15-414: Bug Catching: Automated Program Verification

Lecture Notes on CEGAR & Craig Interpolation

Matt Fredrikson André Platzer

Carnegie Mellon University

Lecture 20

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: Predicate abstraction

Definition 1. 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 2 (Abstract Transition Structure)**.** Given a program α, a set of abstract atomic predicates $\hat{\Sigma}$, and control flow transition relation $\epsilon(\alpha)$, 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. 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.
- $\hat{\curvearrowright} = \{(\langle \ell, A\rangle, \langle \ell', A'\rangle$: for $(\ell, \beta, \ell') \in \epsilon(\alpha)$ where there exist σ, σ' such that $\sigma \in \gamma(A), \sigma' \in \gamma(A')$ 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.

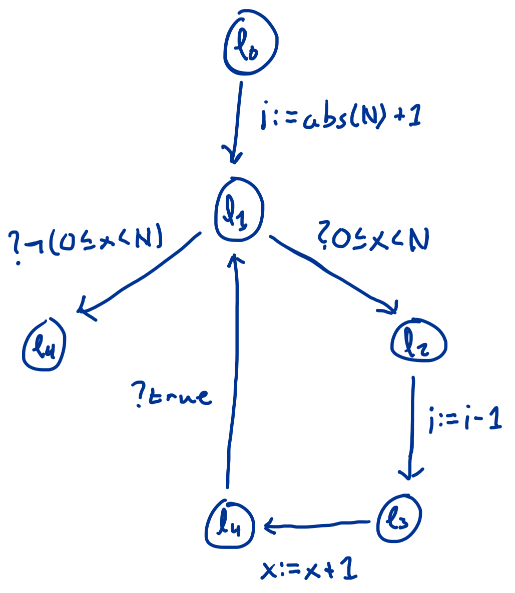
Theorem 3. *For any trace $\langle \ell_0, \sigma_0\rangle, \langle \ell_1, \sigma_1\rangle, \ldots$ of K_α, there exists a corresponding trace of $\hat{K}_\alpha$ $\langle \hat{\ell}_0, A_0\rangle, \langle \hat{\ell}_1, A_1\rangle, \ldots$ such that for all $i \geq 0$, $\ell_i = \hat{\ell}_i$ and $\sigma_i \in \gamma(A_i)$.*

Theorem 4. *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.*

Spurious counterexamples We looked at a modification of the earlier example from bounded model checking.

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

This yields the following control flow transitions.



Consider the following counterexample on the predicate set $\hat{\Sigma} = \{0 \le i\}$.

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

At this point, $\hat{K_\alpha}$ is in a state satisfying $\ell_4 \land \neg(0 \le i)$. As before, we need to determine whether this counterexample is spurious. We consider a path which starts in a state where $0 \le i$, 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:

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

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

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

The validity of this formula tells us that executing the statements in this counterexample will necessarily lead to a program state where $0 \le i$, which does not violate the property $\Box \ell_4 \to 0 \le i$. 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.

3 Automatic Refinement

We'll see how to refine abstractions automatically using information derived from spurious counterexamples. While the refinements computed by this approach may not necessarily be strong enough to verify the original program, they are strong enough to preclude the counterexample that they are derived from. In other words, after refinement, no trace corresponding to the particular spurious counterexample we observed will be in the abstraction. But the abstraction will still be an overapproximation, and subsequent searches may identify more spurious counterexamples.

Craig interpolation Let $P \to Q$ be a valid first-order formula. A *Craig interpolant* for P and Q is a formula C that satisfies the following conditions:

- $P \to C$ and $C \to Q$ are valid.
- All of the variables in C also appear in both P and Q.

Intuitively, a Craig interpolant for P and Q is a fact about the variables shared by P and Q that is "explained" by P, and in turn explains Q. Theorem 5 states that Craig interpolants always exist, and in fact, there are often many interpolants for a given pair of formulas P, Q.

