15-414: Bug Catching: Automated Program Verification

Lecture Notes on Bounded Model Checking

Matt Fredrikson

Carnegie Mellon University
Lecture 21
November 25, 2025

1 Introduction

In this lecture, we will show how we can use solvers to either verify that some program is correct or find a counterexample that shows inputs to the program that may trigger some bug. One such approach is called *bounded model checking*. There are several challenges when trying to verify programs, foremost among them the fact state-space of programs may be infinite. Bounded model checking computes an *under*approximation of the reachable state-space by assuming a fixed computation depth in advance, and treating paths within this depth limit symbolically to explore all possible states. While this approach has its limitations, it can be effectively used in practice and it is a useful technique to have in our collection of verification techniques.

Learning Goals.

In this lecture, you will learn:

- How bounded model checking verifies an under-approximation of a program's semantics against a contract given by a Hoare triple, by leveraging the *strongest postcondition* introduced in Lecture 11.
- A key limitation of bounded model checking, i.e. the fact that it cannot prove the absence of all bugs, can be partially mitigated with *unwinding assertions*.

2 Bounded Model Checking

Bounded Model Checking computes an *underapproximation* of a program's semantics 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 following axiom, which allows us to replace a loop with a conditional statement, within which is a copy of the original loop.

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

The axiom 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 it 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.

```
1  i := N;
2  while(0 ≤ x < N) {
3    i := i - 1;
4    x := x + 1;
5  }
```

Suppose that we want to check that $\mathbf{G}\,terminated \rightarrow 0 \leq i$ is an invariant of this program, i.e. that it holds at the end of the program when it terminates. We will try with $k = 2$ here and unwind the loop twice by using the previous axiom $[\texttt{while}(Q)\,\alpha]P$ to replace while loops by a conditional program. Unwinding twice yields the following program, which reduces loops to one level of conditionals:

$$i := N; \texttt{if}(0 \leq x < N)\, i := i-1; x := x+1; \texttt{if}(0 \leq x < N)\, i := i-1; x := x+1; ?true?true?true$$

This computation unrolls all paths where the loop condition $0 \leq x < N$ becomes true at most twice.

To check whether our intended invariant holds for all terminating executions within this unwinding depth, we will use the *strongest postcondition* operator $\mathrm{sp}_\alpha(P)$ to compute the set of states reachable after running a program α from a set of starting states described by (first-order) formula P. For assignments and nondeterministic assignments, the strongest postcondition is well known and is given as follows:

$$\begin{array}{lcl} \mathrm{sp}_{x:=e}(P) & \equiv & \exists u.\, \big(P[x \leftarrow u] \wedge x = e[x \leftarrow u]\big) \\ \mathrm{sp}_{x:=*}(P) & \equiv & \exists x.\, P \end{array}$$

For tests: $\mathrm{sp}_{?Q}(P) \equiv P \wedge Q$. Moreover, the strongest postcondition for a program block is obtained by composing the strongest postconditions of its parts. For the program $\alpha;\beta$ this is defined by: $\mathrm{sp}_{\alpha;\beta}(P) \equiv \mathrm{sp}_\beta(\mathrm{sp}_\alpha(P))$.

Returning to our program, suppose we initialize N, x, i nondeterministically. Intuitively this means we should consider all initial states. Using the strongest postcondition, we can formalize this as starting with the formula true: $P = \top$. The formula describing the set of states reachable by the program above is given by:

$$\mathrm{sp}_{i:=N;\,\mathtt{if}(0\le x<N)\,i:=i-1;x:=x+1;\mathtt{if}(0\le x<N)\,i:=i-1;x:=x+1;?true?true?true}(\top)$$

The program is a composition $\alpha;\beta$ of $\alpha \equiv i := N$ and

$$\beta \equiv \mathtt{if}(0 \le x < N)\, i := i-1; x := x+1; \mathtt{if}(0 \le x < N)\, i := i-1; x := x+1; ?true?true?true$$

We compute $\mathrm{sp}_{\alpha;\beta}(\top)$ by first computing $\mathrm{sp}_{i:=N}(\top)$ and then $\mathrm{sp}_{\beta}(\,\cdot\,)$ for the resulting formula. We can avoid unnecessary encumbrance on the derivation by applying logical equivalences to simplify as we go. For α we obtain:

$$\mathrm{sp}_{i:=N}(\top) \equiv \exists i\, \top \wedge i = N \equiv i = N.$$

Now to compute the strongest postcondition of the rest of the program β, it will be helpful to use the following proposition.

