: analyze path



Lazy Predicate Abstraction of Software

On-the-fly construction of an ART with CEGAR

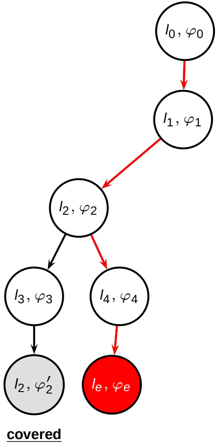
1. Pick an ART node
2. Compute abstract successors
3. If reach the error location: analyze path
   - If path is feasible $\Rightarrow$ program is unsafe



Lazy Predicate Abstraction of Software

On-the-fly construction of an ART with CEGAR

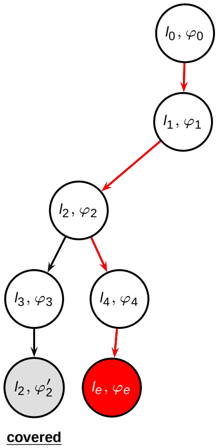
1. Pick an ART node
2. Compute abstract successors
3. If reach the error location: analyze path
   - If path is feasible $\Rightarrow$ program is unsafe
   - If path is spurious:



Lazy Predicate Abstraction of Software

On-the-fly construction of an ART with CEGAR

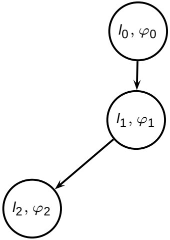
1. Pick an ART node
2. Compute abstract successors
3. If reach the error location:
   analyze path
   - If path is feasible $\Rightarrow$
     program is unsafe
   - If path is spurious:
     - Discover predicates to
       refine abstraction



Lazy Predicate Abstraction of Software

On-the-fly construction of an ART with CEGAR

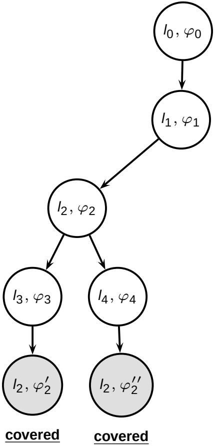
1. Pick an ART node
2. Compute abstract successors
3. If reach the error location: analyze path
   - If path is feasible $\Rightarrow$ program is unsafe
   - If path is spurious:
     - Discover predicates to refine abstraction
     - Undo part of ART



Lazy Predicate Abstraction of Software

On-the-fly construction of an ART with CEGAR

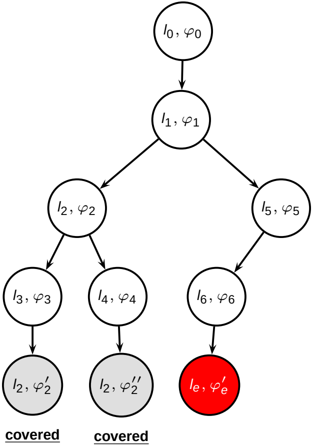
1. Pick an ART node
2. Compute abstract successors
3. If reach the error location: analyze path
   - If path is feasible $\Rightarrow$ program is unsafe
   - If path is spurious:
     - Discover predicates to refine abstraction
     - Undo part of ART
     - Goto 1 to reconstruct subtree



Lazy Predicate Abstraction of Software

On-the-fly construction of an ART with CEGAR

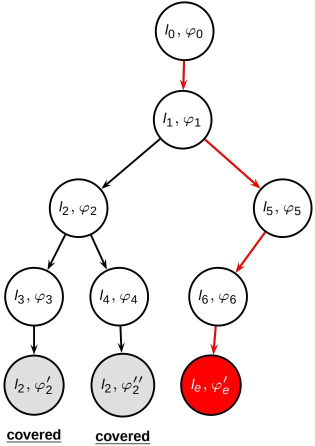
1. Pick an ART node
2. Compute abstract successors
3. If reach the error location: analyze path
   - If path is feasible $\Rightarrow$ program is unsafe
   - If path is spurious:
     - Discover predicates to refine abstraction
     - Undo part of ART
     - Goto 1 to reconstruct subtree



Lazy Predicate Abstraction of Software

On-the-fly construction of an ART with CEGAR

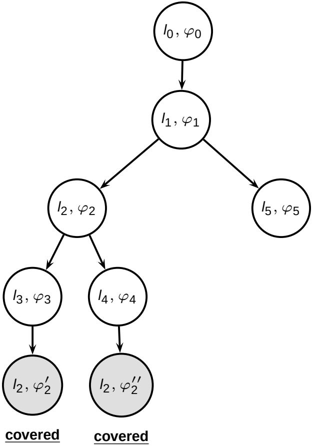
1. Pick an ART node
2. Compute abstract successors
3. If reach the error location: analyze path
    - If path is feasible $\Rightarrow$ program is unsafe
    - If path is spurious:
        - Discover predicates to refine abstraction
        - Undo part of ART
        - Goto 1 to reconstruct subtree



Lazy Predicate Abstraction of Software

On-the-fly construction of an ART with CEGAR

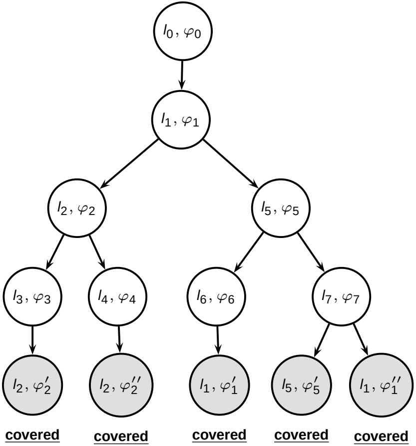
1. Pick an ART node
2. Compute abstract successors
3. If reach the error location: analyze path
   - If path is feasible $\Rightarrow$ program is unsafe
   - If path is spurious:
     - Discover predicates to refine abstraction
     - Undo part of ART
     - Goto 1 to reconstruct subtree



Lazy Predicate Abstraction of Software

On-the-fly construction of an ART with CEGAR

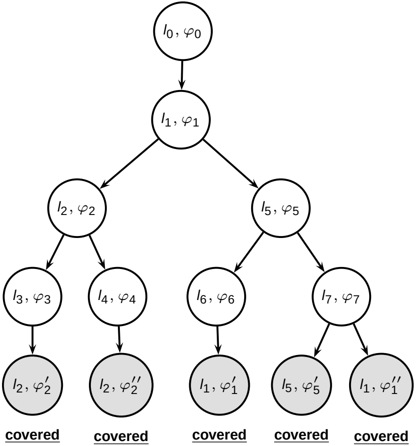
1. Pick an ART node
2. Compute abstract successors
3. If reach the error location: analyze path
   - If path is feasible $\Rightarrow$ program is unsafe
   - If path is spurious:
     - Discover predicates to refine abstraction
     - Undo part of ART
     - Goto 1 to reconstruct subtree



Lazy Predicate Abstraction of Software

On-the-fly construction of an ART with CEGAR

1. Pick an ART node
2. Compute abstract successors
3. If reach the error location: analyze path
   - If path is feasible $\Rightarrow$ program is unsafe
   - If path is spurious:
     - Discover predicates to refine abstraction
     - Undo part of ART
     - Goto 1 to reconstruct subtree
4. ART is safe $\Rightarrow$ program is safe



Key factors for Lazy Predicate Abstraction

- Computation of $SP^{\pi}_{op}(\varphi)$ expensive

Key factors for Lazy Predicate Abstraction

- Computation of $SP^{\pi}_{op}(\varphi)$ expensive
  - Advanced techniques for computation of $SP^{\pi}_{op}(\varphi)$
    - AllSMT [LNO06]
    - Structural Abstraction [CDJR09]
    - Hybrid Abstraction (BDD + SMT) [CCF+07, CFG+10]
    - QE techniques: Fourier-Motzkin [Sch98], Loos-Weispfenning [LW93, Mon08]

Key factors for Lazy Predicate Abstraction

- Computation of $SP^{\pi}_{op}(\varphi)$ expensive
  - Advanced techniques for computation of $SP^{\pi}_{op}(\varphi)$
    - AllSMT [LNO06]
    - Structural Abstraction [CDJR09]
    - Hybrid Abstraction (BDD + SMT) [CCF+07, CFG+10]
    - QE techniques: Fourier-Motzkin [Sch98], Loos-Weispfenning [LW93, Mon08]
  - Limit number of $SP^{\pi}_{op}(\varphi)$ computations
    - Large Block Encoding [BCG+09]
    - Adjustable Block Encoding [BKW10]

Key factors for Lazy Predicate Abstraction

- Computation of $SP^{\pi}_{op}(\varphi)$ expensive
  - Advanced techniques for computation of $SP^{\pi}_{op}(\varphi)$
    - AllSMT [LNO06]
    - Structural Abstraction [CDJR09]
    - Hybrid Abstraction (BDD + SMT) [CCF+07, CFG+10]
    - QE techniques: Fourier-Motzkin [Sch98], Loos-Weispfenning [LW93, Mon08]
  - Limit number of $SP^{\pi}_{op}(\varphi)$ computations
    - Large Block Encoding [BCG+09]
    - Adjustable Block Encoding [BKW10]
- Discovery of new predicates [BHJM07]:
  - Weakest Precondition
  - Unsatisfiable Core
  - Interpolants

Lazy Abstraction with Interpolants

Interpolants:

- Given Φ_1 and Φ_2 such that $\Phi_1 \wedge \Phi_2$ is unsatisfiable
- There exists an interpolant Ψ such that:
  - $\Phi_1 \Rightarrow \Psi$
  - $\Psi \wedge \Phi_2$ is unsatisfiable
  - $\Psi \in \mathcal{L}(\Phi_1) \cap \mathcal{L}(\Phi_2)$

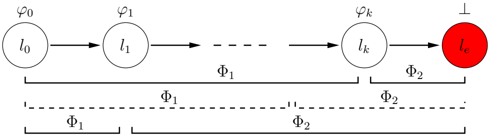
Lazy Abstraction with Interpolants

Interpolants:

- Given Φ_1 and Φ_2 such that $\Phi_1 \wedge \Phi_2$ is unsatisfiable
- There exists an interpolant Ψ such that:
  - $\Phi_1 \Rightarrow \Psi$
  - $\Psi \wedge \Phi_2$ is unsatisfiable
  - $\Psi \in \mathcal{L}(\Phi_1) \cap \mathcal{L}(\Phi_2)$

Lazy abstraction with interpolation:

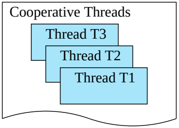
- Similar in spirit to lazy-predicate abstraction
- Avoids computation of $SP_{op}^{\pi}(\varphi)$ by over-approximating reachability regions using interpolants
  - Reachability region of error location set to $\perp$
  - Refine reachability regions on the path using interpolants



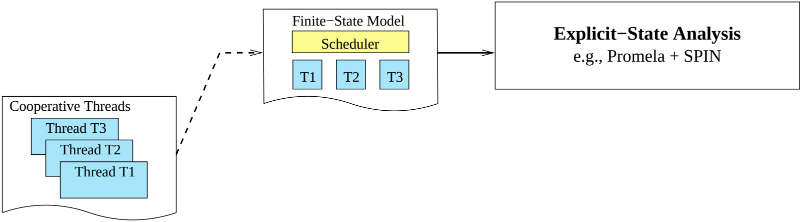
Outline

- Cooperative Threaded Programs (CTPs)
- Background
  - Safe Sequential Programs
  - Model Checking of Sequential Programs
    - Finite Model for Sequential Programs
    - Symbolic Model Checking of Sequential Programs
- Approaches to Model Checking of CTPs
  - Finite-Model for Cooperative Threaded Programs
  - Symbolic Model Checking of Sequential Software
  - Explicit Scheduler and Symbolic Threads (ESST)
- The Kratos Software Model Checker
- Experimental Results
- Related Work
- Conclusions

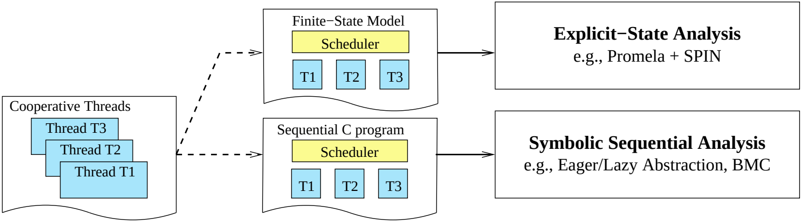
Approaches to Model Checking of CTPs



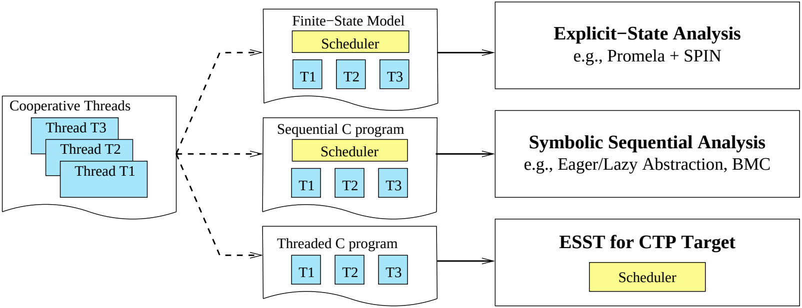
Approaches to Model Checking of CTPs



Approaches to Model Checking of CTPs



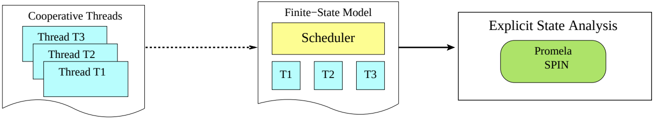
Approaches to Model Checking of CTPs



Finite-Model for Cooperative Threaded Programs

Translate cooperative threads into a Finite-State Model

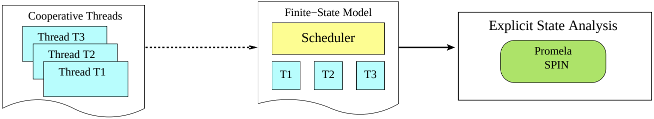
- e.g. Promela [Hol05]



Finite-Model for Cooperative Threaded Programs

Translate cooperative threads into a Finite-State Model

- e.g. Promela [Hol05]



Analysis can be done with Explicit State Model Checker

- e.g, SPIN Model Checker [Hol05]

Finite-Model for Cooperative Threaded Programs

- Input reading encoded as a function that selects non-deterministically a value from a finite-set of values

