Scalable Safety Verification of Statechart-*like* Programs

Kumar Madhukar

Advisors:
Mandayam Srivas
Peter Schrammel

Nov. 26
CMI, Chennai, India

Thanks

TATA CONSULTANCY SERVICES

Scalability issues

- concurrency (Statecharts)
- loops (C programs)
- compositionality in safety refutations (C programs)
- inter-procedural *k*-induction proofs (C programs)

Outline of this talk

- state the problems, and our results
- discuss two problems in details
  - concurrency in Statecharts
  - compositionality in safety refutations (C programs)
- conclusion and prospects

Concurrency in Statecharts

- state-transition diagrams: hierarchical & concurrent (sync)
- **problem**: is a particular configuration reachable?
- widely used in the industry (e.g., in designing vehicle ECUs)
- **existing approaches**: scheduler (sequential); busy-wait (async)
- **our contribution**[1]: effective encoding of the interleavings, in the lazy abstraction framework[2]
- **result**: an order of magnitude speed-up over existing approaches

---

[1] appeared in DATE 2015, Grenoble, France
[2] Ken McMillan, CAV 2006

Loops in C programs

- loops pose a challenge in automated program analysis
- **problem**: does *acceleration*[3] (transitive-closure computation) help existing safety verification tools?
- **existing approaches**: several techniques that tackle loops; could be used in conjunction
- **our contribution**[4]: extensive experimental evaluation to quantify the benefits in off-the-shelf analysis tools

[3] by Kroening et al., FM 2015 & CAV 2013
[4] appeared in FMCAD 2015, Texas, US

Compositional Safety Refutation

- monolithic approach (inlining) does not scale
- **problem**: can safety refutations be arrived at in a modular (inter-procedural) way?
- decomposition is useful, if the results may be composed
- **existing approaches**: assume-guarantee, function summaries
- **our contribution**[5]: developed a framework; implemented and compared three instantiations with different degrees of completeness

[5] appeared in ATVA 2017, Pune, India

Inter-procedural *k*-induction

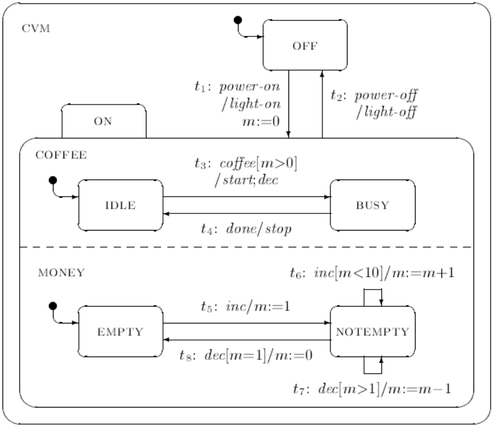
- **problem**: can modular refutation help in doing *k*-induction[6] inter-procedurally?
- the monolithic approach suffers from scalability issues
- **existing approaches**: using abstractions and invariants, assume-guarantee
- **our contribution**: a sound algorithmic framework for inter-procedural *k*-induction with *selective* refinement

[6] Brain et al., SAS 2015

Concurrency in Statecharts

Statecharts

- states: basic, and, or
- labeled transitions ($e[c]/a$); "delayed" actions
- synchronous behaviour
- write-write races are possible
- no read-write races



The context

- **Problem**: Whether a configuration (set of states) is reachable or not?
- The popular approaches include:
  - flattening into a global transition system
  - implementing a scheduler to analyze different interleavings (e.g. Scoot)
- use an analysis tool for asynchronously composed processes (e.g. Impara) by *busy-waiting*

Our approach

- translate Statecharts to multi-threaded C programs
  - to leverage existing tools for analyzing C programs
- extends Lazy Abstraction with Interpolants (LAWI) with support for synchronous concurrency

Statecharts to multi-threaded C programs

- each component is modeled as a separate thread
- program labels for states; transitions are encoded using `goto` jumps from one label to another
- *primed* variables to model delayed writes
- an *environment* process flushes the writes and generates triggers randomly if the system is inactive