Proposition 1. $\mathrm{sp}_{\,\mathtt{if}(Q)\,\alpha\beta}(P) \equiv \mathrm{sp}_{\alpha}(P \wedge Q) \;\vee\; \mathrm{sp}_{\beta}(P \wedge \neg Q)$.

Proposition 1 follows directly from the axiomatic semantics of dynamic logic and the weakest precondition rules (by duality). Using it, we can combine the results to compute the set of states reachable by the program and express the property we want to verify as follows:

$$\big(\mathrm{sp}_{\beta}(\,i = N\,)\big) \to (0 \le i).$$

This reduces the verification problem to checking the validity of the above formula. If it is valid, then the invariant holds for all executions up to unwinding depth $k = 2$. If not, the witness gives a counterexample trace within the bound.

We can apply exactly the same idea to check safety with *unwinding assertions*. The idea is simple: each time we truncate a loop because we have reached the bound k, we assert that the loop condition is false. If a violation is found, the counterexample is either a genuine bug or indicates that the bound was too small.

3 Software Transition Structures (Review)

Definition 2 (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 $\mathit{locs}(\alpha)$, $\iota(\alpha)$ be the *initial* locations of α, and $\kappa(\alpha)$ be the *final* locations of α:

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

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

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.
- $\epsilon(\alpha;\beta) = \epsilon(\alpha) \cup \epsilon(\beta) \cup \{(\ell_f, ?\top, \ell_i) : \ell_i \in \iota(\beta), \ell_f \in \kappa(\alpha)\}$

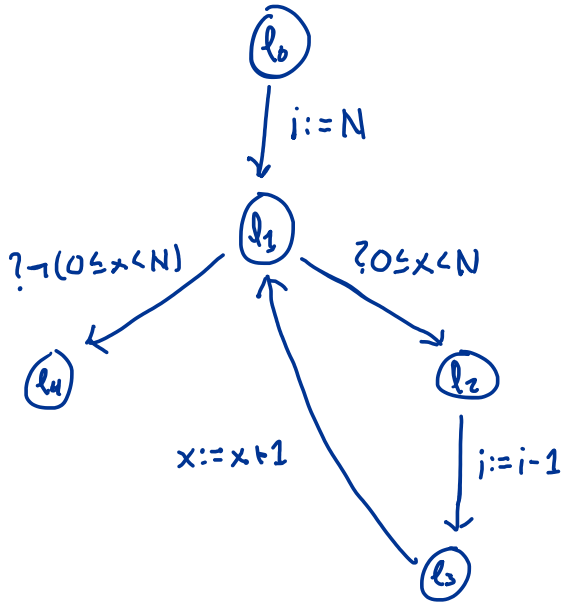
Notice that control flow transitions are associated with statements. Intuitively, the locations at the source of a transition correspond to the state immeidately prior to exeucting a statement, and those at the destination the state immediately after. Then the transition structure $K_\alpha = (W, I,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

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

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

We obtain the ϵ transition relation according to Definition 2 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 2 to make the formalisation easier to understand. We omit it in the diagram below to keep the relation concise.



4 Predicate Abstraction

Recall from the previous lecture the central idea of predicate abstraction: define a set of abstract atomic predicates $\hat{\Sigma}$ that is concise, but still allows us to distinguish all of the traces relevant to our property. We then create a new transition structure whose states correspond to sets of abstract propositions (i.e., elements of $\wp(\hat{\Sigma})$) rather than program states.

