Static Analysis and Verification

Predicate Abstraction

Tomáš Vojnar, Ondřej Lengál

SAV 2026, FIT BUT

Abstraction in Model Checking

Towards Abstraction (1/4)

- In traditional model checking, the states of the Kripke structure of a system being verified are given by all (reachable) combinations of assignments of values to system variables.
- For example, consider a system containing 3 variables x_1, x_2, x_3 whose domains are $D_{x_1} = D_{x_2} = \{0, 1, 2, 3\}$, and $D_{x_3} = \{0, 1\}$. Assume that all combinations of these values are reachable.
  - The KS of this system contains 32 states, which are generated by the Cartesian product of the domains of the system variables: $S = D_{x_1} \times D_{x_2} \times D_{x_3}$.
- In general, the size of the state space of a system containing variables $V = \{x_1, \ldots, x_n\}$ with (finite) domains $D_{x_1}, \ldots, D_{x_n}$ is given by the following formula:

$$|S| = \left| \bigotimes_{i=1}^{n} D_{x_i} \right| = \prod_{i=1}^{n} |D_{x_i}|$$

Towards Abstraction (2/4)

- Obviously, the size of the state space grows exponentially in the number of variables and in their bit-width.
  - E.g., a system containing 3 integer variables (32 bit) can have up to

$$79\,228\,162\,514\,264\,337\,593\,543\,950\,336 \cong 7.9 \cdot 10^{28} \text{ states!}$$

- Thus, it is not feasible to handle that many states explicitly even in the case when most of the state space is not reachable.
  - It is completely impossible if infinite (parametric) domains are allowed.
- Possible solutions include efficient storage of sets of states (e.g., using hierarchical storage or BDDs) or state space reductions (based on not exploring certain states since they and their successors are not relevant for the property to be checked, or their properties are covered by other explored states).
- Another possible solution is to use some kind of abstraction which collapses certain states of the original system that are similar from some point of view.

Towards Abstraction (3/4)

- For a given (possibly infinite) KS $M = (S, S_0, R, L)$, we replace working with the set of concrete states S by working with the set of abstract states $\hat{S}$ obtained using some abstraction function α whose domain includes S:
  - $\hat{S} = \{\hat{s} \mid \exists s \in S \colon \alpha(s) = \hat{s}\}$,
    - hence, each $\hat{s}$ corresponds to an equivalence class of the equivalence $\sim_\alpha$ induced on S by α such that $\forall s_1, s_2 \in S \colon s_1 \sim_\alpha s_2 \iff \alpha(s_1) = \alpha(s_2)$,
  - similarly, $\hat{S}_0 = \{\hat{s} \mid \exists s \in S_0 \colon \alpha(s) = \hat{s}\}$.
- An example: Consider a program with one variable x_1 over the domain $D_{x_1} = \{1, \ldots, 12\}$.
  - Consider the abstraction function: $\alpha = \lambda x.\ \lfloor (x-1)/3 \rfloor + 1$.
  - We obtain 4 abstract states, namely, $\hat{S} = \{\hat{1}, \hat{2}, \hat{3}, \hat{4}\}$, which correspond to the equivalence classes induced by α on the original state space:
    - $\hat{1} \sim \{1, 2, 3\}$,
    - $\hat{2} \sim \{4, 5, 6\}$,
    - $\hat{3} \sim \{7, 8, 9\}$,
    - $\hat{4} \sim \{10, 11, 12\}$.

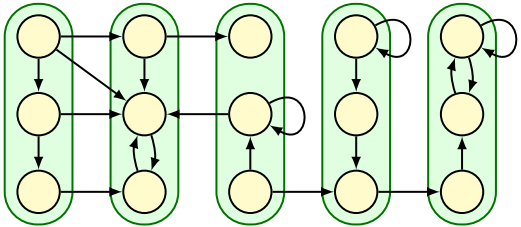
What about Transitions?

- We already know (at least conceptually) how to build the set of abstract states of a given program. Next, we have to add transitions to obtain an entire abstract KS.