An example

```
void coffee()
  idle:

  if(coffee && (m>0))
    start' = 1;
    dec' = 1;
    cState' = busy;
    goto busy;

  else
    inactive = 1;
    cState' = idle;
    goto idle;


  busy:
  ...
```

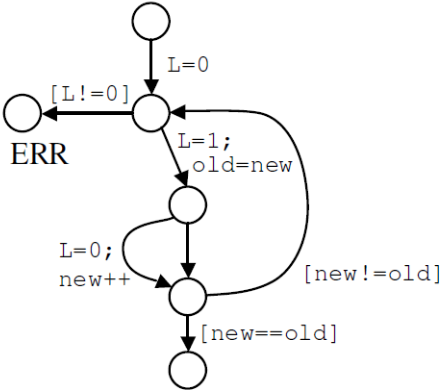
```
void environment()

  start = start';
  dec = dec';

  assert(!((cState == busy) &&
  (mState == empty)));


  if(system is inactive)
    coffee = *;
```

Our approach

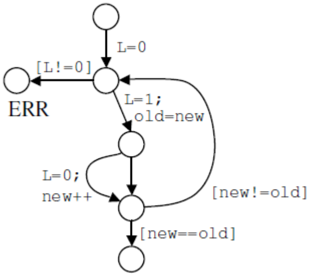
- translate Statecharts to multi-threaded C programs
- extends LAWI to support synchronous concurrency
  - incrementally constructs an abstract reachability tree
  - labels nodes (program points) with interpolants to argue unreachability
  - does not explore states that are already *covered*
  - when the algorithm terminates, the labels are the invariants at those program points

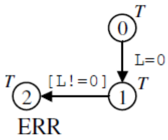
LAWI algorithm

```
L = 0;

do
  assert(L == 0);

  L = 1;     // lock
  old = new;

  if(*)
    L = 0;     // unlock
    new++;

 while(new != old)
```

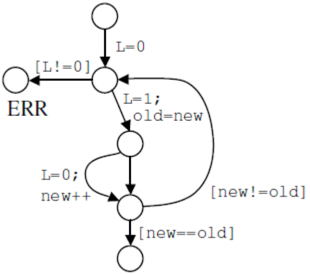


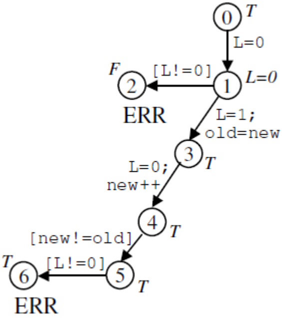
LAWI algorithm: potential error path



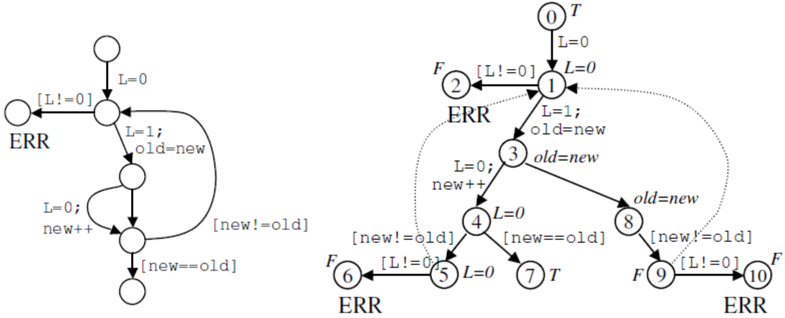


LAWI algorithm: potential error path





LAWI algorithm: cover



SYMPARA & Concurrent SSAs

- encodes the non-determinism of the schedule into the SSA (static single assignment)
- conditional assignments of the form $\underline{x = b?v : v'}$
- *concurrent SSA* allows checking for races (eagerly)
- proposed a modified check for *cover*

Experiments

- compare SYMPARA with CBMC (with *k*-induction) and IMPARA
- encode each example in three different ways, to suit each tool
- the executables and examples are available at `http://www.cmi.ac.in/~madhukar/sympara/`

Comparing the time taken..

