Semantic Foundations for the Static Analysis of Program Revisions

Dakota Bryan[1] and Bor-Yuh Evan Chang*[1,2]

[1]University of Colorado Boulder, USA, {dakota.bryan, evan.chang}@colorado.edu
[2]Amazon, USA

Abstract. Software development is a process of gradually modifying program behavior; thus, most programs are revisions of other programs. Developers informally reason about a revision in terms of which behaviors are removed, added, or preserved. We formalize these notions by providing a collecting semantics for program revisions, parameterized by a user-specified state correspondence, which classifies reachable states as "removed", "added", or "related". Intuitively, reasoning about added and removed states requires showing that some reachable state in one program has no corresponding state in the other. Therefore, proving safety properties over removed and added states requires an underapproximation of one program and an overapproximation of the other. Existing relational Hoare-style program logics provide limited support for reasoning about this class of properties. To address this gap, we present a system for proving safety properties of "added", "removed", and "related" states that combines over- and underapproximating program logics.

1 Introduction

During the software development process, programmers create pull requests (PRs) to modify the behavior of a system. These PRs are subsequently reviewed to determine whether the changes introduced to the code produce the intended consequences. Often in these reviews, the semantics of the change are described informally, and manual code review is used to check these informal semantics. Here, static program analysis tools can be used to formally verify properties of the revision. For these tools to be easily adopted, developers should be able to specify assertions in a way similar to their informal notions of revision semantics. One way to enable this is for these tools to prove properties about a representation of the revision, instead of a single program or a pair of programs. This requires a formal semantics for program revisions that captures developer intuitions, which is the first problem we address. Then, we develop a system for proving safety properties over our semantics, which serves as a foundation for the static analysis of program revisions.

* Bor-Yuh Evan Chang holds concurrent appointments at the University of Colorado Boulder and as an Amazon Scholar. This paper describes work performed at the University of Colorado Boulder and is not associated with Amazon.

During a PR, a revision may be described as adding a feature that has some semantics and not removing any behavior. We consider a simple imperative programming language (IMP) and represent program behavior with reachable states. With these semantics, an example of the informal revision description above is "in the updated program, the value of `result` can be `10` which wasn't possible in the old program"(semantics of a new feature), and "all values of `result` that were possible in the old program are still possible in the new program" (no removed behavior). For the rest of the paper, we use the term "new" to refer to the updated program in the revision and its reachable states, and "old" to refer to the original program and its reachable states.

To design a proof system for these properties, or use an existing one, we must first characterize them. They fall into a class of program properties that generalizes *refinement*. Refinement asserts that all possible input-output state pairs of one program are contained within those of another program. Even with our reachable state semantics, we retain the input-output style reasoning since final states contain all information from the input states. Thus, our "no removed behavior" property is refinement for a projection of the state. It asserts that the values of `result` in all reachable old states are contained within the possible values of `result` in all new states. Our "added behavior" assertion cannot be formulated as a refinement; instead, it is an assertion on all states of the new program which do not satisfy a refinement property. Refinement focuses on what *doesn't change* between two versions of a program, but we argue that it is useful to reason about what *does change*, as seen in our "added behavior" assertion.

The properties discussed above cannot be proven in systems that derive 2-safety properties such as relational Hoare logic [3]. BiKAT [1] provides a unified algebraic framework for reasoning about relational properties like ours in terms of *simulation*, and there is other work [21,5] on program logics for proving this style of property. In these existing works, revision-specific properties such as refinement are mentioned as applications of relational reasoning systems, but few systems are designed specifically for revisions. Most of the few revision-specific systems [11,10,2] generate automatic reports about program differences and equivalences. This does not leave space for developers to specify properties about the revision or influence the analysis in any way. Thus, we seek to develop a revision-specific proof system that leaves space for user insights and is expressive enough to capture the class of properties outlined above, which go beyond refinement and are relevant to revisions.

