1  Introduction

In the previous lecture we saw how to create a Kripke structure whose language is equivalent to the trace semantics of a program. However, this is problematic for model checking due to the fact that there are an infinite number of states in the structure. We began describing a way to address this using predicate abstraction, which overapproximates the Kripke structure by partitioning Kripke states into a finite number of abstract states.

Today we will continue with predicate abstraction, and see how to create an abstract transition structure for an arbitrary program. 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 see how to incrementally fix this using a technique called refinement, which leads to interesting new questions about automated software verification.

2  Review: Software transition structures

Definition 1 (Transition Structure of a Program). Given a program $\alpha$ over program states $S$, let $L$ be a set of locations given by the inductively-defined function $\text{locs}(\alpha)$, $\iota(\alpha)$ be the initial locations of $\alpha$, and $\kappa(\alpha)$ be the final locations of $\alpha$:

- $\text{locs}(x := e) = \{\ell_i, \ell_f\}$, $\iota(x := e) = \{\ell_i\}$, $\kappa(x := e) = \{\ell_f\}$
- $\text{locs}(?Q) = \{\ell_i, \ell_f\}$, $\iota(?Q) = \{\ell_i\}$, $\kappa(?Q) = \{\ell_f\}$
L20.2 Abstraction and Refinement

associated with that location. The control flow transition relation \( K \) structure a statement, and those at the destination the state immediately after. Then the transition \( t \) transitions at the source of a transition correspond to the state immediately prior to executing

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

- \( W = \text{locs}(\alpha) \times \{S\}, I = \{\ell_i, \sigma : \ell_i \in \iota(\alpha)\} \).
- \( \sqcap = \{((\ell, \sigma), (\ell', \sigma')) : \text{for } (\ell, \beta, \ell') \in \epsilon(\alpha) \text{ where } (\sigma, \sigma') \in \\llbracket \beta \rrbracket\} \).
- \( \nu((\ell, \sigma)) = \ell \land \bigwedge_{v \in \text{vars}} v = \sigma(v) \). 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.

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

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

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

- \( W = \text{locs}(\alpha) \times \{S\}, I = \{\ell_i, \sigma : \ell_i \in \iota(\alpha)\} \).
- \( \sqcap = \{((\ell, \sigma), (\ell', \sigma')) : \text{for } (\ell, \beta, \ell') \in \epsilon(\alpha) \text{ where } (\sigma, \sigma') \in \\llbracket \beta \rrbracket\} \).
- \( \nu((\ell, \sigma)) = \ell \land \bigwedge_{v \in \text{vars}} v = \sigma(v) \). 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.

\[ \ell_0: \quad i := N; \]
\[ \ell_1: \quad \text{while}(0 \leq x < N) \{ \]
\[ \ell_2: \quad i := i - 1; \]
\[ \ell_3: \quad x := x + 1; \]
\[ \ell_4: \} \]

We obtain the \( \epsilon \) transition relation according to Definition 1 below. Notice that the construction technically calls for another state after \( \ell_2 \), which transitions to \( \ell_3 \) on \( ? \top \). This is not necessary, and is only specified in Definition 1 to make the formalisation easier to understand. We omit it in the diagram below to keep the relation concise.

3 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 \( K_\alpha \) at \( \ell_0 \) where \( 0 \leq i \). Likewise, \( \langle \ell_0, 0 \rangle \) corresponds to any state at \( \ell_0 \) where \( 0 > i \). An abstract state labeled \( \langle \ell_0, \{0, 0 \leq i\} \rangle \) corresponds to any state at \( \ell_0 \) satisfying \( 0 > i \land 0 \leq i \), 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 \( K_\alpha \) 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 2.** Given a set of predicates \( A \in \hat{\Sigma} \), let \( \gamma(A) \) be the set of program states \( \sigma \in S \) that satisfy the conjunction of predicates in \( A \):

\[
\gamma(A) = \{ \sigma \in S : \sigma \models \bigwedge_{a \in A} a \}
\]

**Definition 3 (Abstract Transition Structure).** Given a program \( \alpha \), a set of abstract atomic predicates \( \Sigma \), and control flow transition relation \( \epsilon(\alpha) \) (Def. 1), let \( L \) be a set of locations given by the inductively-defined function \( \text{locs}(\alpha) \), \( i(\alpha) \) be the initial locations of \( \alpha \), and \( \kappa(\alpha) \) be the final locations of \( \alpha \) as given in Definition 1. The abstract transition structure \( K_\alpha = (W, I, \gamma, \hat{\nu}) \) is a tuple containing:

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

**Theorem 4.** For any trace \( \langle \ell_0, \sigma_0 \rangle, \langle \ell_1, \sigma_1 \rangle, \ldots \) of \( K_\alpha \), there exists a corresponding trace of \( \hat{K}_\alpha \langle \ell_0, A_0 \rangle, \langle \ell_1, A_1 \rangle, \ldots \) such that for all \( i \geq 0, \ell_i = \ell_i \) 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 \( K_\alpha \).