Theorem 5 (Craig's Interpolation Theorem (1957) [Cra57])**.** *Let $P \to Q$ be a valid first order formula, where P and Q share at least one variable symbol. Then there exists a formula C containing only variable symbols in both P and Q such that $P \to C$ and $C \to Q$.*

We present a proof of Theorem 5 for the restricted case of propositional logic, i.e., P and Q only contain propositional literals.

Proof. In the following, let $\mathrm{Atoms}(P)$ be the set of atomic propositions (i.e., variables) in P.

First, notice that if $\mathrm{Atoms}(P) \cap \mathrm{Atoms}(Q) = \emptyset$, then either $\neg P$ is valid or Q is valid. In either case, any formula will work as an interpolant. So assume that $\mathrm{Atoms}(P) \cap \mathrm{Atoms}(Q) \neq \emptyset$, and proceed by induction on the size of the set $\mathrm{Atoms}(P) \setminus \mathrm{Atoms}(Q)$.

Base case: In the base case $\mathrm{Atoms}(P) \setminus \mathrm{Atoms}(Q) = \emptyset$. Then P is an interpolant because $P \to P$ is valid, and $P \to Q$ is valid by assumption.

Inductive case: Let $D \in \mathrm{Atoms}(P) \setminus \mathrm{Atoms}(Q)$. Then define $P_{D\mapsto\top}$ to be the same as P, but with all instances of D replaced with $\top$, and $P_{D\mapsto\bot}$ be the same as P with all instances of D replaced with $\bot$. Then $|\mathrm{Atoms}(P_{D\mapsto\top} \vee P_{D\mapsto\bot}) \setminus \mathrm{Atoms}(Q)| = |\mathrm{Atoms}(P) \setminus \mathrm{Atoms}(Q)| - 1$, so by the inductive hypothesis $P_{D\mapsto\top} \vee P_{D\mapsto\bot} \to Q$ has an interpolant C. Because $P_{D\mapsto\top} \vee P_{D\mapsto\bot} \equiv P$, C is also an interpolant for $P \to D$.

□

The proof given above for Theorem 5 is constructive: it tells us how to find a Craig interpolant for $P \to Q$. However, because the size of this interpolant may be exponential due to the splitting of P into disjunctive cases at each step, this interpolant may not be suitable for use in practice for predicate abstraction. There exist similar constructive proofs for the more general case of propositional logic, and rather than increasing the size of the formula exponentially they may introduce quantifiers. Because we would like to use automated decision procedures to answer queries about our abstractions, interpolants with quantifiers are also not always useful in practice. Automated techniques for finding concise, quantifier-free interpolants is a challenge, and remains an active research topic in the verification literature.

Note that it is equivalent to define a Craig interpolant for P, Q as a formula C that is implied by P ($P \to C$), inconsistent with Q ($C \land Q$ is unsatisfiable), and contains only shared symbols of P, Q. This formulation is more commonly used in the model checking literature, and we will use it from this point forward.

Examples Let's look at a few examples of Craig interpolants before seeing how they can help with refinement. Consider P, Q, where:

$$\begin{aligned} P &\equiv \neg(A \land B) \to (\neg D \land B) \\ Q &\equiv (E \to A) \land (E \to \neg D) \end{aligned}$$

We can find an interpolant in this case by applying the procedure outlined in the proof of Theorem 5 for propositional logic. The only variable in $\text{Atoms}(P) \setminus \text{Atoms}(Q)$ is B. So splitting P on B we obtain:

$$\begin{aligned} & (\neg(A \land \top) \to (\neg D \land \top)) \lor (\neg(A \land \bot) \to (\neg D \land \bot)) \\ \leftrightarrow \quad & (\neg A \to \neg D) \\ \leftrightarrow \quad & A \lor \neg D \end{aligned}$$

Now consider the formula P, Q, where the literals are no longer propositional variables.

$$\begin{aligned} P &\equiv (c \neq 0 \lor d = 0) \land c = 0 \\ Q &\equiv d = 0 \end{aligned}$$