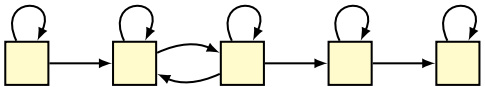
- There are two major ways of defining abstract transitions over the set of abstract states $\hat{S}$ obtained using some abstraction function α over a concrete KS $M = (S, S_0, R, L)$:
  1. Via the so-called existential abstraction (may abstraction):
     - $\hat{R}_{may} = \{(\hat{s}_1, \hat{s}_2) \mid \exists s_1, s_2 \in S \colon \alpha(s_1) = \hat{s}_1 \wedge \alpha(s_2) = \hat{s}_2 \wedge (s_1, s_2) \in R\}$,
     - i.e., whenever some state s_1 can go to some state s_2 in the original KS, then also $\alpha(s_1)$ can go to $\alpha(s_2)$ in the abstract KS,
     - the abstract KS allows more behaviour than the original one: it over-approximates the concrete KS: can prove ACTL* properties (CTL* with only universal path quantifiers and negations in literals).
  2. Via the so-called universal abstraction (must abstraction):
     - $\hat{R}_{must} = \{(\hat{s}_1, \hat{s}_2) \mid \forall s_1 \in S \colon \alpha(s_1) = \hat{s}_1 \Rightarrow \exists s_2 \in S \colon \alpha(s_2) = \hat{s}_2 \wedge (s_1, s_2) \in R\}$,
     - i.e., $\hat{s_1}$ can go to $\hat{s_2}$ in the abstract KS when each s_1 such that $\alpha(s_1) = \hat{s_1}$ can go to some s_2 in the original KS such that $\alpha(s_2) = \hat{s_2}$,
     - the abstract KS allows less behaviour than the original one: it under-approximates the concrete KS: can prove ECTL* properties (CTL* with only existential path quantifiers and negations in literals).

An Example

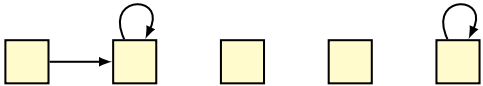
- Consider the following concrete KS where states equal wrt some abstraction function are encircled in green:



- The (over-approximating) may abstraction:



- The (under-approximating) must abstraction:



Towards Abstraction (4/4)

- However, a problem is how to obtain a suitable abstraction:
  - One possibility is to define it manually (requires insight, time consuming, error-prone).
  - A better way is when it is derived automatically or at least its parameters are derived automatically.
    - The principle of the abstraction may be fixed but in a parametric way — e.g., consider equal all states satisfying the same predicates where the set of predicates is the parameter: predicate abstraction.
    - If possible, the abstraction should be relevance-driven, i.e., it should preserve only those features of the system that are important for the verification question at hand.
    - Possible, e.g., within the above mentioned predicate abstraction.

Predicate Abstraction

Predicate Abstraction

- In 1997, S. Graf and H. Saïdi proposed in their paper "Construction of Abstract State Graphs with PVS" a new type of abstraction called predicate abstraction.
- This abstraction collapses all states of the original state space in which the valuation of a certain (finite) set of predicates is the same.
- More precisely, let $P = \{p_1, \ldots, p_n\}$ be an indexed set of predicates over program (system) variables (e.g., $x < y$). Then, the set of abstract states is the set

$$\{l_1 \wedge \cdots \wedge l_n \mid \forall 1 \leq i \leq n \colon l_i \in \{p_i, \neg p_i\}\}$$

  and the abstraction function α is defined as

$$\alpha = \lambda s. f(p_1, s) \wedge \cdots \wedge f(p_n, s)$$

  where
  - $f(p, s) = p$ if p holds in s (i.e., $s \models p$) and
  - $f(p, s) = \neg p$ otherwise.
- Basic predicate abstraction is suitable for verification of safety properties; for verification of termination (liveness properties), transition predicates have later been introduced by Podelski and Rybalchenko to avoid problems with artificial loops introduced by collapsing states — *we, however, limit ourselves to safety in what follows.*

