15-414: Bug Catching: Automated Program Verification

Lecture Notes on Software Model Checking

Matt Fredrikson André Platzer

Carnegie Mellon University
Lecture 19

1 Introduction

So far we've focused on model checking algorithms that assume a computation structure is given. It should come as no surprise that our goal is to perform model checking of programs given as code, so today we'll describe techniques that allow us to apply model checking in this setting. There are several challenges to doing so, foremost among them the fact that the statespace of programs may be infinite. We'll describe two approaches for dealing with this: *bounded model checking* and *predicate abstraction*.

Each of these techniques addresses the problem by computing an approximation. Bounded model checking computes an *under*approximation of the reachable statespace by assuming a fixed computation depth in advance, and treating paths within this depth limit symbolically to explore all possible states. Predicate abstraction computes an *over*approximation of reachable states by constructing a transition structure that treats distinct program states identically, in a way that makes it possible to reason over a finite number of states. While either approach has its limitations, both are used effectively in practice, and are the core techniques that make software model checking possible.

2 Review: Transition structures, LTL

Definition 1 (Trace semantics of programs)**.** The *trace semantics,* $\tau(\alpha)$*, of a program* α, is the set of all its possible traces and is defined inductively as follows:

1. $\tau(x := e) = \{(\omega, \nu) \ : \ \nu = \omega$ except that $\nu(x) = \omega[\![e]\!]$ for $\omega \in \mathcal{S}\}$
2. $\tau(?Q) = \{(\omega) \ : \ \omega \models Q\} \cup \{(\omega, \Lambda) \ : \ \omega \not\models Q\}$

3. $\tau(\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta) = \{\sigma \in \tau(\alpha) \,:\, \sigma_0 \models Q\} \cup \{\sigma \in \tau(\beta) \,:\, \sigma_0 \not\models Q\}$

4. $\tau(\alpha;\beta) = \{\sigma \circ \varsigma \,:\, \sigma \in \tau(\alpha)\,,\, \varsigma \in \tau(\beta)\}$;
the composition of $\sigma = (\sigma_0, \sigma_1, \sigma_2, \dots)$ and $\varsigma = (\varsigma_0, \varsigma_1, \varsigma_2, \dots)$ is
$$\sigma \circ \varsigma := \begin{cases} (\sigma_0, \dots, \sigma_n, \varsigma_1, \varsigma_2, \dots) & \text{if } \sigma \text{ terminates in } \sigma_n \text{ and } \sigma_n = \varsigma_0 \\ \sigma & \text{if } \sigma \text{ does not terminate} \\ \text{not defined} & \text{otherwise} \end{cases}$$

5. $\tau(\texttt{while}(Q)\,\alpha) = \{\sigma^{(0)} \circ \sigma^{(1)} \circ \dots \circ \sigma^{(n)} \,:\,$ for some $n \geq 0$ such that for all $0 \leq i < n$:
① the loop condition is true $\sigma_0^{(i)} \models Q$ and ② $\sigma^{(i)} \in [\![\alpha]\!]$ and ③ $\sigma^{(n)}$ either does not terminate or it terminates in $\sigma_m^{(n)}$ and $\sigma_m^{(n)} \not\models Q$ in the end$\}$
$\cup\, \{\sigma^{(0)} \circ \sigma^{(1)} \circ \sigma^{(2)} \circ \dots \,:\,$ for all $i \in \mathbb{N}$: ① $\sigma_0^{(i)} \models Q$ and ② $\sigma^{(i)} \in [\![\alpha]\!]\}$
$\cup\, \{(\omega) \,:\, \omega \not\models Q\}$
That is, the loop either runs a nonzero finite number of times with the last iteration either terminating or running forever, or the loop itself repeats infinitely often and never stops, or the loop does not even run a single time.

6. $\tau(\alpha^*) = \bigcup_{n\in\mathbb{N}} \tau(\alpha^n)$ where $\alpha^{n+1} \stackrel{\text{def}}{\equiv} (\alpha^n;\alpha)$ for $n \geq 1$, and $\alpha^1 \stackrel{\text{def}}{\equiv} \alpha$ and $\alpha^0 \stackrel{\text{def}}{\equiv} (?\mathit{true})$.