Finite-Model for Cooperative Threaded Programs

- Input reading encoded as a function that selects non-deterministically a value from a finite-set of values
- Encode thread communication primitives opening their definition

Finite-Model for Cooperative Threaded Programs

- Input reading encoded as a function that selects non-deterministically a value from a finite-set of values
- Encode thread communication primitives opening their definition
- Encode `Scheduler` as a process

Finite-Model for Cooperative Threaded Programs

- Input reading encoded as a function that selects non-deterministically a value from a finite-set of values
- Encode thread communication primitives opening their definition
- Encode `Scheduler` as a process
- Encode thread body as functions

Finite-Model for Cooperative Threaded Programs

- Input reading encoded as a function that selects non-deterministically a value from a finite-set of values
- Encode thread communication primitives opening their definition
- Encode `Scheduler` as a process
- Encode thread body as functions
  - Thread suspension as function thread_suspend()
    - Implementation varies from the encodings

```
wait(...);  ⟹      thread_state = WAITING;
                    thread_pc = NEXT_LOC;
                    global = local;
                    thread_suspend();
                  NEXT_LOC_LABEL:
                    local = global;
```

```
inline thread_body() {
  if
   :: (thread_pc == NEXT_LOC) ->
                    goto NEXT_LOC_LABEL;
   :: ...
   :: else -> skip;
  fi
  /** Thread body **/
}
```

Finite-Model for Cooperative Threaded Programs

- Input reading encoded as a function that selects non-deterministically a value from a finite-set of values
- Encode thread communication primitives opening their definition
- Encode `Scheduler` as a process
- Encode thread body as functions
  - Thread suspension as function thread_suspend()
    - Implementation varies from the encodings

```
                    thread_state = WAITING;
                    thread_pc = NEXT_LOC;
wait(...);  ==>     global = local;
                    thread_suspend();
                  NEXT_LOC_LABEL:
                    local = global;
```

```
inline thread_body() {
  if
   :: (thread_pc == NEXT_LOC) ->
                  goto NEXT_LOC_LABEL;
   :: ...
   :: else -> skip;
  fi
  /** Thread body **/
}
```

[CCNR11, CNR13] shows finite-model for SystemC designs

Encoding of primitive functions

Channel update

```
#define ITE(C,T,E) { if ::C -> T; ::else -> E; fi }

inline p_to_c_update() {
 ITE(p_to_c_new != p_to_c_old,
     p_to_c_old = p_to_c_new; e_p_to_c = NOTIFIED_DELTA, skip)
}
```

Encoding of primitive functions

Channel update

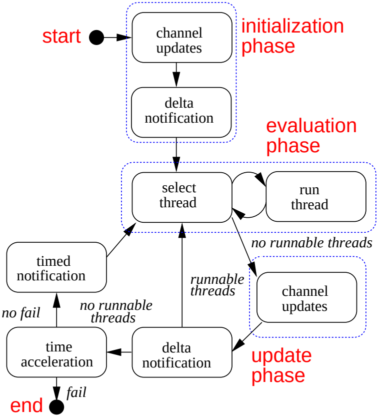
```
#define ITE(C,T,E) { if ::C -> T; ::else -> E; fi }

inline p_to_c_update() {
 ITE(p_to_c_new != p_to_c_old,
     p_to_c_old = p_to_c_new; e_p_to_c = NOTIFIED_DELTA, skip)
}
```

Event Notification

```
bool p_write_notified, p_read_notified, c_read_and_ack_notified;

inline is_p_write_notified(notified) {
  ITE(((p_write_pc == wait_1 && e_p_write_state == NOTIFIED) ||
       (p_write_pc == wait_2 && e_state         == NOTIFIED)),
       notified = true, notified = false);
}

inline notify_threads() {
  is_p_write_notified(p_write_notified);
  ITE(p_write_notified, p_write_state = RUNNABLE, skip);
  is_p_read_notified(p_read_notified);
  ITE(p_read_notified, p_read_state = RUNNABLE, skip);
  is_c_read_and_ack_notified(c_read_and_ack_notified);
  ITE(c_read_and_ack_notified, c_read_and_ack_state = RUNNABLE, skip);
}

inline e_notify() {
  e_state = NOTIFIED; notify_threads(); e_state = NONE;
}
```

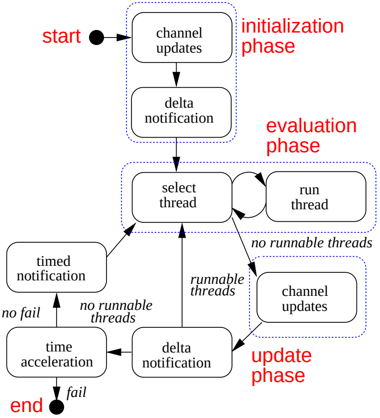
Encoding of the scheduler

For SystemC



Encoding of the scheduler

For SystemC



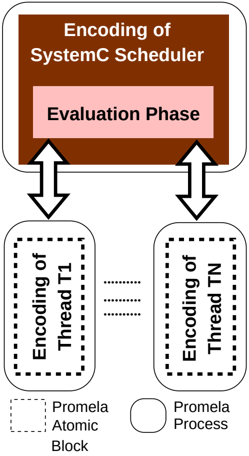

```
proctype Scheduler() {
 bool runnable, accelerated;
 channel_updates();
 delta_notification();
start_DeltaCycle:
 exists_runnable_threads(runnable);
 ITE(!runnable,goto TimedNotification,skip);
 evaluation_phase();
 channel_updates();
progress_DeltaCycle:
 delta_notification();
 ITE(runnable,goto start_DeltaCycle,skip);
TimedNotification:
 time_acceleration(accelerated);
 ITE(accelerated,goto SchedulerExit,skip);
 timed_notification();
 goto start_DeltaCycle;
SchedulerExit:
}
```

Finite-Model for Cooperative Threaded Programs (II)

- Encode threads and thread suspension/resume depending on the encoding of the synchronization thread-scheduler
  - Thread-To-Process
  - Thread-To-Atomic-Block
  - One-Atomic-Block

Thread-To-Process Encoding

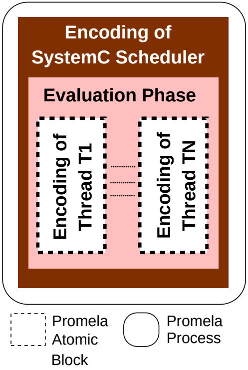
Threads and scheduler as separate processes, synchronization scheduler-threads through token exchange on a rendezvous channel




```
inline thread_suspend () {thread_chan!TK;
                          thread_chan?TK;}
active proctype thread_1() {
 thread_1_entry:
  atomic { thread_1_chan?TK; thread_1_body(); }
 thread_1_exit:
  goto thread_1_entry;
}
inline evaluation_phase() {
 do
  :: thread_1_state == RUNNABLE ->
     thread_1_state = RUNNING;
     thread_1_chan!TK; thread_1_chan?TK;
  ...
  :: thread_N_state == RUNNABLE ->
     thread_N_state = RUNNING;
     thread_N_chan!TK; thread_N_chan?TK;
  :: else -> break;
 od;
}
```

Thread-To-Atomic-Block Encoding

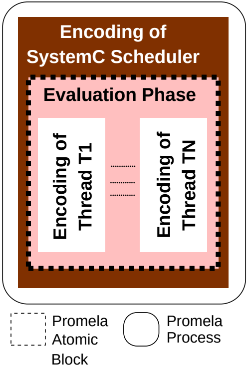
Threads and scheduler embedded in a unique process, thread suspension through jump to the exit location, no need of the rendezvous channel. Each thread body enclosed in an atomic block




```
inline thread_1() {
  atomic {  thread_1_body();  }
 thread_1_exit:
  skip;
}

inline evaluation_phase() {
 do
   :: thread_1_state == RUNNABLE ->
       thread_1_state = RUNNING;  thread_1();
   ...
   :: thread_N_state == RUNNABLE ->
       thread_N__state = RUNNING;  thread_N();
   :: else -> break;
 od;
}
```

One-Atomic-Block Encoding

Derived from Thread-To-Atomic-Block enclosing whole evaluation phase into an atomic block



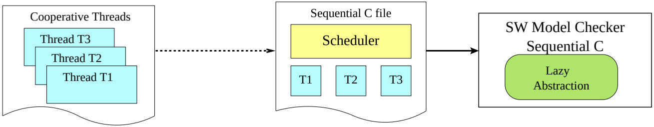

```
proctype scheduler() {
  ...
  atomic {
    evaluation_phase();
  }
  ...
}
```

Limitations of Finite State Model for CTPs

- Under-approximation
  - There might be different inputs for which the property is violated
- Partial Order Reduction (POR) within model checker can be ineffective
  - POR should be carried out at the level of the Scheduler
  - Explicit state model checkers (e.g. SPIN) do POR at process level
  - Useful domain information lost in the encoding
- State explosion

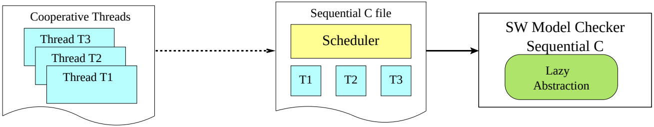
Symbolic Model Checking of Sequential Software

Translate cooperative threads into sequential C program



Symbolic Model Checking of Sequential Software

Translate cooperative threads into sequential C program



Analysis can be based on:

- Bounded Model Checking [CKL04]
- Lazy Predicate Abstraction [HJMS02]
- Lazy Abstraction with Interpolants [McM06]

Sequentializing Cooperative Threads

- Encode each thread as a function

Sequentializing Cooperative Threads

- Encode each thread as a function
  - Thread suspension as function return:

```
wait(...);   ⟹     thread_state = WAITING;
                    thread_pc = NEXT_LOC;
                    global = local;
                    return;
                  NEXT_LOC_LABEL:
                    local = global;
```

```
void thread() {
  if (thread_pc == NEXT_LOC)
    goto NEXT_LOC_LABEL;

  /** Thread body **/
}
```

Sequentializing Cooperative Threads

- Encode each thread as a function
  - Thread suspension as function return:

```
wait(...);    ⟹    thread_state = WAITING;
                   thread_pc = NEXT_LOC;
                   global = local;
                   return;
                 NEXT_LOC_LABEL:
                   local = global;
```

```
void thread() {
  if (thread_pc == NEXT_LOC)
    goto NEXT_LOC_LABEL;

  /** Thread body **/
}
```

- Encode thread communication primitives opening their definition

Sequentializing Cooperative Threads

- Encode each thread as a function
  - Thread suspension as function return:

```
wait(...);   ⟹     thread_state = WAITING;
                    thread_pc = NEXT_LOC;
                    global = local;
                    return;
                  NEXT_LOC_LABEL:
                    local = global;
```

```
void thread() {
  if (thread_pc == NEXT_LOC)
    goto NEXT_LOC_LABEL;

  /** Thread body **/
}
```

- Encode thread communication primitives opening their definition
- Encode `Scheduler` as a function:

Sequentializing Cooperative Threads

- Encode each thread as a function
  - Thread suspension as function return:

```
wait(...);   ⟹     thread_state = WAITING;
                   thread_pc = NEXT_LOC;
                   global = local;
                   return;
                 NEXT_LOC_LABEL:
                   local = global;
```

```
void thread() {
  if (thread_pc == NEXT_LOC)
    goto NEXT_LOC_LABEL;

  /** Thread body **/
}
```

- Encode thread communication primitives opening their definition
- Encode `Scheduler` as a function:
  - Must allow for exploring all possible thread interleavings:

```
while ( exists_runnable_thread() ) {
  if ( thread_i_state == RUNNABLE && nondet() )
    thread_i();
  ...
  if ( thread_j_state == RUNNABLE && nondet() )
    thread_j();
}
```

Sequentializing Cooperative Threads

- Encode each thread as a function
  - Thread suspension as function return:

```
wait(...);   ⟹     thread_state = WAITING;
                   thread_pc = NEXT_LOC;
                   global = local;
                   return;
                 NEXT_LOC_LABEL:
                   local = global;
```

```
void thread() {
  if (thread_pc == NEXT_LOC)
    goto NEXT_LOC_LABEL;

  /** Thread body **/
}
```

- Encode thread communication primitives opening their definition
- Encode `Scheduler` as a function:
  - Must allow for exploring all possible thread interleavings:

```
while ( exists_runnable_thread() ) {
  if ( thread_i_state == RUNNABLE && nondet() )
    thread_i();
  ...
  if ( thread_j_state == RUNNABLE && nondet() )
    thread_j();
}
```

[CMNR10, CNR13] shows sequentialization for SystemC

Limitations of Sequentialization for CTPs

- Bounded Model Checking requires too deep analysis and often blows up even on small programs

Limitations of Sequentialization for CTPs

- Bounded Model Checking requires too deep analysis and often blows up even on small programs
- Initial abstractions are often too aggressive
  - Many refinements are needed to recover details of models

Limitations of Sequentialization for CTPs

- Bounded Model Checking requires too deep analysis and often blows up even on small programs
- Initial abstractions are often too aggressive
  - Many refinements are needed to recover details of models
- Precise scheduler and its states are often needed:

Limitations of Sequentialization for CTPs

- Bounded Model Checking requires too deep analysis and often blows up even on small programs
- Initial abstractions are often too aggressive
  - Many refinements are needed to recover details of models
- Precise scheduler and its states are often needed:
  - Lazy predicate abstraction
    - Need to keep track of predicates
      `thread_state == WAITING`,
      `thread_state == RUNNABLE`
      for every thread

Limitations of Sequentialization for CTPs

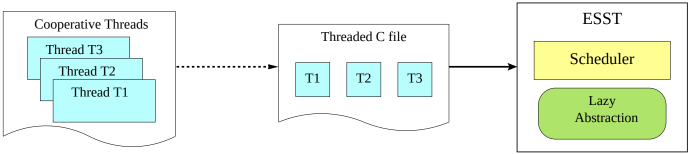
- Bounded Model Checking requires too deep analysis and often blows up even on small programs
- Initial abstractions are often too aggressive
  - Many refinements are needed to recover details of models
- Precise scheduler and its states are often needed:
  - Lazy predicate abstraction
    - Need to keep track of predicates
      `thread_state == WAITING`,
      `thread_state == RUNNABLE`
      for every thread
    - The more predicates to keep track, the more expensive the abstractions

Limitations of Sequentialization for CTPs

- Bounded Model Checking requires too deep analysis and often blows up even on small programs
- Initial abstractions are often too aggressive
  - Many refinements are needed to recover details of models
- Precise scheduler and its states are often needed:
  - Lazy predicate abstraction
    - Need to keep track of predicates `thread_state == WAITING`, `thread_state == RUNNABLE` for every thread
    - The more predicates to keep track, the more expensive the abstractions
  - Lazy abstraction with interpolants
    - Slow convergence
    - Large interpolants

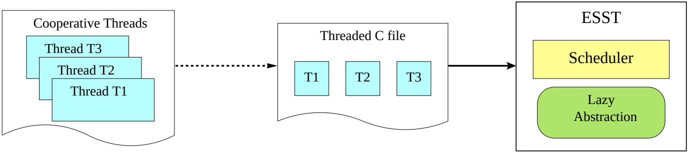
Model Checking with ESST

Explicit-Scheduler Symbolic-Thread (ESST) algorithm



Model Checking with ESST

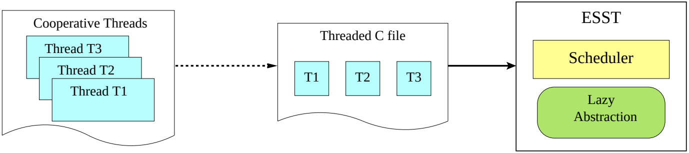
Explicit-Scheduler Symbolic-Thread (ESST) algorithm



In a nutshell . . .