An Example

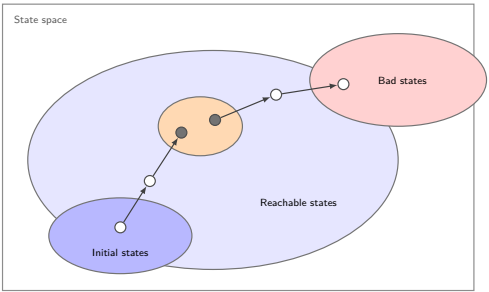
- Suppose that a program has two variables x, y with the domains $D_x = D_y = \{0, 1, 2\}$, and we use the set of predicates $P = \{p_1 \colon (x = y),\ p_2 \colon (x < y),\ p_3 \colon (y = 2)\}$.
- Now, the abstract state space may contain up to 8 $(= 2^3)$ states only:

$$
\begin{array}{lcl}
\neg p_1 \wedge \neg p_2 \wedge \neg p_3 & \sim & \{(1,0),(2,0),(2,1)\}, \\
\neg p_1 \wedge \neg p_2 \wedge p_3 & \sim & \emptyset, \\
\neg p_1 \wedge p_2 \wedge \neg p_3 & \sim & \{(0,1)\}, \\
\neg p_1 \wedge p_2 \wedge p_3 & \sim & \{(0,2),(1,2)\}, \\
p_1 \wedge \neg p_2 \wedge \neg p_3 & \sim & \{(0,0),(1,1)\}, \\
p_1 \wedge \neg p_2 \wedge p_3 & \sim & \{(2,2)\}, \\
p_1 \wedge p_2 \wedge \neg p_3 & \sim & \emptyset, \\
p_1 \wedge p_2 \wedge p_3 & \sim & \emptyset.
\end{array}
$$

- Note: When the set of predicates remains the same, the set of possibly reachable abstract states of the program stays the same even if x and y are 64 bit integer variables (or when the bit-width is a parameter yielding an infinite-state concrete system).

Predicate Abstraction — Continued

- Predicate abstraction is usually implemented as an over-approximating may abstraction. (The labelling function typically associates states with the predicates used by the predicate abstraction.)
- Hence, verification over abstract KS obtained by predicate abstraction is sound for ACTL* properties. This means that once the model checker answers "a system is correct", then the system is indeed correct wrt the specification (the abstract system contains all original behaviour). Formally, $\hat{M} \models \varphi \Rightarrow M \models \varphi$.
- On the other hand, the abstract KS can contain a path leading to an error state which is not present in the original system: a so-called spurious counterexample.

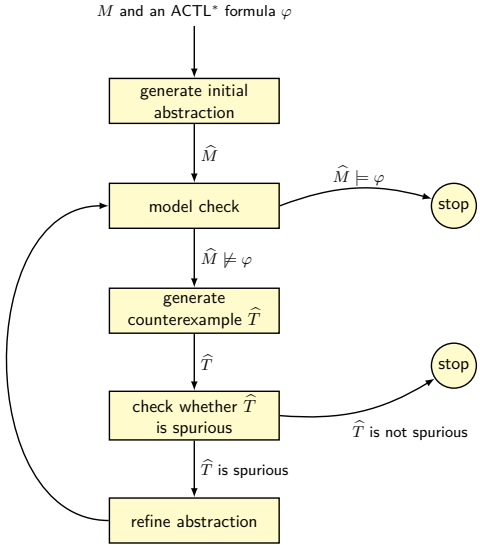


CEGAR

CEGAR

- To cope with spurious counterexamples, the approach of the so-called **C**ounter**e**xample-**G**uided **A**bstraction **R**efinement was first proposed by E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith in 2000.
- For a given path representing a possible error, one can try to decide whether it is spurious or not by executing the path in the original system.
  - A symbolic execution with states represented, e.g., by suitable logic formulae (but without abstraction) is needed if the original system is infinite-state (or large) and non-deterministic (e.g., due to reading some input).
- If the path is found executable in the original system, then the system does not satisfy the required property: a real error was found.
- If the path is not executable in the original system, then the abstract state space should be refined by using additional predicates obtained from the spurious counterexample to avoid the given error path, and the process should be repeated.
- Hopefully, at some point, the model checker answers either the "system is correct", or it gives a real counterexample.
- However, convergence is usually not guaranteed.

The CEGAR Loop



Generating the Abstraction

- Generating first the concrete KS and then abstracting it is of course not practical at all:
  - either an abstract system to be verified is first generated (e.g., abstracting a C program to a Boolean program with Boolean variables representing the predicates), or
  - abstract states are generated on-the-fly from abstract predecessors states w.r.t. concrete transitions to be executed.