Definition 2 (Kripke structure)**.** A *Kripke frame* $(W, I, \rightsquigarrow)$ consists of a set W with a transition relation $\rightsquigarrow \subseteq W \times W$ where $s \rightsquigarrow t$ indicates that there is a direct transition from s to t in the Kripke frame $(W, \rightsquigarrow)$. The elements $s \in W$ are also called states, and $I \subseteq W$ are the set of initial states. A *Kripke structure* $K = (W, I, \rightsquigarrow, v)$ is a Kripke frame $(W, \rightsquigarrow)$ with a mapping $v : W \to \Sigma \to \{\mathit{true}, \mathit{false}\}$ assigning truth-values to all the propositional atoms in all states.

The program semantics $[\![\alpha]\!]$ which was defined as a relation of initial and final states in Lecture 3 is an example of a Kripke structure.

Definition 3 (Computation structure)**.** A Kripke structure $K = (W, I, \rightsquigarrow, v)$ is called a *computation structure* if W is a finite set of states, $I \subseteq W$ is a set of initial states, and every element $s \in W$ has at least one direct successor $t \in W$ with $s \rightsquigarrow t$. A (computation) *path* in a computation structure is an infinite sequence $s_0, s_1, s_2, s_3, \dots$ of states $s_i \in W$ such that $s_i \rightsquigarrow s_{i+1}$ for all i.

A computation structure is a special case of a Kripke structure. Although we could simply refer to both types of structures as Kripke structures, we will often refer to the general class of all such structures as "transition structures" when we don't care to distinguish between the two.

3 Bounded Model Checking

The first approach that we consider computes an *underapproximation* of $\tau(\alpha)$: not all possible traces will appear in the approximation, but all those that do appear are certain to be in the true trace semantics. In principle Bounded Model Checking (BMC)

can be used to verify arbitrary temporal properties, but it is most commonly used to check invariants of the form $\Box \mathit{terminated} \rightarrow P$, and we will focus on this case for the remainder.

The way in which BMC approximates $\tau(\alpha)$ is by assuming that all loops in the program are unrolled to some fixed, pre-determined finite depth k. There are two useful ways to think about this operation. The first, which might have occurred to you naturally before having taken this course, is to transform the original program, which may contain loops, into a loop-free program using the bound k. Recall from a much earlier lecture the [unwind] axiom, which allows us to replace a loop with a conditional statement, within which is a copy of the original loop.

([unwind]) $[\texttt{while}(Q)\,\alpha]P \leftrightarrow [\texttt{if}(Q)\,\{\alpha; \texttt{while}(Q)\,\alpha\}]P$

Axiom [unwind] tells us that it is perfectly acceptable when reasoning about a safety property to replace `while` statements with `if` statements in this way. To perform bounded model checking, we first apply [unwind] to each loop in the program up to k times. When we are finished, we replace any remaining loops with `skip` statements (or equivalently, $?Q$).

Let's see an example. Consider the following program, which doesn't do anything useful but is simple enough to illustrate the key ideas here.

```
i := N;
while(0 ≤ x < N) {
  i := i - 1;
  x := x + 1;
}
```

Suppose that we want to check that $\Box \mathit{terminated} \rightarrow 0 \leq i$ holds, up to a bound of $k = 2$. We begin by applying [unwind] twice to the loop. When we stop, we replace the remaining loop with an empty statement.

```
i := N;
if(0 ≤ x < N) {
  i := i - 1;
  x := x + 1;
  if(0 ≤ x < N) {
    i := i - 1;
    x := x + 1;
  }
}
```

With all of the loops removed from the program, verification is straightforward using the deductive techniques covered earlier in the semester: the formula we need to prove is just $[\alpha]0 \leq i$. In particular, we can apply [if], [;], and [:=] repeatedly until we are left with a term containing no modalities and literals involving only integer operations. In

the current example, we have the following after applying the necessary steps.

$$\begin{array}{ll} & (\neg(0 \leq x < N) \rightarrow 0 \leq N) \\ \wedge & (0 \leq x < N \rightarrow \neg(0 \leq x + 1 < N) \rightarrow 0 \leq N - 1) \\ \wedge & (0 \leq x < N \rightarrow 0 \leq x + 1 < N \rightarrow 0 \leq N - 2) \end{array}$$

If this formula is valid (which it is not), then the original property holds. In practice, these formulas are converted into satisfiability problems and given to a decision procedure.