Consider the example above, and suppose that we select $\hat{\Sigma} = \{0 \leq i\}$. Then the states in the abstraction will correspond to:

$$\{\ell_0, \ell_1, \ell_2, \ell_3, \ell_4\} \times \{\emptyset, 0 \leq i\}$$

Intuitively, the state $\langle \ell_0, 0 \leq i \rangle$ corresponds to any state in K_α at ℓ_0 where $0 \leq i$. Likewise, $\langle \ell_0, \emptyset \rangle$ corresponds to any state at ℓ_0 where $0 > i$. An abstract state labeled

$\langle \ell_0, \{\emptyset, 0 \leq i\}\rangle$ corresponds to any state at ℓ_0 satisfying $0 > i \wedge 0 \leq i$, which in fact means no concrete states due to the contradiction. Generally, an abstract state that does *not* contain a predicate $P \in \hat{\Sigma}$ is interpreted as corresponding to concrete states in K_α that satisfy the negation of P. If an abstract state corresponds to more than one predicate, then we interpret it as corresponding to concrete states that satisfy the conjunction of those predicates.

Definition 3. Given a set of predicates $A \in \hat{\Sigma}$, let $\gamma(A)$ be the set of program states $\sigma \in \mathcal{S}$ that satisfy the conjunction of predicates in A:

$$\gamma(A) = \{\sigma \in \mathcal{S} : \sigma \models \textstyle\bigwedge_{a \in A} a\}$$

Definition 4 (Abstract Transition Structure)**.** Given a program α, a set of abstract atomic predicates $\hat{\Sigma}$, and control flow transition relation $\epsilon(\alpha)$ (Def. 2), 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 as given in Definition 2. The abstract transition structure $\hat{K_\alpha} = (\hat{W}, \hat{I}, \hat{\curvearrowright}, \hat{v})$ is a tuple containing:

- $\hat{W} = locs(\alpha) \times \wp(\hat{\Sigma})$ are the states defined as pairs of program locations and sets of abstraction predicates.
- $\hat{I} = \{\langle \ell, A\rangle \in \hat{W} : \ell \in \iota(\alpha)\}$ are the initial states corresponding to initial program locations.
- $\hat{\curvearrowright} = \{(\langle \ell, A\rangle, \langle \ell', A'\rangle : \text{ for } (\ell, \beta, \ell') \in \epsilon(\alpha) \text{ where there exist } \sigma, \sigma' \text{ such that } \sigma \in \gamma(A), \sigma' \in \gamma(A') \text{ and } (\sigma, \sigma') \in [\![\beta]\!]\}$ is the transition relation.
- $\hat{v}(\langle \ell, A\rangle) = \langle \ell, A\rangle$ is the labeling function, which is in direct correspondence with states.

Theorem 5. *For any trace $\langle \ell_0, \sigma_0\rangle, \langle \ell_1, \sigma_1\rangle, \ldots$ of K_α, there exists a corresponding trace of $\hat{K_\alpha}$ $\langle \hat{\ell}_0, A_0\rangle, \langle \hat{\ell}_1, A_1\rangle, \ldots$ such that for all $i \geq 0$, $\ell_i = \hat{\ell}_i$ and $\sigma_i \in \gamma(A_i)$.*

Proof. We proceed by induction on the length of the trace $\langle \ell_0, \sigma_0\rangle, \langle \ell_1, \sigma_1\rangle, \ldots$ of K_α.

Length=1: By Definition 2, the trace is $\langle \ell_0, \sigma_0\rangle$ where $\ell_0 \in \iota(\alpha)$. Then let A be such that $\sigma_0 \in \gamma(A)$; we know that such an A exists, because $\wp(\hat{\Sigma})$ covers the entire statespace $\mathcal{S}$. Then $\langle \ell_0, A\rangle$ is an initial state of $\hat{K_\alpha}$ as well, so it is a trace of length 1 in $\hat{K_\alpha}$.