One possible Craig interpolant is $d = 0$, which we see by applying the same approach as in the previous example but this time treating the literals $c \neq 0$, $d = 0$, $c = 0$ as we did propositional variables previously. To save steps, we treat $c = 0$ and $c \neq 0$ as one atom, non-negated in the first case and negated in the latter.

$$\begin{aligned} & ((\neg\top \lor d = 0) \land \top) \lor ((\neg\bot \lor d = 0) \land \bot) \\ \leftrightarrow \quad & d = 0 \end{aligned}$$

Another possibile interpolant is $d \leq 0$. Both interpolants describe a fact implied by P sufficient to prove Q. Note that because the interpolant only contains variables in both formulas, it does not reflect anything about P that is irrelevent to Q. It is a concise accounting of why P implies Q, which will be useful for our purposes.

In the previous example, the second interpolant $d \leq 0$ could not be derived using the technique from the proof because $d \leq 0$ is not an atom in either formula P, Q. The interpolant doesn't necessarily need to mention the literals from the original formula, and in some cases there is no atom shared between P and Q. Consider the following P, Q:

$$\begin{aligned} P &\equiv x = c_0 \land c_1 = c_0 + 1 \land y = c_1 \\ Q &\equiv x = m \land y = m + 1 \end{aligned}$$

An interpolant in this case is $y = x + 1$. Although there are automated techniques for producing such literals for formulas in most first-order theories relevant to verification [HJMM04, BKRW10], we will not cover them in this course.

Refinements from interpolants Let us return to the example from previous lecture. Recall that we found a spurious counterexample for the property $\Box \ell_4 \to 0 \le i$ using the abstraction set $\hat{\Sigma} = \{0 \le i\}$.

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

We know that this is a spurious counterexample because we can construct a formula that is valid if and only if the path corresponding to the counterexample satisfies the safety property, which in this case is that $0 \le i$ at the end of the path. Namely, the following is valid:

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

Consider the following incomplete deduction:

$${\scriptstyle [;],[:=]_=}\dfrac{{\scriptstyle [?],\to R}\dfrac{{\scriptstyle [:=]_=,[:=]_=}\dfrac{{\scriptstyle [?]}\dfrac{i_0 = |N| + 1, 0 \le x < N, i_1 = i_0 - 1, x_1 = x + 1, 0 \le x_1 < N \vdash 0 \le i_1}{i_0 = |N| + 1, 0 \le x < N, i_1 = i_0 - 1, x_1 = x + 1 \vdash [?\neg(0 \le x_1 < N)] 0 \le i_1}}{i_0 = |N| + 1, 0 \le x < N \vdash [i := i_0 - 1][x := x + 1][?\neg(0 \le x < N)] 0 \le i}}{i_0 = |N| + 1 \vdash [?0 \le x < N][i := i_0 - 1][x := x + 1][?\neg(0 \le x < N)] 0 \le i}}{\vdash [i := \mathtt{abs}(N) + 1; ?0 \le x < N; i := i - 1; x := x + 1; ?\neg(0 \le x < N)] 0 \le i}$$

The last sequent is valid if and only if the following formula is unsatisfiable:

$$i_0 = |N| + 1 \land 0 \le x < N \land i_1 = i_0 - 1 \land x_1 = x + 1 \land 0 \le x_1 < N \land 0 > i_1$$

This is called the *path formula* for the counterexample. Each conjunct on the left of the implication corresponds to a constraint imposed by executing one of the statements on the path. Assignments introduce equalities, and tests introduce direct assertions; updates to variables are accounted for by indexing variable names in the manner of static single-assignment (SSA) form. The final conjunct is the negated safety property

Now consider the interpolant corresponding to:

$$\begin{aligned} P &\equiv i_0 = |N| + 1 \\ Q &\equiv 0 \le x < N \land i_1 = i_0 - 1 \land x_1 = x + 1 \land 0 \le x_1 < N \land 0 > i_1 \end{aligned}$$

Such an interpolant is a fact about the program state immediately after the assignment $i := \texttt{abs}(N) + 1$ that must hold after executing the assignment. One such fact is $i_0 = |N| + 1$ (i.e., just P). But another is $i > |N| - x$. We can do this for each step of the counterexample, deriving interpolants along the way to learn useful facts.