Some More Technical Details

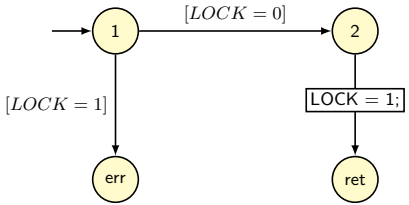
Control Flow Automata (CFA)

- To present predicate abstraction in more detail, we will consider verification of (sequential) programs encoded by the so-called control flow automata.
- A CFA is a directed graph where
  - nodes correspond to control locations in a given program,
    - there is a designated initial node (associated with a possible constraint on initial values of program variables),
  - edges represent transitions between control locations; each edge is labelled
    - by a basic block of instructions that are executed to move between the source and destination location (no branching in the middle, no jump into the middle) — depicted in boxes in what follows,
    - by an assume predicate corresponding to a branch condition that must be true for that edge to be taken — depicted in brackets in what follows.
- A basic block can contain a (non-recursive) call to a function which is described by its own CFA (with designated return nodes). The whole system is then described by a set of CFAs closed under function calls with a specified top-level CFA.
- The (safety) properties that we want to verify for a given program are encoded by reachability of designated error locations.

A Locking Example

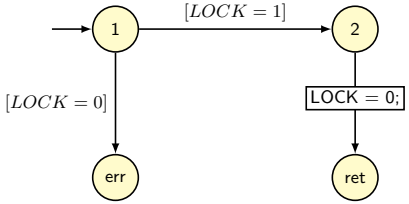
- `lock()` can be called when the *lock* is not held:

```
lock() {
1:    if (LOCK == 0) {
2:        LOCK = 1;
      } else {
err:      ERROR
      }
}
```



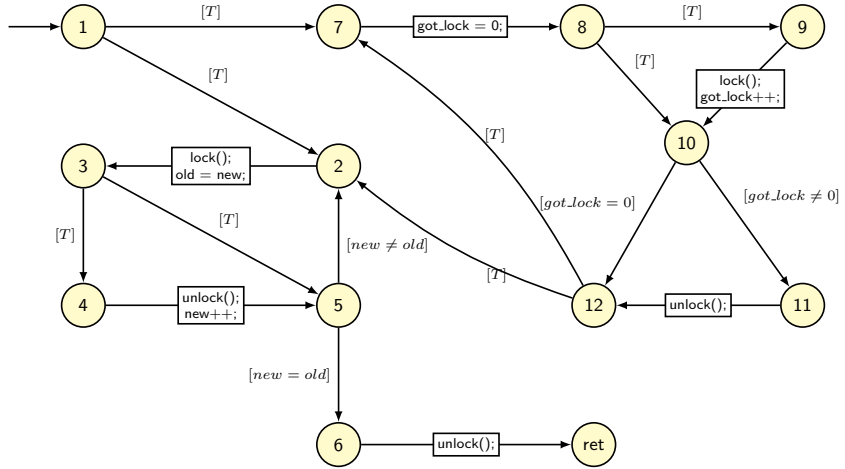
- `unlock()` can be called when the *lock* is held:

```
unlock() {
1:    if (LOCK == 1) {
2:        LOCK = 0;
      } else {
err:      ERROR
      }
}
```



A Locking Example — Continued

```
example() {
1:  if (*) {
7:    do {
        got_lock = 0;
8:      if (*) {
9:        lock();
          got_lock++;
        }
10:     if (got_lock) {
11:       unlock();
        }
12:   } while (*);
    }
2:  do {
      lock();
      old = new;
3:    if (*) {
4:      unlock();
        new++;
      }
5:  } while (new != old);
6:  unlock();
}
```