**Length=1:** By Definition 1, the trace is \( \langle \ell_0, \sigma_0 \rangle \) where \( \ell_0 \in i(\alpha) \). Then let \( A \) be such that \( \sigma_0 \in \gamma(A) \); we know that such an \( A \) exists, because \( \varphi(\Sigma) \) covers the entire state space \( S \). 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 \( K_\alpha \).

**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 \( K_\alpha \). By the inductive hypothesis, there must exist a trace \( \langle \ell_0, A_0 \rangle, \ldots, \langle \ell_n, A_n \rangle \) 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 \( \langle \ell_n, \sigma_n \rangle \cap (\ell_{n+1}, \sigma_{n+1}) \), we know that there exists \( \langle \ell_n, \beta, \ell_{n+1} \rangle \in \epsilon(\alpha) \) where \( (\sigma_n, \sigma_{n+1}) \in [\beta] \). Then by Definition 3, it must be that \( \langle \ell_n, A_n \rangle \hat{\gamma} (\ell_{n+1}, A_{n+1}) \). So \( \langle \ell_0, A_0 \rangle, \ldots, \langle \ell_{n+1}, A_{n+1} \rangle \) 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 4 tells us that $\hat{K}_\alpha$ can be used to deduce properties about $K_\alpha$: any trace in $K_\alpha$ is also in $\hat{K}_\alpha$, so any property of $K_\alpha$ is also one of $\hat{K}_\alpha$. However, Theorem 4 also tells us that $\hat{K}_\alpha$ overapproximates $K_\alpha$, so some properties of $\hat{K}_\alpha$ may not be properties of $K_\alpha$.

Definition 3 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 $|\text{locs}(\alpha)| \times 2^{|\hat{\Sigma}|}$ 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 $\sigma, \sigma'$ to satisfy the condition on $\wedge$ is not feasible.

When deciding whether to add a transition to $\hat{K}_\alpha$, we only care about the existence of $\sigma, \sigma'$ that satisfy the requirements of Definition 3. It is thus sufficient for our purposes to determine whether there are any $\sigma' \in \gamma(A')$ that are reachable from executing $\beta$ 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 $\beta$. Note that this is exactly the same as determining the validity of $\bigwedge_{a \in A} a \rightarrow [\beta] \bigvee_{b \in B} \neg b$.

**Theorem 5.** Let $A, B \subseteq \hat{\Sigma}$ be sets of predicates over program states, and $\beta$ 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 $\bigwedge_{a \in A} a \rightarrow [\beta] \bigvee_{b \in B} \neg b$ is not valid.

**Proof.** First we prove that $\bigwedge_{a \in A} a \rightarrow [\beta] \bigvee_{b \in B} \neg b$ not valid implies that $\exists \sigma, \sigma'. \sigma \in \gamma(A) \land \sigma' \in \gamma(B) \land (\sigma, \sigma') \in [\beta]$. 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 $\beta$ 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 $\sigma, \sigma'$ where $\sigma \in \gamma(A) \land \sigma' \in \gamma(B) \land (\sigma, \sigma') \in [\beta]$, then $\bigwedge_{a \in A} a \rightarrow [\beta] \bigvee_{b \in B} \neg b$ 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 $\sigma \models \bigwedge_{a \in A} a$ reach a final state in $\bigvee_{b \in B} \neg b$ after running $\beta$, which finishes the proof in this direction. \hfill $\square$

Theorem 5 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 1, 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 $[\cdot]=\cdot$ or $[\cdot]$, the DL formula stipulated in Theorem 5 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.

- \( \langle \ell_0, 0 > i \rangle \bowtie \langle \ell_1, 0 > i \rangle \): The program between \( \ell_0 \) and \( \ell_1 \) is \( i := N \). By Theorem 5, we must decide the validity of \( 0 > i \rightarrow [i := N]0 \leq i \). By \([=]\), we can reduce this to \( 0 > i \rightarrow 0 \leq N \), which is not valid: it is falsified by setting \( i = -1, N = 0 \). So this edge is added to \( \bowtie \).

- \( \langle \ell_2, 0 > i \rangle \bowtie \langle \ell_3, 0 > i \rangle \): The program between \( \ell_2 \) and \( \ell_3 \) is \( i := i - 1 \). By Theorem 5, we must decide the validity of \( 0 > i \rightarrow [i := i - 1]0 > i \). By \([=]\), we can reduce this to \( 0 > i \rightarrow 0 > i - 1 \), which is valid. So this edge is not added to \( \bowtie \).