Notice that there are three clauses in this formula, one for each possible path through the program after unwinding at $k = 2$. What bounded model checking essentially does is to "symbolically" evaluate each path through the program up to the unwinding depth. Each path corresponds to a conjunctive clause, so that if the formula is not valid, there will be a clause that the model checker can identify as being at fault. The corresponding path gives a counterexample, and a satisfying solution to its negation a valuation of the input variables that will violate the property.

In the example above, we see that the first clause is already invalid. We negate it to look for a satisfying solution:

$$\neg(\neg(0 \leq x < N) \rightarrow 0 \leq N) \leftrightarrow (\neg(0 \leq x < N) \wedge \neg(0 \leq N))$$

A satisfying solution to the above is $x = 0, N = -1$. Notice that if we run the original program starting in a state that matches this assignment, then it terminates immediately without executing the loop, leaving $i = -1$.

Limitations Because bounded model checking is an underapproximation, it might not consider some traces that are in the trace semantics of the program. This means that if it does not find a property violation, we cannot necessarily conclude that the program is bug-free. However, in some cases we can. Consider the following variation of the above example.

```
i := 3;
while(0 ≤ x < 3) {
  i := i - 1;
  x := x + 1;
}
```

While a bound of $k = 2$ is insufficient to conclude that there are no bugs in this program, setting $k = 3$ is in fact sufficient. Furthermore, we can modify the unwinding process slightly so that if no bugs are found up to a particular depth, *and* we've chosen a sufficiently large enough k, we will conclude as much. Likewise, if no bugs are found but we chose an inadequately large k, we'll know that to be the case as well.

The approach uses what are called *unwinding assertions*. Whereas before when we finished applying [unwind], we replaced the remaining loop with an empty statement, now we will replace it with a statement that violates safety if the unwinding is insufficient. In the above example, we would have the following for $k = 2$.

```
i := 3;
if(0 ≤ x < 3) {
  i := i - 1;
  x := x + 1;
  if(0 ≤ x < 3) {
    i := i - 1;
    x := x + 1;
    assert(¬(0 ≤ x < 3));
  }
}
```

Although we haven't talked about assertions before, we can model them using existing constructs and safety properties. To check that an assertion isn't violated, we replace the `assert` statement with a corresponding conditional, which makes an assignment to a special variable whenever its condition is true.

```
error := 0;
i := 3;
if(0 ≤ x < 3) {
  i := i - 1;
  x := x + 1;
  if(0 ≤ x < 3) {
    i := i - 1;
    x := x + 1;
    if(0 ≤ x < 3) error := 1;
  }
}
```

We can then check the validity of the formula $[\alpha]\texttt{error} = 0$. In this case, the formula would be invalid, because x is at most 2 on the path containing the `assert`. This means that the unwinding assertion fails to hold, and so we should not conclude that the program is bug-free by unwinding up to $k = 2$.

4 Transition Structures for Programs

Moving on, we'll now look at a different technique that builds an abstraction of the program's reachable states. This abstraction will be in the form of a transition structure. Until now, we've been rather informal about the fact that the programs we've discussed all semester can be modeled as transition structures. Now let's get serious about it and write the definition.

Definition 4 (Transition Structure of a Program)**.** Given a program α over program states $\mathcal{S}$, let L be a set of *locations* given by the inductively-defined function $locs(\alpha)$, $\iota(\alpha)$ be the *initial* locations of α, and $\kappa(\alpha)$ be the *final* locations of α:

- $locs(x := e) = \{\ell_i, \ell_f\}$,
  $\iota(x := e) = \{\ell_i\}$,
  $\kappa(x := e) = \{\ell_f\}$

