An insight into

SMT-based model checking techniques

for formal software verification of synchronous dataflow programs

Jonathan Laurent

École Normale Supérieure, National Institute of Aerospace, NASA Langley Research Center

August 11^{th}, 2014

The Copilot Language

An embedded language for monitoring embedded systems

The Copilot language

Copilot is a *synchronous dataflow* language embedded in *Haskell*. It is designed for writing monitors for real-time C programs.

A Copilot program can be :

- Interpreted.
- Compiled to a constant-space C program which can be linked with the monitored code.
- Inspected with some static analysis tools.

A Copilot program consists of :

- A list of mutually recursive stream equations.
  External C variables can be imported as external streams.
- A list of triggers, which are external C functions and which should be called when some special event occurs.

The Copilot language

A *stream* of type T is an infinite sequence of values of type T. Available types include

`Bool`, `Word8`, `Word64`, `Float`, `Double` ...

Literal constants should be interpreted as constant streams. Standard operators are defined pointwise over streams. For instance :

```
x :: Stream Word64
x = 5 + 5
y :: Stream Word64
y = x * x
z :: Stream Bool
z = (x == 10) && (y < 200)
t :: Stream Word64
t = if z then x + 1 else y
```

```
x = { 10, 10, 10 ...}
y = { 100, 100, 100 ...}
z = { true, true ...}
t = { 11, 11, 11 ...}
```

The Copilot language

Two temporal operators are provided.

The `++` operator delays a stream by prepending a finite list of values to it

```
x :: Stream Word64
x = 10
y :: Stream Word64
y = [1, 2, 3] ++ x
```

```
x = { 10, 10, 10 ...}
y = { 1, 2, 3, 10, 10 ... }
```

For an integer `n`, `drop n` strips the `n` first values of a stream

```
z :: Word64
z = drop 2 y
```

```
z = { 3, 10, 10, 10 ... }
```

The Copilot language

It is possible to use recursive definitions :

```
evens :: Stream Word64
evens = [0] ++ (1 + odds)
odds  = 1 + evens
```

```
evens = { 0, 2, 4, 6 ...}

odds = { 1, 3, 5, 7 ... }
```

The Fibonacci sequence can be defined as

```
fib :: Stream Word64
fib = [1, 1] ++ (fib + drop 1 fib)
```

For comparison, valid Haskell code with the same purpose :

```
fib :: [Int]
fib = [1, 1] ++ zipWith (+) fib (tail fib)
```

Metaprogramming with Copilot

Copilot enables us to do some metaprogramming :

```
bitCounter :: Int -> [Stream Bool]
bitCounter n
  | iszero n  = []
  | otherwise = bn : cnts
  where bn   = [False] ++ if conj cnts then not bn else bn
        cnts = bitCounter (n - 1)
        conj = foldl (&&) true
```

For instance, `bitCounter 4` evaluates to a list of four streams $[s_0, s_1, s_2, s_3]$ yielding

s_0	0	1	0	1	0	1	0	1	0	1	...
s_1	0	0	1	1	0	0	1	1	0	0	...
s_2	0	0	0	0	1	1	1	1	0	0	...
s_3	0	0	0	0	1	1	1	1	0	0	...

where 0 stands for `false` and 1 for `true`.

The Copilot language

Safety properties on Copilot programs are expressed with standard boolean streams. For instance,

```
x  = [1] ++ (1 + x)
ok = x /= 0
```

We then use an external tool which takes as an input a program and the name of a stream *s* and tries to prove that *s* is equal to the constant stream `true`.

A real world example

We want to ensure the following behavior :

If the engine temperature probe exceeds 250 degrees, then the cooler is engaged and remains engaged until the engine temperature probe drops to below the limit. Otherwise, the immediate shutdown of the engine is triggered.

```
engineMonitor :: Spec
engineMonitor = do
  trigger "shutoff" (not ok) []
  where
    exceed   = (externW8 "tmp_probe" Nothing) > 250
    ok       = exceed ==> extern "cooler" Nothing
```

In real world, we don't trust only one temperature probe. We read many of them and use a majority vote algorithm, like the Boyer Moore algorithm.

```
engineMonitor :: Spec
engineMonitor = do
  trigger "shutoff"  (not ok) []
  where
    probes =
      [ externW8 "tmp_probe_1" Nothing
      , externW8 "tmp_probe_2" Nothing
      , externW8 "tmp_probe_3" Nothing ]

    exceed     = map (> 250) probes
    maj        = majority exceed
    checkMaj   = hasMajority maj exceed
    ok         = (maj && checkMaj) ==> extern "cooler" Nothing
```

```haskell
majority :: forall a . (Typed a, Eq a) => [Stream a] -> Stream a
majority [] = error "empty list"
majority (x : xs) = aux x 1 xs
  where
  aux :: Stream a -> Stream Word8 -> [Stream a] -> Stream a
  aux p _s [] = p
  aux p s (l : ls) =
    local (if s == 0 then l else p) $ \p' ->
    local (if s == 0 || l == p then s + 1 else s - 1) $ \s' ->
    aux p' s' ls

holdsMajority :: forall a . (Typed a, Eq a) =>
  [Stream a] -> Stream a -> Stream Bool
holdsMajority maj l =
  (2 * count maj l) <= length l
  where
    count :: Stream a -> [Stream a] -> Stream Word8
    count _ [] = 0
    count e (x : xs) = (if x == e then 1 else 0) + count e xs
```

```haskell
engineMonitor :: Spec
engineMonitor = do
  trigger "shutoff"  (not ok) []
  prop    "prop_maj" ( forAllBoolCste $ \b ->
                       (maj /= b) ==> not (hasMajority b exceed) )
  where
    probes =
      [ externW8 "tmp_probe_1" Nothing
      , externW8 "tmp_probe_2" Nothing
      , externW8 "tmp_probe_3" Nothing ]

    exceed    = map (> 250) probes
    maj       = majority exceed
    checkMaj  = hasMajority maj exceed
    ok        = (maj && checkMaj) ==> extern "cooler" Nothing
```

State transition systems

A natural formalism for model checking

State transition systems

Some basic terminology

A *state transition system* is a triple $(Q, I, \to)$ where

- Q is a set of states
- I is a set of initial states
- $\to$ is a transition relation over Q

A few definitions :

- A state $s \in Q$ is said to be *k-reachable* if there exists a path from an initial state $i \in I$ to state s of length at most k.
- It is said to be *reachable* if k-reachable for some $k \in \mathbb{N}$.
- Likewise, a set of states $P \subset Q$ is said to be reachable if there exists a reachable state in P.