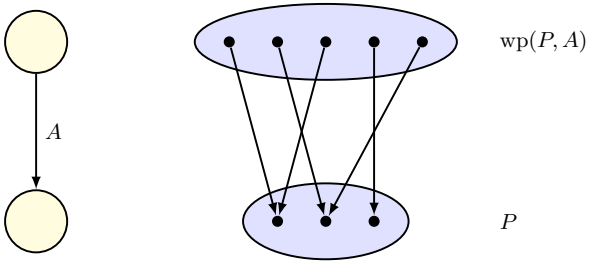


Computing Successors in Predicate Abstraction

- Model checking based on predicate abstraction can in fact be viewed as a form of symbolic model checking which works with sets of states represented by formulae built over some predicates and their complements.
- In standard model checking over a program written in some programming language, successors of a state being explored are generated by simply systematically executing the given program (while taking into account all possible context switches, etc.).
- In predicate abstraction, to compute the abstract successor Q of some abstract state P and an action A, we have to compute the so-called strongest postcondition over the given predicates, i.e., the strongest formula over the given predicates describing the set of states reachable from the set of states described by P after executing A.
- However, computing the strongest postcondition may be expensive. Instead, one can use a Cartesian approximation based on
  - checking for each predicate p and its negation $\neg p$ in isolation whether the weakest precondition of p or $\neg p$ to hold after executing A holds in P (i.e., before executing A),
  - followed by considering as the successor Q the conjunction of those predicates and their negations whose weakest preconditions hold in P (provided the action is executable).

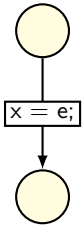
Weakest Precondition

- A formula P' is the weakest precondition of a formula P wrt an action A, denoted $\mathrm{wp}(P, A)$, iff P' is the weakest formula s.t.:
  - if P' holds before A is executed from its source location, then the execution of A terminates in the target location of A and P holds after that.



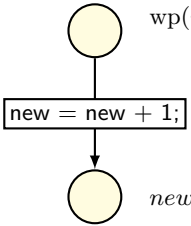
Computing Weakest Preconditions

- The weakest precondition of an assignment statement without side effects and pointer aliasing can be computed as follows:



$\mathrm{wp}(P, x \leftarrow e) = P[e/x]$

P



$\mathrm{wp}(new = old, new \leftarrow new + 1) =$
$new + 1 = old$

$new = old$

WP and Pointer Aliasing

- Unfortunately, the variable assignments are not always that easy. For example, one also has to take care of the situations containing pointers, which can but need not be aliased.
- For example, let $x \leftarrow 0$ be a statement. Then, the weakest precondition of $*p = 1$ is:

$$\mathrm{wp}(*p = 1, x \leftarrow 0) = (p = \&x \wedge 0 = 1) \vee (p \neq \&x \wedge *p = 1)$$

- In general, for k pointers, the weakest precondition would contain $O(2^k)$ disjuncts, each considering a possible alias scenario of the k addresses with x.
- To improve the efficiency, it is possible to use some conservative pointer analysis (Steensgaard, Andersen, ...). If it says that x and y cannot be aliased, then all disjuncts with x and y aliased can be taken away from the formula.
- Other complications include dealing with side-effects, dynamic allocation, dynamic linked data structures, arrays, type-casting, ...
  - *Beyond the scope of the lecture*, often subject to further research, may be handled outside of the basic predicate abstraction framework.
  - It is possible to try to *abstract away* all such data manipulation (replacing program conditions based on them by a non-deterministic choice).

Computing the Successors — Continued

- As we already mentioned, computing the abstract successors precisely may be expensive. Therefore the abstract successors are sometimes overapproximated using the weakest precondition computation separately for each predicate and its negation.
- Given a set of predicates $\{p_1, p_2, \ldots, p_n\}$, an abstract source state P and a program action A (other than an assumption), we compute the abstract successor Q using a procedure denoted $AbsSucc$ as follows. First, we set $Q = true$ and then we iterate over the set of predicates and for every p_i modify Q in the following way:

$$Q \leftarrow \begin{cases} Q \wedge p_i & \text{if } P \Rightarrow \mathrm{wp}(p_i, A) \\ Q \wedge \neg p_i & \text{if } P \Rightarrow \mathrm{wp}(\neg p_i, A) \\ Q & \text{otherwise} \end{cases}$$

- To check the implications on the right-hand side, we use some automatic theorem prover.
- In the third case, we may actually overapproximate the strongest postcondition over the given predicates (due to we use a single conjunction only).
  - We also generalise the abstract states a bit by allowing some predicate to be completely missing.
- Assume statements will be dealt with in a special way.

Forward Search with Predicate Abstraction

- In the first phase of model checking based on predicate abstraction and CEGAR, given a program described by a CFA M (with correctness criteria built in in the form of error locations) and an initial set of predicates P (may be empty), an abstract reachability tree (ART) T is (partially) built — the construction is stopped when an error is reached.
- An abstract reachability tree (used instead of KS):
  - a computation tree obtained by unwinding a CFA from its start state, with nodes labelled by program locations and reachable regions, and edges labelled by basic blocks or assumptions according to the corresponding edges in the CFA.
- A reachable region (RR) R
  - is given by a conjunction of predicates from P and their negations,
  - it represents a set of concrete states of the original program that satisfy R.