- $locs(?Q) = \{\ell_i, \ell_f\}$,
  $\iota(?Q) = \{\ell_i\}$,
  $\kappa(?Q) = \{\ell_f\}$
- $locs(\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta) = \{\ell_i\} \cup \{\ell_t : \forall \ell \in locs(\alpha)\} \cup \{\ell_f : \forall \ell \in locs(\beta)\}$,
  $\iota(\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta) = \{\ell_i\}$,
  $\kappa(\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta) = \kappa(\alpha) \cup \kappa(\beta)$
- $locs(\alpha;\beta) = \{\ell_0 : \forall \ell \in locs(\alpha)\} \cup \{\ell_1 : \forall \ell \in locs(\beta)\}$,
  $\iota(\alpha;\beta) = \iota(\alpha)$,
  $\kappa(\alpha;\beta) = \kappa(\beta)$
- $locs(\texttt{while}(Q)\,\alpha) = \{\ell_i, \ell_f\} \cup \{\ell_t : \forall \ell \in locs(\alpha)\}$,
  $\iota(\texttt{while}(Q)\,\alpha) = \{\ell_i\}$,
  $\kappa(\texttt{while}(Q)\,\alpha) = \{\ell_f\}$

As a convenient shorthand, given a location ℓ we will write α_ℓ to denote the statement associated with that location. The control flow transition relation $\epsilon(\alpha) \subseteq locs(\alpha) \times progs \times locs(\alpha)$ is given by:

- $\epsilon(x := e) = \{(\ell_i, x := e, \ell_f) : \ell_i \in \iota(x := e), \ell_f \in \kappa(x := e)\}$
- $\epsilon(?Q) = \{(\ell_i, ?Q, \ell_f) : \ell_i \in \iota(?Q), \ell_f \in \kappa(?Q)\}$
- $\epsilon(\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta) = \{(\ell_i, ?Q, \ell_{ti}) : \ell_i \in \iota(\cdot), \ell_{ti} \in \iota(\alpha)\} \cup \{(\ell_i, ?\neg Q, \ell_{fi}) : \ell_i \in \iota(\cdot), \ell_{fi} \in \iota(\beta)\} \cup \epsilon(\alpha) \cup \epsilon(\beta)$, where $\iota(\cdot) = \iota(\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta)$.
  In other words, transitions go from the initial location ℓ_i to the initial locations of α and β.
- $\epsilon(\texttt{while}(Q)\,\alpha) = \{(\ell_i, ?\neg Q, \ell_f) : \ell_i \in \iota(\cdot), \ell_f \in \kappa(\cdot)\} \cup \{(\ell_i, ?Q, \ell_{ti}) : \ell_i \in \iota(\cdot), \ell_{ti} \in \iota(\alpha)\} \cup \{(\ell_f, ?\top, \ell_i) : \ell_i \in \iota(\cdot), \ell_f \in \kappa(\alpha)\} \cup \epsilon(\alpha)$.
  In other words, transitions go from the initial location ℓ_i to the initial location of α, as well as from the initial location ℓ_i to the final location ℓ_f and the final location of the loop body to the initial location of the loop.
- $\epsilon(\alpha;\beta) = \epsilon(\alpha) \cup \epsilon(\beta) \cup \{(\ell_f, ?\top, \ell_i) : \ell_i \in \iota(\beta), \ell_f \in \kappa(\alpha)\}$

Notice that control flow transitions are associated with statements. Intuitively, the locations at the source of a transition correspond to the state immeidately prior to exeucting a statement, and those at the destination the state immediately after. Then the transition structure $K_\alpha = (W, I, \curvearrowright, v)$ itself is given by:

- $W = locs(\alpha) \times \{\mathcal{S}\}$, $I = \{\langle \ell_i, \sigma\rangle : \ell_i \in \iota(\alpha)\}$.
- $\curvearrowright = \{(\langle \ell, \sigma\rangle, \langle \ell', \sigma'\rangle) :$ for $(\ell, \beta, \ell') \in \epsilon(\alpha)$ where $(\sigma, \sigma') \in [\![\beta]\!]\}$.
  In other words, a transition in K_α is possible whenever there is a corresponding edge in $(\ell, \beta, \ell') \in \epsilon(\alpha)$, and the program state components σ, σ' in the pre- and post-states of the transition are in the semantics of β.