Problem :

Given a transition system and a set of *safe* states P, are all the reachable states of the system in P ? In this case, P is said to be an *invariant*.

State transition systems

Some reminders and notation for logics

- $\mathcal{L}$ is a logic such that the satisfiability of quantifier-free formulas is decidable. Its set of values is denoted by $V_{\mathcal{L}}$.
- Bold letters indicate vectors. Therefore,
  - $F[x_1, \cdots, x_n]$ is written $F[\mathbf{x}]$
  - $G[x_1, \cdots, x_n, y_1, \cdots, y_p]$ is written $G[\mathbf{x}, \mathbf{y}]$
- $F[\mathbf{v}]$ stands for $F[\mathbf{x}]$ where all occurences of x_i are replaced by v_i simultaneously.
- $F[\mathbf{x}] \models G[\mathbf{x}]$ iff $F[\mathbf{x}] \wedge \neg G[\mathbf{x}]$ is not satisfiable

Problem

How to deal with large sets of states?

We focus on transition systems such that a state is fully described by the value of *n state variables*. In other words,

$$Q \simeq V_{\mathcal{L}}^{n}$$

Moreover, we assume that there are two formulas $I[\mathbf{x}]$ and $T[\mathbf{x}, \mathbf{x}']$ such that :

- $\mathbf{s}$ is an initial state iff $I[\mathbf{s}]$ holds
- $\mathbf{s} \rightarrow \mathbf{s}'$ iff $T[\mathbf{s}, \mathbf{s}']$ holds

Copilot programs as state transition systems

Giving Copilot a small-step operational semantics

We first transform the program such that all equations are of the form

```
s = l ++ e
```

with a stream name `s`, a list `l` of size at most 1, and an expression `e` without any temporal operator. For instance :

```
fib = [1, 1] ++ (fib + drop 1 fib)
```

is flattened into

```
fib0 = [1] ++ fib1
fib1 = [1] ++ (fib1 + fib0)
```

and is translated into a transition system where

$$
\begin{aligned}
Q &= \mathbb{Z}^2 \\
I\,[\, f_0, f_1 \,] &= (f_0 = 1) \wedge (f_1 = 1) \\
T\,[\, f_0, f_1, f_0', f_1' \,] &= (f_0' = f_1) \wedge (f1' = f_0 + f_1)
\end{aligned}
$$

Copilot programs as state transition systems

Giving Copilot a small-step operational semantics

- Handling non-determinism : just introduce some unconstrained state variables. It is useful to let the user write assumptions on the non-deterministic streams of the program.
- Handling if-then-else constructions :

  ```
  y = externW8 "y" Nothing
  x = 1 + (if y < 0 then -y else y)
  ```

  becomes

$$
T\,[\,x, y, i, x', y', i'\,] = \begin{array}{cl} & (\,x' = 1 + i'\,) \\ \wedge & (\,y' < 0 \;\Rightarrow\; i' = -y'\,) \\ \wedge & (\,\neg\,(y' < 0) \;\Rightarrow\; i' = y'\,) \end{array}
$$

- Handling operators absent in $\mathcal{L}$: if $\mathcal{L}$ handles *uninterpreted functions*, use one for each unknown operator.

The k-induction algorithm

A *nice* proving strategy for *nice* properties

Bounded model checking

From now on, we assume (Q, I, T) is a transition system, and P a set of safe states.

The basic idea of *Bounded Model Checking (BMC)* is to check the following entailment (ie. logical consequence)

$$I[\mathbf{x_0}] \wedge T[\mathbf{x_0}, \mathbf{x_1}] \wedge \cdots \wedge T[\mathbf{x_{k-1}}, \mathbf{x_k}] \models P[\mathbf{x_k}]$$

for increasing values of k.

In the case $\neg P$ is reachable, the SMT solver will provide an assignment for

$$I[\mathbf{x_0}] \wedge T[\mathbf{x_0}, \mathbf{x_1}] \wedge \cdots \wedge T[\mathbf{x_{k-1}}, \mathbf{x_k}] \wedge \neg P[\mathbf{x_k}]$$

which can be used as a counterexample trace.

The k-induction algorithm

The most natural way to prove P is invariant is to proceed by induction, checking the two entailments :

$$I\,[\,\mathbf{x}\,] \models P\,[\,\mathbf{x}\,] \quad \text{(initiation)}$$

$$P\,[\,\mathbf{x}\,] \wedge T\,[\,\mathbf{x}, \mathbf{x}'\,] \models P\,[\,\mathbf{x}'\,] \quad \text{(consecution)}$$

We can generalize this idea with k-induction, testing these two entailments for increasing values of k :

$$I\,[\,\mathbf{x_0}\,] \wedge T\,[\,\mathbf{x_0}, \mathbf{x_1}\,] \wedge \cdots \wedge T\,[\,\mathbf{x_{k-1}}, \mathbf{x_k}\,] \;\models\; P\,[\,\mathbf{x_k}\,] \quad \text{(i.)}$$

$$P\,[\,\mathbf{x_0}\,] \wedge \cdots \wedge P\,[\,\mathbf{x_{k-1}}\,] \wedge T\,[\,\mathbf{x_0}, \mathbf{x_1}\,] \wedge \cdots \wedge T\,[\,\mathbf{x_{k-1}}, \mathbf{x_k}\,] \;\models\; P\,[\,\mathbf{x_k}\,] \quad \text{(c.)}$$