Model Checking with ESST

Explicit-Scheduler Symbolic-Thread (ESST) algorithm

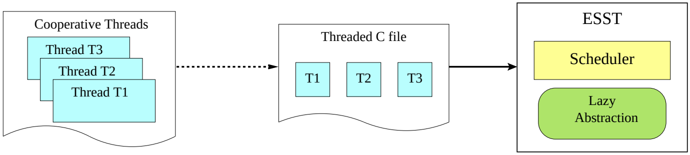


In a nutshell . . .

- Analyze threads symbolically using lazy predicate abstraction.

Model Checking with ESST

Explicit-Scheduler Symbolic-Thread (ESST) algorithm

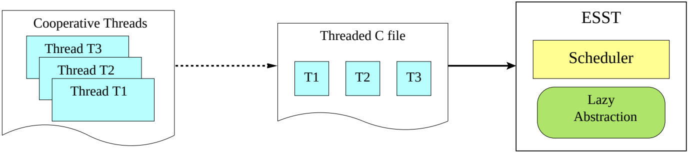


In a nutshell . . .

- Analyze threads symbolically using lazy predicate abstraction.
- Analyze scheduler using explicit-state techniques:

Model Checking with ESST

Explicit-Scheduler Symbolic-Thread (ESST) algorithm

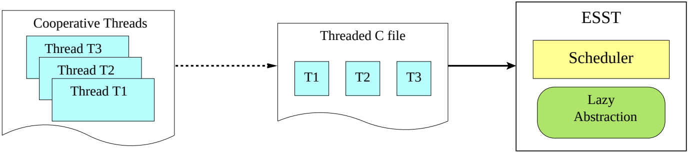


In a nutshell . . .

- Analyze threads symbolically using lazy predicate abstraction.
- Analyze scheduler using explicit-state techniques:
  - Keep track of the scheduler states explicitly

Model Checking with ESST

Explicit-Scheduler Symbolic-Thread (ESST) algorithm

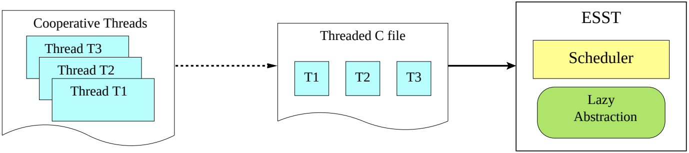


In a nutshell . . .

- Analyze threads symbolically using lazy predicate abstraction.
- Analyze scheduler using explicit-state techniques:
  - Keep track of the scheduler states explicitly
- Scheduler is part of the model-checking algorithm

Model Checking with ESST

Explicit-Scheduler Symbolic-Thread (ESST) algorithm



In a nutshell . . .

- Analyze threads symbolically using lazy predicate abstraction.
- Analyze scheduler using explicit-state techniques:
  - Keep track of the scheduler states explicitly
- Scheduler is part of the model-checking algorithm

[CMNR10, CNR13] shows ESST for SystemC, [CNR12a] for FairThreads

Abstract Reachability Forest (ARF)

- An abstract reachability forest (ARF) consists of connected abstract reachability trees ART’s
  - Each ART is obtained by unwinding the CFG of running thread

Abstract Reachability Forest (ARF)

- An abstract reachability forest (ARF) consists of connected abstract reachability trees ART's
  - Each ART is obtained by unwinding the CFG of running thread
- An ARF node with N threads:

$$(\langle l_1, \varphi_1 \rangle, \ldots, \langle l_N, \varphi_N \rangle, \varphi, S)$$

  - $\langle l_i, \varphi_i \rangle$ where l_i CFG location and φ_i region of thread i
  - φ is global region (e.g., for shared variables)
  - S is scheduler state: mapping from variables to values

Primitive Executor

- Primitive executor

$$\textsc{SExec} : \mathit{SchedulerState} \times \mathit{PrimitiveCall} \to \mathit{SchedulerState}$$

Example:

$S' = \textsc{SExec}(S, \texttt{wait\_event(e)})$, such that

$$S' = S[t_{state} \mapsto \mathit{WAITING}, t_{event} \mapsto \mathrm{e}]$$

Scheduler

- Scheduler

$$\textsc{Sched} : \mathit{SchedulerState} \rightarrow \mathcal{P}(\mathit{SchedulerState})$$

$\{S_1, \ldots, S_m\} = \textsc{Sched}(S)$

  - No running thread in S
  - Each S_i for $i = 1, \ldots, m$ has exactly one running thread

ESST Algorithm: ARF Construction

Computing successor nodes involves:

- Computing abstract strongest post-condition *SP*
- Executing primitive functions
- Running the scheduler

ESST Algorithm: ARF Analysis

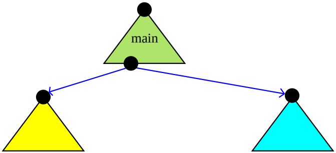
On-the-fly construction of an ARF with CEGAR

1. Pick an ARF node

ESST Algorithm: ARF Analysis

On-the-fly construction of an ARF with CEGAR

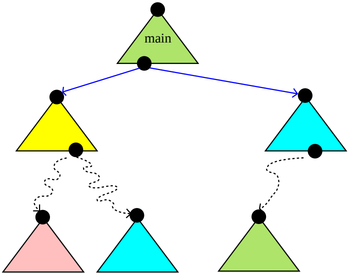
1. Pick an ARF node
2. Compute abstract successors



ESST Algorithm: ARF Analysis

On-the-fly construction of an ARF with CEGAR

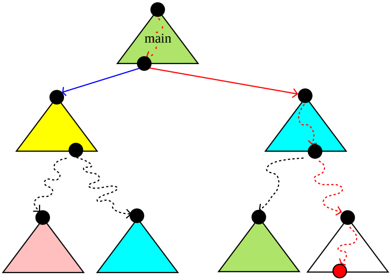
1. Pick an ARF node
2. Compute abstract successors



ESST Algorithm: ARF Analysis

On-the-fly construction of an ARF with CEGAR

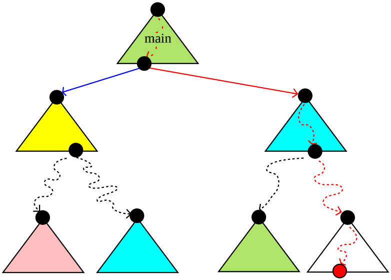
1. Pick an ARF node
2. Compute abstract successors
3. If reach the error location: analyze path



ESST Algorithm: ARF Analysis

On-the-fly construction of an ARF with CEGAR

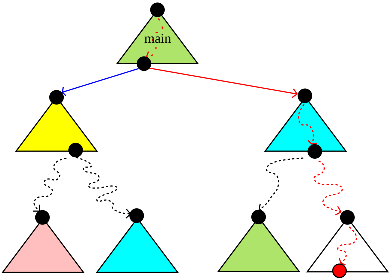
1. Pick an ARF node
2. Compute abstract successors
3. If reach the error location: analyze path
   - If path is feasible $\Rightarrow$ program is unsafe



ESST Algorithm: ARF Analysis

On-the-fly construction of an ARF with CEGAR

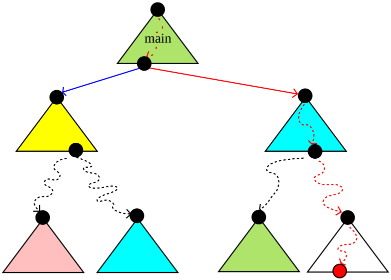
1. Pick an ARF node
2. Compute abstract successors
3. If reach the error location: analyze path
   - If path is feasible $\Rightarrow$ program is unsafe
   - If path is spurious:



ESST Algorithm: ARF Analysis

On-the-fly construction of an ARF with CEGAR

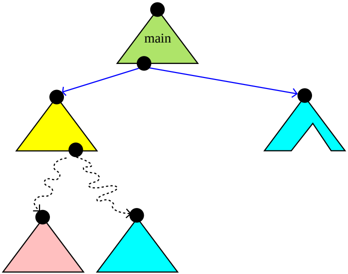
1. Pick an ARF node
2. Compute abstract successors
3. If reach the error location: analyze path
   - If path is feasible $\Rightarrow$ program is unsafe
   - If path is spurious:
     - Discover predicates to refine abstraction



ESST Algorithm: ARF Analysis

On-the-fly construction of an ARF with CEGAR

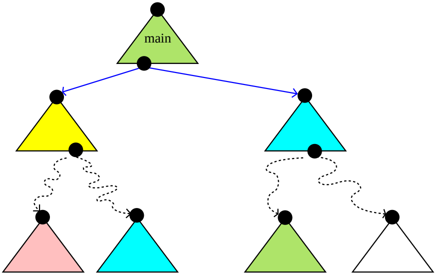
1. Pick an ARF node
2. Compute abstract successors
3. If reach the error location: analyze path
   - If path is feasible $\Rightarrow$ program is unsafe
   - If path is spurious:
     - Discover predicates to refine abstraction
     - Undo part of ARF



ESST Algorithm: ARF Analysis

On-the-fly construction of an ARF with CEGAR

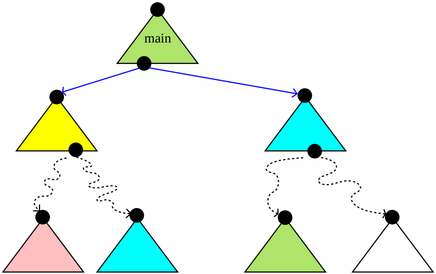
1. Pick an ARF node
2. Compute abstract successors
3. If reach the error location: analyze path
    - If path is feasible $\Rightarrow$ program is unsafe
    - If path is spurious:
        - Discover predicates to refine abstraction
        - Undo part of ARF
        - Goto 1 to reconstruct ARF



ESST Algorithm: ARF Analysis

On-the-fly construction of an ARF with CEGAR

1. Pick an ARF node
2. Compute abstract successors
3. If reach the error location: analyze path
   - If path is feasible $\Rightarrow$ program is unsafe
   - If path is spurious:
     - Discover predicates to refine abstraction
     - Undo part of ARF
     - Goto 1 to reconstruct ARF
4. ARF is safe $\Rightarrow$ program is safe



ESST Algorithm: ARF Construction (Rule 1)

Thread i is the running thread in S, and op in the CFA edge (l_i, op, l'_i) is not a primitive function call

$$(\langle l_1, \varphi_1 \rangle, \quad \ldots \quad \langle l_i, \varphi_i \rangle, \quad \ldots \quad \langle l_n, \varphi_n \rangle, \quad \varphi, \quad S)$$

ESST Algorithm: ARF Construction (Rule 1)

Thread i is the running thread in S, and op in the CFA edge (l_i, op, l_i') is not a primitive function call

$$
\begin{array}{ccccccc}
(\langle l_1, \varphi_1 \rangle, & \ldots & \langle l_i, \varphi_i \rangle, & \ldots & \langle l_n, \varphi_n \rangle, & \varphi, & S) \\
 & & \Big\downarrow SP^{\pi_i}_{op}(\psi_i) & & & & \\
 & & \langle l_i', \varphi_i' \rangle, & & & &
\end{array}
$$

- $\psi_i \Leftrightarrow \varphi_i \wedge \varphi$

ESST Algorithm: ARF Construction (Rule 1)

Thread i is the running thread in S, and op in the CFA edge (l_i, op, l'_i) is not a primitive function call

$$
\begin{array}{ccccccc}
(\langle l_1, \varphi_1 \rangle, & \ldots & \langle l_i, \varphi_i \rangle, & \ldots & \langle l_n, \varphi_n \rangle, & \varphi, & S) \\
 & & \Big\downarrow SP^{\pi_i}_{op}(\psi_i) & & & \Big\downarrow SP^{\pi}_{op}(\varphi) & \\
 & & \langle l'_i, \varphi'_i \rangle, & & & \varphi', &
\end{array}
$$

- $\psi_i \Leftrightarrow \varphi_i \wedge \varphi$

ESST Algorithm: ARF Construction (Rule 1)

Thread i is the running thread in S, and op in the CFA edge (l_i, op, l'_i) is not a primitive function call

$$
\begin{array}{ccccccccc}
(\langle l_1, \varphi_1 \rangle, & \ldots & \langle l_i, \varphi_i \rangle, & \ldots & \langle l_n, \varphi_n \rangle, & \varphi, & S) \\
\downarrow SP^{\pi_1}_{\text{HAVOC}(op)}(\psi_1) & & \downarrow SP^{\pi_i}_{op}(\psi_i) & & \downarrow SP^{\pi_n}_{\text{HAVOC}(op)}(\psi_n) & \downarrow SP^{\pi}_{op}(\varphi) & \\
(\langle l_1, \varphi'_1 \rangle, & \ldots & \langle l'_i, \varphi'_i \rangle, & \ldots & \langle l_n, \varphi'_n \rangle, & \varphi', &
\end{array}
$$

- $\psi_i \Leftrightarrow \varphi_i \wedge \varphi$
- Havoc HAVOC $(g := e) = (g := f)$
  - g is a global variable
  - f is a fresh variable

ESST Algorithm: ARF Construction (Rule 1)