To this end, we provide a collecting semantics to enable reasoning about program revisions that interprets a revision in terms of "removed", "related", and "added" behavior (states). Our "no removed behavior" assertion would ensure that the revision produces no "removed" states, and our "new feature" assertion is a property of "added" states. An old program state is removed if it is not related to any new program state, and the way the framework determines when one state is related to another is user-specified, which we call the *state correspondence*. Developers may also want to reason about behavior which the revision did not change, which are "related" states in our framework. These are state

pairs (one from the old program, and one from the new) that are related under the state correspondence. In this collecting semantics and subsequent proof system, refinement is expressible by reasoning about removed and added states, and 2-safety is expressible by reasoning about related states. More properties are expressible in this system, and they are all framed in terms of a mental model for revisions which we believe many developers have.

To prove safety properties over the sets of removed, added, and related states defined by our collecting semantics, both over- and underapproximations of reachable states are required. In particular, reasoning about added states requires underapproximating the old program's reachable states, since an overapproximation may introduce spurious states that are related to new states, thereby *shrinking* the set of added states. To establish these approximations in our proof system, we use Hoare logic [9] and incorrectness logic [15].

Our main contributions can be summarized as follows:

- A collecting semantics for program revisions which categorizes reachable states as "removed", "related", or "added" up to a user-defined state correspondence.
- A sound proof system for deriving safety properties over our collecting semantics.
- A mechanized development of our collecting semantics and proof system in Lean.

2 Motivating Example

In this section, we present a simple program revision to understand how natural descriptions of revisions inform our collecting semantics and proof system.

```
if (x >= 0)
    sgn := 1
else
    sgn := −1
```

```
if (x >= 0)
    sgn := 1
else
    sgn := −1
if (x == 0)
    sgn := 0
```

Fig. 1. A revision to a sign calculation program that changes the sign of 0 from 1 to 0

Consider the two programs in figure 1 which together form a revision. For the remainder of the paper, we will only discuss programs written in a simple imperative language (IMP) with non deterministic assignment.

As we saw in the section 1, developers can view a program revision in terms of the behavior which it removed, added, and kept the same. For IMP programs, one natural characterization of program behavior is the set of reachable states, so we

will describe the semantic effect of a revision by characterizing reachable states of the two programs as removed, related, or added. For this approach, we could consider an old program state removed if it is not reachable by the new program. In this case, $\sigma_- = \mathtt{x}_- \mapsto 0, \mathtt{sgn}_- \mapsto 1$ would be the only removed state, and $\sigma_+ = \mathtt{x}_+ \mapsto 0, \mathtt{sgn}_+ \mapsto 0$ would be the only added state. This characterization represents program behavior in terms of how input values are transformed to output values, which is consistent with most previous work. However, a developer may view the effect of the revision as adding the functionality that the sign of a number can now be zero, in which case no behavior is removed. To accommodate this, we could consider an old program state removed if some slice of the state (only $\mathtt{sgn}_-$) is not reachable by the new program. Then, no states are removed, and $\sigma_+ = \mathtt{x}_+ \mapsto 0, \mathtt{sgn}_+ \mapsto 0$ is the only added state.

To allow for our collecting semantics to have this flexibility, we parameterized it by a user-supplied relation that determines when an old state is "equivalent", or *related* to a new state, which is the state correspondence (definition 1) and variable correspondence (definition 4). For behavior that the revision did not alter, we "pair up" a reachable old state with a reachable new state if they are related to each other under the state correspondence. This represents preserved behavior because the state correspondence represents equivalent behavior for how the user views the semantics of the programs. Thus, the "related" set in our collecting semantics will consist of state pairs, instead of individual states as in the "removed" and "added" sets.

Given these collecting semantics (definition 6), we provide a proof system for deriving safety properties over reachable removed, related, and added states in definition 12. Using this, users can reason about other aspects of the state(s) that led to, or result from, the particular state(s) being removed, added, or related. For example, all added states described above will have $\mathtt{x}_+ = 0$ and all related state pairs will have $\mathtt{x}$ values with the same sign.

3 Collecting Semantics

A static analysis for program revisions should allow users to reason about the change itself instead of the semantics of a single program or pair of programs. In this section, we formulate a collecting semantics for program revisions which serves as a foundation for analyses that treat revisions as the objects of study. We only consider simple imperative programs where program behavior is represented by reachable states.