The k-induction algorithm

An example

We want to prove the invariant `ok` in the following program :

```
x  = [1] ++ y
y  = [0] ++ x
ok = (x == 0) || (x == 1)
```

For this, we start two SMT-solvers in parallel, one for BMC and the other to check consecution. We will look at the latter.

```
(set-logic QF_LIA)
(declare-fun x  () Int)
 ...
(declare-fun y' () Int)

(assert (or (= x 0) (= x 1)))
(assert (and (= x' y) (= y' x)))

(push 1)
(assert) (not (or (= x' 0) (= x' 1)))
(check-sat)
(pop 1)

(declare-fun x'' () Int)
(declare-fun y'' () Int)

(assert (or (= x' 0) (= x' 1)))
(assert (and (= x'' y') (= y'' x')))

(push 1)
(assert) (not (or (= x'' 0) (= x'' 1)))
(check-sat)
```

```
x  = [1] ++ y
y  = [0] ++ x
ok = (x == 0) || (x == 1)
```

$T[x, y, x', y'] \wedge P[x, y]$
$\wedge \neg P[x', y']$

```
> sat
```

$\{x = 0,\ y = 2,\ x' = 2,\ y' = 0\}$

$T[x, y, x', y'] \wedge T[x', y', x'', y'']$
$\wedge P[x, y] \wedge P[x', y']$
$\wedge \neg P[x'', y'']$

```
> unsat
```

Path compression

If P is not invariant, a counterexample trace exists such that :

- It is cycle-free
- Only its first state belongs to I

Therefore, we define :

$$C_k\,[\,\mathbf{x_0}\,,\,\cdots\,,\,\mathbf{x_k}\,] = \bigwedge_{i \neq j} \mathbf{x_i} \neq \mathbf{x_j} \;\wedge\; \bigwedge_{i > 0} \neg I\,[\,\mathbf{x_i}\,]$$

where the equality is defined pointwise over vectors.

We can now strengthen the left hand side of the *continuation* entailment, which becomes :

$$\bigwedge_{i=0}^{k-1} P\,[\,\mathbf{x_i}\,] \;\wedge\; \bigwedge_{i=0}^{k-1} T\,[\,\mathbf{x_i}, \mathbf{x_{i+1}}\,] \;\wedge\; C_k\,[\,\mathbf{x_0}\,,\,\cdots\,,\,\mathbf{x_k}\,] \models\; P\,[\,\mathbf{x_k}\,]$$

Path compression

Now, we can make the algorithm complete for bounded state spaces by testing the entailement :

$$I\,[\,\mathbf{x_0}\,] \wedge T\,[\,\mathbf{x_0}, \mathbf{x_1}\,] \wedge \cdots \wedge T\,[\,\mathbf{x_{k-1}}, \mathbf{x_k}\,] \;\models\; \neg C_k\,[\,\mathbf{x_0}\,,\,\cdots\,,\,\mathbf{x_{k-1}}\,]$$

after the k^{th} BMC iteration.

If it holds and P is k-invariant, then P is invariant.

Example

In the following program, we want to prove that `ok` is an invariant :

```
x  = [4] ++ if x == 10 then 0 else x + 2
ok = (x < 11)
```

This cannot be done with the basic k-induction algorithm. With path compression, the periodicity of `x` is determined.

Structural abstraction

An important *limiting factor* of the k-induction algorithm is the *computing power* needed by the SMT solver.

In order to scale our approach, we need to reduce the size of the problems discharged. One important idea for this is *structural abstraction*.

The idea is to replace T by a weaker approximation $T^\sharp$ by removing some clauses. Then,

- If P is invariant for $T^\sharp$, it is for T and we are done.
- Otherwise, we use the counterexample given by the SMT solver to refine $T^\sharp$ by restoring some well-chosen clauses.

Problem

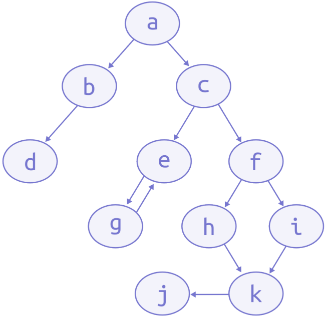
What clauses to choose?

Structural abstraction

We build a *dependency graph* where

- Each variable is given a vertex
- There is an edge $x \rightarrow y$ iff y appears in the *definition* of x

```
a = (b == 0) || (c == 1)
b = if d <= 0 then 0 else d
c = e + f + 1
d = extern "d" Nothing
e = [0] ++ g
f = if h == i then -1 else 1
g = [0] ++ e
h = k + 1
i = k + 1
j = extern "j" Nothing
k = -j
```

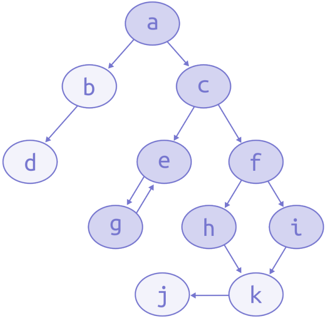


Structural abstraction

We build a *dependency graph* where

- Each variable is given a vertex
- There is an edge $x \to y$ iff y appears in the *definition* of x

```
a = (b == 0) || (c == 1)
b = if d <= 0 then 0 else d
c = e + f + 1
d = extern "d" Nothing
e = [0] ++ g
f = if h == i then -1 else 1
g = [0] ++ e
h = k + 1
i = k + 1
j = extern "j" Nothing
k = -j
```



Structural abstraction

If a counterexample $(\mathbf{s_0}\,, \cdots, \mathbf{s_k})$ is found with $T^\sharp$, we check whether or not

$$\bigwedge_{i=0}^{k} \bigwedge_{j=1}^{n} x_{ij} = s_{ij} \ \wedge \ I\,[\,\mathbf{x}\,] \ \wedge \ \bigwedge_{i=0}^{k-1} T\,[\,\mathbf{x_i}, \mathbf{x_{i+1}}\,]$$

is satisfiable.

