15-414: Bug Catching: Automated Program Verification

Lecture Notes on Model Checking Abstractions

Matt Fredrikson

Carnegie Mellon University
Lecture 23

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 an approach for dealing with this called *predicate abstraction*.

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. The good news is that it is always feasible to do so, as there are a finite number of states and the transitions can be computed using familiar techniques. The bad news is that often it is the case that crucial information gets lost in the approximation, leaving us unable to find real bugs or verify their absence. We'll eventually see how to incrementally fix this using a technique called *abstraction refinement*, which leads to interesting new questions about automated software verification.

2 Review: trace semantics

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 $\alpha^0 \stackrel{\text{def}}{\equiv} (?\mathit{true})$.

3 Computation structures of programs

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 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:

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

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 $\epsilon(\alpha) \subseteq \mathit{locs}(\alpha) \times \mathit{progs} \times \mathit{locs}(\alpha)$ 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:

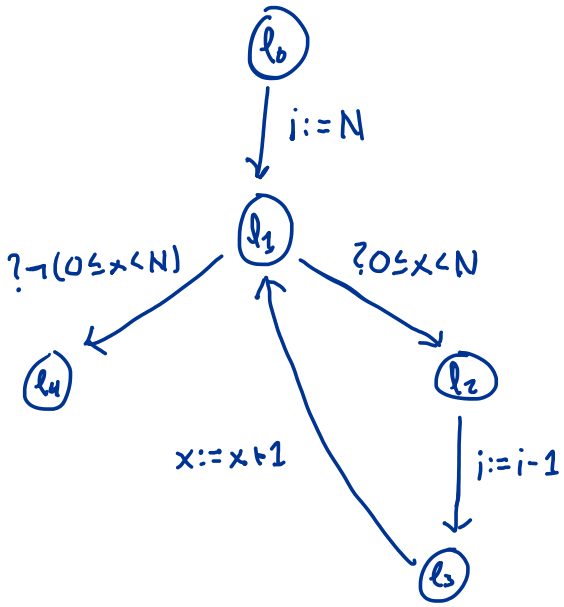
- $W = \mathit{locs}(\alpha) \times \{\mathcal{S}\}$, $I = \{\langle \ell_i, \sigma\rangle : \ell_i \in \iota(\alpha)\}$.
- $\curvearrowright = \{(\langle \ell, \sigma\rangle, \langle \ell', \sigma'\rangle) :$ for $(\ell, \beta, \ell') \in \epsilon(\alpha)$ where $(\sigma, \sigma') \in [\![\beta]\!]\}$.
  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 2 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. 2 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 following simple program, 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 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.