- \( \langle \ell_1, 0 > i \rangle \bowtie \langle \ell_4, 0 > i \rangle \): The program between \( \ell_1 \) and \( \ell_4 \) is \( \neg (0 \leq x < N) \). By Theorem 5, we must decide the validity of \( 0 > i \rightarrow [\neg (0 \leq x < N)0 \leq i \). By \([?]\), we can reduce this to \( 0 > i \land \neg (0 \leq x < N) \rightarrow 0 \leq i \), which is not valid: it is falsified by \( i = -1, x = 0 \). So this edge is added to \( \bowtie \).

- \( \langle \ell_0, 0 > i \rangle \bowtie \langle \ell_1, 0 \leq i \rangle \): The program between \( \ell_0 \) and \( \ell_1 \) is \( i := N \). By Theorem 5, we must decide the validity of \( 0 > i \rightarrow [i := N]0 \leq i \). By \([=]\), we can reduce this to \( 0 > i \rightarrow 0 > N \), which is not valid: it is falsified by setting \( i = -1, N = -1 \). So this edge is added to \( \bowtie \).

- \( \langle \ell_0, 0 \leq i \rangle \bowtie \langle \ell_2, 0 > i \rangle \): The program between \( \ell_0 \) and \( \ell_2 \) is \( ? 0 \leq x < N \). By Theorem 5, we must decide the validity of \( 0 \leq i \rightarrow [? 0 \leq x < N]0 \leq i \). By \([?]\), we can reduce this to \( 0 \leq i \land 0 \leq x < N \rightarrow 0 \leq i \), which is not valid: it is falsified by setting \( x = 0, i = -1 \). So this edge is added to \( \bowtie \).

Now suppose that we want to verify the property from before when we discussed bounded model checking: \( \square \ell_4 \rightarrow 0 \leq i \). Notice from what we just worked out above that there is a counterexample path in \( \hat{K}_\alpha \):

\[
\langle \ell_0, 0 > i \rangle \bowtie \langle \ell_1, 0 > i \rangle \bowtie \langle \ell_4, 0 > i \rangle
\]

Because \( \hat{K}_\alpha \) overapproximates the true transition structure \( K_\alpha \), we need to determine whether this does in fact correspond to a path in \( K_\alpha \), 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 \( K_\alpha \) does not violate the safety property. Equivalently, it would mean the following formula must be valid:

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

Applying \([=]\),\([?]\),\([:=]\), the formula above reduces to \( 0 \geq i \rightarrow \neg (0 \leq x < N) \rightarrow 0 \leq N \). 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.

$\ell_0$: $i := \text{abs}(N)+1;$
$\ell_1$: while ($0 \leq x < N$) {
$\ell_2$: $i := i - 1;$
$\ell_3$: $x := x + 1;$
$\ell_4$: }

Now the counterexample from before no longer works, because there is no edge from $(\ell_0, 0 > i) \vdash (\ell_1, 0 > i)$. To see why, observe that from Theorem 5 we reason:

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

is valid.

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

1. $(\ell_0, 0 \leq i) \vdash (\ell_1, 0 \leq i)$. This edge is in $K_\alpha$ because $0 \leq i \rightarrow [i := \text{abs}(N) + 1]0 > i$ is equivalent to $0 \leq i \rightarrow 0 > \text{abs}(N) + 1$, which is not valid.

2. $(\ell_1, 0 \leq i) \vdash (\ell_2, 0 \leq i)$. This edge exists because $0 \leq i \rightarrow [?0 \leq x < N]0 > i$ is equivalent to $0 \leq i \rightarrow 0 \leq x < N \rightarrow 0 > i$, which is not valid.

3. $(\ell_2, 0 \leq i) \vdash (\ell_3, 0 > i)$. This edge exists because $0 \leq i \rightarrow [i := i - 1]0 \leq i$ is equivalent to $0 \leq i \rightarrow 0 \leq i - 1$ and is not valid, seen from the assignment $i = 0$.

4. $(\ell_3, 0 > i) \vdash (\ell_4, 0 > i)$. This edge exists because $0 > i \rightarrow [x := x + 1]0 \leq i$ is equivalent to $0 > i \rightarrow 0 \leq i$, which is not valid.

5. $(\ell_4, 0 > i) \vdash (\ell_4, 0 > i)$. This edge exists because $0 > i \rightarrow [? \neg(0 \leq x < N)]0 \leq i$ is equivalent to $0 > i \rightarrow \neg(0 \leq x < N) \rightarrow 0 \leq i$ and is not valid.