We begin by presenting a grammar for the metavariables used in this section.

$$
\begin{aligned}
\text{revision} \quad & rs ::= s_- \pm s_+ \\
\text{statement} \quad & s ::= s_- \mid s_+ \\
\text{set of state pairs} \quad & \Sigma_2 ::= \Sigma_2 , \sigma_2 \mid \circ \\
\text{set of states} \quad & \Sigma_1 ::= \Sigma_1 , \sigma_1 \mid \circ \\
\text{state pair} \quad & \sigma_2 ::= (\sigma_- , \sigma_+) \\
\text{unary state} \quad & \sigma_1 ::= \sigma_- \mid \sigma_+ \\
\text{state} \quad & \sigma
\end{aligned}
$$

For our underlying programming language IMP, we use the metavariable s to represent a statement, σ to represent a state, and Σ for a set of states. We write $s_- \pm s_+$ for a program revision where s_- is the "old" program and s_+ is the "new" program in the revision. States and set of states will usually be associated with a program, so we follow the same convention where σ_- is a state from program s_- and Σ_- is a set of σ_- states. We will write σ_{rm} and σ_{add} for added and removed program states, respectively, and Σ_{rm} and Σ_{add} for sets of those states. When a state is non associated with a program, we write σ_1 to denote that it is a single state. For a pair of states, which unless otherwise noted will be an old state and a new state (σ_-, σ_+), we sometimes write σ_2 and Σ_2 for a set of them. Otherwise, when a pair of states is a "related" pair, we write σ_{rel} and Σ_{rel}. Thus, a set of single states and a set of state pairs can be disambiguated by noticing that Σ_2 and Σ_{rel} are the only cases where Σ represents a set of pairs instead of a set of states. For single states and state sets, one can tell what program it came from via the subscript. To differentiate between metavariables of the same type, we add a superscript prime, *e.g.* σ'_-.

3.1 Decomposing Reachable States via State Correspondence

As we saw in sections 1 and 2, one way to view a program revision is in terms of the behavior it removed, added, and kept the same. Thus, our collecting semantics will characterize reachable states of the "old" and "new" programs in the revision as "removed", "related", or "added". To form this decomposition, a relation that determines if two states are related to one another is required, and it should be user-specified to allow for various interpretations of the revision. This relation, which we call the *state correspondence*, is a parameter to our collecting semantics.

Definition 1 (State Correspondence). *A state correspondence is a binary relation states.*

$$\sim \in \Sigma_1 \times \Sigma_1$$

Here, Σ_1 is the set of all possible states. The state correspondence is not usually reflexive since when it is used, one of the states will always be from an old state set (Σ_-) and the other will always be from a new state set (Σ_+). We use infix notation where the left side is always an old program state.

$$\sigma_- \sim \sigma_+$$

Using the state correspondence, we now define what it means for a single state to be removed from a set of states.

Definition 2 (Removed Relation). *Given a state correspondence ($\sim$), we define the removed relation $\not\approx$.*

$$\frac{\textit{forall } \sigma'_1 \in \Sigma'_1.\ \sigma_1 \not\sim \sigma'_1}{\sigma_1 \not\approx \Sigma'_1}$$

$\sigma \not\sim \sigma'$ iff $\sigma \sim \sigma'$ does not hold. $\sigma_1 \not\approx \Sigma_1'$ holds whenever σ_1 is removed from Σ_1' up to $\sim$.

For a new state to be added, it must be removed from the set of reachable old states up to the state correspondence. Thus, we can use the judgment given above to define when a state is added. Based on definition 2, $\sigma_+ \not\approx \Sigma_-$ would place the new state on the left side of $\sim$, which is not how we defined the state correspondence. To avoid clutter, when we write $\sigma_+ \not\approx \Sigma_-$, it is assumed that $\sigma_- \not\sim \sigma_+$ will be checked for all old states.

Definition 3 (Decomposition via State Correspondence). *Given the set of reachable states of an old program (Σ_-) and a new program (Σ_+), and a state correspondence ($\sim$), we define the removed, added, and related sets of states.*

$$\Sigma_{\text{rm}} := \{\sigma_- \in \Sigma_- \mid \sigma_- \not\approx \Sigma_+ \}$$
$$\Sigma_{\text{rel}} := \{(\sigma_-, \sigma_+) \in \Sigma_- \times \Sigma_+ \mid \sigma_- \sim \sigma_+\}$$
$$\Sigma_{\text{add}} := \{\sigma_+ \in \Sigma_+ \mid \sigma_+ \not\approx \Sigma_- \}$$

Every reachable state for the new program (Σ_+) is added iff it is not related to any reachable old state (Σ_-) via the state correspondence. Thus, every reachable state from both programs ($\Sigma_- \uplus \Sigma_+$) will either be in Σ_{rm}, Σ_{add}, or one side of a pair in Σ_{rel}. Since pairs of states in Σ_{rel} are the pairs in the product of reachable states that are related under $\sim$, one state could appear in multiple pairs in Σ_{rel}.