Thread i is the running thread in S, and op in the CFA edge (l_i, op, l'_i) is not a primitive function call

$$
\begin{tikzcd}
(\langle l_1, \varphi_1 \rangle, \arrow[d, "SP^{\pi_1}_{\textsc{havoc}(op)}(\psi_1)"] & \ldots & \langle l_i, \varphi_i \rangle, \arrow[d, "SP^{\pi_i}_{op}(\psi_i)"] & \ldots & \langle l_n, \varphi_n \rangle, \arrow[d, "SP^{\pi_n}_{\textsc{havoc}(op)}(\psi_n)"] & \varphi, \arrow[d, "SP^{\pi}_{op}(\varphi)"] & S) \arrow[d] \\
(\langle l_1, \varphi'_1 \rangle, & \ldots & \langle l'_i, \varphi'_i \rangle, & \ldots & \langle l_n, \varphi'_n \rangle, & \varphi', & S)
\end{tikzcd}
$$

- $\psi_i \Leftrightarrow \varphi_i \wedge \varphi$
- Havoc $\textsc{havoc}(\mathtt{g} := e) = (\mathtt{g} := f)$
  - g is a global variable
  - f is a fresh variable

ESST Algorithm: ARF Construction (Rule 2)

Thread i is the running thread in S, and op in the CFA edge (l_i, op, l_i') is ~~not~~ a primitive function call.

$$(\langle l_1, \varphi_1 \rangle, \quad \ldots \quad \langle l_i, \varphi_i \rangle, \quad \ldots \quad \langle l_n, \varphi_n \rangle, \qquad \varphi, \qquad S)$$

ESST Algorithm: ARF Construction (Rule 2)

Thread i is the running thread in S, and op in the CFA edge (l_i, op, l'_i) is ~~not~~ a primitive function call.

$$\begin{array}{ccccccc} (\langle l_1,\varphi_1\rangle, & \ldots & \langle l_i,\varphi_i\rangle, & \ldots & \langle l_n,\varphi_n\rangle, & \varphi, & S) \\ \downarrow & & \downarrow & & \downarrow & \downarrow & \\ (\langle l_1,\varphi_1\rangle, & \ldots & \langle l'_i,\varphi_i\rangle, & \ldots & \langle l_n,\varphi_n\rangle, & \varphi, & \end{array}$$

ESST Algorithm: ARF Construction (Rule 2)

Thread i is the running thread in S, and op in the CFA edge (l_i, op, l'_i) is ~~not~~ a primitive function call.

$$
\begin{array}{cccccccc}
(\langle l_1, \varphi_1 \rangle, & \ldots & \langle l_i, \varphi_i \rangle, & \ldots & \langle l_n, \varphi_n \rangle, & \varphi, & S) & \\
\downarrow & & \downarrow & & \downarrow & \downarrow & \downarrow & \textsc{Sexec}(S, op) \\
(\langle l_1, \varphi_1 \rangle, & \ldots & \langle l'_i, \varphi_i \rangle, & \ldots & \langle l_n, \varphi_n \rangle, & \varphi, & S') &
\end{array}
$$

ESST Algorithm: ARF Construction (Rule 3)

No running thread in S

$$(\ldots, S)$$

ESST Algorithm: ARF Construction (Rule 3)

No running thread in S

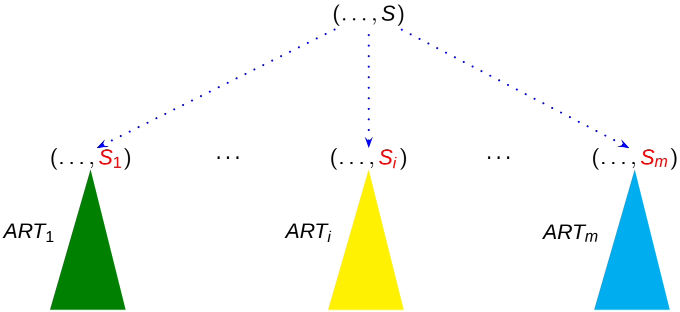
$$(\ldots, S)$$

$$(\ldots, S_1) \qquad \cdots \qquad (\ldots, S_i) \qquad \cdots \qquad (\ldots, S_m)$$

- $\{S_1, \ldots, S_m\} = \textsc{Sched}(S)$

ESST Algorithm: ARF Construction (Rule 3)

No running thread in S



- $\{S_1, \ldots, S_m\} = \textsc{Sched}(S)$
- $(\ldots, S) \dashrightarrow (\ldots, S_i)$ is ARF connector

ESST Algorithm: Coverage

- Coverage check

$$(\langle l_1, \varphi_1 \rangle, \quad \ldots \quad \langle l_i, \varphi_i \rangle, \quad \ldots \quad \langle l_n, \varphi_n \rangle, \qquad \varphi, \qquad S)$$

$$(\langle l_1, \varphi'_1 \rangle, \quad \ldots \quad \langle l_i, \varphi'_i \rangle, \quad \ldots \quad \langle l_n, \varphi'_n \rangle, \qquad \varphi', \qquad S')$$

ESST Algorithm: Coverage

- Coverage check

$$
\begin{array}{ccccccc}
(\langle l_1, \varphi_1 \rangle, & \ldots & \langle l_i, \varphi_i \rangle, & \ldots & \langle l_n, \varphi_n \rangle, & \varphi, & S) \\
\Big\downarrow \text{covered} & & \Big\downarrow \text{covered} & & \Big\downarrow \text{covered} & \Big\downarrow \models & \Big\downarrow = \\
(\langle l_1, \varphi'_1 \rangle, & \ldots & \langle l_i, \varphi'_i \rangle, & \ldots & \langle l_n, \varphi'_n \rangle, & \varphi', & S')
\end{array}
$$

ESST Algorithm: Feasibility Check

- Feasibility check

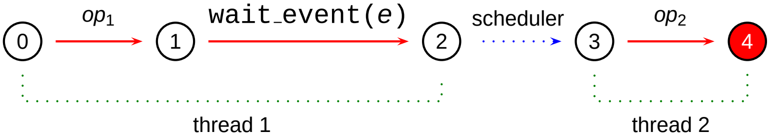
ARF Path π

$$0 \xrightarrow{op_1} 1 \xrightarrow{\texttt{wait\_event}(e)} 2 \xrightarrow{\text{scheduler}} 3 \xrightarrow{op_2} 4$$

ESST Algorithm: Feasibility Check

- Feasibility check

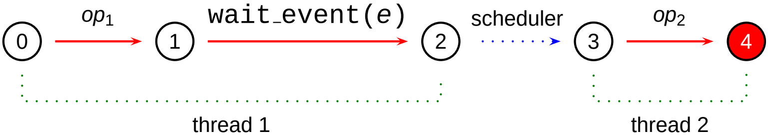
ARF Path π



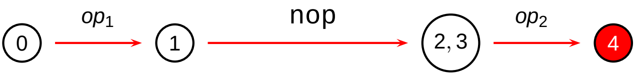
ESST Algorithm: Feasibility Check

- Feasibility check

ARF Path π



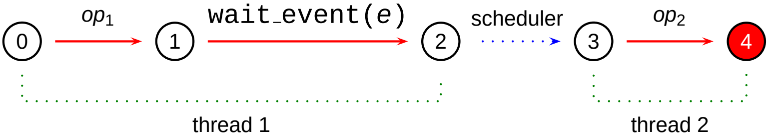
ARF path π'



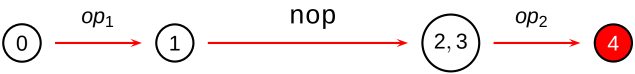
ESST Algorithm: Feasibility Check

- Feasibility check

ARF Path π



ARF path π'



π is feasible iff so is π'

Correctness of ESST

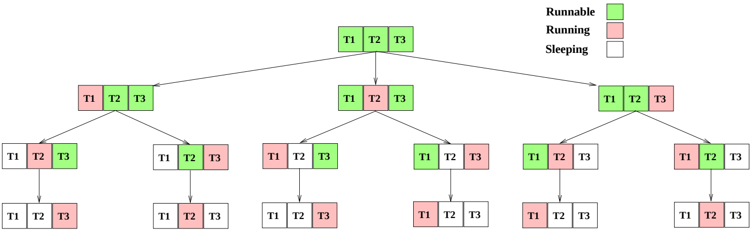
Theorem

Let P be a threaded sequential program. For every terminating execution of $\mathrm{ESST}(P)$*, we have the following properties:*

1. *If* $\mathrm{ESST}(P)$ *returns a feasible counter-example path* $\hat{\rho}$*, then we have* $\gamma \xrightarrow{\hat{\rho}} \gamma'$ *for an initial configuration* γ *and an error configuration* γ' *of* P
2. *If* $\mathrm{ESST}(P)$ *returns a safe ARF* $\mathcal{F}$*, then for every configuration* $\gamma \in \mathit{Reach}(P)$*, there is an ARF node* $\eta \in \mathit{Nodes}(\mathcal{F})$ *such that* $\gamma \models \eta$

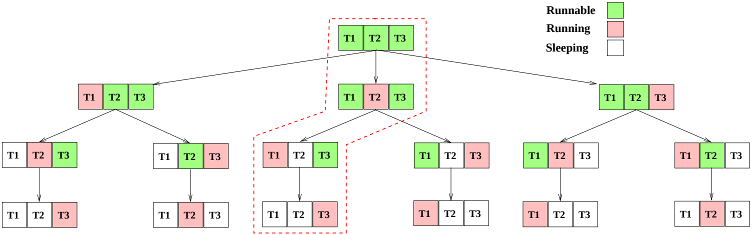
Limitations of ESST for CTPs

Limitations of ESST for CTPs



Given n threads: $n!$ interleavings, at least 2^n abstract states

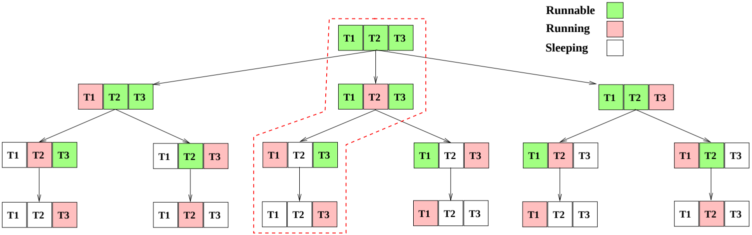
Limitations of ESST for CTPs



Given n threads: n! interleavings, at least 2^n abstract states

Impacts on ESST:

Limitations of ESST for CTPs

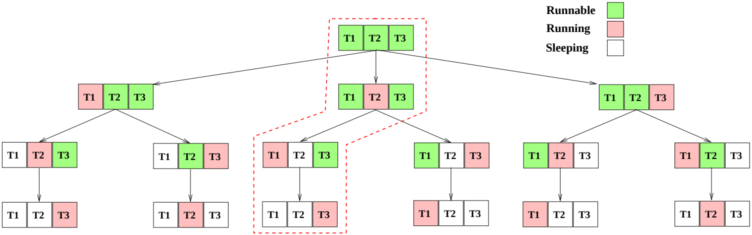


Given n threads: $n!$ interleavings, at least 2^n abstract states

Impacts on ESST:

- More abstract states to explore
  - Expensive abstract post image computations

Limitations of ESST for CTPs

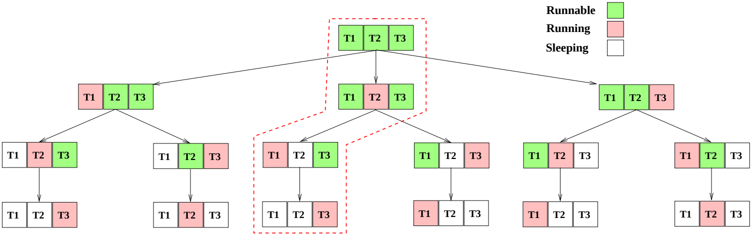


Given n threads: n! interleavings, at least 2^n abstract states

Impacts on ESST:

- More abstract states to explore
  - Expensive abstract post image computations
- More refinements, more predicates to keep track

Limitations of ESST for CTPs



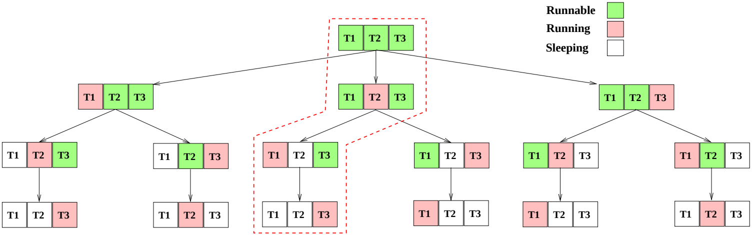
Given n threads: $n!$ interleavings, at least 2^n abstract states

Impacts on ESST:

- More abstract states to explore
  - Expensive abstract post image computations
- More refinements, more predicates to keep track

$\Rightarrow$ Degrade performace of ESST + State explosion

Limitations of ESST for CTPs



Given *n* threads: *n*! interleavings, at least 2^n abstract states

Impacts on ESST:

- More abstract states to explore
  - Expensive abstract post image computations
- More refinements, more predicates to keep track

⇒ Degrade performace of ESST + State explosion

- Apply partial-order reduction to ESST [CNR11]
  - Allow ESST to explore only representative interleavings

Partial-Order Reduction (POR)

Idea of POR

Exploit independence and commutativity of transitions

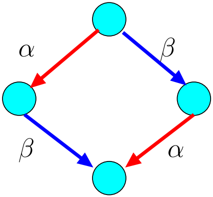
Partial-Order Reduction (POR)

Idea of POR

Exploit independence and commutativity of transitions

Two transitions are independent if

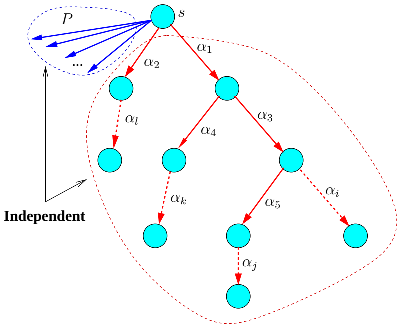
1. they neither disable nor enable each other
2. they commute



Persistent Set

Persistent Set

A set P of transitions is persistent in a state s if the transitions are independent of every $\alpha_i \notin P$ reachable from s