- If it is, then the counterexample is also valid for T.
- Otherwise, the counterexample is *spurious*.
  It is possible to get an explanation of *why* by asking the SMT solver for an *unsatisfiable core*. The state variables which are candidates for refinement are then the x_j such that a clause $x_{ij} = s_{ij}$ was kept in this core.

Among these variables, we pick one, usually the deepest in the dependency graph. We then use a heuristic to refine some of its transitive successors.

The IC3 Algorithm

A property directed reachability algorithm

Inductive properties

A property P is said to be inductive if

$$P[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models P[\mathbf{x}']$$

Otherwise there exists states $\mathbf{s}$ and $\mathbf{s}'$ such that

$$P[\mathbf{s}] \wedge T[\mathbf{s}, \mathbf{s}'] \wedge \neg P[\mathbf{s}']$$

and $\mathbf{s}$ is said to be a CTI (Counterexample To the Inductiveness)

P is said to be inductive relative to A iff

$$A[\mathbf{x}] \wedge P[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models P[\mathbf{x}']$$

We can generalize these notions to k-inductiveness.

Inductive properties

An example

For instance, let's say we want to prove $y \geq 1$ on the following program :

```
x = [0] ++ (1 + x)
y = [1] ++ (x + y)
```

This property is not inductive. A CTI is

$$\{x = -2, y = 1\}$$

However, it is inductive relative to $x \geq 0$.

Therefore, we can prove $y \geq 1$ after having proved the *lemma* $x \geq 0$.

The FSIS algorithm

A more incremental approach to model checking

- We try to prove that P is inductive.
- In case of failure, if $\mathbf{s}$ is a CTI, we *find* an inductive invariant ϕ_0 such that $\phi_0[\mathbf{s}]$ does not hold.
- We go on finding such lemmas ϕ_0, $\phi_1, \cdots$ until there are no more CTIs.
- At the end, as P is inductive relative to $\bigwedge_i \phi_i$, it is proven invariant if it holds for the initial states.

Note that each ϕ_i has only to be proven inductive relative to the previous ones. Thanks to the following property, it is even sufficient to prove ϕ_i inductive relative to $\{\phi_j\}_{j<i} \wedge P$.

Property

If P is inductive relative to Q and Q is inductive relative to P, then $P \wedge Q$ is inductive.

The IC3 algorithm

Main idea

As it is difficult to find inductive lemmas, we will only search for lemmas which are inductive relative to a formula R_k that over-approximates the set of k-reachable states. We then try to propagate these lemmas as k increases, stopping when the sequence (R_k) reaches a fixed point.

We want to find a superset R of the set of reachable states such that :

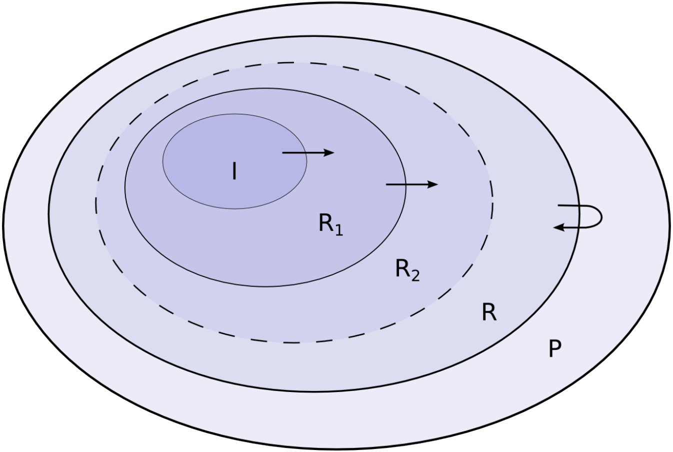
$$I\,[\,\mathbf{x}\,] \models R\,[\,\mathbf{x}\,] \qquad R\,[\,\mathbf{x}\,] \wedge T\,[\,\mathbf{x},\mathbf{x}'\,] \models R\,[\,\mathbf{x}'\,] \qquad R\,[\,\mathbf{x}\,] \models P\,[\,\mathbf{x}\,]$$

We will build R as the fixed point of an increasing sequence of *frames* (R_k). Each R_k can be seen as a set of formulas called *lemmas* or as a single formula which is the conjunction of all its lemmas. This sequence has the following properties :

$$R_0 = \{I\} \qquad R_{i+1} \subset R_i$$

$$R_i\,[\,\mathbf{x}\,] \wedge T\,[\,\mathbf{x},\mathbf{x}'\,] \models R_{i+1}\,[\,\mathbf{x}'\,] \qquad P \in R_i \text{ for } i > 0$$

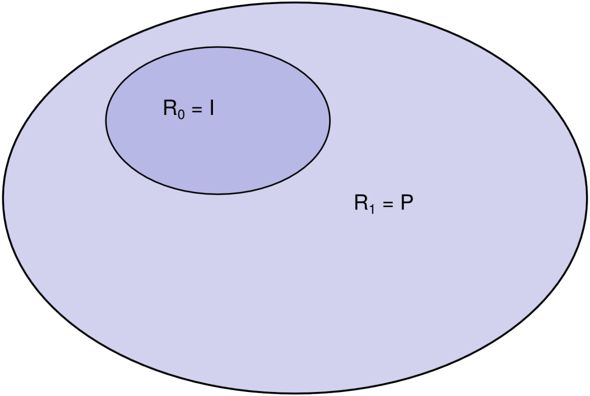
If all of this holds and (R_k) admits a fixed point, it is clear that P is invariant.



$$R_0 = \{I\} \qquad R_{i+1} \subset R_i$$

$$R_i\,[\,\mathbf{x}\,] \wedge T\,[\,\mathbf{x}, \mathbf{x}'\,] \models R_{i+1}\,[\,\mathbf{x}'\,] \qquad P \in R_i \text{ for } i > 0$$