No.	Example	Property	Tools		
			SYMPARA	IMPARA	CBMC + k-Induction
			time taken (in seconds); timeout = 900 seconds		
1.	mutex	correctness	5.77	timeout	timeout
2.	mutex	stability	8.02	timeout	timeout
3.	vw_alarm	non-determinism	1.80	timeout	timeout
4.	vw_alarm	stability	214.82	timeout	timeout
5.	sc_alarm	sensitivity	1.35	8.65	2.66
6.	sc_alarm	independence	1.36	9.96	0.46
7.	dragon	correctness	0.64	timeout	timeout
8.	switch	correctness	0.02	0.39	0.12
9.	prod_cons	correctness	0.03	timeout	timeout

- examples are representative of real-life instances

Summarizing this part..

- a technique tailored for verifying synchronous reactive systems
- ANSI-C based input offers applicability to a variety of formalisms (e.g. Lustre, Esterel, Stateflow, SCADE)
- concurrent SSAs encode synchronous concurrency and enable on-the-fly race analysis
- significant performance advantage over CBMC and IMPARA

Compositional Safety Refutation

The context

- divide & conquer approaches effective in scaling verification algorithms
- compositional approaches based on contracts and summaries
- we look at safety refutation in *procedure-modular* decompositions
- *assumptions*: non-recursive programs; loops unwound a finite number of times

An example

```
void main(int x)
  if(x < 10)
    x = foo(x);
    x = foo(x);
    bar(x);
```

```
int foo(int y)
  return y+1;
```

```
void bar(int z)
  assert(z > 10);
```

Entry-exit guards

```
void main(int x)
  if(x < 10)
    x = foo(x);
    x = foo(x);
    bar(x);
```

$$
\begin{aligned}
T_{main}((x_0, g^{in}_{main}), (g^{out}_{main})) \quad &\equiv \quad g^{in}_{foo_0} = (g^{in}_{main} \wedge (x_0 < 10)) \wedge \\
&\quad\;\; foo_0((x_0, g^{in}_{foo_0}), (x_1, g^{out}_{foo_0})) \wedge \\
&\quad\;\; g^{in}_{foo_1} = g^{out}_{foo_0} \wedge \\
&\quad\;\; foo_1((x_1, g^{in}_{foo_1}), (x_2, g^{out}_{foo_1})) \wedge \\
&\quad\;\; g^{in}_{bar} = g^{out}_{foo_1} \wedge \\
&\quad\;\; bar((x_2, g^{in}_{bar}), (g^{out}_{bar})) \wedge \\
&\quad\;\; g^{out}_{main} = ((g^{in}_{main} \wedge \neg(x_0 < 10)) \vee g^{out}_{bar}) \\
Props_{main} \quad &\equiv \quad true
\end{aligned}
$$

```
int foo(int y)
  return y+1;
```

$$
\begin{aligned}
T_{foo}((y, g^{in}_{foo}), (r, g^{out}_{foo})) \quad &\equiv \quad (r = y+1) \wedge (g^{in}_{foo} = g^{out}_{foo}) \\
Props_{foo} \quad &\equiv \quad true
\end{aligned}
$$

```
void bar(int z)
  assert(z > 10);
```

$$
\begin{aligned}
T_{bar}((z, g^{in}_{bar}), (g^{out}_{bar})) \quad &\equiv \quad g^{out}_{bar} = (g^{in}_{bar} \wedge (z > 10)) \\
Props_{bar} \quad &\equiv \quad g^{in}_{bar} \Rightarrow (z > 10)
\end{aligned}
$$