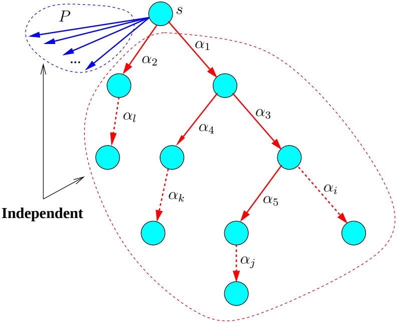


Persistent Set

Persistent Set

A set P of transitions is persistent in a state s if the transitions are independent of every $\alpha_i \notin P$ reachable from s

$\Rightarrow$ One only needs to explore P

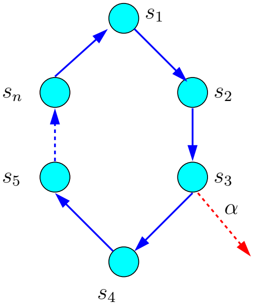


Requirements for Verifying Safety Properties

1. Successor-state condition: persistent set *P* in state *s* is empty iff no enabled transitions in *s*

Requirements for Verifying Safety Properties

1. Successor-state condition: persistent set *P* in state *s* is empty iff no enabled transitions in *s*
2. Cycle condition: disallow



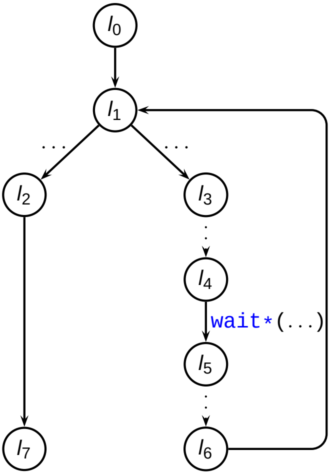
α is enabled in s_i but not in the persistent sets of $s_1, \ldots, s_n$

Identifying Atomic Blocks

An atomic block correspond to a non-interleaved transition

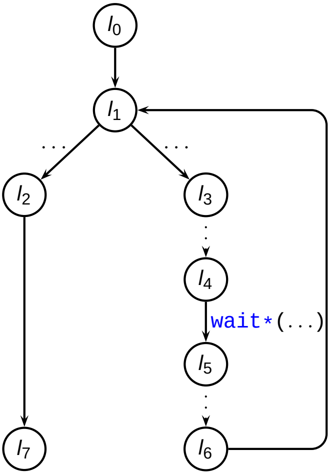
Identifying Atomic Blocks

An atomic block correspond to a non-interleaved transition



Identifying Atomic Blocks

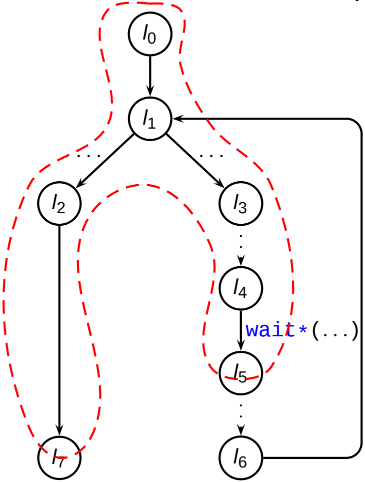
An atomic block correspond to a non-interleaved transition



Fragment between two wait*(...)

Identifying Atomic Blocks

An atomic block correspond to a non-interleaved transition



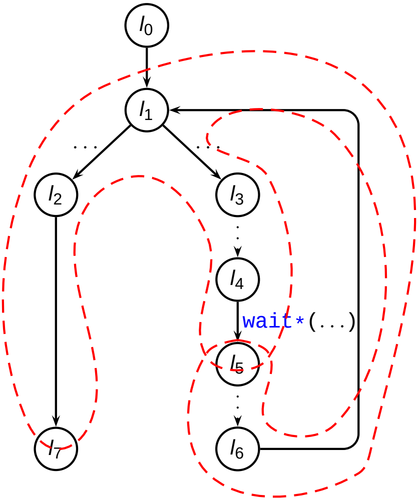
Fragment between two wait*(...)

Identify an atomic block by its entry:

1. Entry: l_o, Exit: l_5, l_7

Identifying Atomic Blocks

An atomic block correspond to a non-interleaved transition



Fragment between two wait*(...)

Identify an atomic block by its entry:

1. Entry: l_o, Exit: l_5, l_7
2. Entry: l_5, Exit: l_5, l_7

Atomic Block (In)dependence

Atomic blocks α and β are dependent if

- α writes to global *g*, and β writes to or reads from *g*

Atomic Block (In)dependence

Atomic blocks α and β are dependent if

- α writes to global *g*, and β writes to or reads from *g*
- α immediately notifies event *e*, and β waits for *e*

Atomic Block (In)dependence

Atomic blocks α and β are dependent if

- α writes to global *g*, and β writes to or reads from *g*
- α immediately notifies event *e*, and β waits for *e*
- α delay notifies event *e*, and β cancels *e*'s notification

The Function PERSISTENT

Compute persistent scheduler states:

$$\text{PERSISTENT} : \mathit{ARFNodes} \rightarrow \mathit{SchedulerStates}$$

The Function PERSISTENT

Compute persistent scheduler states:

$$\text{PERSISTENT} : \mathit{ARFNodes} \rightarrow \mathit{SchedulerStates}$$

Let $N = (\langle l_1, \varphi_1 \rangle, \ldots, \langle l_n, \varphi_n \rangle, \varphi, S)$

PERSISTENT(N):

The Function PERSISTENT

Compute persistent scheduler states:

$$\textsc{Persistent} : \textit{ARFNodes} \rightarrow \textit{SchedulerStates}$$

Let $N = (\langle l_1, \varphi_1 \rangle, \ldots, \langle l_n, \varphi_n \rangle, \varphi, S)$

$\textsc{Persistent}(N)$:

1. Let $\mathcal{S} = \textsc{Sched}(S)$.

The Function PERSISTENT

Compute persistent scheduler states:

$$\textsc{Persistent} : \mathit{ARFNodes} \rightarrow \mathit{SchedulerStates}$$

Let $N = (\langle l_1, \varphi_1 \rangle, \ldots, \langle l_n, \varphi_n \rangle, \varphi, S)$

PERSISTENT(N):

1. Let $\mathcal{S} = \textsc{Sched}(S)$.
2. Collect enabled atomic blocks:
   $AB = \{l_i \mid \exists S \in \mathcal{S}.S(t_i) = \mathit{Running}\}$

The Function PERSISTENT

Compute persistent scheduler states:

$$\textsc{Persistent} : \mathit{ARFNodes} \rightarrow \mathit{SchedulerStates}$$

Let $N = (\langle l_1, \varphi_1 \rangle, \ldots, \langle l_n, \varphi_n \rangle, \varphi, S)$

PERSISTENT(N):

1. Let $\mathcal{S} = \textsc{Sched}(S)$.
2. Collect enabled atomic blocks:
   $AB = \{l_i \mid \exists S \in \mathcal{S}. S(t_i) = \mathit{Running}\}$
3. Let $\mathit{PersistentAB} \subseteq AB$
   - Reuse existing techniques for explicit-state model checking!

The Function PERSISTENT

Compute persistent scheduler states:

$$\textsc{Persistent} : \mathit{ARFNodes} \rightarrow \mathit{SchedulerStates}$$

Let $N = (\langle l_1, \varphi_1 \rangle, \ldots, \langle l_n, \varphi_n \rangle, \varphi, S)$

PERSISTENT(N):

1. Let $\mathcal{S} = \textsc{Sched}(S)$.
2. Collect enabled atomic blocks:
   $AB = \{l_i \mid \exists S \in \mathcal{S}.S(t_i) = \mathit{Running}\}$
3. Let $\mathit{PersistentAB} \subseteq AB$
   - Reuse existing techniques for explicit-state model checking!
4. Let $\mathcal{S}' = \{S \in \mathcal{S} \mid S(t_i) = \mathit{Running} \wedge l_i \in \mathit{PersistentAB}\}$

The Function PERSISTENT

Compute persistent scheduler states:

$$\text{PERSISTENT} : \mathit{ARFNodes} \rightarrow \mathit{SchedulerStates}$$

Let $N = (\langle l_1, \varphi_1 \rangle, \ldots, \langle l_n, \varphi_n \rangle, \varphi, S)$

PERSISTENT(N):

1. Let $\mathcal{S} = \text{SCHED}(S)$.
2. Collect enabled atomic blocks:
   $AB = \{l_i \mid \exists S \in \mathcal{S}.S(t_i) = \mathit{Running}\}$
3. Let $\mathit{PersistentAB} \subseteq AB$
   - Reuse existing techniques for explicit-state model checking!
4. Let $\mathcal{S}' = \{S \in \mathcal{S} \mid S(t_i) = \mathit{Running} \wedge l_i \in \mathit{PersistentAB}\}$
5. Return $\mathcal{S}'$

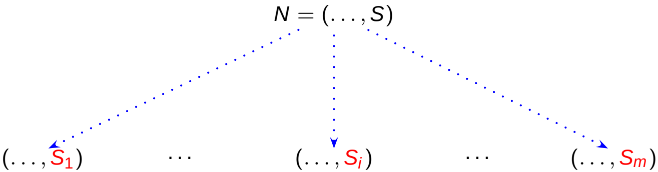
Extension of ESST Rule: Persistent Set

No running thread in S

$$N = (\ldots, S)$$

Extension of ESST Rule: Persistent Set

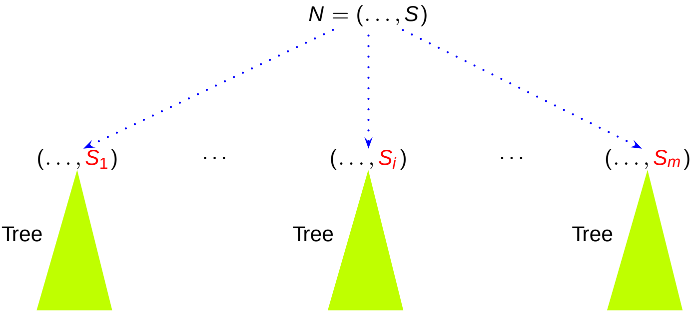
No running thread in S



- $\{S_1, \ldots, S_m\} = \textsc{Sched}(S)$

Extension of ESST Rule: Persistent Set

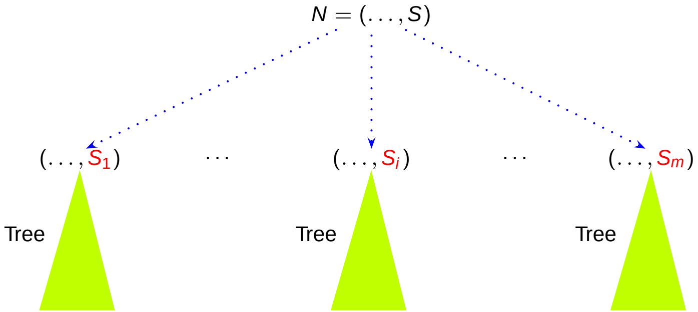
No running thread in S



- $\{S_1, \ldots, S_m\} = \textsc{Sched}(S)$
- $(\ldots, S) \dashrightarrow (\ldots, S_i)$ connects two trees

Extension of ESST Rule: Persistent Set

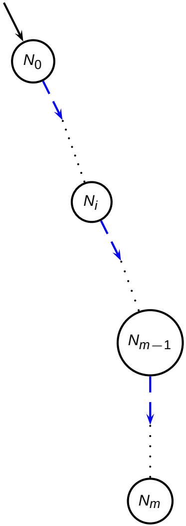
No running thread in S



- $\{S_1, \ldots, S_m\} = \cancel{\textsc{Sched}(S)}\quad \textsc{Persistent}(N)$
- $(\ldots, S) \dashrightarrow (\ldots, S_i)$ connects two trees

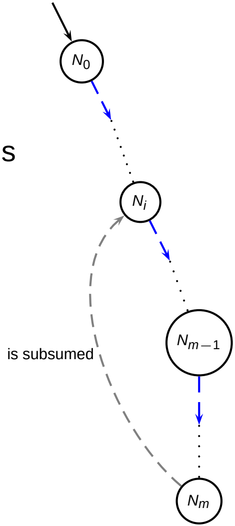
Extension of ESST Rule: Cycle Condition

- Use persistent set for node expansions
- $N_0, \ldots, N_m$ are non-running



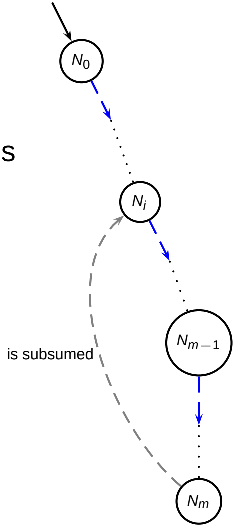
Extension of ESST Rule: Cycle Condition

- Use persistent set for node expansions
- $N_0, \ldots, N_m$ are non-running
- If N_m is subsumed by N_i



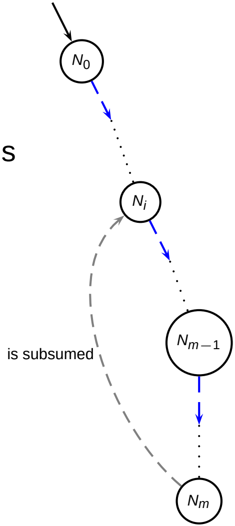
Extension of ESST Rule: Cycle Condition

- Use persistent set for node expansions
- $N_0, \ldots, N_m$ are non-running
- If N_m is subsumed by N_i
  - There is a potential cycle



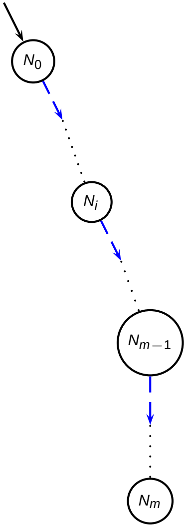
Extension of ESST Rule: Cycle Condition

- Use persistent set for node expansions
- $N_0, \ldots, N_m$ are non-running
- If N_m is subsumed by N_i
  - There is a potential cycle
  - Optional: check feasibility



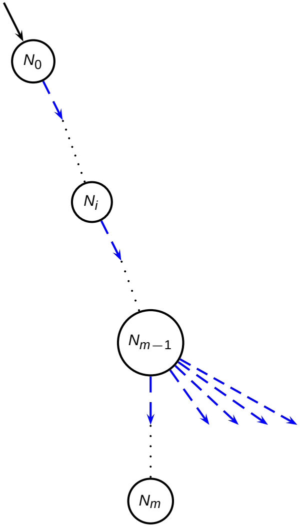
Extension of ESST Rule: Cycle Condition

- Use persistent set for node expansions
- $N_0, \ldots, N_m$ are non-running
- If N_m is subsumed by N_i
  - There is a potential cycle
  - Optional: check feasibility
- If N_{m-1} is not fully expanded ($\textsc{Persistent}(N_{m-1}) \subset \textsc{Sched}(S_{m-1})$):