- $P \equiv i_0 = |N| + 1$,
  $Q \equiv 0 \leq x < N \land i_1 = i_0 - 1 \land x_1 = x + 1 \land 0 \leq x_1 < N \land 0 > i_1$
  $C \equiv i_0 > 0$
- $P \equiv i_0 = |N| + 1 \land 0 \leq x < N$,
  $Q \equiv i_1 = i_0 - 1 \land x_1 = x + 1 \land 0 \leq x_1 < N \land 0 > i_1$,
  $C \equiv i_0 > |N| - x \land 0 \leq x < N$
- $P \equiv i_0 = |N| + 1 \land 0 \leq x < N \land i_1 = i_0 - 1$,
  $Q \equiv x_1 = x + 1 \land 0 \leq x_1 < N \land 0 > i_1$,
  $C \equiv i_1 \geq |N| - x \land 0 \leq x < N$
- $P \equiv i_0 = |N| + 1 \land 0 \leq x < N \land i_1 = i_0 - 1 \land x_1 = x + 1$,
  $Q \equiv 0 \leq x_1 < N \land 0 > i_1$,
  $C \equiv i_1 > |N| - x_1 \land 0 \leq x_1 \leq N$
- $P \equiv i_0 = |N| + 1 \land 0 \leq x < N \land i_1 = i_0 - 1 \land x_1 = x + 1 \land 0 \leq x_1 < N$,
  $Q \equiv 0 > i_1$,
  $C \equiv 0 \leq i_1$

We can obtain a set of predicates to refine our abstraction with by dropping the subscripts from each interpolant above. We would then be left with,

$$\hat{\Sigma} = \{0 \leq i, i > 0, i > |N| - x, i \geq |N| - x, 0 \leq x < N, 0 \leq x \leq N\}$$

Using these predicates will ensure that we do not encounter the same spurious counterexample again in future attempts at model checking. To see why, notice the following sequence of observations.

1. After executing the first assignment $i := \texttt{abs}(N) + 1$, we have that $i > 0$ must hold. In other words, $[i := \texttt{abs}(N) + 1]i > 0 \leftrightarrow |N| + 1 > 0$ is valid.
2. After executing the test $?0 \leq x < N$ (i.e., entering the loop) starting in a state where $i > 0$ holds, $i > |N| - x \land 0 \leq x < N$ must hold. This follows from the validity of $(i > 0 \rightarrow [?0 \leq x < N]i > |N| - x \land 0 \leq x < N) \leftrightarrow (i > 0 \land 0 \leq x < N \rightarrow i > |N| - |x| \land 0 \leq x < N)$.
3. Starting in a state where $i > |N| - x \land 0 \leq x < N$ holds and executing the assignment $i := i - 1$ lands in a state where $i \geq |N| - x \land 0 \leq x < N$ holds. This follows from the validity of $(i > |N| - x \land 0 \leq i < N \rightarrow [i := i-1]i \geq |N| - x \land 0 \leq x < N) \leftrightarrow (i > |N| - x \land 0 \leq x < N \rightarrow i - 1 \geq |N| - x \land 0 \leq x < N)$.

4. Starting in a state where $i \geq |N| - x \wedge 0 \leq x < N$ holds and executing the assignment $x := x + 1$ yields a state where $i > |N| - x \wedge 0 \leq x \leq N$ holds. This follows from the validity of $(i \geq |N| - x \wedge 0 \leq x < N \rightarrow [x := x + 1]i > |N| - x \wedge 0 \leq x \leq N) \leftrightarrow (i \geq |N| - x \wedge 0 < x < N \rightarrow i > |N| - (x+1) \wedge 0 \leq x + 1 \leq N)$.