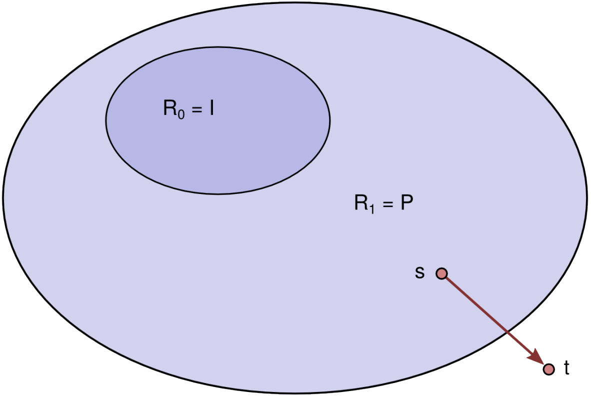
$k = 1$



Each new frame R_k is initialized with P. We then

- Strengthen R_k and the previous frames such that an error state is not reachable in one step from R_k.
- Propagate as many lemmas as possible from the previous frame.

$k = 1$

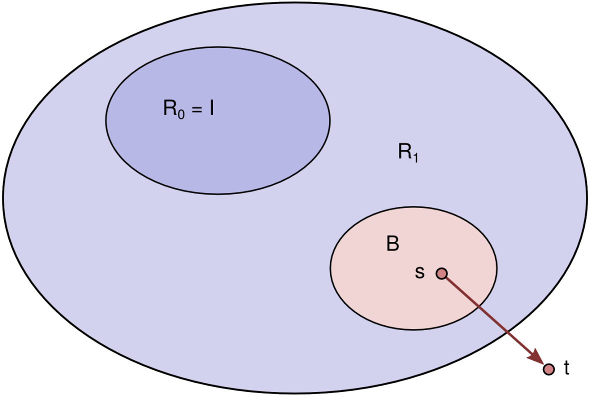


To strengthen a new frame, we first check the entailment

$$R_k\,[\,\mathbf{x}\,] \wedge T\,[\,\mathbf{x}, \mathbf{x}'\,] \models P\,[\,\mathbf{x}'\,]$$

If it holds, the *strengthening* terminates. Otherwise, Let $(\mathbf{s}, \mathbf{t})$ be a cex.

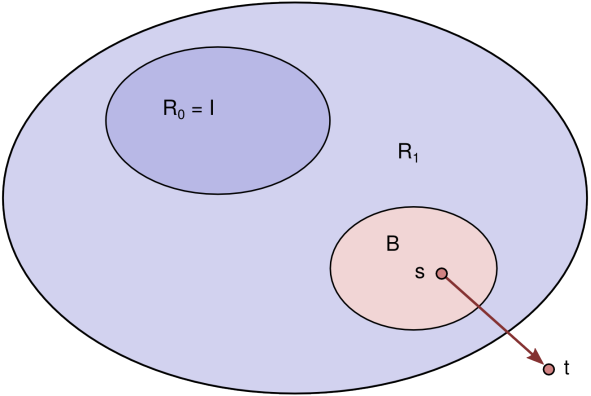
$k = 1$



We generalize the *counterexample*, finding a set B of *bad states* such that each state of B leads to an error-state in one step too.

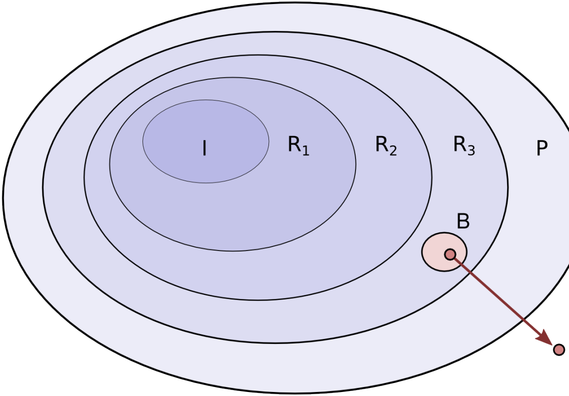
Now, we could want to prove B unreachable and add $\neg B$ as a lemma in R_k.

$k = 1$



However, B may not be inductive. Therefore, we'll just try to prove it doesn't intersect with I and it is inductive relative to the *previous* frame. In particular, this would imply that B is not k-reachable, as each R_i over-approximate the set of i-reachable states. This is easy for the special case $k = 1$.

$k = 3$

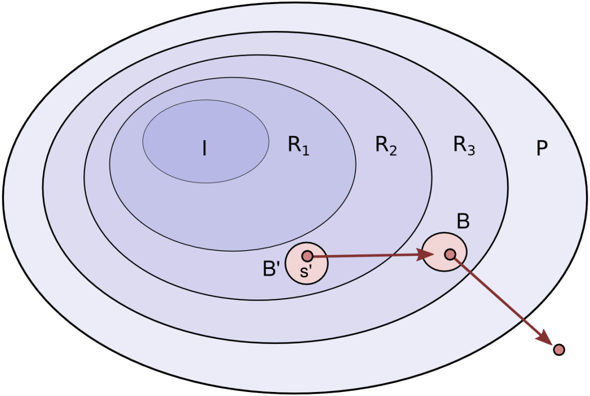


In a more general situation, we check these two entailments :

$$I\,[\,\mathbf{x}\,] \models \neg B\,[\,\mathbf{x}\,] \qquad R_{k-1}\,[\,\mathbf{x}\,] \wedge \neg B\,[\,\mathbf{x}\,] \wedge T\,[\,\mathbf{x}, \mathbf{x}'\,] \models \neg B[x']$$

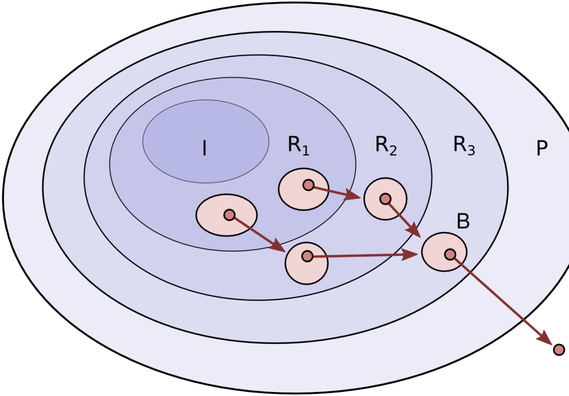
If the first fails, P is not an invariant.