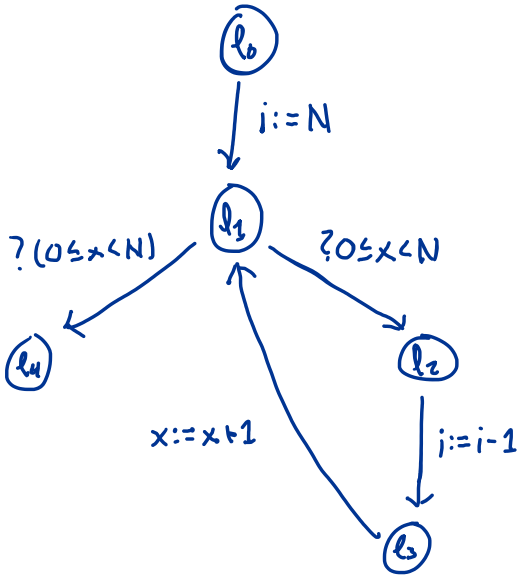
- $v(\langle \ell, \sigma \rangle) = \ell \wedge \bigwedge_{v \in \mathbf{vars}} v = \sigma(v)$. In other words, states are labeled with formulas that describe their location and valuation. We assume that program locations correspond to literals in such formulas.

Definition 4 is consistent with Def. 1, in that if we start at an initial state and transcribe the program state component in the label of each state entered moving along a possible transition, then we will generate exactly the trace semantics of K_α. However, note that we will never obtain a computation structure using Def. 4 because the state space is infinite: there is at least one state in K_α for each possible valuation of variables as integers. The model checking techniques that we have discussed all assume that the computation we work with is described by a computation structure, which seems to pose problems for us now.

Example 1 Consider the program we looked at in the context of bounded model checking. Below is a version annotated with location labels.

```
ℓ0:   i := N;
ℓ1:   while(0 ≤ x < N) {
ℓ2:     i := i - 1;
ℓ3:     x := x + 1;
ℓ4:   }
```

We obtain the ϵ transition relation according to Definition 4 below. Notice that the construction technically calls for another state after ℓ_2, which transitions to ℓ_3 on $?\top$. This is not necessary, and is only specified in Definition 4 to make the formalisation easier to understand. We omit it in the diagram below to keep the relation concise.



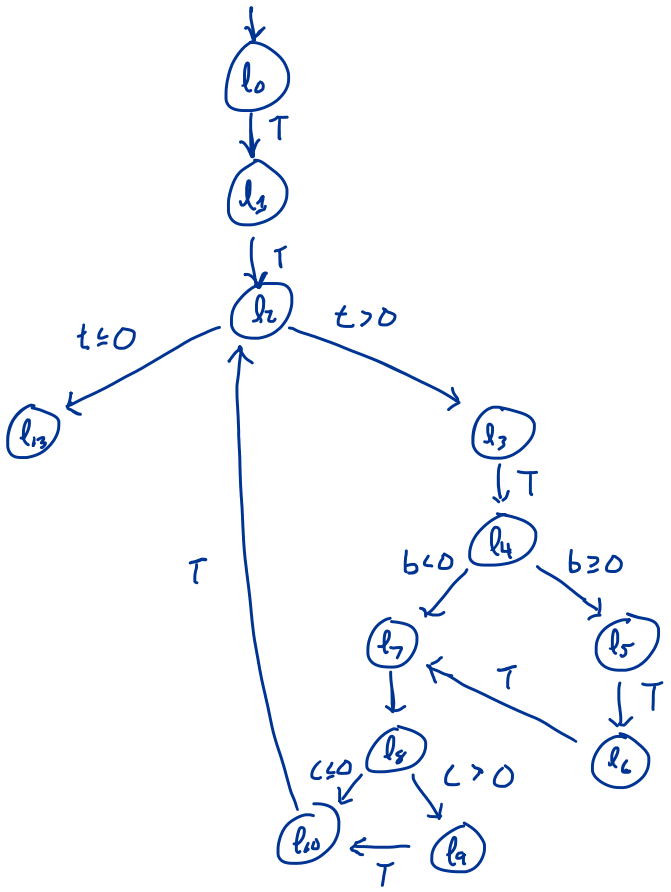
Example 2 Consider the following example, which uses a variable L in an attempt at a simple mutual exclusion protocol.

```
L := 0;
C := 0;
while(t > 0) {
  b := *;
  if(b >= 0) {
    L := 1;
    C := C + 1;
    // critical section
  }
  if(C > 0)
    L := 0;
  t := t - 1;
}
```