Length=n+1: We have that $\langle \ell_0, \sigma_0\rangle, \ldots, \langle \ell_{n+1}, \sigma_{n+1}\rangle$ is a trace of K_α. By the inductive hypothesis, there must exist a trace $\langle \hat{\ell}_0, A_0\rangle, \ldots, \langle \hat{\ell}_n, A_n\rangle$ of $\hat{K_\alpha}$ such that for all $0 \leq i \leq n$, $\ell_i = \hat{\ell}_i$ and $\sigma_i \in \gamma(A_i)$. Then let A_{n+1} be such that $\sigma_{n+1} \in \gamma(A_{n+1})$. Because $\langle \ell_n, \sigma_n\rangle \curvearrowright \langle \ell_{n+1}, \sigma_{n+1}\rangle$, we know that there exists $(\ell_n, \beta, \ell_{n+1} \in \epsilon(\alpha)$ where $(\sigma_n, \sigma_{n+1}) \in [\![\beta]\!]$. Then by Definition 4, it must be that $\langle \hat{\ell}_n, A_n\rangle \hat{\curvearrowright} \langle \hat{\ell}_{n+1}, A_{n+1}\rangle$. So $\langle \hat{\ell}_0, A_0\rangle, \ldots, \langle \hat{\ell}_{n+1}, A_{n+1}\rangle$ is a trace in $\hat{K_\alpha}$ where for $0 \leq i \leq n+1$ we have that $\sigma_i \in \gamma(A_i)$.

□

Theorem 5 tells us that $\hat{K_\alpha}$ can be used to deduce properties about K_α: any trace in K_α is also in $\hat{K_\alpha}$, so any property of K_α is also one of $\hat{K_\alpha}$. However, Theorem 5 also tells us that $\hat{K_\alpha}$ overapproximates K_α, so some properties of $\hat{K_\alpha}$ may not be properties of K_α.

Definition 4 tells us what an abstract transition structure for a program is, given a set $\hat{\Sigma}$ of predicates. We are ultimately interested in computing the structure, for use in model checking. On initial inspection, this seems quite feasible as there are $|locs(\alpha)| \times 2^{|\hat{\Sigma}|}$ states in $\hat{K_\alpha}$, so enumerating them is not an issue as long as we keep $\hat{\Sigma}$ small. But what about the transitions? There are still an infinite number of program states to contend with, so naive searching of σ, σ' to satisfy the condition on $\hat{\rightsquigarrow}$ is not feasible.

When deciding whether to add a transition to $\hat{K_\alpha}$, we only care about the existence of σ, σ' that satisfy the requirements of Definition 4. It is thus sufficient for our purposes to determine whether there are *any* $\sigma' \in \gamma(A')$ that are reachable from executing β starting in $\sigma \in \gamma(A)$. Equivalently, we can determine whether it is always the case that when starting in $\sigma \in \gamma(A)$, we end up in $\sigma' \in \gamma(A')$ after executing β. Note that this is exactly the same as determining the validity of $\bigwedge_{a \in A} a \to [\beta] \bigvee_{a' \in A'} \neg a'$.

Theorem 6. *Let* $(\ell, \beta, \ell') \in \epsilon(\alpha)$. *Then* $\langle \ell, A \rangle \hat{\rightsquigarrow} \langle \ell', A' \rangle$ *iff* $\bigwedge_{a \in A} a \to [\beta] \bigvee_{a' \in A'} \neg a'$ *is* not *valid.*

Theorem 6 tells us that we can reason about transitions in $\hat{K_\alpha}$ by determining the validity of first order dynamic logic formulas. Moreover, looking at the construction of $\epsilon(\alpha)$ given in Definition 2, we see that the only programs forms that can appear on transitions in $\epsilon(\alpha)$ are assignments and tests; there are no loops, conditionals, or even composition operators. This means that by a single application if [:=] or [?], the DL formula stipulated in Theorem 6 is reducible to an arithmetic formula that can be solved with a decision procedure.

Example continued Let us derive the edges of the abstract transition structure $\hat{K_\alpha}$ for our example when $\hat{\Sigma} = \{0 \leq i\}$. We evaluate potential edges of the form $\langle \ell, A \rangle \hat{\rightsquigarrow} \langle \ell', A' \rangle$ by checking the corresponding formula in Theorem 6.