$k = 3$



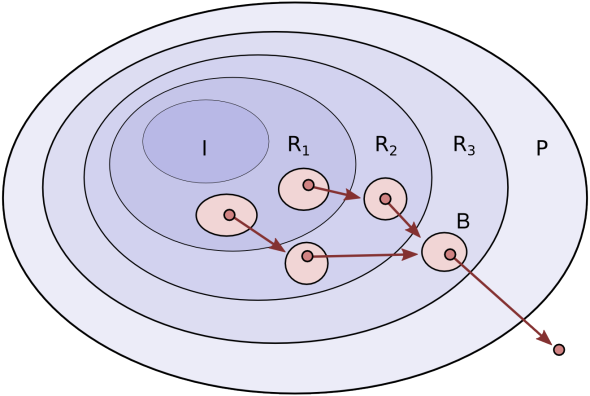
If the second fails, it means that B is reachable from a state s' of the previous frame. We generalize s' into a set B' of states belonging to the previous frame which lead to B in one step. Now, we have to recursively refine R_{k-1} to eliminate B' from it.

$k = 3$



And we go on, performing kind of a backward-reachability analysis.

$k = 3$



This process stops when one of the following events happen :

- We reach $R_0 = I$ and then P is not invariant
- All the bad states recursively generated are proven to be $k-$unreachable

Once the strengthening process completes, which means no error state is reachable in one step from the last frame, we try to propagate the new lemmas we've discovered, from each frame to its successor. It is done by checking the entailment

$$R_j\,[\,\mathbf{x}\,] \wedge T\,[\,\mathbf{x},\mathbf{x}'\,] \models C\,[\,\mathbf{x}'\,]$$

for all $C \in R_j \backslash R_{j+1}$, with j increasing from 0 to k.

If all the lemmas of R_{k-1} are successfuly propagated to R_k, the (R_j) sequence reaches a fixed point and P is proven invariant.

Otherwise, we add a new frame R_{k+1} initialized with $\{P\}$ and the algorithm continue.

Counterexample generalization

If B is a set of bad states and R a frame, we often check an entailment of the form :

$$F[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models \neg B[\mathbf{x}'] \qquad (*)$$

If it does not hold, we try to find some additional lemmas discarding all the counterexamples to $(*)$.

In fact, a single lemma $\neg G$ is sufficient if

$$G[\mathbf{x}] = \exists\, \mathbf{x}' .\, F[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \wedge B[\mathbf{x}']$$

Unfortunately, G is not quantifier-free.

Basic quantifier elimination

Definition

A logic $\mathcal{L}$ is said to admit *quantifier elimination* if for all formulas ϕ, there exists a quantifier-free formula ψ such that $\models_{\mathcal{L}} \phi \Leftrightarrow \psi$.

Theorem

$\mathcal{L}$ admits QE iff every formula of the form $\exists\, x.\ \phi\,[\,x\,]$ where ϕ is quantifier-free admits a quantifier-free equivalent.

The first-order theory of Presburger arithmetic[1], $V_{\mathcal{L}}$ is just the set of integer constants. does **not** admit QE. For instance,

$$\exists\, t.\ x = 2t$$

has no quantifier-free equivalent. However,

Theorem

The Presburger theory extended with the predicates $\{D_n\,[\,x\,] = n \mid x\}_{n\in\mathbb{N}}$ **does** admit QE.

[1] *Presburger arithmetic* : theory of equality, inequality and addition over integers

A quantifier-free formula is said to be in standard form if :

- It only uses the operators $\wedge, \vee$ and $\neg$
- Each literal is of the form $0 < \Sigma_i\, a_i x_i \,+\, c$ or $0 = \Sigma_i\, a_i x_i \,+\, c$

Moreover, if ϕ is a quantifier-free formula in standard form and t a variable apppearing free in ϕ, then there exists a quantifier-free formula ϕ' such that :

- $\exists\, t \,.\, \phi\,[\,t\,] \iff \exists\, t \,.\, \phi'\,[\,t\,]$
- t appears only with the coefficients $\{-1, 1\}$ in ϕ'

$$
\begin{aligned}
& \exists\, t\,.\; x \geq 2t \;\wedge\; x \leq 3t \\
\Leftrightarrow\; & \exists\, t\,.\; 3x \geq 6t \;\wedge\; 2x \leq 6t \\
\Leftrightarrow\; & \exists\, t, t'\,.\; t' = 6t \;\wedge\; 3x \geq t' \;\wedge\; 2x \leq t' \\
\Leftrightarrow\; & \exists\, t'\,.\; 6 \mid t' \;\wedge\; 3x \geq t' \;\wedge\; 2x \leq t'
\end{aligned}
$$

Theorem (Cooper)

Let ϕ be a quantifier-free formula in standard form, such that x only appears with the coefficients $\{-1, 1\}$. Then

$$\exists x \,.\, \phi[x] \iff \bigvee_{j=1}^{m} \phi_{-\infty}[j] \;\vee\; \bigvee_{j=1}^{m} \bigvee_{b \in B} \phi'[b+j]$$

where

- m is the LCM of $\{\, k \;:\; k \mid t$ is a subformula of ϕ containing $x \,\}$
- $\phi_{-\infty}$ is derived from ϕ by replacing :
  - $0 = t$ by $\perp$ if $1 \cdot x$ belongs to t
  - $0 < t$ by $\perp$ if $1 \cdot x$ belongs to t and by $\top$ if $-1 \cdot x$ belongs to t
- B is the set of *boundary points*, a boundary point being associated to some literals of ϕ which are not divisibility predicates :
  - $0 = x + t$ is associated to the value of $-t - 1$
  - $\neg(0 = x + t)$ is associated to the value of $-t$
  - $0 < x + t$ is associated to the value of $-t$