Forward Search: Initialization

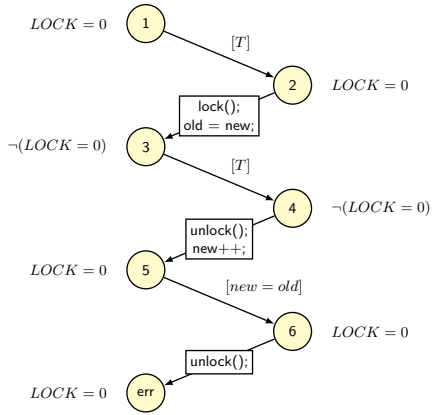
- Construct the root of the ART as a node labelled by the initial location of M and a region corresponding to possible constraints on the initial state of M,
  - the initial region can be constructed by checking implications between the initial constraints on M and the predicates from P and their negations;
  - if there is no initial constraint (and the program initialisation is a part of the program), start with $true$.

Forward Search

- Process nodes of T in the depth-first order as follows — let the node n being processed be labelled with an RR R:
  - Skip n if it has already been covered by another ART node (i.e., if its RR R implies the RR of some other node with the same program location).
  - Examine edges between the CFA node corresponding to n in M and its successors:
    - For an edge labelled by an assume predicate AP: check using a decision procedure (automatic theorem prover) whether $R \wedge AP$ is satisfiable; if not, do nothing. Otherwise, create an $[AP]$-labelled edge to a child node labelled by a conjunction of the predicates or their negations that are implied by $R \wedge AP$.
    - For a basic block B, create a B-labelled edge to a successor node labelled with a region R' obtained by iterating the $AbsSucc$ procedure over B.
- Terminate when all nodes in T are covered (i.e., the program is correct, no error location is reached), or some error location is reached.

Forward Search: An Example

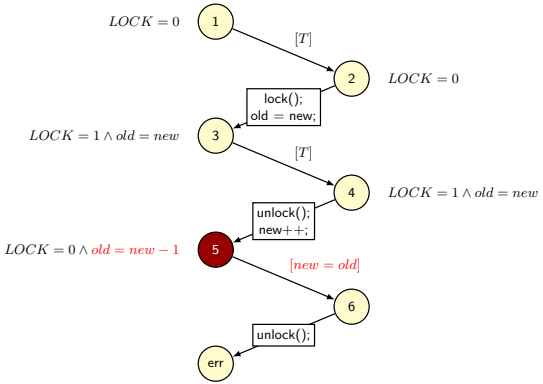
- Consider the locking example with predicates $P = \{LOCK = 0\}$ and an initial assumption $LOCK = 0$.



Forward Counterexample Analysis

- When an error location is hit within the forward search, the suspected error path has to be analysed — one way how to do it is a concrete (symbolic) forward execution of the given sequence of statements.
  - More precisely, given a (partial) ART T and a suspected counterexample path in T leading to some error location, we check in a forward way whether the path is real or spurious (i.e., whether the corresponding sequence of transitions can be executed in the concrete system or not).
- A sketch of the algorithm:
  - Start at the root of T in the initial (concrete) state of the system given by the initial constraint of the CFA.
  - Traverse the sequence of edges that appear in the suspected error path:
    - For an edge labelled with an assume predicate AP, check whether $S \wedge AP$ is satisfiable for S being the constraint describing the concrete states reached so far; if not, then the counterexample is spurious, otherwise conjoin P to S and continue from there on.
    - For an edge labelled with a basic block B, perform a concrete symbolic execution of B from the current symbolic state using the strongest postcondition calculus — $\mathrm{sp}(S, x \leftarrow E) = \exists x' \colon S[x'/x] \wedge x = E[x'/x]$ for E with no side-effects, ...

Forward Counterexample Analysis: An Example



- Refinement when a spurious counterexample is detected: take as new predicates atomic formulae appearing in the failed concrete execution of the spurious counterexample.

Lazy Abstraction

Lazy Abstraction

- Lazy abstraction was introduced by T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre in 2002.
- Uses different sets of predicates (and hence different degrees of precision) in different parts of the ART:
  - abstraction refinement is done for a part of the system only (for which it is too coarse and produces spurious counterexamples),
  - avoids repetition of work between iterations of the CEGAR loop (parts of the system which have already been proven safe need not be re-checked using a refined abstraction).
- A backward counterexample analysis is used.