5. Starting in a state where $i > |N| - x \wedge 0 \leq x \leq N$ holds and executing the test $?\top$ (i.e., going back to the top of the loop) leads to a state where $i > 0$ necessarily holds. This follows from the validity of $(i > |N| - x \wedge 0 \leq x \leq N \rightarrow [?\top]i > 0) \leftrightarrow (i > |N| - x \wedge 0 \leq x \leq N \rightarrow i > 0)$.

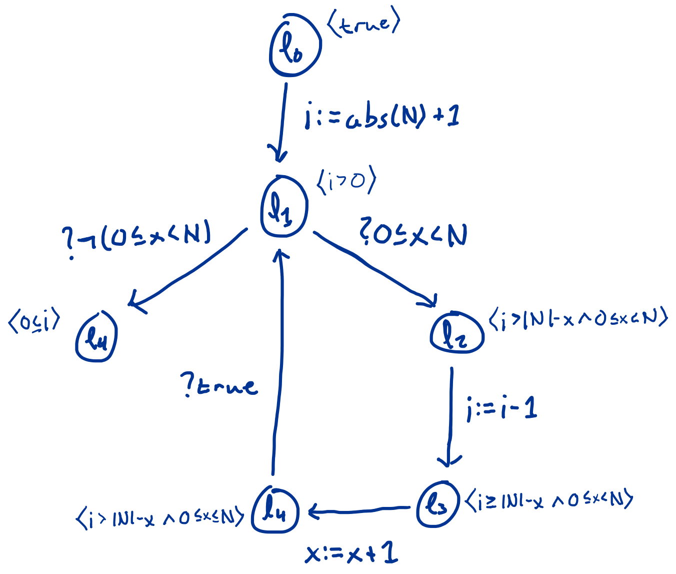
6. Starting in a state where $i > 0$ holds and executing the test $?\neg(0 \leq x < N)$ (i.e., exiting the loop) leads to a state where $0 \leq i$ necessarily holds. This follows from the validity of $(i > 0 \rightarrow [?\neg(0 \leq x < N)]0 \leq i) \leftrightarrow (i > 0 \wedge \neg(0 \leq x < N) \rightarrow 0 \leq i)$.

Most automated techniques for deriving interpolants from counterexample path formulas do so in a way that the interpolant for at each step through the path formula is sufficient to prove the interpolant at the following [HJMM04]. In other words, if C_i, C_{i+1} are the interpolants for steps i and $i+1$ of the counterexample, and α is the statement executed at that step, then $C_i \rightarrow [\alpha]C_{i+1}$. This guarantees that the counterexample used to refine the abstraction will no longer be a trace in the new transition structure.

Localizing abstraction The way in which we derived these predicates gives us more information yet. In particular, we derived each interpolant by splitting the counterexample path formula at all of the program locations along the path. So in addition to telling us which predicates are relevant to refining out the counterexample, this procedure also tells us where in the program they are relevant at. We can use this information to greatly reduce the statespace of the abstraction by localizing the set of abstraction predicates to each control flow location. So rather than using as the statespace of the abstraction $\hat{W} = \mathcal{S} \times \wp(\hat{\Sigma})$, we define $\hat{\Sigma}$ to be a function from control flow locations to sets of predicates:

$$\hat{\Sigma} : \mathit{locs}(\alpha) \rightarrow \wp(\text{predicates over program states}),$$
$$\text{where } \hat{\Sigma}(\ell) \text{ is the set of predicates interpolated at location } \ell$$

Applying this to our running example, we are left with the abstraction depicted below. Notice that there are no paths to a state where $0 > i$ holds, so we can perform exhaustive model checking to conclude that the safety property always holds.



This approach is called *lazy abstraction* [HJMS02], because the abstraction is constructed as needed and only in relevant locations as required by counterexamples. It is the standard approach used for unbounded software model checking, and has been implemented in numerous tools available today.

References

[BKRW10] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. An interpolating sequent calculus for quantifier-free presburger arithmetic. In *Proceedings of the 5th International Conference on Automated Reasoning*, 2010.

[Cra57] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symbolic Logic*, 22(3):269–285, 09 1957.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.