- $\langle \ell_0, 0 > i \rangle \hat{\rightsquigarrow} \langle \ell_1, 0 > i \rangle$: The program between ℓ_0 and ℓ_1 is $i := N$. By Theorem 6, we must decide the validity of $0 > i \to [i := N]0 \leq i$. By [:=], we can reduce this to $0 > i \to 0 \leq N$, which is not valid: it is falsified by setting $i = -1, N = 0$. So this edge is added to $\hat{\rightsquigarrow}$.

- $\langle \ell_2, 0 > i \rangle \hat{\rightsquigarrow} \langle \ell_3, 0 \leq i \rangle$: The program between ℓ_2 and ℓ_3 is $i := i - 1$. By Theorem 6, we must decide the validity of $0 > i \to [i := i - 1]0 > i$. By [:=], we can reduce this to $0 > i \to 0 > i - 1$, which is valid. So this edge is *not* added to $\hat{\rightsquigarrow}$.

- $\langle \ell_1, 0 > i \rangle \hat{\rightsquigarrow} \langle \ell_4, 0 > i \rangle$: The program between ℓ_1 and ℓ_4 is $?\neg(0 \leq x < N)$. By Theorem 6, we must decide the validity of $0 > i \to [?\neg(0 \leq x < N)]0 \leq i$. By [?],

we can reduce this to $0 > i \land \neg(0 \leq x < N) \to 0 \leq i$, which is not valid: it is falsified by $i = -1, x = 0$. So this edge is added to $\hat{\curvearrowright}$.

- $\langle \ell_0, 0 > i\rangle \hat{\curvearrowright} \langle \ell_1, 0 \leq i\rangle$: The program between ℓ_0 and ℓ_1 is $i := N$. By Theorem 6, we must decide the validity of $0 > i \to [i := N]0 > i$. By [:=], we can reduce this to $0 > i \to 0 > N$, which is not valid: it is falsified by setting $i = -1, N = -1$. So this edge is added to $\hat{\curvearrowright}$.
- $\langle \ell_1, 0 \leq i\rangle \hat{\curvearrowright} \langle \ell_2, 0 > i\rangle$: The program between ℓ_1 and ℓ_2 is $?0 \leq x < N$. By Theorem 6, we must decide the validity of $0 \leq i \to [?0 \leq x < N]0 \leq i$. By [?], we can reduce this to $0 \leq i \land 0 \leq x < N \to 0 \leq i$, which is not valid: it is falsified by setting $x = 0, i = -1$. So this edge is added to $\hat{\curvearrowright}$.

We can use these edges to find potential counterexamples that could point to real bugs. For instance, consider the path

$$\langle \ell_0, 0 > i\rangle \hat{\curvearrowright} \langle \ell_1, 0 > i\rangle \hat{\curvearrowright} \langle \ell_4, 0 > i\rangle$$

Because $\hat{K_\alpha}$ overapproximates the true transition structure K_α, we need to determine whether this does in fact correspond to a path in K_α, or whether it is merely an artifact of the overapproximation. If it is a spurious artifact, then we can reason that the corresponding path in K_α does *not* violate the safety property. Equivalently, it would mean the the following formula must be valid:

$$0 > i \land \neg(0 \leq x < N) \to 0 \leq i.$$

But, indeed, this formula is not valid, since $i = -1$ and $x = 0$ are a witness rendering it false. The abstract counterexample corresponds to a real counterexample as well and our safety property was not satisfied by the program.

To see a case where the abstraction *does* introduce a spurious counterexample, consider a different abstraction with $\hat{\Sigma} = \{0 > i\}$. Then for the same program we might derive the following path:

$$\langle \ell_0, 0 > i\rangle \hat{\curvearrowright} \langle \ell_1, 0 > i\rangle \hat{\curvearrowright} \langle \ell_4, 0 > i\rangle$$

Because this path is not feasible in the concrete program, this example illustrates how abstraction can introduce spurious behaviors that must be ruled out by refinement.