For instance, we find with the Cooper formula that

$\exists\, t\,.\; x < t \,\wedge\, t < y$ is equivalent to $y - x > 1$

Another example :

$$\exists\, t \,.\, x \geq 2t \,\wedge\, x \leq 3t$$

is preprocessed into

$$\exists\, t \,.\, \phi[\, t\,] \quad \text{with} \quad \phi[\, t\,] = 6 \mid t \,\wedge\, 3x \geq t \,\wedge\, 2x \leq t$$

and becomes

$$\bigvee_{i=1}^{6} \phi[\, 2x + i\,]$$

which can be simplified into

$$\bigvee_{i=1}^{6} (\, 6 \mid 2x + i\,) \,\wedge\, (\, i \leq x\,)$$

and then

$$x \geq 2$$

Counterexample generalization

If B is a set of bad states and R a frame, we often check an entailment of the form :

$$F[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models \neg B[\mathbf{x}'] \qquad (*)$$

If it does not hold, we try to find some additional lemmas discarding all the counterexamples to $(*)$.

In fact, a single lemma $\neg G$ is sufficient if

$$G[\mathbf{x}] = \exists\, \mathbf{x}' .\, F[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \wedge B[\mathbf{x}']$$

Unfortunately, G is not quantifier-free.

Counterexample generalization

With approximate QE

Rather than trying to find a quantifier-free equivalent of

$$G[\mathbf{x}] = \exists\, \mathbf{x}' .\, F[\mathbf{x}] \wedge T\left[\mathbf{x}, \mathbf{x}'\right] \wedge B\left[\mathbf{x}'\right]$$

We search for a quantifier-free approximation of G, that is a cube[2] K s.t.

$$K[\mathbf{x}] \models G[\mathbf{x}]$$

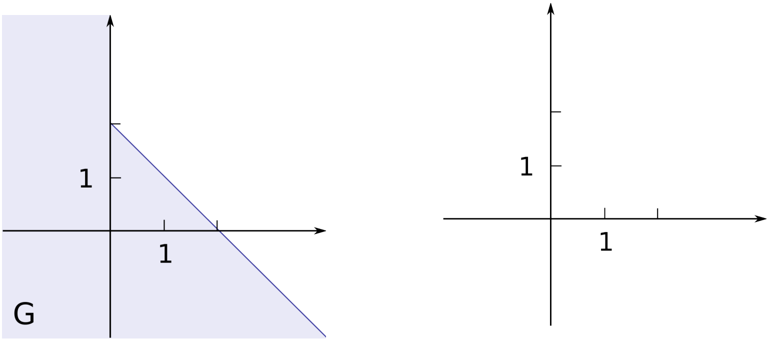
Moreover, we require that K contains at least the counterexample $(\mathbf{s}, \mathbf{t})$ previously given by the SMT solver.

For this, we eliminate one by one the existentially quantified variables of G, approximating the result of each step by a cube prefixed with existential quantifiers. An approximation is made each time a disjunction is encountered in the process. In this case, only a disjunct satisfied by $(\mathbf{s}, \mathbf{t})$ is kept.

[2] A *cube* is a conjunction of literals

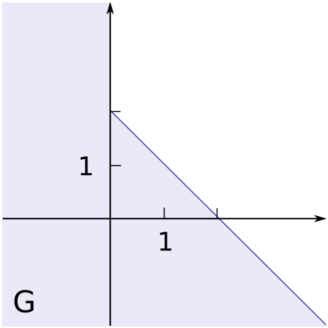
Counterexample generalization

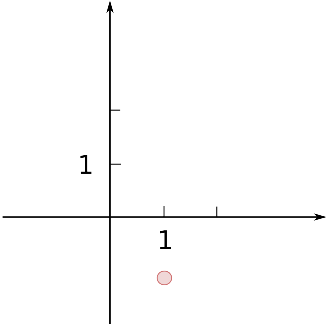
An example



Counterexample generalization

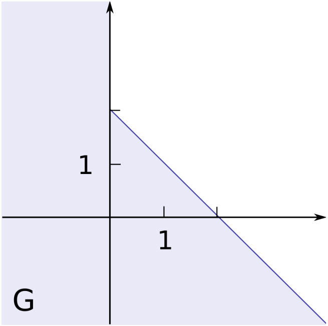
An example

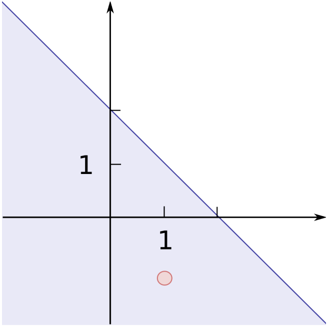




Counterexample generalization

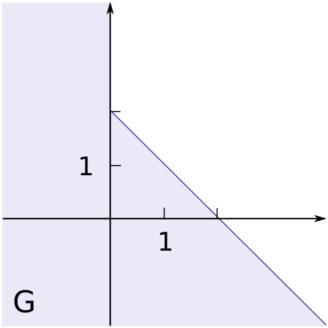
An example

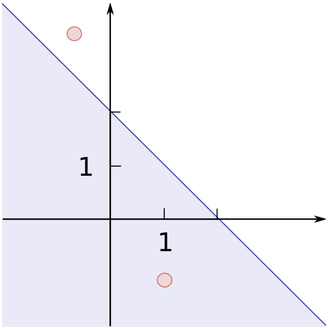




Counterexample generalization

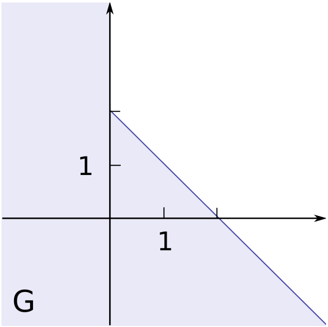
An example

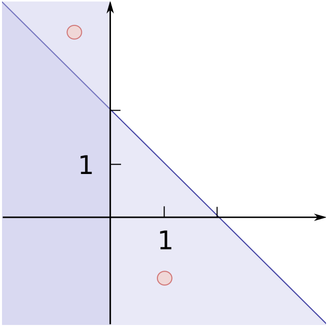




Counterexample generalization

An example





Lemma tightening

Let's assume we found a set of bad states B and we proved $\neg B$ doesn't intersect with I and is inductive relative to the previous frame, that is