Backward Counterexample Analysis

- We will use the so-called bad regions (BR) as an additional labelling of nodes in a (partial) ART T which appear on the path of the suspected counterexample.
  - A BR labelling an ART node n that is introduced wrt some suffix π of the suspected error path is a formula representing the set of states that can execute the actions labelling the edges of π starting from the control location of n in the CFA and leading to an error.
  - In figures, we will draw BRs in curly braces.
- A pivot node is a node where the intersection of the reachable region and the bad region becomes empty (i.e., the conjunction of the corresponding formulas becomes unsatisfiable) for the first time when going backwards through the suspected error path.

Backward Counterexample Analysis — Continued

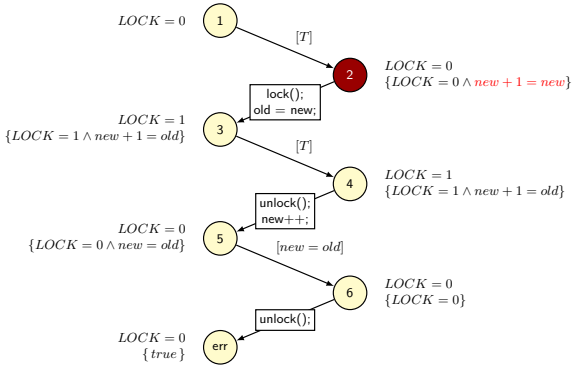
- The backward counterexample analysis starts with the error node at the end of a given suspected error path and labels it with the BR $true$.
- Edges of the suspected error path are examined in the reverse order as follows — assume that the backward computation has currently reached a BR B:
  - In case the next edge to be backtracked through is labelled with an assume predicate AP, let the new bad region be $B \wedge AP$.
  - In case the next edge to be backtracked through is labelled with a basic block BB, let the new BR be $\mathrm{wp}(B, BB)$.
- At each step of the backward computation which reaches a BR B at an ART node labelled with an RR R, satisfiability of $B \wedge R$ is checked. If the satisfiability check fails, a pivot has been found.
  - At the initial node, one needs to check satisfiability of the conjunction of B and the concrete initial state too.

Backward Counterexample Analysis — Continued

- If a pivot node is found, the counterexample is spurious.
- To introduce new predicates and thus get rid of the spurious counterexample in the next round of the CEGAR loop, one can use a theorem prover which provides a proof of unsatisfiability and use the predicates which appear in the proof.
- In order to maintain the syntactic form of predicates in which they originate from assignments, tests, etc. (instead of having all this composed into a complex predicate whose validity would not be that useful at different lines of the code), it is advantageous to compute the weakest preconditions using explicit substitutions:
  - $\mathrm{wp}(\varphi, x \leftarrow E) = \exists x' \colon x' = E \wedge \varphi[x'/x]$.

Backward Counterexample Analysis: An Example

- When using the classical weakest precondition computation in the backward analysis of the already mentioned suspected counterexample in the locking example, we get:



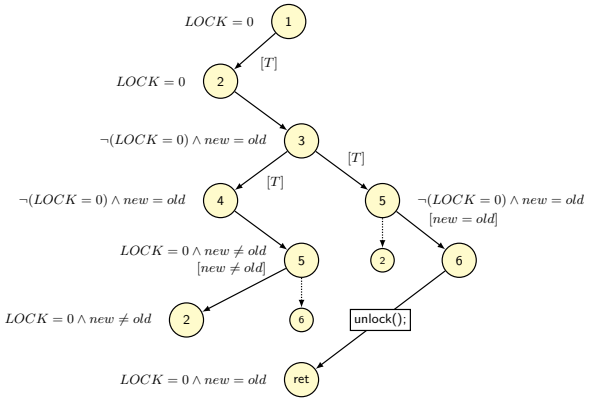
Backward Counterexample Analysis: An Example

- When using the explicit substitution, we get the weakest precondition:

$$(\exists LOCK'' \colon LOCK'' = 1 \wedge (\exists old' \colon old' = new \wedge (\exists LOCK' \colon LOCK' = 0 \wedge \\ (\exists new' \colon new' = new + 1 \wedge LOCK' = 0 \wedge new' = old')) \wedge LOCK'' = 1)) \wedge LOCK = 0$$

- From this, we learn that we should use the new predicate $old = new$.