To make it easier to read

```
void main(int x)
  if(x < 10)
    x = foo(x);
    x = foo(x);
    bar(x);
```

$$
\begin{array}{rcl}
T_{main}((x_0, g_0), (g_5)) & \equiv & g_1{=}(g_0 \wedge (x_0{<}10)) \wedge \\
 & & foo_0((x_0, g_1), (x_1, g_2)) \wedge \\
 & & foo_1((x_1, g_2), (x_2, g_3)) \wedge \\
 & & bar((x_2, g_3), (g_4)) \wedge \\
 & & g_5{=}(g_0 \wedge \neg(x_0{<}10) \vee g_4) \\
Props_{main} & \equiv & true
\end{array}
$$

```
int foo(int y)
  return y+1;
```

$$
\begin{array}{rcl}
T_{foo}((y, g_6), (r, g_7)) & \equiv & (r{=}y{+}1) \wedge (g_6{=}g_7) \\
Props_{foo} & \equiv & true
\end{array}
$$

```
void bar(int z)
  assert(z > 10);
```

$$
\begin{array}{rcl}
T_{bar}((z, g_8), (g_9)) & \equiv & g_9{=}(g_8 \wedge (z{>}10)) \\
Props_{bar} & \equiv & g_8 \Rightarrow (z{>}10)
\end{array}
$$

Formal definition (monolithic case)

- inline every procedure call at its call site
- then to refute safety, we must show that the following is satisfiable:

$$\forall \hat{X} : \bigwedge_{j \in CS} g^{in}_{f_{entry}} \wedge T_{fn(j)}(\vec{x}^{in}_j, \vec{x}^{out}_j) \wedge InlineRec_{fn(j)} \wedge \neg Props_{fn(j)}(\vec{x}_j)$$

- solving this is often intractable
- decompose this into smaller subformulae that are faster to solve
  - usually decomposition is over paths e.g. KLEE, UAutomizer

Summaries and Calling Contexts

- approximate the input-output behavior of procedures
- from the callee's and caller's perspective, resp.
- $Sum_{foo}((y, g_6), (r, g_7)) = (y < max\_int \Rightarrow r > y)$
- $CallCtx_{foo_0}((x_0, g_1), (x_1, g_2)) = (g_1 \Rightarrow x_0 < 10)$

Modular Safety Refutation

- a summary for each procedure f – while considering only f and the summaries for the procedures called in f
- start with negation of properties
- compute *maximal* summary and contexts that lead the program to an error state

$$
\begin{aligned}
\max Sum_f, \overbrace{CallCtx_j, \ldots}^{\text{for all } j \in CS_f} : \forall X_f :& \\
& Sum_f(\vec{x}_f^{in}, \vec{x}_f^{out}) \wedge \\
& \bigwedge_{j \in CS_f} CallCtx_j(\vec{x}_j^{p\_in}, \vec{x}_j^{p\_out}) \Longrightarrow \quad (CallCtx_f(\vec{x}_f^{in}, \vec{x}_f^{out}) \vee \neg Props_f) \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad T_f(\vec{x}_f^{in}, \vec{x}_f^{out}) \wedge Sums_f
\end{aligned}
\tag{1}
$$

- piece-wise calling contexts and summaries are combined disjunctively

Working on the example

We start with

$$Sum_{main}((x_0, g_0), (g_5)) = \neg g_0,$$
$$Sum_{foo}((y, g_6), (r, g_7)) = \neg g_6,$$
$$Sum_{bar}((z, g_8), (g_9)) = \neg g_8,$$

$$CallCtx^{*}_{main}((x_0, g_0), (g_5)) = \neg g_5,$$
$$CallCtx_{foo}((y, g_6), (r, g_7)) = false,$$
$$CallCtx_{bar}((z, g_8), (g_9)) = false$$

$$\begin{array}{rcl}
T_{main}((x_0, g_0), (g_5)) & \equiv & g_1{=}(g_0 \wedge (x_0{<}10)) \wedge \\
 & & foo_0((x_0, g_1), (x_1, g_2)) \wedge \\
 & & foo_1((x_1, g_2), (x_2, g_3)) \wedge \\
 & & bar((x_2, g_3), (g_4)) \wedge \\
 & & g_5{=}(g_0 \wedge \neg(x_0{<}10) \vee g_4) \\
Props_{main} & \equiv & true \\
 & & \\
T_{foo}((y, g_6), (r, g_7)) & \equiv & (r{=}y{+}1) \wedge (g_6{=}g_7) \\
Props_{foo} & \equiv & true \\
 & & \\
T_{bar}((z, g_8), (g_9)) & \equiv & g_9{=}(g_8 \wedge (z{>}10)) \\
Props_{bar} & \equiv & g_8 \Rightarrow (z{>}10)
\end{array}$$