$$I[\mathbf{x}] \models \neg B[\mathbf{x}] \qquad R_{k-1}[\mathbf{x}] \wedge \neg B[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models \neg B[\mathbf{x}']$$

Previously, we took $\neg B$ as a lemma. In fact, we can do better.

Indeed, if $B = \bigwedge_i B_i$ and

$$J_1 = \{ \ j : B_j[\mathbf{x}] \quad \text{belongs to the unsat. core of} \ \ I[\mathbf{x}] \wedge B[\mathbf{x}] \ \}$$

$$J_2 = \{ \ j : B_j[\mathbf{x}'] \quad \cdots \ R_{k-1}[\mathbf{x}] \wedge \neg B[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \wedge B[\mathbf{x}'] \ \}$$

Then we have the better invariant

$$\neg \left(\bigwedge_{j \in J_1} B_j[\mathbf{x}] \ \vee \ \bigwedge_{j \in J_2} B_j[\mathbf{x}] \right)$$

Lemma tightening

An example

Let's consider the following situation :

$$\begin{array}{rcl} I\,[x,y] & = & (x = y = 0) \\ T\,[x,y,x',y'] & = & (x' = x+1) \wedge (y' = y) \\ \\ B\,[x,y] & = & (x < 0) \wedge (y = 1) \\ R_{k-1}\,[x,y] & = & (y = 1) \end{array}$$

- The entailment $I\,[\mathbf{x}] \models \neg B\,[\mathbf{x}]$ holds iff the following is unsat. : $(x = y = 0)\,(x < 0)\,(y = 1)$
- Similarly concerning $R_{k-1}\,[\mathbf{x}] \wedge \neg B\,[\mathbf{x}] \wedge T\,[\mathbf{x},\mathbf{x}'] \models \neg B\,[\mathbf{x}']$ and : $(y = 1)\,(x \geq 0 \vee y \neq 1)\,(x' = x+1)\,(y' = y)\,(x' < 0)\,(y' = 1)$

Lemma tightening

An example

Let's consider the following situation :

$$\begin{array}{rcl} I\,[x, y\,] & = & (x = y = 0) \\ T\,[x, y, x', y'\,] & = & (x' = x + 1) \wedge (y' = y) \\ \\ B\,[x, y\,] & = & (x < 0) \wedge (y = 1) \\ R_{k-1}\,[x, y\,] & = & (y = 1) \end{array}$$

- The entailment $I\,[\,\mathbf{x}\,] \models \neg B\,[\,\mathbf{x}\,]$ holds iff the following is unsat. :
  $(x = y = 0)\,(x < 0)\,(y = 1)$
- Similarly concerning $R_{k-1}\,[\,\mathbf{x}\,] \wedge \neg B\,[\,\mathbf{x}\,] \wedge T\,[\,\mathbf{x}, \mathbf{x}'\,] \models \neg B\,[\,\mathbf{x}'\,]$ and :
  $(y = 1)\,(x \geq 0 \vee y \neq 1)\,(x' = x + 1)\,(y' = y)\,(x' < 0)\,(y' = 1)$

Therefore, we add the lemma $\neg\,(x < 0)$ instead of $\neg\,(x < 0 \wedge y = 1)$

Some limitations of the current version of *Kind2*

All of this is not sufficient to prove the example we gave at the beginning of the section, that is proving that $y \geq 0$ in

```
x = [0] ++ (1 + x)
y = [1] ++ (x + y)
```

In fact, the current version of the Kind2 model-checker fails at solving an even simpler problem :

```
x  = [1] ++ (1 + x)
ok = x /= 0
```

If you run the IC3 algorithm on this example,

$$R_k \Leftrightarrow x \notin [\![-k\,,\,\cdots\,,\,-1]\!]$$

We can see there is only one CTI discovered at each step and the much general lemma $x \geq 1$ is not infered.

Some ideas to enhance lemma tightening

A naive patch to solve the last case

The last case can be solved if we replace all literals of the form

$$a = b$$

in the set B of bad states by the equivalent conjunction

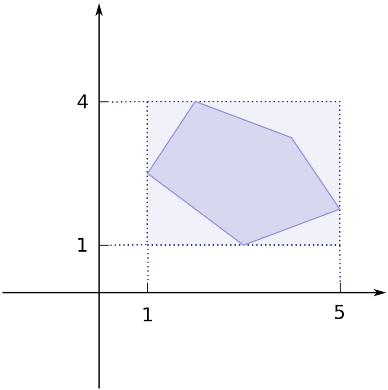
$$(\, a \leq b \,) \wedge (\, b \leq a \,)$$

before searching for an unsatisfiable core in the lemma tightening part.

The inverse transformation should be performed after the tightened lemma has been computed.

Of course, this idea is not general enough to solve many practical cases.

If B is a cube of bad states, we compute a best over-approximation of B as a set of literals containing one variable each. We then check if this generalization is still provable relative to the current frame.



Geometrically speaking,
in the case of linear arithmetic,
it is just finding the minimal
axis-aligned bounding box of a
convex polyhedra.

Thank you

Special thanks to Natasha, Andrew and Alwyn