The Locking Example: A Refined Forward Search



- One more refinement is then needed in the other possible continuation of node 1 (adding a predicate $got_lock = 0$).

Craig Interpolation

Path Formulae

- Given a suspected error path, the corresponding path formula (PF) is constructed by transforming the path into the static single-assignment (SSA) form in the following way:
  - For each variable x and each statement i of the path, we create a new variable x_i.
  - For each statement i of the path, we build a statement formula which is defined as follows:
    - if the statement is an assignment $x = E$, then the formula is

$$\left(\bigwedge_{y \in Var \setminus \{x\}} y_i = y_{i-1} \right) \wedge x_i = E[x_{i-1}/x, y_{i-1}/y, \ldots]$$

      ,
    - if the statement is an assumption $[C]$, the formula is

$$\left(\bigwedge_{y \in Var} y_i = y_{i-1} \right) \wedge C[x_{i-1}/x, y_{i-1}/y, \ldots]$$

      .
  - The path formula is the conjunction of all statement formulae.
  - In practice, the number of variables may be reduced by introducing a fresh variable only when the value of some variable changes.
- Clearly, a PF is satisfiable iff the corresponding path is feasible in the concrete program.

An Example of a Path Formula

- A simplified PF for the infeasible error path in the locking example:

pc	path	SSA form	path formula
`init`		$LOCK_0 = 0$	$LOCK_0 = 0 \wedge$
`1`	`assume(true)`	$assume(true)$	$true \wedge$
`2-L1`	`assume(LOCK==0)`	$assume(LOCK_0 == 0)$	$LOCK_0 = 0 \wedge$
`2-L2`	`LOCK = 1;`	$LOCK_1 = 1;$	$LOCK_1 = 1 \wedge$
`2`	`old = new;`	$old_1 = new_0;$	$old_1 = new_0 \wedge$
`3`	`assume(true)`	$assume(true)$	$true \wedge$
`4-U1`	`assume(LOCK==1)`	$assume(LOCK_1 == 1)$	$LOCK_1 = 1 \wedge$
`4-U2`	`LOCK = 0;`	$LOCK_2 = 0;$	$LOCK_2 = 0 \wedge$
`4`	`new++;`	$new_1 = new_0 + 1;$	$new_1 = new_0 + 1 \wedge$
`5`	`assume(old==new)`	$assume(old_1 == new_1)$	$old_1 = new_1 \wedge$
`6-U1`	`assume(LOCK==0)`	$assume(LOCK_2 == 0)$	$LOCK_2 = 0$
`6-err`	`ERROR`	$ERROR$	

New Predicates via Craig Interpolation

- Over an unsatisfiable path formula, one can use the so-called Craig interpolation to find out which predicates should be tracked at which location to get rid of the path:
  - these predicates summarise and abstract the information present in the path in a way sufficient to show infeasibility at any chosen location,
  - this decreases the number of predicates to be tracked and allows their localisation.
- A Craig interpolant ψ is a formula that, for a given inconsistent pair of formulae φ^-, φ^+ (i.e., such that $\varphi^- \wedge \varphi^+$ is not satisfiable), satisfies:
  1. $\varphi^- \Rightarrow \psi$,
  2. $\neg(\psi \wedge \varphi^+)$,
  3. ψ contains only those non-logical symbols (variables, uninterpreted function and predicate symbols) that are common to both φ^- and φ^+.
- Given an infeasible PF ϕ,
  - we can cut ϕ at each location and create φ^- (containing the fragment of ϕ up to the given location) and φ^+ containing the rest,
  - for each pair φ^-, φ^+, we may compute an interpolant ψ that likely contains good predicates for refining the abstraction at the given location.

Computing Craig Interpolants

- Algorithms for computing Craig interpolants exist for various logical fragments, enabling a use of Craig interpolation if the given PF is in the appropriate fragment.
- Interpolation is implemented within (or on top of) various SMT solvers (e.g., MathSat, interpolating Z3, SMTinterpol, ...) as well as various automatic theorem provers (Princess, Vampire).
- Like with SMT/automated theorem proving, generation of interpolants is still being improved.