At this point, $K_\alpha$ is in a state satisfying $\ell_4 \land \neg(0 \leq i)$. 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 $\ell_0, \ell_1, \ell_2, \ell_3, \ell_4$, ending in a state where $0 > i$. This leads us to ask whether the following DL formula is valid:

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

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

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

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 $\square \ell_4 \rightarrow 0 \leq i$. So this counterexample is spurious: it exists in the abstraction $K_\alpha$, but not in the true transition system $K_\alpha$ corresponding to the program.
4 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 $K_\alpha$ 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 $K_\alpha$. 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 conclude 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 $i = \text{abs}(N) + 1$. 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. $\langle \ell_0, 0 \leq i \rangle \land \langle \ell_1, 0 \leq i \land i = \text{abs}(N) + 1 \rangle$. This edge is in $\hat{K}_\alpha$ because $0 \leq i \rightarrow [i := \text{abs}(N) + 1] \land (0 \leq i \land i = \text{abs}(N) + 1)$ is equivalent to $0 \leq i \rightarrow \neg(0 \leq \text{abs}(N) + 1) \land \text{abs}(N) + 1 = \text{abs}(N) + 1$, which is not valid.

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

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

4. $\langle \ell_3, 0 \leq i \rangle \land \langle \ell_1, 0 \leq i \rangle$. 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 $\square \ell_4 \rightarrow 0 \leq i$, we need to find a set of predicates that characterizes the relationship between $i, x,$ and $N$. Consider the following:

- $a_0 \equiv i \geq |N| - |x|$
- $a_1 \equiv i > |N| - |x|$
- $a_2 \equiv 0 \leq x \leq N$
- $a_3 \equiv 0 \leq x < N$

Using these predicates, we can reason as follows.

- The only transition from an $\ell_0$ state to an $\ell_1$ state also contains predicate $a_1$ in the $\ell_1$ state. We see that $true \rightarrow [i := \text{abs}(N)+1] \neg a_1$ 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 $[i := \text{abs}(N)+1] \neg (a_1 \land A') \leftrightarrow [i := \text{abs}(N)+1] a_1 \lor \neg A'$, which is valid.

- The only transition from an $\ell_1$ state to an $\ell_2$ state is one in which $a_2$ holds, and if $a_0$ or $a_1$ held previously in the $\ell_1$ state, then they will also hold in the $\ell_2$ state. This is obvious because the test $0 \leq x < N$ does not change the value of $i$ or $N$.

- Any transition from an $\ell_2$ state where $a_1$ holds will land in an $\ell_3$ state where $a_0$ holds. We see that $a_1 \rightarrow [i := i - 1] \neg a_0$ is equivalent to $i > |N| - |x| \rightarrow i - 1 < |N| - |x|$, which is not valid. Furthermore, $a_0$ must hold in the post-state, because $a_1 \rightarrow [i := i - 1] a_0$ is equivalent to $i > |N| - |x| \rightarrow i - 1 \geq |N| - |x|$, which is valid.

- Any transition from $\ell_3$ to $\ell_1$ where $a_0$ holds in $\ell_3$ will result in $a_1$ holding in $\ell_1$. We have $a_0 \rightarrow [x := x + 1] \neg a_1$ is equivalent to $i \geq |N| - |x| \rightarrow i \leq |N| - |x + 1|$ which is not valid.

- Any transition from $\ell_3$ to $\ell_1$ where $a_3$ holds in $\ell_3$ will result in either $a_2$ or $a_3$ holding in $\ell_1$.

- Any transition from $\ell_1$ to $\ell_4$ where $a_1$ and $a_2$ hold in $\ell_1$ will result in $a_0$ and $a_2$ at $\ell_4$. We see that $a_1 \land a_2 \rightarrow [?] \neg (0 \leq x < N) \land a_0 \lor \neg a_2$ is equivalent to $i > |N| - |x| \land 0 \leq x \leq N \rightarrow \neg (0 \leq x < N) \rightarrow i < |N| - |x| \lor \neg (0 \leq x \leq N)$ is not valid.

- Any state where $a_0$ and $a_2$ hold must also be one where $0 \leq i$ holds, because $i \geq |N| - |x| \land 0 \leq x \leq N \rightarrow 0 \leq i$ is valid.

From this reasoning, we see that all reachable $\ell_1$ states have $a_1 \equiv i > |N| - |x|$ and either $a_2$ or $a_3$. The only reachable $\ell_4$ states must go through $\ell_1 \land a_1 \land a_2$, and so must have $a_0 \land a_2$, which combined imply $0 \leq i$. Thus, there are no counterexample traces in the abstraction. Because $K_\alpha$ overapproximates $K_\alpha$, we know that any trace of $K_\alpha$ is also one of $K_\alpha$. We can then conclude that there are no counterexamples in $K_\alpha$, 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.