To understand the essence of the reachable state decomposition from defintion 3 let's take a simple property, $\sigma_-(\mathtt{x}) > \sigma_+(\mathtt{x})$. This is a predicate on an old state and new state, so it naturally defines a state correspondence: $\sigma_- \sim \sigma_+$ iff $\sigma_-(\mathtt{x}) > \sigma_+(\mathtt{x})$. This state correspondence will generate a Σ_{rm} of old states whose value of `x` is *less than or equal to* every possible value of `x` in the new program states since $\not\sim$ holds when $\sigma_-(\mathtt{x}) \leq \sigma_+(\mathtt{x})$. Σ_{add} will contain states whose value of `x` is *greater than or equal* to all possible values of `x` in reachable old program states. All Σ_{rel} state pairs satisfy our state correspondence property.

This decomposition can be viewed as internalizing a property (the state correspondence) into the removed, related, and added set of states framework. In this way, it is a revision oriented abstraction of the cross product of reachable states. The state correspondence is as a relation on two states, so our decomposition can use any property that is about two reachable states to construct this abstraction. These properties are fairly expressive; if our reachable state sets are overapproximating, then these are 2-safety properties. One natural generalization is for the state correspondence to express properties about the two programs using k states for each program. For example, non-interference is a 2-safety property for *one* program. If the state correspondence was a relation on pairs of states from each program, we could express that $\sigma_-, \sigma_-' \sim \sigma_+, \sigma_+'$ holds iff both pairs satisfy non-interference. Then, if the revision removed all instances violating non-interference, Σ_{add} would be empty and Σ_{rm} would contain all old state pairs that do not have the non-interference property.

3.2 Revision-Specific Collecting Semantics

While generalizing the state correspondence is an interesting direction and useful intuition for how the state correspondence properties get lifted into our revision framework, we keep it as defined in definition 1, and further restrict it to the simplest form that aligns with how revisions are understood. This restriction allows only for the comparison of the values of variables via equality. This is quite natural, since our state decomposition lifts equality to a state inclusion check. For example, if we have $\sigma_- \sim \sigma_+$ iff $\sigma_-(\mathtt{x}) = \sigma_+(\mathtt{x})$, then $\sigma_- \not\sim \Sigma_+$ will hold iff $\sigma_-(\mathtt{x}) \notin \Sigma_+(\mathtt{x})$ where $\Sigma_+(\mathtt{x})$ is the set of all possible values of $\mathtt{x}$ in Σ_+. We provide an intuitive syntax for developers to express state correspondences of this from, which, taking inspiration from [16], we call the *variable correspondence*.

Definition 4 (Variable Correspondence).

$$vc ::= (\overrightarrow{x}, \overrightarrow{y})$$
$$\overrightarrow{x} \in \mathtt{Var}^* \text{ and } \overrightarrow{y} \in \mathtt{Var}^*$$

We can define the state correspondence induced by a variable correspondence.

$$\frac{vc = (\overrightarrow{x}, \overrightarrow{y}) \quad |\overrightarrow{x}| = |\overrightarrow{y}| \quad \textit{forall } i \in \{1, \ldots, |\overrightarrow{x}|\}.\ \sigma_1(\overrightarrow{x}_i) = \sigma_1'(\overrightarrow{y}_i)}{\sigma_1 \sim_{vc} \sigma_1'}$$

$\sigma_1(\overrightarrow{x}_i)$ is the value of the ith variable in variable sequence $\overrightarrow{x}$, in store σ_1. We lift $\sim_{vc}$ to $\not\sim_{vc}$ using definition 2. For the remainder of the paper, variable correspondence can refer to either the syntactic object or the state correspondence induced by the variable correspondence when the distinction is clear from context.