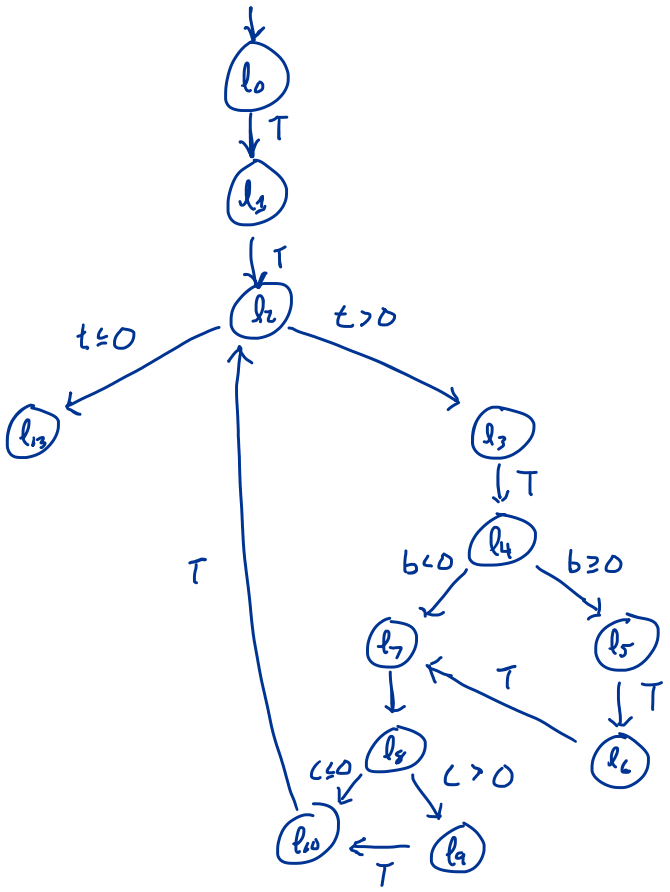
Example 2 Consider the following example, which uses a variable `L` in an attempt at a simple mutual exclusion protocol.

```
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. We omit the assignment statements on edges to avoid clutter in the diagram, but it is easy to find the appropriate statement for a transition if we need to. 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*.

4 Predicate Abstraction

Predicate abstraction gives an *over*approximation to the program's computation structure K_α in the sense that for any path in K_α, there exists a corresponding path in its abstraction. However, the abstraction may contain some paths for which there is no correspondent in K_α, as it is an approximation.

Thinking about what this means, model checking such an abstraction may result in finding bugs that do not correspond to real ones, but doing so 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 2 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 $v(s) \models \ell_2 \wedge \texttt{b} > 0$.

4.1 Computing abstractions

Following on the observations from this example, 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 $\hat{v}(s) = \hat{s}, \hat{v}(s') = \hat{s}'$.

Consider the first example from earlier, 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 $0 \leq i$. 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. So for example, $\langle \ell_0, \emptyset \rangle$ corresponds to any state at ℓ_0 where $0 > i$. 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 $\sigma_i \in \gamma(A_i)$.

This completes the proof. □

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 $\hat{\curvearrowright}$ 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 $A, B \subseteq \hat{\Sigma}$ be sets of predicates over program states, and β be a program. Then for $\sigma \in \gamma(A)$, there exists a state $\sigma' \in \gamma(B)$ such that $(\sigma, \sigma') \in [\![\beta]\!]$ if and only if $\bigwedge_{a \in A} a \to [\beta] \bigvee_{b \in B} \neg b$ is not valid.*

Proof. First we prove that $\bigwedge_{a \in A} a \to [\beta] \bigvee_{b \in B} \neg b$ not valid implies that $\exists \sigma, \sigma' . \sigma \in \gamma(A) \wedge \sigma' \in \gamma(B) \wedge (\sigma, \sigma') \in [\![\beta]\!]$. First we know that there is some $\sigma \in \gamma(A)$ because the formula is not valid, so $\bigwedge_{a \in A} a$ is not equivalent to false. Then by the semantics of $[\cdot]$, we know that there exists some $\sigma' \models \bigwedge_{b \in B} b$ reachable by running β starting in a state $\sigma \models \bigwedge_{a \in A} a$, i.e., $(\sigma, \sigma') \in [\![\beta]\!]$. So then $\sigma \in \gamma(A)$, and $\sigma' \in \gamma(B)$, finishing the proof in this direction.

Now in the other direction, we show that if there exists σ, σ' where $\sigma \in \gamma(A) \wedge \sigma' \in \gamma(B) \wedge (\sigma, \sigma') \in [\![\beta]\!]$, then $\bigwedge_{a \in A} a \to [\beta] \bigvee_{b \in B} \neg b$ is not valid. Because $\sigma' \in \gamma(B)$, we know that $\sigma' \models \bigwedge_{b \in B} b$ and like wise because $\sigma \in \gamma(A)$ that $\sigma \models \bigwedge_{a \in A} a$. So not all states $\sigma \models \bigwedge_{a \in A}$ reach a final state in $\bigvee_{b \in B} \neg b$ after running β, which finishes the proof in this direction. □

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 Let us go back to the program from before, and again use $\hat{\Sigma} = \{0 \leq i\}$. For clarity, we will be explicit about the abstract conjunctions in each state, and consider the state space of our abstraction $\hat{K_\alpha}$ to be $\{\ell_0, \ell_1, \ell_2, \ell_3, \ell_4\} \times \{0 > i, 0 \leq i\}$. Now we must decide the transitions. We will work out several of them in some detail to demonstrate the reasoning, but leave the rest as an exercise due to the large number of possible transitions.

- $\langle \ell_0, 0 > i \rangle \hat{\curvearrowright} \langle \ell_1, 0 > i \rangle$: 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 $\hat{\curvearrowright}$.