Working on the example

We start with

$$
\begin{aligned}
&Sum_{main}((x_0, g_0), (g_5)) = \neg g_0,\\
&Sum_{foo}((y, g_6), (r, g_7)) = \neg g_6,\\
&Sum_{bar}((z, g_8), (g_9)) = \neg g_8,
\end{aligned}
$$

$$
\begin{aligned}
&CallCtx^*_{main}((x_0, g_0), (g_5)) = \neg g_5,\\
&CallCtx_{foo}((y, g_6), (r, g_7)) = false,\\
&CallCtx_{bar}((z, g_8), (g_9)) = false
\end{aligned}
$$

$$
\begin{aligned}
T_{main}((x_0, g_0), (g_5)) &\equiv g_1{=}(g_0 \wedge (x_0{<}10))\wedge\\
&\quad foo_0((x_0, g_1), (x_1, g_2))\wedge\\
&\quad foo_1((x_1, g_2), (x_2, g_3))\wedge\\
&\quad bar((x_2, g_3), (g_4))\wedge\\
&\quad g_5{=}(g_0 \wedge \neg(x_0{<}10) \vee g_4)\\
Props_{main} &\equiv true\\[1ex]
T_{foo}((y, g_6), (r, g_7)) &\equiv (r{=}y{+}1) \wedge (g_6{=}g_7)\\
Props_{foo} &\equiv true\\[1ex]
T_{bar}((z, g_8), (g_9)) &\equiv g_9{=}(g_8 \wedge (z{>}10))\\
Props_{bar} &\equiv g_8 \Rightarrow (z{>}10)
\end{aligned}
$$

The backward traversal begins with *main*. We solve (1):

$$
\begin{aligned}
&\max Sum_{main}, CallCtx_{foo_0}, CallCtx_{foo_1}, CallCtx_{bar} : \forall X_{main} :\\
&\qquad Sum_{main}((x_0, g_0), (g_5))\wedge\\
&\qquad CallCtx_{foo_0}((x_0, g_1), (x_1, g_2))\wedge\\
&\qquad CallCtx_{foo_1}((x_1, g_2), (x_2, g_3))\wedge\\
&\qquad CallCtx_{bar}((x_2, g_3), (g_4)) \implies \quad (\neg g_5 \vee \neg true)\wedge\\
&\qquad\qquad g_1 = (g_0 \wedge (x_0 < 10))\wedge\\
&\qquad\qquad g_5 = ((g_0 \wedge \neg(x_0 < 10)) \vee g_4)\wedge\\
&\qquad\qquad \neg g_1 \wedge \neg g_2 \wedge \neg g_3
\end{aligned}
$$

Working on the example

$$CallCtx_{bar} = \neg g_4,$$

$$CallCtx_{foo_1} = \neg g_3,$$

$$CallCtx_{foo_0} = \neg g_2,$$

$$Sum_{main} = \neg g_0 \wedge \neg g_5$$

$$\begin{array}{rcl}
T_{main}((x_0, g_0), (g_5)) & \equiv & g_1{=}(g_0 \wedge (x_0{<}10)) \wedge \\
 & & foo_0((x_0, g_1), (x_1, g_2)) \wedge \\
 & & foo_1((x_1, g_2), (x_2, g_3)) \wedge \\
 & & bar((x_2, g_3), (g_4)) \wedge \\
 & & g_5{=}(g_0 \wedge \neg(x_0{<}10) \vee g_4) \\
Props_{main} & \equiv & true \\
 & & \\
T_{foo}((y, g_6), (r, g_7)) & \equiv & (r{=}y{+}1) \wedge (g_6{=}g_7) \\
Props_{foo} & \equiv & true \\
 & & \\
T_{bar}((z, g_8), (g_9)) & \equiv & g_9{=}(g_8 \wedge (z{>}10)) \\
Props_{bar} & \equiv & g_8 \Rightarrow (z{>}10)
\end{array}$$