Extension of ESST Rule: Cycle Condition

- Use persistent set for node expansions
- $N_0, \ldots, N_m$ are non-running
- If N_m is subsumed by N_i
  - There is a potential cycle
  - Optional: check feasibility
- If N_{m-1} is not fully expanded ($\textsc{Persistent}(N_{m-1}) \subset \textsc{Sched}(S_{m-1})$):
  - Fully expand N_{m-1}



Correctness of ESST+POR

Theorem

Let P be a threaded sequential program. For every terminating executions of $\mathrm{ESST}(P)$ *and* $\mathrm{ESST}_{\mathrm{POR}}(P)$*, we have that* $\mathrm{ESST}(P)$ *reports safe iff so does* $\mathrm{ESST}_{\mathrm{POR}}(P)$*.*

Limitations of ESST +POR

- POR could interact negatively with ESST

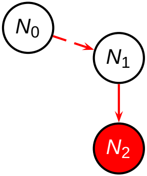
Limitations of ESST +POR

- POR could interact negatively with ESST
- Example: longer counter example

Limitations of ESST +POR

- POR could interact negatively with ESST
- Example: longer counter example

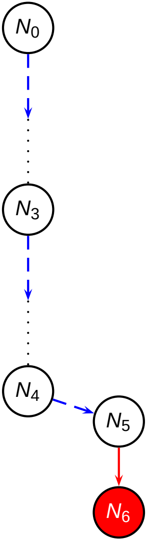
Limitations of ESST +POR



- POR could interact negatively with ESST
- Example: longer counter example

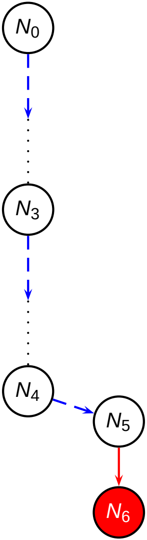
Limitations of ESST +POR

- POR could interact negatively with ESST
- Example: longer counter example



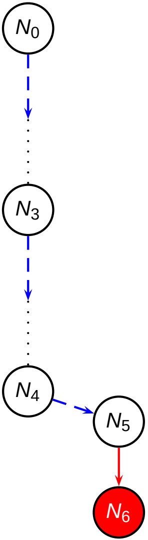
Limitations of ESST +POR

- POR could interact negatively with ESST
- Example: longer counter example
- Impacts on ESST:



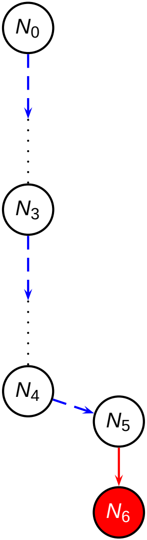
Limitations of ESST +POR

- POR could interact negatively with ESST
- Example: longer counter example
- Impacts on ESST:
  - More abstract states to explore



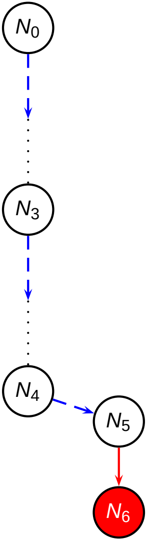
Limitations of ESST +POR

- POR could interact negatively with ESST
- Example: longer counter example
- Impacts on ESST:
  - More abstract states to explore
  - More refinements



Limitations of ESST +POR

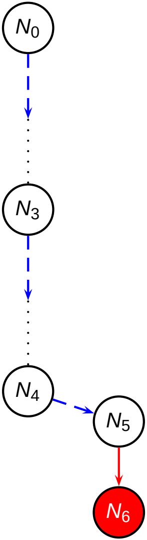
- POR could interact negatively with ESST
- Example: longer counter example
- Impacts on ESST:
  - More abstract states to explore
  - More refinements
  - More predicates to keep track



Limitations of ESST +POR

- POR could interact negatively with ESST
- Example: longer counter example
- Impacts on ESST:
  - More abstract states to explore
  - More refinements
  - More predicates to keep track

$\Rightarrow$ Degrade performance of ESST
  - Experimental evaluation does not show this behavior

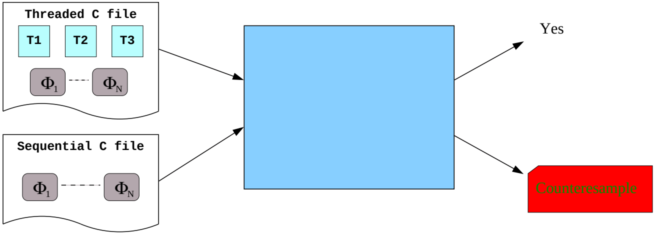


Outline

- Cooperative Threaded Programs (CTPs)
- Background
  - Safe Sequential Programs
  - Model Checking of Sequential Programs
    - Finite Model for Sequential Programs
    - Symbolic Model Checking of Sequential Programs
- Approaches to Model Checking of CTPs
  - Finite-Model for Cooperative Threaded Programs
  - Symbolic Model Checking of Sequential Software
  - Explicit Scheduler and Symbolic Threads (ESST)
- The Kratos Software Model Checker
- Experimental Results
- Related Work
- Conclusions

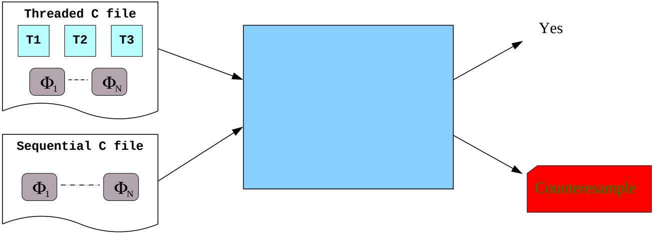
KRATOS: Overview

- KRATOS is a software model checker for sequential and threaded programs with cooperative scheduler



KRATOS: Overview

- KRATOS is a software model checker for sequential and threaded programs with cooperative scheduler



- KRATOS verifies safety properties in the form of program assertion

KRATOS: Overview (II)

- Analyses for sequential programs:
  - Sequential analysis:
    - Lazy abstraction [HJMS02]
    - Lazy abstraction with interpolation [McM06]
    - Property Driven Reachability for Software [CG12]
  - Symbolic Model Checking via reduction to NUSMV
  - BDD based reachability analysis for finite domains programs

KRATOS: Overview (II)

- Analyses for sequential programs:
  - Sequential analysis:
    - Lazy abstraction [HJMS02]
    - Lazy abstraction with interpolation [McM06]
    - Property Driven Reachability for Software [CG12]
  - Symbolic Model Checking via reduction to NUSMV
  - BDD based reachability analysis for finite domains programs
- Analyses for threaded programs:
  - Reduction to finite model and analysis with SPIN [CCNR11, CNR13]
  - Reduction to sequential analysis [CNR13, CMNR10]
  - Concurrent analysis:
    - Explicit-Scheduler/Symbolic Threads [CNR12a, CMNR10]
    - Semi-Symbolic-Scheduler/Symbolic-Threads [CNR12b]

KRATOS: Overview (II)

- Analyses for sequential programs:
  - Sequential analysis:
    - Lazy abstraction [HJMS02]
    - Lazy abstraction with interpolation [McM06]
    - Property Driven Reachability for Software [CG12]
  - Symbolic Model Checking via reduction to NUSMV
  - BDD based reachability analysis for finite domains programs
- Analyses for threaded programs:
  - Reduction to finite model and analysis with SPIN [CCNR11, CNR13]
  - Reduction to sequential analysis [CNR13, CMNR10]
  - Concurrent analysis:
    - Explicit-Scheduler/Symbolic Threads [CNR12a, CMNR10]
    - Semi-Symbolic-Scheduler/Symbolic-Threads [CNR12b]
- State-of-the-art SMT techniques for abstractions and refinements

KRATOS: Overview (II)

- Analyses for sequential programs:
  - Sequential analysis:
    - Lazy abstraction [HJMS02]
    - Lazy abstraction with interpolation [McM06]
    - Property Driven Reachability for Software [CG12]
  - Symbolic Model Checking via reduction to NUSMV
  - BDD based reachability analysis for finite domains programs
- Analyses for threaded programs:
  - Reduction to finite model and analysis with SPIN [CCNR11, CNR13]
  - Reduction to sequential analysis [CNR13, CMNR10]
  - Concurrent analysis:
    - Explicit-Scheduler/Symbolic Threads [CNR12a, CMNR10]
    - Semi-Symbolic-Scheduler/Symbolic-Threads [CNR12b]
- State-of-the-art SMT techniques for abstractions and refinements
- Advanced techniques for handling multiple assertions [CCL+12]

Software Model Checking of Multiple Assertions

Typically SW model checkers stops once they find a violated assertion, and returns a counterexample

- For finding all violated assertions: one assertion at a time in the program under analysis

Software Model Checking of Multiple Assertions

Typically SW model checkers stops once they find a violated assertion, and returns a counterexample

- For finding all violated assertions: one assertion at a time in the program under analysis

Verifying one assertion at a time may be unfeasible

- Computation starts from scratch each time

Software Model Checking of Multiple Assertions

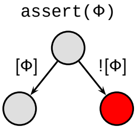
Typically SW model checkers stops once they find a violated assertion, and returns a counterexample

- For finding all violated assertions: one assertion at a time in the program under analysis

Verifying one assertion at a time may be unfeasible

- Computation starts from scratch each time

Software model checkers may fail to discover all the violated assertions because the way they interpret "`assert`"



Software Model Checking of Multiple Assertions

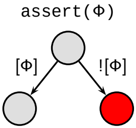
Typically SW model checkers stops once they find a violated assertion, and returns a counterexample

- For finding all violated assertions: one assertion at a time in the program under analysis

Verifying one assertion at a time may be unfeasible

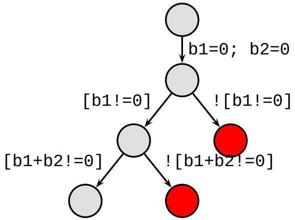
- Computation starts from scratch each time

Software model checkers may fail to discover all the violated assertions because the way they interpret “`assert`”



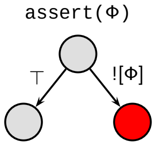

```
int b1=0, b2=0;
assert(b1 != 0);
assert(b1+b2 != 0);
```

2nd assertion cannot be violated: it is blocked by the 1st one



Software Model Checking of Multiple Assertions (II)

- Modify the interpretation of `assert` to enable handling of multiple assertions and produce a counterexample for all violated assertions
  - Interpret assertions-as-properties

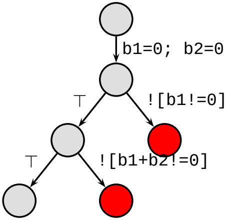


- Extend SW model checking via lazy-predicate abstraction to deal with multiple assertions
  - Two search techniques
    - All-in-one-go
    - One-at-a-time
  - Both interpret assertion as properties

SW MC Multiple Assertions: All-In-One-Go

- When an assertion violation reached, the assertion is disabled, and the search continues for other possible violations of other assertions
  - Search terminates when the ART/ARF is complete

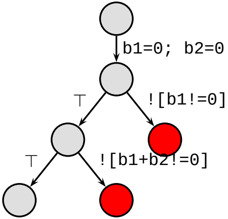
```
int b1=0, b2=0;
assert(b1 != 0);
assert(b1+b2 != 0);
```



SW MC Multiple Assertions: All-In-One-Go

- When an assertion violation reached, the assertion is disabled, and the search continues for other possible violations of other assertions
  - Search terminates when the ART/ARF is complete

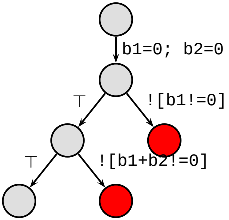
```
int b1=0, b2=0;
assert(b1 != 0);
assert(b1+b2 != 0);
```

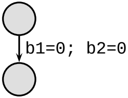


SW MC Multiple Assertions: All-In-One-Go

- When an assertion violation reached, the assertion is disabled, and the search continues for other possible violations of other assertions
  - Search terminates when the ART/ARF is complete

```
int b1=0, b2=0;
assert(b1 != 0);
assert(b1+b2 != 0);
```

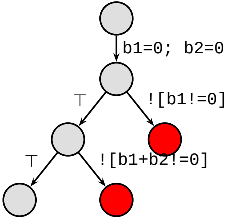


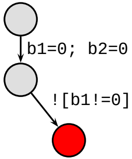


SW MC Multiple Assertions: All-In-One-Go

- When an assertion violation reached, the assertion is disabled, and the search continues for other possible violations of other assertions
  - Search terminates when the ART/ARF is complete

```
int b1=0, b2=0;
assert(b1 != 0);
assert(b1+b2 != 0);
```

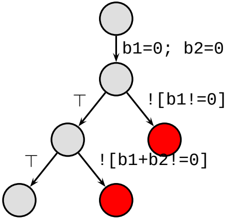


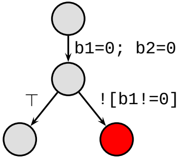


SW MC Multiple Assertions: All-In-One-Go

- When an assertion violation reached, the assertion is disabled, and the search continues for other possible violations of other assertions
  - Search terminates when the ART/ARF is complete

```
int b1=0, b2=0;
assert(b1 != 0);
assert(b1+b2 != 0);
```

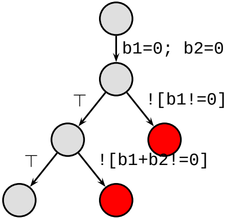


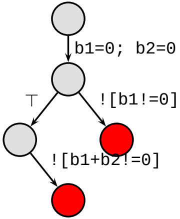


SW MC Multiple Assertions: All-In-One-Go

- When an assertion violation reached, the assertion is disabled, and the search continues for other possible violations of other assertions
  - Search terminates when the ART/ARF is complete

```
int b1=0, b2=0;
assert(b1 != 0);
assert(b1+b2 != 0);
```

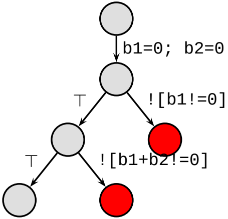


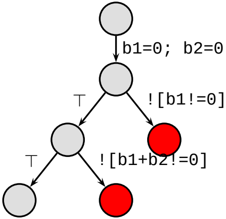


SW MC Multiple Assertions: All-In-One-Go

- When an assertion violation reached, the assertion is disabled, and the search continues for other possible violations of other assertions
  - Search terminates when the ART/ARF is complete

```
int b1=0, b2=0;
assert(b1 != 0);
assert(b1+b2 != 0);
```

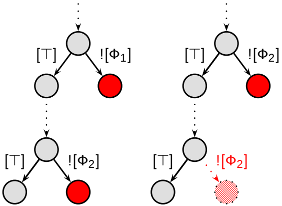




SW MC Multiple Assertions: One-At-A-Time

- One assertion at a time is checked
  - Disabling other assertions

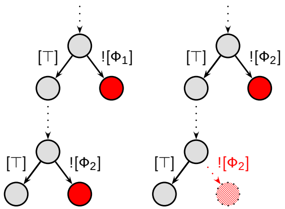
...
assert(Φ_1);
...
assert(Φ_2);
...