In order for $\sim_{vc}$ to hold, the sequences must have the same length, and the value sequence from σ_1's variable sequence must match that of σ_1''s. State correspondences of this form induce decompositions where states in Σ_{rm} and Σ_{add} are such that they have a sequence of variable values that is not reachable by the other program. This interpretation is distinct from [16] because they use the variable correspondence to compare pairs of individual states and consider specific regions of a single state as being removed or added.

We now turn our attention to using the variable correspondence in a revision-specific collecting semantics. While the decomposition of definition 3 captures the desired relational structure, a collecting semantics must also account for program execution. This will provide a concrete semantic foundation that a program logic with preconditions can be defined as an abstraction of. To achieve this, we use the semantics of the underlying programming language that our programs (s) are written in.

Definition 5 (IMP Collecting Semantics). *Given an imperative program, s, we represent its operational semantics as a relation on input states and output states,*

$$\sigma_1 \vdash s \Downarrow \sigma_1'$$

holds whenever s executed with σ_1 can result in σ_1'. We lift these semantics to sets.

$$\frac{\Sigma_1' := \{\sigma_1' \mid \sigma_1 \in \Sigma_1 \text{ and } \sigma_1 \vdash s \Downarrow \sigma_1'\}}{\Sigma_1 \vdash s \Downarrow^{\{\}} \Sigma_1'}$$

Sometimes, we use the denotational semantics of IMP, which is defined as follows,

$$(\sigma_1, \sigma_1') \in [\![s]\!] \text{ iff } \sigma_1 \vdash s \Downarrow \sigma_1'$$

In order for final states to have all information from pre states, we require that a well-founded program must not modify a variable that was not defined in the program.

Now, we can write a concrete collecting semantics for a revision, $s_- \pm s_+$, and a variable correspondence, vc. For the rest of the paper, we only consider restricted state correspondences that are induced by variable correspondences.

Definition 6 (Collecting Semantics). *Parameterized by a vc.*

$$\frac{\begin{array}{c}\Sigma_- \vdash s_- \Downarrow^{\{\}} \Sigma_-' \qquad \Sigma_+ \vdash s_+ \Downarrow^{\{\}} \Sigma_+' \\ \Sigma_{\text{rel}} := \{(\sigma_-, \sigma_+) \in \Sigma_-' \times \Sigma_+' \mid \sigma_- \sim_{vc} \sigma_+\} \\ \Sigma_{\text{rm}} := \{\sigma_- \in \Sigma_-' \mid \sigma_- \not\sim_{vc} \Sigma_+' \} \qquad \Sigma_{\text{add}} := \{\sigma_+ \in \Sigma_+' \mid \sigma_+ \not\sim_{vc} \Sigma_-' \}\end{array}}{\Sigma_-, \Sigma_+ \vdash s_- \pm s_+ \Downarrow_{vc} \Sigma_{\text{rm}}, \Sigma_{\text{rel}}, \Sigma_{\text{add}}}$$

We now demonstrate the expressivity of these collecting semantics by encoding two often-sought properties of revisions: refinement and 2-safety.

Example 1 (Refinement). Refinement properties can be expressed in our collecting semantics (definition 6) as follows.

$$\top, \top \vdash s_- \pm s_+ \Downarrow_{vc} \_, \_, \emptyset$$

Here $\top$ represents the set of all possible states, and $\_$ indicates that those state(s) sets are arbitrary, *i.e.* they are inconsequential for the refinement property.

Our framework naturally expresses refinement as, "nothing was added". If a program has an input variable, `in`, and an output variable, `out`, we can use the variable correspondence, $vc := ((\texttt{in}, \texttt{out}), (\texttt{in}, \texttt{out}))$, to express when the new version of the program refines the old with the formulation in example 1. Notice that keeping the collecting semantics relation the same but altering the variable correspondence expresses refinement for only certain projections of the state that a developer may care about. In section 4, we present a proof system for deriving safety properties over the collecting semantics, in which case refinement can be shown with a $\bot$ assertion on added states. If we change Σ_{add} to something other than $\emptyset$, we are expressing properties about the states which are "left out of the refinement", or are *added*. Then, in our overapproximating proof system, one can specify safety properties over these added states.