Working on the example

$$CallCtx_{bar} = \neg g_4,$$
$$CallCtx_{foo_1} = \neg g_3,$$
$$CallCtx_{foo_0} = \neg g_2,$$
$$Sum_{main} = \neg g_0 \wedge \neg g_5$$

$$\begin{array}{rcl}
T_{main}((x_0, g_0), (g_5)) & \equiv & g_1 = (g_0 \wedge (x_0 < 10)) \wedge \\
& & foo_0((x_0, g_1), (x_1, g_2)) \wedge \\
& & foo_1((x_1, g_2), (x_2, g_3)) \wedge \\
& & bar((x_2, g_3), (g_4)) \wedge \\
& & g_5 = (g_0 \wedge \neg(x_0 < 10) \vee g_4) \\
Props_{main} & \equiv & true \\
\\
T_{foo}((y, g_6), (r, g_7)) & \equiv & (r = y + 1) \wedge (g_6 = g_7) \\
Props_{foo} & \equiv & true \\
\\
T_{bar}((z, g_8), (g_9)) & \equiv & g_9 = (g_8 \wedge (z > 10)) \\
Props_{bar} & \equiv & g_8 \Rightarrow (z > 10)
\end{array}$$

Then we recur into *bar* with (1) instantiated as:

$$\begin{array}{l}
\max Sum_{bar} : \forall z, g_8, g_9 : \\
\quad Sum_{bar}((z, g_8), (g_9)) \Longrightarrow \quad (\neg g_9 \vee \neg(g_8 \Rightarrow z > 10)) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad (g_9 = (g_8 \wedge z > 10))
\end{array}$$

Working on the example

$$CallCtx_{bar} = \neg g_4,$$
$$CallCtx_{foo_1} = \neg g_3,$$
$$CallCtx_{foo_0} = \neg g_2,$$
$$Sum_{main} = \neg g_0 \wedge \neg g_5$$

$$\begin{array}{rcl}
T_{main}((x_0, g_0), (g_5)) & \equiv & g_1{=}(g_0 \wedge (x_0{<}10)) \wedge \\
 & & foo_0((x_0, g_1), (x_1, g_2)) \wedge \\
 & & foo_1((x_1, g_2), (x_2, g_3)) \wedge \\
 & & bar((x_2, g_3), (g_4)) \wedge \\
 & & g_5{=}(g_0 \wedge \neg(x_0{<}10) \vee g_4) \\
Props_{main} & \equiv & true \\
 & & \\
T_{foo}((y, g_6), (r, g_7)) & \equiv & (r{=}y{+}1) \wedge (g_6{=}g_7) \\
Props_{foo} & \equiv & true \\
 & & \\
T_{bar}((z, g_8), (g_9)) & \equiv & g_9{=}(g_8 \wedge (z{>}10)) \\
Props_{bar} & \equiv & g_8 \Rightarrow (z{>}10)
\end{array}$$

Then we recur into *bar* with (1) instantiated as:

$$\begin{array}{l}
\max Sum_{bar} : \forall z, g_8, g_9 : \\
\quad Sum_{bar}((z, g_8), (g_9)) \Longrightarrow \quad (\neg g_9 \vee \neg(g_8 \Rightarrow z > 10)) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad (g_9 = (g_8 \wedge z > 10))
\end{array}$$

Hence, we get for Sum_{bar}: $(g_8 \Rightarrow \neg(z > 10)) \wedge \neg g_9$

Working on the example

- and so on ...
- we update $CallCtx_{foo}$, and compute Sum_{foo} to be $(g_6 \Rightarrow \neg(r > 10) \wedge g_7) \wedge (r = y + 1)$
- and, after another invocation of the composition operator, $Sum_{main} = (g_0 \Rightarrow \neg(x_0 > 8)) \wedge \neg g_5$
- hence, $(x \leq 8)$ is a (maximal) refutation witness