- $\langle \ell_2, 0 > i \rangle \hat{\curvearrowright} \langle \ell_3, 0 \leq i \rangle$: 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 $\hat{\curvearrowright}$.
- $\langle \ell_1, 0 > i \rangle \hat{\curvearrowright} \langle \ell_4, 0 > i \rangle$: 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 $0 > i \wedge \neg(0 \leq x < N) \to 0 \leq i$, 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 $0 \leq i \wedge 0 \leq x < N \to 0 \leq i$, which is not valid: it is falsified by setting $x = 0, i = -1$. So this edge is added to $\hat{\curvearrowright}$.

Now suppose that we want to verify the property: $\Box \ell_4 \to 0 \leq i$. Notice from what we just worked out above that there is a counterexample path in $\hat{K_\alpha}$:

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

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:

$$0 > i \to [i := N; ?\neg(0 \leq x < N)]0 \leq i$$

Applying [;],[?],[:=], the formula above reduces to $0 \geq i \to \neg(0 \leq x < N) \to 0 \leq N$. This is not valid, which we see from the assignment $i = 0, x = 0, N = -1$. So in fact the counterexample is correct, and we conclude that the property does not hold.

Spurious counterexamples Now let's consider modifying the example a bit, by changing the first assignment such that `i` always takes a positive value at first.

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

Now the counterexample from before no longer works, because there is no edge from $\langle \ell_0, 0 > i \rangle \hat{\curvearrowright} \langle \ell_1, 0 > i \rangle$. To see why, observe that from Theorem 6 we reason:

$$(0 > i \to [i := \texttt{abs}(N) + 1]0 \leq i) \leftrightarrow (0 > i \to 0 \leq \texttt{abs}(N) + 1) \text{ is valid}$$

But there is another counterexample, which we see taking the following steps.

1. $\langle \ell_0, 0 \leq i \rangle \hat{\curvearrowright} \langle \ell_1, 0 \leq i \rangle$. This edge is in $\hat{K_\alpha}$ because $0 \leq i \to [i := \texttt{abs}(N) + 1]0 > i$ is equivalent to $0 \leq i \to 0 > \texttt{abs}(N) + 1$, which is not valid.
2. $\langle \ell_1, 0 \leq i \rangle \hat{\curvearrowright} \langle \ell_2, 0 \leq i \rangle$. This edge exists because $0 \leq i \to [?0 \leq x < N]0 > i$ is equivalent to $0 \leq i \to 0 \leq x < N \to 0 > i$, which is not valid.
3. $\langle \ell_2, 0 \leq i \rangle \hat{\curvearrowright} \langle \ell_3, 0 > i \rangle$. This edge exists because $0 \leq i \to [i := i - 1]0 \leq i$ is equivalent to $0 \leq i \to 0 \leq i - 1$ and is not valid, seen from the assignment $i = 0$.
4. $\langle \ell_3, 0 > i \rangle \hat{\curvearrowright} \langle \ell_1, 0 > i \rangle$. This edge exists because $0 > i \to [x := x + 1]0 \leq i$ is equivalent to $0 > i \to 0 \leq i$, which is not valid.
5. $\langle \ell_1, 0 > i \rangle \hat{\curvearrowright} \langle \ell_4, 0 > i \rangle$. This edge exists because $0 > i \to [?\neg(0 \leq x < N)]0 \leq i$ is equivalent to $0 > i \to \neg(0 \leq x < N) \to 0 \leq i$ is not valid.

At this point, $\hat{K_\alpha}$ is in a state satisfying $\ell_4 \wedge \neg(0 \leq i)$. As before, we need to determine whether this counterexample is spurious. We consider a path which starts in a state where $0 \leq i$, and transitions through $\ell_0, \ell_1, \ell_2, \ell_3, \ell_1, \ell_4$, ending in a state where $0 > i$. This leads us to ask whether the following DL formula is valid:

$$0 \leq i \to [i := \texttt{abs}(N) + 1; ?0 \leq x < N; i := i - 1; x := x + 1; ?\neg(0 \leq x < N)]0 \leq i$$

Multiple applications of [;],[?],[:=] leave us with the valid formula:

$$0 \leq i \to 0 \leq x < N \to \neg(0 \leq x + 1 < N) \to 0 \leq \texttt{abs}(N)$$

The validity of this formula tells us that executing the statements in this counterexample will necessarily lead to a program state where $0 \leq i$, which does not violate the property $\Box \ell_4 \to 0 \leq i$. So this counterexample is *spurious*: it exists in the abstraction $\hat{K_\alpha}$, but not in the true transition system K_α corresponding to the program.