SW MC Multiple Assertions: One-At-A-Time

- One assertion at a time is checked
  - Disabling other assertions
- ART/ARF used for checking one assertion re-used for proving the others

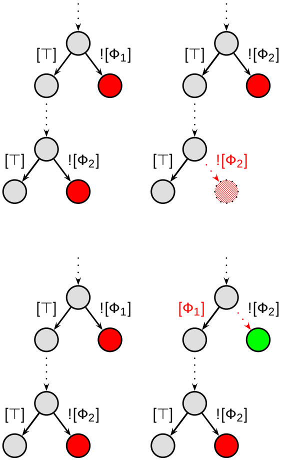
...
assert(Φ_1);
...
assert(Φ_2);
...



SW MC Multiple Assertions: One-At-A-Time

- One assertion at a time is checked
  - Disabling other assertions
- ART/ARF used for checking one assertion re-used for proving the others
- When an assertion proved to be safe, CFG strengthened turning assertion into “standard” semantics

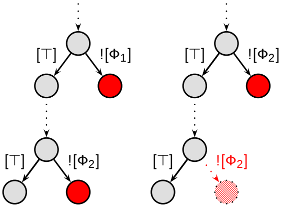
...
assert(Φ_1);
...
assert(Φ_2);
...

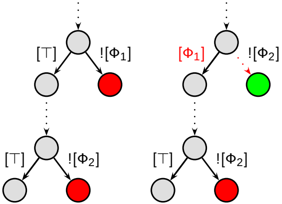


SW MC Multiple Assertions: One-At-A-Time

- One assertion at a time is checked
  - Disabling other assertions
- ART/ARF used for checking one assertion re-used for proving the others
- When an assertion proved to be safe, CFG strengthened turning assertion into “standard” semantics

...
assert(Φ_1);
...
assert(Φ_2);
...





- Enables for several optimizations
  - On-the-fly slicing with respect to the checked assertion
  - Partitioning the predicates used to prove each assertion
  - Collecting loop invariants from the constructed ART/ARF to be used to possibly strengthen the successive searches

Architecture of KRATOS

- Front-end:
  - Parser and Type checker
  - CFG encoder: single-block, basic-block and large-block [BCG+09] encodings
  - Optimization: constant propagation, dead-code elimination, cone-of-influence reduction

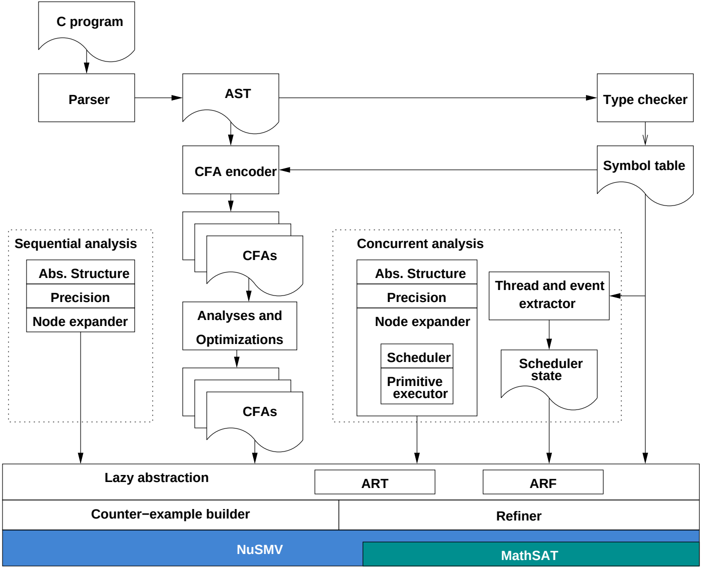
Architecture of KRATOS

- Front-end:
  - Parser and Type checker
  - CFG encoder: single-block, basic-block and large-block [BCG+09] encodings
  - Optimization: constant propagation, dead-code elimination, cone-of-influence reduction
- Analysis:
  - Abstraction structure: CFG locations, data states as formulas, call stack
  - Node expander: compute successor abstract states
  - Scheduler: implement some scheduling policy
  - Primitive executor: execute API for querying and updating scheduler states (SystemC and FairThreads)
  - Counter-example builder and Refiner

Architecture of KRATOS

- Front-end:
  - Parser and Type checker
  - CFG encoder: single-block, basic-block and large-block [BCG+09] encodings
  - Optimization: constant propagation, dead-code elimination, cone-of-influence reduction
- Analysis:
  - Abstraction structure: CFG locations, data states as formulas, call stack
  - Node expander: compute successor abstract states
  - Scheduler: implement some scheduling policy
  - Primitive executor: execute API for querying and updating scheduler states (SystemC and FairThreads)
  - Counter-example builder and Refiner
- Backend: NUSMV and MATHSAT
  - Advanced techniques for boolean predicate abstraction
  - Entailment checks in abstract state coverage
  - Feasibility checks of counter-examples (via SMT)

Architecture of KRATOS



KRATOS: Availability

- KRATOS can be downloaded at
  http://es.fbk.eu/tools/kratos

KRATOS: Availability

- KRATOS can be downloaded at
  `http://es.fbk.eu/tools/kratos`
- Free for academic and research purposes

Outline

Experimental Evaluation Setup

- Benchmarks:
  - SystemC benchmarks taken and extended from literature
  - FairThreads benchmarks taken from the literature and adapted from SystemC
  - Industrial benchmarks from railway application from Ansaldo STS

Experimental Evaluation Setup

- Benchmarks:
  - SystemC benchmarks taken and extended from literature
  - FairThreads benchmarks taken from the literature and adapted from SystemC
  - Industrial benchmarks from railway application from Ansaldo STS
- Evaluated ESST w.r.t. sequentialization

Experimental Evaluation Setup

- Benchmarks:
  - SystemC benchmarks taken and extended from literature
  - FairThreads benchmarks taken from the literature and adapted from SystemC
  - Industrial benchmarks from railway application from Ansaldo STS
- Evaluated ESST w.r.t. sequentialization
- Evaluated POR techniques
  - Persistent set, Sleep set, Persistent + Sleep set

Experimental Evaluation Setup

- Benchmarks:
  - SystemC benchmarks taken and extended from literature
  - FairThreads benchmarks taken from the literature and adapted from SystemC
  - Industrial benchmarks from railway application from Ansaldo STS
- Evaluated ESST w.r.t. sequentialization
- Evaluated POR techniques
  - Persistent set, Sleep set, Persistent + Sleep set
- Evaluated finite-model and analysis with SPIN

Experimental Evaluation Setup

- Benchmarks:
  - SystemC benchmarks taken and extended from literature
  - FairThreads benchmarks taken from the literature and adapted from SystemC
  - Industrial benchmarks from railway application from Ansaldo STS
- Evaluated ESST w.r.t. sequentialization
- Evaluated POR techniques
  - Persistent set, Sleep set, Persistent + Sleep set
- Evaluated finite-model and analysis with SPIN
- Resource limit: time limit 1000s, memory limit 2GB

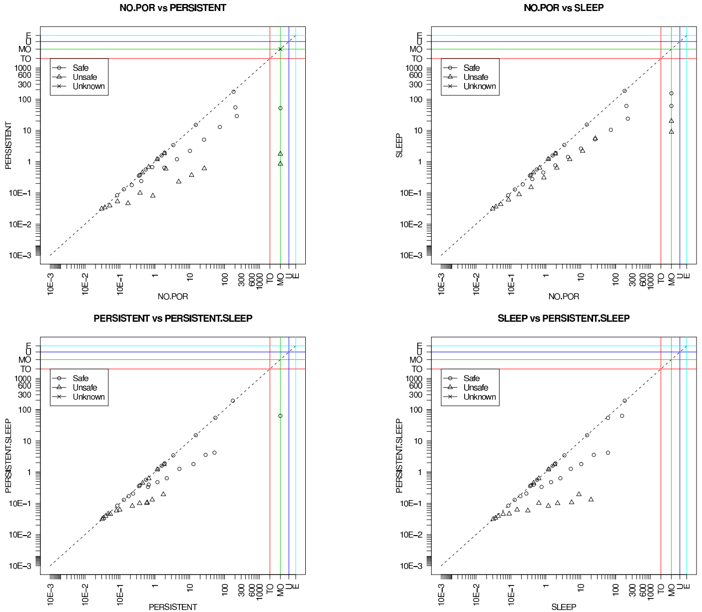
SystemC Benchmarks

Benchmarks	V	Finite-State Models			Sequentialization							Threaded
		PROMELA +SPIN			Eager	Lazy PA			Lazy AWI		BMC	ESST
		T2P	T2AB	1AB	SATABS	BLAST	CPACHECKER	KRATOS	WOLVERINE	KRATOS	CBMC	
bist-cell	S	-	-	-	31.740	5.300	7.340	0.400	T.O.	**0.300**	-	1.390
kundu-bug-1	U	**0.001**	0.010	**0.001**	33.730	105.850	24.310	25.000	205.370	25.590	1.080	0.590
kundu-bug-2	U	**0.001**	**0.001**	**0.001**	79.160	Err	17.710	0.890	580.990	11.200	2.450	0.500
kundu	S	-	-	-	96.460	Err	35.620	151.490	T.O.	T.O.	-	**1.090**
mem-slave-tlm.1	S	-	-	-	69.150	80.360	120.060	139.790	78.920	40.590	-	**3.500**
mem-slave-tlm.3	S	-	-	-	385.410	745.690	M.O.	T.O.	470.890	596.250	-	**42.690**
mem-slave-tlm.5	S	-	-	-	T.O.	Err	T.O.	T.O.	T.O.	T.O.	-	**280.260**
mem-slave-tlm-bug.1	U	**0.001**	**0.001**	**0.001**	83.140	84.420	42.190	13.800	90.710	27.790	53.750	2.600
mem-slave-tlm-bug.3	U	0.010	**0.001**	**0.001**	719.070	763.640	M.O.	T.O.	505.100	687.150	55.850*	33.390
mem-slave-tlm-bug.5	U	**0.001**	**0.001**	0.010	T.O.	Err	M.O.	T.O.	T.O.	T.O.	56.890*	207.970
mem-slave-tlm-bug2.1	U	**0.001**	**0.001**	**0.001**	75.610	82.070	33.160	2.790	85.830	18.000	54.770	1.400
mem-slave-tlm-bug2.3	U	0.150	**0.130**	0.300	391.900	T.O.	71.680	18.390	T.O.	401.870	56.400*	12.290
mem-slave-tlm-bug2.5	U	21.000	**17.000**	43.300	T.O.	Err	158.580	85.090	T.O.	T.O.	58.960*	40.490
pc-sfifo-1	S	-	-	-	3.490	20.350	16.960	3.300	13.690	3.590	-	**0.300**
pc-sfifo-2	S	-	-	-	4.810	34.650	25.820	8.400	32.430	25.590	-	**0.500**
pipeline-bug	U	**0.001**	**0.001**	**0.001**	737.320	T.O.	54.840	13.600	T.O.	103.290	-	6.400
pipeline	S	-	-	-	T.O.	T.O.	**67.630**	T.O.	T.O.	T.O.	-	81.790
token-ring.1	S	-	-	-	9.970	6.360	11.940	4.300	49.880	3.500	-	**0.100**
token-ring.5	S	-	-	-	814.160	T.O.	M.O.	T.O.	T.O.	T.O.	-	**0.400**
token-ring.9	S	-	-	-	T.O.	T.O.	M.O.	T.O.	T.O.	T.O.	-	**1.100**
token-ring.13	S	-	-	-	T.O.	M.O.	M.O.	T.O.	T.O.	T.O.	290.450	**4.500**
token-ring-bug.1	U	**0.001**	**0.001**	**0.001**	5.460	3.300	14.870	1.500	T.O.	2.590	1.620	**0.001**
token-ring-bug.5	U	**0.001**	**0.001**	**0.001**	748.250	T.O.	M.O.	T.O.	T.O.	T.O.	15.060	0.100
token-ring-bug.9	U	**0.001**	**0.001**	**0.001**	T.O.	T.O.	M.O.	T.O.	T.O.	T.O.	95.460	0.300
token-ring-bug.13	U	**0.001**	**0.001**	**0.001**	T.O.	M.O.	M.O.	T.O.	T.O.	T.O.	288.940	1.790
token-ring-bug2.1	U	0.010	**0.001**	4.100	5.940	2.480	13.980	2.000	T.O.	1.500	1.090	**0.001**
token-ring-bug2.5	U	T.O.	T.O.	T.O.	819.060	T.O.	M.O.	T.O.	T.O.	T.O.	15.370	**0.100**
token-ring-bug2.9	U	M.O.	M.O.	T.O.	T.O.	T.O.	M.O.	T.O.	T.O.	T.O.	97.980	**0.390**
token-ring-bug2.13	U	M.O.	M.O.	T.O.	T.O.	M.O.	M.O.	T.O.	T.O.	T.O.	312.380	**2.700**
toy-bug-1	U	5.550	5.340	4.560	23.570	241.240	45.650	10.200	T.O.	T.O.	1.430	**0.490**
toy-bug-2	U	5.690	5.290	4.560	19.560	144.610	44.810	3.890	T.O.	T.O.	1.410	**0.200**
toy	S	-	-	-	22.150	Err	195.620	T.O.	T.O.	T.O.	-	**1.800**
transmitter.1	U	**0.001**	**0.001**	**0.001**	2.280	1.190	17.060	1.090	T.O.	0.800	0.430	**0.001**
transmitter.5	U	**0.001**	**0.001**	**0.001**	224.070	T.O.	353.480	409.670	T.O.	T.O.	10.080	**0.001**
transmitter.9	U	**0.001**	0.010	**0.001**	T.O.	T.O.	M.O.	T.O.	T.O.	T.O.	74.420	0.100
transmitter.13	U	**0.001**	**0.001**	**0.001**	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	259.060	0.090