The example

```
void main(int x)
  if(x < 10)
    x = foo(x);
    x = foo(x);
    bar(x);
```

$$
\begin{aligned}
T_{main}((x_0, g_0), (g_5)) &\equiv g_1{=}(g_0 \wedge (x_0{<}10))\wedge \\
&\quad\ foo_0((x_0, g_1), (x_1, g_2))\wedge \\
&\quad\ foo_1((x_1, g_2), (x_2, g_3))\wedge \\
&\quad\ bar((x_2, g_3), (g_4))\wedge \\
&\quad\ g_5{=}(g_0 \wedge \neg(x_0{<}10) \vee g_4) \\
Props_{main} &\equiv true
\end{aligned}
$$

```
int foo(int y)
  return y+1;
```

$$
\begin{aligned}
T_{foo}((y, g_6), (r, g_7)) &\equiv (r{=}y{+}1) \wedge (g_6{=}g_7) \\
Props_{foo} &\equiv true
\end{aligned}
$$

```
void bar(int z)
  assert(z > 10);
```

$$
\begin{aligned}
T_{bar}((z, g_8), (g_9)) &\equiv g_9{=}(g_8 \wedge (z{>}10)) \\
Props_{bar} &\equiv g_8 \Rightarrow (z{>}10)
\end{aligned}
$$

We prove that

Theorem

$Sum_{f_{entry}}$ would be false in modular refutation if and only if the monolithic refutation formula is unsat.

Proof.

Induction on the size of call stack □

Examples of Refutation Algorithms

- three different instantiations of the compositional framework
  - domain used to propagate summaries and calling contexts
- *concrete* backward interpretation
  - a single constant value for each variable

$$
\begin{aligned}
&\max Sum_{bar} : \forall z, g_8, g_9 : \\
&\quad Sum_{bar}((z, g_8), (g_9)) \Longrightarrow \quad (\neg g_9 \vee \neg(g_8 \Rightarrow z > 10)) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad (g_9 = (g_8 \wedge z > 10))
\end{aligned}
$$

$$
\begin{aligned}
&\exists d : \forall z, g_8, g_9 : \\
&\quad (g_8 \Rightarrow z{=}d) \Longrightarrow \quad (\neg g_9 \vee \neg(g_8 \Rightarrow (z > 10))) \wedge \\
&\qquad\qquad\qquad\qquad\quad (g_9 = (g_8 \wedge z > 10))
\end{aligned}
$$

The example

```
void main(int x)
  if(x < 10)
    x = foo(x);
    x = foo(x);
    bar(x);
```

$$
\begin{aligned}
T_{main}((x_0, g_0), (g_5)) &\equiv g_1{=}(g_0 \wedge (x_0{<}10)) \wedge \\
&\quad\; foo_0((x_0, g_1), (x_1, g_2)) \wedge \\
&\quad\; foo_1((x_1, g_2), (x_2, g_3)) \wedge \\
&\quad\; bar((x_2, g_3), (g_4)) \wedge \\
&\quad\; g_5{=}(g_0 \wedge \neg(x_0{<}10) \vee g_4) \\
Props_{main} &\equiv true
\end{aligned}
$$

```
int foo(int y)
  return y+1;
```

$$
\begin{aligned}
T_{foo}((y, g_6), (r, g_7)) &\equiv (r{=}y{+}1) \wedge (g_6{=}g_7) \\
Props_{foo} &\equiv true
\end{aligned}
$$

```
void bar(int z)
  assert(z > 10);
```

$$
\begin{aligned}
T_{bar}((z, g_8), (g_9)) &\equiv g_9{=}(g_8 \wedge (z{>}10)) \\
Props_{bar} &\equiv g_8 \Rightarrow (z{>}10)
\end{aligned}
$$

Examples of Refutation Algorithms

- three different instantiations of the compositional framework
  - domain used to propagate summaries and calling contexts
- *concrete* backward interpretation
  - a single constant value for each variable
- *abstract* backward interpretation
  - disjunction of intervals for variables

Examples of Refutation Algorithms

- three different instantiations of the compositional framework
  - domain used to propagate summaries and calling contexts
- *concrete* backward interpretation
  - a single constant value for each variable
- *abstract* backward interpretation
  - disjunction of intervals for variables
- *symbolic* backward interpretation (weakest precondition with slicing)
  - summaries may be as big as the procedures themselves