Example 2 (Relational Reasoning). Purely relational properties can be expressed in our collecting semantics as follows.

$$vc := () \qquad\qquad \top, \top \vdash s_- \pm s_+ \Downarrow_{vc} \_, \Sigma_{\text{rel}}, \_$$

In the example above, any two states will be related since the variable correspondence is empty, so Σ_{rel} will contain all reachable state pairs ($\Sigma'_- \times \Sigma'_+$). Thus, a 2-safety property can be proven by ensuring that it holds over Σ_{rel}, which is still sound if Σ_{rel} is overapproximated as in our proof system. This allows a relational Hoare logic triple with precondition φ_2 and postcondition φ'_2 to be expressed in our system by asserting that $\varphi_2 \Rightarrow \varphi'_2$ holds over Σ_{rel} with an empty vc.

4 Proving Revision Safety Properties

In this section, we consider the problem of proving *safety properties* over our collecting semantics. We show that our formulation of the collecting semantics yields safety properties over Σ_{rm} and Σ_{add} that are not expressible in existing relational program logics. Then, we present our proof system for soundly deriving safety properties over our collecting semantics which relies on valid Hoare and incorrectness logic triples to establish over and underapproximations of the reachable states.

We begin with a grammar for an assertion language so users can write assertions over states and state pairs.

$$\begin{aligned} \text{state predicates } \varphi ::= {} & \top \mid \bot \mid b \mid \varphi \wedge \varphi' \mid \varphi \vee \varphi' \mid \\ & \varphi \Rightarrow \varphi' \mid \neg\varphi \mid \forall x.\ \varphi \mid \exists x.\ \varphi \\ \text{boolean expressions } b & \\ \text{expressions } e & \\ \text{variables } x & \end{aligned}$$

Boolean expressions and expressions are as defined for the underlying programming language, IMP. Any lowercase greek letter except σ can be used for a state predicate. Similar to states, we subscript predicates with $-$ for predicates over old program (s_-) states. A generic single store predicate is φ_1, and we write φ_2 for a predicate over a state pair (σ_-, σ_+). Variables are subscripted with $-$ or $+$ to denote which state these variables should be looked up in. φ_- must only contain variables subscripted with $-$ to be well-founded. φ_{rm} and φ_{add} is used for predicates on removed and added states, respectively. φ_{rel} is used to denote a predicate on related state pairs (σ_-, σ_+).

We write, $\sigma_1 \models \varphi_1$ and $\sigma_2 \models \varphi_2$ to denote that a state or pair of states models a state(s) predicate. These are defined in the standard way, where we look up variables that appear free in φ_1 in the state σ_1, then rely on the semantics of boolean and logical connectives. Quantification in our assertion language is over the domain of the values in our programming language, which is the codomain of our states.

4.1 Safety Properties of the Collecting Semantics

We must taxonomize the safety properties over our collecting semantics in order to see if existing program logics can be used to derive them. First, let's consider related states, where φ_{rel} is a binary state predicate that we want to prove as a safety property over the states in Σ_{rel}. We must also have state predicates for our initial states, which we call φ_- and φ_+. Since φ_{rel} is a safety property, our proof theory must be sound with respect to weakening it, which means if φ'_{rel} is proven to be a safety property over Σ_{rel} and $\varphi'_{\text{rel}} \Rightarrow \varphi_{\text{rel}}$, then φ_{rel} is a safety property over Σ_{rel}. This is the same ensuring that for any postcondition that are proof theory could prove, φ_2, the set of states that satisfy it, Σ_2, overapproximates the concrete set of reachable states in our collecting semantics, $\Sigma_{\text{rel}} \subseteq \Sigma_2$. Notice that this is precisely the semantics of relational Hoare logic, so we can use that to prove our related states safety properties.