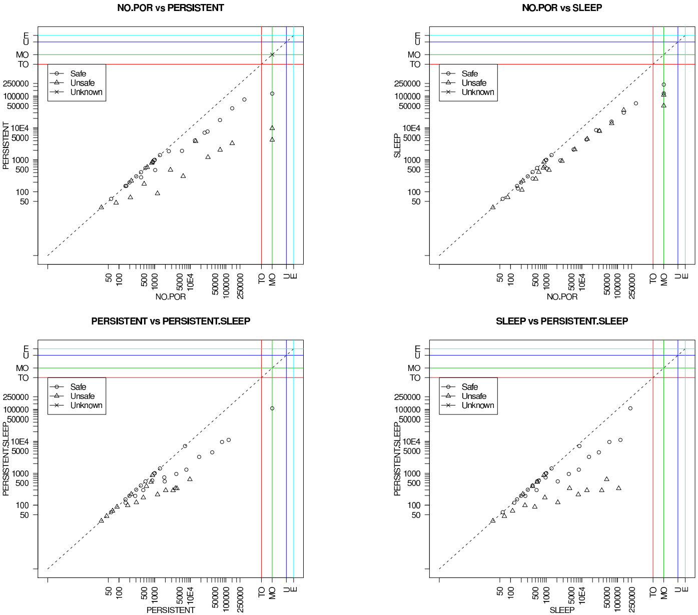
FairThreads Benchmarks

Name	V	SatAbs	CpaChecker	Kratos Seq	CBMC	Kratos ESST
fact1	S	9.07	14.26	2.90	-	**0.01**
fact1-bug	U	22.18	8.06	0.39	15.09	**0.01**
fact1-mod	S	4.41	8.18	0.50	-	**0.40**
fact2	S	69.05	17.25	15.40	-	**0.01**
gear-box	S	T.O	T.O	T.O	-	T.O
ft-pc-sfifo1	S	57.08	56.56	44.49	-	**0.30**
ft-pc-sfifo2	S	715.31	T.O	T.O	-	**0.39**
ft-token-ring.3	S	115.66	T.O	T.O	-	**0.48**
ft-token-ring.4	S	448.86	T.O	T.O	-	**5.20**
ft-token-ring.5	S	T.O	T.O	T.O	-	**213.37**
ft-token-ring.6	S	T.O	T.O	T.O	-	T.O
ft-token-ring.7	S	T.O	T.O	T.O	-	T.O
ft-token-ring.8	S	T.O	T.O	T.O	-	T.O
ft-token-ring.9	S	T.O	T.O	T.O	-	T.O
ft-token-ring.10	S	T.O	T.O	T.O	-	T.O
ft-token-ring-bug.3	U	111.10	T.O	T.O	158.76	**0.10**
ft-token-ring-bug.4	U	306.41	T.O	T.O	*407.36	**1.70**
ft-token-ring-bug.5	U	860.29	T.O	T.O	*751.44	**66.09**
ft-token-ring-bug.6	U	T.O	T.O	T.O	T.O	T.O
ft-token-ring-bug.7	U	T.O	T.O	T.O	T.O	T.O
ft-token-ring-bug.8	U	T.O	T.O	T.O	T.O	T.O
ft-token-ring-bug.9	U	T.O	T.O	T.O	T.O	T.O
ft-token-ring-bug.10	U	T.O	T.O	T.O	T.O	T.O

ESST vs ESST+POR: Run time



ESST vs ESST+POR: Explored abstract states



Industrial Benchmarks from Ansaldo STS

Embedded Software from Logica di Sicurezza (LDS) a generic subsystem of ERTMS developed by Ansaldo STS

- An LDS specification
  - An entity description of the physical and logical entities
  - A configuration describing a particular physical layout
- LDS is specified in VELOS
  - Structured programming language with a C++ like syntax developed by Ansaldo STS
    - Classes for representing Components, Points, EOAs,. . .
    - Member variables represent the state of the entity
    - Member functions represent actions to modify member variables

All properties	BLAST	SATABS	CPACHECKER	CBMC	KRATOS
Solved	0	0	8	20	53
Safe	0	0	8	-	33
Unsafe	0	0	0	20	20
Time out	2	52	0	0	0
Memory out	43	0	45	0	0
Total time	-	-	17m:7s	2h:41m:22s	**28m:46s**
Max space	-	-	8.4Gb	728.1Mb	**5.2Gb**

Results presented at [CCL+12]

Outline

Related Work

- Sequential Software
  - Many software model checker for sequential software (in C)
    - CPACHECKER [BK11], BLAST [BHJM07], IMPACT [McM06], WOLVERINE [WKM12], LLBMC [FMS13] UFO [AGL+13], SATABS [CKSY05], CBMC [CKL04], . . .
  - Growing interest:
    - Software Model Checking competition `http://sv-comp.sosy-lab.org/2014/`
- Cooperative Threaded Programs
  - SystemC via reduction to NUSMV [Moy05, MMMC05], and to PROMELA [TCMM07, MJM10]
  - SystemC via reduction to software model checking [KS05]
  - SystemC via reduction to Timed Automata [HFG08]
  - SystemC via reduction to CADP [GHPS09]
  - FairThreads via reduction to SIGNAL [JBGT10]
  - OSEK/VDX via reduction to timed automata [WH08]
  - SPECC via CEGAR with NUSMV [CJK07]

Outline

Conclusions

- Three directions for software model checking of cooperative threaded programs
  - Finite-model encoding and analysis with SPIN
  - Translation from cooperative threaded programs to sequential C programs and analysis with any state-of-the-art software model checker
  - ESST algorithms
    - With and without POR

Conclusions

- Three directions for software model checking of cooperative threaded programs
  - Finite-model encoding and analysis with SPIN
  - Translation from cooperative threaded programs to sequential C programs and analysis with any state-of-the-art software model checker
  - ESST algorithms
    - With and without POR
- Instantiated ESST for SystemC and FairThreads

Conclusions

- Three directions for software model checking of cooperative threaded programs
  - Finite-model encoding and analysis with SPIN
  - Translation from cooperative threaded programs to sequential C programs and analysis with any state-of-the-art software model checker
  - ESST algorithms
    - With and without POR
- Instantiated ESST for SystemC and FairThreads
- Implemented the KRATOS software model checker
  - Good performance w.r.t. competitors
  - Successfully applied in industrial setting

Conclusions

- Three directions for software model checking of cooperative threaded programs
  - Finite-model encoding and analysis with SPIN
  - Translation from cooperative threaded programs to sequential C programs and analysis with any state-of-the-art software model checker
  - ESST algorithms
    - With and without POR
- Instantiated ESST for SystemC and FairThreads
- Implemented the KRATOS software model checker
  - Good performance w.r.t. competitors
  - Successfully applied in industrial setting
    - Under integration within Ansaldo STS Design and V&V flow

Future Work

- Semi-Symbolic-Scheduler/Symbolic-Threads (S3ST)
  - Non-constant arguments to primitive function calls
    - Preliminary results for SystemC are positive and encouraging [CNR12b]
- Find safety regions of parametric designs exploiting S3ST
- Apply ESST paradigm to other specification languages and application domains
  - PLC, Automotive, Robotics, etc.

Questions?

Bibliography I

Aws Albarghouthi, Arie Gurfinkel, Yi Li, Sagar Chaki, and Marsha Chechik.
Ufo: Verification with interpolants and abstract interpretation - (competition contribution).
In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 637–640. Springer, 2013.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu.
Bounded model checking.
Advances in Computers, 58:117–148, 2003.

Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani.
Software model checking via large-block encoding.
In *FMCAD*, pages 25–32. IEEE, 2009.

D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar.
The software model checker Blast.
STTT, 9(5-6):505–525, 2007.

D. Beyer and M. E. Keremoglu.
CPAchecker: A Tool for Configurable Software Verification.
In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011.

Bibliography II

D. Beyer, M. E. Keremoglu, and P. Wendler.
Predicate abstraction with adjustable-block encoding.
In R. Bloem and N. Sharygina, editors, *FMCAD*, pages 189–197. IEEE, 2010.

R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar.
Computing Predicate Abstractions by Integrating BDDs and SMT Solvers.
In *FMCAD*, pages 69–76. IEEE, 2007.

Alessandro Cimatti, Raffaele Corvino, Armando Lazzaro, Iman Narasamdya, Tiziana Rizzo, Marco Roveri, Angela Sanseviero, and Andrei Tchaltsev.
Formal verification and validation of ertms industrial railway train spacing system.
In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 378–393. Springer, 2012.

D. Campana, A. Cimatti, I. Narasamdya, and M. Roveri.
An analytic evaluation of SystemC encodings in promela.
In A. Groce and M. Musuvathi, editors, *SPIN*, volume 6823 of *LNCS*, pages 90–107. Springer, 2011.

A. Cimatti, J. Dubrovin, T. Junttila, and M. Roveri.
Structure-aware computation of predicate abstraction.
In *FMCAD*, pages 9–16. IEEE, 2009.

Bibliography III

A. Cimatti, A. Franzén, A. Griggio, K. Kalyanasundaram, and M. Roveri.
Tighter integration of BDDs and SMT for Predicate Abstraction.
In *Proc. of DATE*, pages 1707–1712. IEEE, 2010.

Lucas Cordeiro, Bernd Fischer, and João Marques-Silva.
Smt-based bounded model checking for embedded ansi-c software.
IEEE Trans. Software Eng., 38(4):957–974, 2012.

Alessandro Cimatti and Alberto Griggio.
Software model checking via ic3.
In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2012.

E. M. Clarke, H. Jain, and D. Kroening.
Verification of SpecC using predicate abstraction.
Formal Methods in System Design, 30(1):5–28, 2007.

E. M. Clarke, D. Kroening, and F. Lerda.
A Tool for Checking ANSI-C Programs.
In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav.
SATABS: SAT-Based Predicate Abstraction for ANSI-C.
In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.

Bibliography IV

A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri.
Verifying systemc: A software model checking approach.
In R. Bloem and N. Sharygina, editors, *FMCAD*, pages 51–59. IEEE, 2010.

A. Cimatti, I. Narasamdya, and M. Roveri.
Boosting Lazy Abstraction for SystemC with Partial Order Reduction.
In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*, volume 6605 of *LNCS*, pages 341–356. Springer, 2011.

Alessandro Cimatti, Iman Narasamdya, and Marco Roveri.
Software model checking with explicit scheduler and symbolic threads.
Logical Methods in Computer Science, 8(2), 2012.

Alessandro Cimatti, Iman Narasamdya, and Marco Roveri.
Verification of parametric system designs.
In Gianpiero Cabodi and Satnam Singh, editors, *FMCAD*, pages 122–130. IEEE, 2012.

Alessandro Cimatti, Iman Narasamdya, and Marco Roveri.
Software model checking systemc.
IEEE Trans. on CAD of Integrated Circuits and Systems, 32(5):774–787, 2013.

Bibliography V

Stephan Falke, Florian Merz, and Carsten Sinz.
Llbmc: Improved bounded model checking of c programs using llvm - (competition contribution).
In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 623–626. Springer, 2013.

H. Garavel, C. Helmstetter, O. Ponsini, and Wendelin Serwe.
Verification of an industrial SystemC/TLM model using LOTOS and CADP.
In *MEMOCODE*, pages 46–55. IEEE Computer Society, 2009.

P. Godefroid.
Software Model Checking: The VeriSoft Approach.
F. M. in Sys. Des., 26(2):77–101, 2005.

P. Herber, J. Fellmuth, and S. Glesner.
Model checking SystemC designs using timed automata.
In C. H. Gebotys and G. Martin, editors, *CODES+ISSS*, pages 131–136. ACM, 2008.

T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre.
Lazy abstraction.
In *POPL*, pages 58–70. ACM, 2002.

G. J. Holzmann.
Software model checking with SPIN.
Advances in Computers, 65:78–109, 2005.

Bibliography VI

K. Johnson, L. Besnard, T. Gautier, and J. P. Talpin.
A synchronous approach to threaded program verification.
In *Proc. of the 10th International Workshop on Automated Verification of Critical Systems*, 2010.

D. Kroening and N. Sharygina.
Formal verification of SystemC by automatic hardware/software partitioning.
In *MEMOCODE*, pages 101–110. IEEE, 2005.

S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras.
SMT techniques for fast predicate abstraction.
In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 424–437. Springer, 2006.

Rüdiger Loos and Volker Weispfenning.
Applying linear quantifier elimination.
Computer Journal, 36(5):450–462, 1993.

K. L. McMillan.
Lazy abstraction with interpolants.
In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

Bibliography VII

K. Marquet, B. Jeannet, and M. Moy.
Efficient Encoding of SystemC/TLM in Promela.
Technical report, Verimag, 2010.
Verimag Research Report no TR-2010-7.

M. Moy, F. Maraninchi, and L. Maillet-Contoz.
Lussy: A toolbox for the analysis of systems-on-a-chip at the transactional level.
In *ACSD*, pages 26–35. IEEE, 2005.

David Monniaux.
A Quantifier Elimination Algorithm for Linear Real Arithmetic.
In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - LPAR*, volume 5330 of *LNCS*, pages 243–257. Springer, 2008.

M. Moy.
Techniques and tools for the verification of systems-on-a-chip at the transaction level.
Technical report, INPG, Grenoble, Fr, Dec 2005.

Alexander Schrijver.
Theory of Linear and Integer Programming.
J. Wiley & Sons, 1998.

Bibliography VIII

C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi.
A SystemC/TLM Semantics in Promela and Its Possible Applications.
In D. Bosnacki and S. Edelkamp, editors, *SPIN*, volume 4595 of *LNCS*, pages 204–222. Springer, 2007.

L. Waszniowski and Z. Hanzálek.
Formal verification of multitasking applications based on timed automata model.
Real-Time Systems, 38(1):39–65, 2008.

Georg Weissenbacher, Daniel Kroening, and Sharad Malik.
Wolverine: Battling bugs with interpolants - (competition contribution).
In Cormac Flanagan and Barbara König, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 556–558. Springer, 2012.