How does compositional refutation help

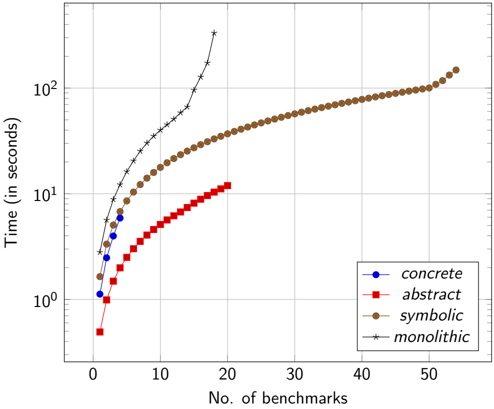
- spuriousness checks often succeed faster (without having to go all the way up to the entry function)
- summaries could be cached and reused

Experiments

- implemented these techniques in 2LS (a tool built on the CPROVER framework)
- sources at `https://github.com/kumarmadhukar/2ls/tree/atva17`
- 265 benchmarks from SV-COMP (*product-lines*)
- reasonably complex – 83 procedures per benchmark (on avg.)
- chose an unwinding depth of 5

Results

- compositional approaches faster than monolithic
- more complete techniques solve more examples
- concrete is slower (but uses SAT; can be made faster)



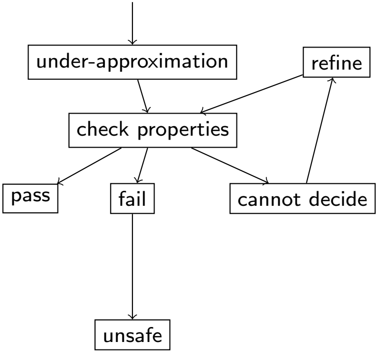
Benefits of this approach

- spuriousness checks often succeed faster (without having to go all the way up to the entry function)
- summaries can be cached and reused

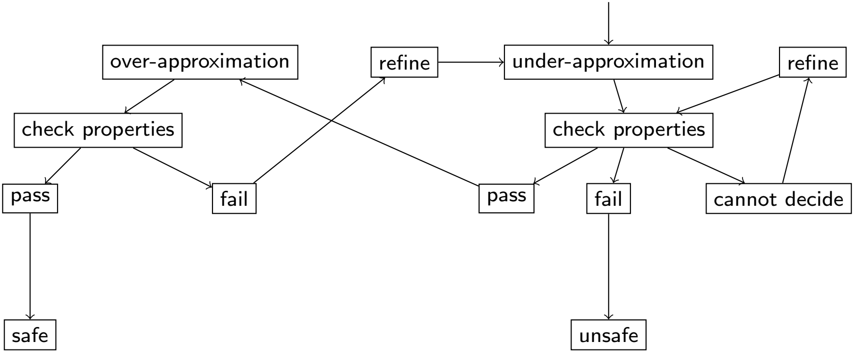
Summarizing this part..

- compositional refutation in procedure-modular decompositions
- the abstract technique solves as many benchmarks as the monolithic one, even with only interval domains
- semantic decompositions – independent of the program's syntactic structure (*future work*)

How this fits in..



How this fits in..



Conclusion

- this thesis looks at techniques that aim at exploiting source-level syntactic and/or semantic structure of the input program
- opens up some interesting directions of future work
- for example, in the modular approaches that we have explored, the division is syntactic; it would be interesting to explore semantic decompositions

Thank you!

Questions?

Appendix

Acceleration helping invariant generation

- experimental evidence that proofs are easier to obtain
- consider, for example, the lazy abstraction with interpolants approach
- interpolants blocking all the paths in the loops may be obtained at once (due to the accelerated fragment), as compared to iterative strengthening

Handling multiple and nested loops

- multiple and nested loops are handled with different unwindings for each loop
- we may also have a summary (or an invariant) for the inner loop and an unwinding for the outer loop
- refinement may strengthen the summary, or may add more unwindings to the loops
- we propose to do this *selectively* – by localizing the problem found during the spuriousness check