5 Abstraction Refinement

What do we do when we encounter a spurious counterexample? In practical terms, these pose a real problem. We can't verify the absence of bugs in the system using $\hat{K_\alpha}$ because we know that there are traces in the abstraction that violate the property. We could simply ignore the spurious counterexample, and continue searching for valid counterexamples in the abstraction. If we ever come across one, then we stop knowing that the program has at least one trace that actually violates the property. If we

exhaust all of the counterexamples in $\hat{K_\alpha}$ without finding a valid counterexample, then we conclude that K_α satisfies the property.

The problem with this approach is that there may be an infinite number of spurious counterexamples in $\hat{K_\alpha}$. Consider the most recent example from the previous section. There are an infinite number of counterexample traces in the abstraction because of the cycle introduced by the loop. None of them is a valid counterexample, which we know because the program satisfies the property.

Instead, we can attempt to make the abstraction a better approximation of K_α. At the moment, $\hat{K_\alpha}$ only keeps track of one fact about the program's state: whether or not $0 \leq i$. This fact alone is not strong enough to conclue that after executing $i := i - 1$ possibly multiple times within the loop, $0 \leq i$ will continue to hold when the loop terminates. Concretely, if all that we know before executing $i := i - 1$ is that $0 \leq i$, then we have to allow for the possibility that $i = 0$ and so $0 > i$ holds after the assignment. This is what gives rise to the spurious counterexamples in our abstraction.

We refine the abstraction by considering additional predicates to keep track of facts about the program state that are necessary to remove the counterexample. In the most recent counterexample trace, we know that after executing $i := i - 1$ it still holds that $0 \leq i$, because when the assignment occurs $i = \texttt{abs}(N) + 1$. Suppose that we add this predicate to our abstraction set in addition to $0 \leq i$. Then going back to what would occur on our counterexample trace, we have the following.

1. $\langle \ell_0, 0 \leq i \rangle \hat{\rightsquigarrow} \langle \ell_1, 0 \leq i \land i = \texttt{abs}(N) + 1 \rangle$. This edge is in $\hat{K_\alpha}$ because $0 \leq i \rightarrow [i := \texttt{abs}(N) + 1] \neg (0 \leq i \land i = \texttt{abs}(N) + 1 \rangle)$ is equivalent to $0 \leq i \rightarrow \neg (0 \leq \texttt{abs}(N) + 1) \land \texttt{abs}(N) + 1 = \texttt{abs}(N) + 1) \rangle$, which is not valid.

2. $\langle \ell_1, 0 \leq i \land i = \texttt{abs}(N) + 1 \rangle \hat{\rightsquigarrow} \langle \ell_2, 0 \leq i \land i = \texttt{abs}(N) + 1 \rangle$. This edge exists because $0 \leq i \land i = \texttt{abs}(N) + 1 \rightarrow [?0 \leq x < N] \neg (0 \leq i \land i = \texttt{abs}(N) + 1)$, which is not valid.

3. $\langle \ell_2, 0 \leq i \land i = \texttt{abs}(N) + 1 \rangle \hat{\rightsquigarrow} \langle \ell_3, 0 \leq i \rangle$. This edge exists because $0 \leq i \land i = \texttt{abs}(N) + 1 \rightarrow [i := i - 1] 0 > i$ is equivalent to $0 \leq i \land i = \texttt{abs}(N) + 1 \rightarrow 0 > i - 1$ and is not valid.

4. $\langle \ell_3, 0 \leq i \rangle \hat{\rightsquigarrow} \langle \ell_1, 0 \leq i \rangle$. This edge exists because $0 \leq i \rightarrow [x := x + 1] 0 > i$ is equivalent to $0 \leq i \rightarrow 0 > i$, which is not valid.

However, at this point $\hat{K_\alpha}$ is back in states where it is only true that $0 \leq i$. Another iteration of the loop will lead to entry of states there $0 > i$ after the assignment to i, and we will find another spurious counterexample.

In order to derive an abstraction that is useful in proving the correctness of this program with respect to the property $\Box \ell_4 \rightarrow 0 \leq i$, we need to find a set of predicates that characterizes the relationship between i, x, and N. Consider the following:

$$\begin{aligned} a_0 &\equiv i \geq |N| - |x| \\ a_1 &\equiv i > |N| - |x| \\ a_2 &\equiv 0 \leq x \leq N \\ a_3 &\equiv 0 \leq x < N \end{aligned}$$

Using these predicates, we can reason as follows.

- The only transition from an ℓ_0 state to an ℓ_1 state also contains predicate a_1 in the ℓ_1 state. We see that $\mathit{true} \to [i := \texttt{abs}(N)+1]\neg a_1$ is equivalent to $|N|+1 \leq |N|-|x|$, which is not valid. Any state *not* containing a_1 will result in a validity test of the form $[i := \texttt{abs}(N)+1]\neg(\neg a_1 \wedge A') \leftrightarrow [i := \texttt{abs}(N)+1]a_1 \vee \neg A'$, which is valid.
- The only transition from an ℓ_1 state to an ℓ_2 state is one in which a_3 holds, and if a_0 or a_1 held previously in the ℓ_1 state, then they will also hold in the ℓ_2 state. This is obvious because the test $?0 \leq x < N$ does not change the value of i or N.
- Any transition from an ℓ_2 state where a_1 holds will land in an ℓ_3 state where a_0 holds. We see that $a_1 \to [i := i-1]\neg a_0$ is equivalent to $i > |N| - |x| \to i - 1 < |N| - |x|$, which is not valid. Furthermore, a_0 *must* hold in the post-state, because $a_1 \to [i := i-1]a_0$ is equivalent to $i > |N| - |x| \to i - 1 \geq |N| - |x|$, which is valid.
- Any transition from ℓ_3 to ℓ_1 where a_0 holds in ℓ_3 will result in a_1 holding in ℓ_1. We have $a_0 \to [x := x+1]\neg a_1$ is equivalent to $i \geq |N| - |x| \to i \leq |N| - |x+1|$ which is not valid.
- Any transition from ℓ_3 to ℓ_1 where a_3 holds in ℓ_3 will result in either a_2 or a_3 holding in ℓ_1.
- Any transition from ℓ_1 to ℓ_4 where a_1 and a_2 hold in ℓ_1 will result in a_0 and a_2 at ℓ_4. We see that $a_1 \wedge a_2 \to [?\neg(0 \leq x < N)]\neg a_0 \vee \neg a_2$ is equivalent to $i > |N| - |x| \wedge 0 \leq x \leq N \to \neg(0 \leq x < N) \to i < |N| - |x| \vee \neg(0 \leq x \leq N)$ is not valid.
- Any state where a_0 and a_2 hold must also be one where $0 \leq i$ holds, because $i \geq |N| - |x| \wedge 0 \leq x \leq N \to 0 \leq i$ is valid.

From this reasoning, we see that all reachable ℓ_1 states have $a_1 \equiv i > |N| - |x|$ and either a_2 or a_3. The only reachable ℓ_4 states must go through $\ell_1 \wedge a_1 \wedge a_2$, and so must have $a_0 \wedge a_2$, which combined imply $0 \leq i$. Thus, there are no counterexample traces in the abstraction. Because $\hat{K_\alpha}$ *over*approximates K_α, we know that any trace of K_α is also one of $\hat{K_\alpha}$. We can then conclude that there are no counterexamples in K_α, and the program satisfies the property.

Automatic Refinement We've now shown that it's possible, at least in principle, to construct a predicate abstraction that is a close enough approximation to the true transition structure to conclude that there are no bugs in a system. But in the example we just saw, we needed to refine the set of predicates with those containing enough information about the inductive properties of the loop to rule out spurious counterexamples. It is not a coincidence that identifying those predicates felt a bit like coming up with a loop invariant for deductive verification, because that is essentially what we did.

When doing deductive verification, we did not expect to derive a procedure for automatically finding loop invariants. So how is abstraction refinement useful for model

checking, where the primary goal is to verify programs (or find bugs) automatically? First of all, model checking is not a magic bullet: there is no guarantee that it will be able to prove the absence of bugs. And for good reason, because that problem is undecidable.

But in the next lecture we will look at techniques for automatic abstraction refinement that work well on many interesting programs and properties. The general approach is called *Counterexample-Guided Abstraction Refinement* (CEGAR), and uses the information contained in spurious counterexamples to derive useful predicates for refinement.