This program uses nondeterminism to simulate the fact that a process may not be granted a lock when requested, in the event that another process already holds it. We begin by annotating the program with locations.

```
ℓ0:          L := 0;
ℓ1:          C := 0;
ℓ2:          while(t > 0) {
ℓ3:            b := *;
ℓ4:            if(b >= 0) {
ℓ5:              L := 1;
ℓ6:              C := C + 1;
                 // critical section
ℓ7:            }
ℓ8:            if(C > 0)
ℓ9:              L := 0;
ℓ10:           t := t - 1;
ℓ11:         }
ℓ13:
```

The control flow transitions, with guards, are shown below. Note that we can add a self-loop to the final location ℓ_{13} to ensure that all states in this structure have a post-state.



The transitions we've constructed so far correspond to $\epsilon(\alpha)$. Now to construct K_α, we need one state for each location paired with each possible program valuation. However, we cannot hope to compute such a structure in its entirety, or write it down because of its infinite size. To address this, we will need to approximate the infinite statespace of K_α with a finite one using a technique called *predicate abstraction*.

5 Predicate Abstraction

Bounded model checking approximates the statespace "from below", by computing a subset of the reachable states and verifying that the ensuing traces always satisfy the property. In this sense, bounded model checking gives an underapproximation: any property violations that it finds are sure to be real, but because not all reachable states are explored, we might not discover some real violations.

Predicate abstraction is a different approach to the infinite statespace problem, which in a similar sense gives an *over*approximation. That is, it may "find" errors that do not correspond to real ones, but it will never miss an error that actually exists in the program. The main idea used in predicate abstraction is to merge states in K_α that have the same labeling of atomic propositions. This may not seem to get us very far

at first, as the labels used in Definition 4 were the the original source of the infinite statespace problem. However, by selecting the set of atomic propositions wisely, we can sidestep this problem while at the same time, in many cases, significantly reducing the overall number of states that need to be explored.

By example. Consider the mutual exclusion program from before. Crucial to this sort of protocol is that the lock be taken (ℓ_2) and released (ℓ_3) in proper order: a process that does not own a lock should not release it, as this could lead to violation of mutual exclusion safety.

To check this, we want to ensure what whenever the lock is taken by assigning $\texttt{L} := 1$ on ℓ_2, it is currently the case that $\texttt{L} = 0$. Likewise, whenever the lock is released on ℓ_3, then it must be that $\texttt{L} = 1$. This gives us two LTL safety properties.

$$\Box \ell_2 \to \texttt{L} = 0 \tag{1}$$
$$\Box \ell_4 \to \texttt{L} = 1 \tag{2}$$

In the above, we use the shorthand ℓ_i to denote any state $\langle \ell_i, \sigma \rangle$, for any σ. Likewise, $\texttt{L} = x$ denotes any state $\langle \ell, \texttt{L} = x \rangle$, for any ℓ.

Let's consider these formulas one at a time. In order to check (1), what states of K_α could we possibly need to explore? Before the first sequence of assignments are executed, `L` and `C` could take any values. It stands to reason that we must consider any initial state s where $v(s) \models \ell_0$. But after executing these assignments, we know that both variables will take value 0, so we must only consider in addition at this stage states s where $v(s) \models \ell_1 \wedge \texttt{L} = 0 \wedge \texttt{C} = 0$. Similarly, the only states that matter at ℓ_2 are those where $v(s) \models \ell_2 \wedge \texttt{b} > 0$.

Following on these observations, we come to the central idea of predicate abstraction: find a set of atomic predicates and corresponding *abstract* labeling function that is concise but sufficient to capture all of the *relevant* traces in the program. We then merge all of the states in the "concrete" transition structure K_α that share the same abstract labeling into one, and allow transitions liberally. In particular, if $\hat{s}$ and $\hat{s}'$ are abstract states and $\hat{v}$ an abstract labeling, then we draw a transition from $\hat{s} \hat{\curvearrowright} \hat{s}'$ iff there are concrete states s and s' where $s \curvearrowright s'$, and additionally $\hat{v}(s) = \hat{s}, \hat{v}(s') = \hat{s}'$.

We will continue with predicate abstraction in the next lecture, picking up where we left off.