Now, to see the features of a safety property over Σ_{rm} or Σ_{add}, consider the following formula which always holds if φ_{rm} is a safety property of Σ_{rm}.

$$\begin{aligned} &\textit{forall } \sigma_-, \sigma_+, \sigma_-.\ (\sigma_-, \sigma'_-) \in [\![s_-]\!] \textit{ and } \sigma_- \models \varphi_- \textit{ and } \sigma_+ \models \varphi_+ \textit{ and} \\ &\quad (\textit{forall } \sigma'_+.\ (\sigma_+, \sigma'_+) \in [\![s_+]\!] \textit{ implies } \sigma'_- \not\sim_{vc} \sigma'_+) \textit{ implies } \sigma'_- \models \varphi_{\text{rm}} \end{aligned}$$

The formula is obtained by "unrolling" Σ_{rm} from definition 6 and using denotation semantics instead of collecting semantics for IMP. Notice that the guard to determine when an old state is checked with φ_{rm} is a *global* condition over all reachable new states, instead of a predicate on a single state or pair of states. This is distinct from, to the best of our knowledge, all existing relational Hoare-style logics that reason about two separate programs. Instead, in these logics, pre- and postconditions are evaluated on individual matched execution pairs that can only be obtained by various quantification structures over states to consider. In our validity definition above, we cannot determine when to check $\sigma_- \models \varphi_{\text{rm}}$ by only filtering with state(s) predicates and reachability, which is required for the property to be expressible in these existing logics. It should be noted that in classical, but not intuitionistic, logic we can rewrite our validity definition to fit this structure with quantifiers only at the outermost scope. We would universally quantify over old program states, exesitentially quantify over new program states, then assert $\sigma_- \sim_{vc} \sigma_+$ *or* $\sigma_- \models \varphi_{\text{rm}}$ on matched executions. Despite this potential rewrite, we decide to continue with the natural formulation of our removed property since the few program logics that support these $\forall\exists$ properties either support a limited form of nondeterminism [5] or do not have verification condition generators [1].

4.2 Proof System for Revision Safety Properties

Here, we develop a proof system for deriving safety properties of our collectings semantics given a variable correspondence and a revision. We specify this a triple, which we call a dassert triple (d for differential).

Definition 7 (Validity). *Validity of an overapproximate dassert triple.*

$$\begin{array}{l}\models_{vc} \{\phi_-, \phi_+\}\ s_- \pm s_+\ \{\phi_{\text{rm}}, \phi_{\text{rel}}, \phi_{\text{add}}\}\ \textit{iff} \\ \quad \sigma_- \in \Sigma_+\ \textit{iff}\ \sigma_- \models \phi_-\ \textit{and}\ \sigma_+ \in \Sigma_+\ \textit{iff}\ \sigma_+ \models \phi_+\ \textit{and} \\ \quad \Sigma_-, \Sigma_+ \vdash s_- \pm s_+ \Downarrow_{vc} \Sigma_{\text{rm}}, \Sigma_{\text{rel}}, \Sigma_{\text{add}}\ \textit{implies} \\ \qquad \textit{forall}\ \sigma_{\text{rm}} \in \Sigma_{\text{rm}}.\ \sigma_{\text{rm}} \models \phi_{\text{rm}}\ \textit{and forall}\ \sigma_{\text{add}} \in \Sigma_{\text{add}}.\ \sigma_{\text{add}} \models \phi_{\text{add}}\ \textit{and} \\ \qquad \textit{forall}\ \sigma_{\text{rel}} \in \Sigma_{\text{rel}}.\ \sigma_{\text{rel}} \models \phi_{\text{rel}}\end{array}$$

This definition is in terms of our collecting semantics from definition 6. It builds the sets of pre states, then runs them through our collecting semantics, and finally asserts that our safety postconditions hold over all post states from our collecting semantics.

Our strategy for developing a proof system that derives valid dassert triples is to combine the postconditions from Hoare and incorrectness logic triples to form predicates that overapproximate Σ_{rm}, Σ_{rel}, and Σ_{add}. This strategy does not take advantage of the structure of our collecting semantics (*i.e.* validity definition), and thus should scale to proving arbitrary properties over the sets of reachable states, which we leave as future work. Let's first consider deriving a predicate that overapproximates Σ_{rm}, which will also demonstrate the approach for Σ_{add}.

If we have a predicate that represents reachable new program states, then we can define when *any* old state would be removed from those new program states for a variable correspondence, apart from reachable old states. If all of those old states model some φ_-, then φ_- is a predicate that overapproximates all possible removed states. This can be seen as a direct translation of our removed relation from definition 2 to predicates instead of concrete state sets.

Definition 8 (Removed Relation Predicate). *We define the judgment $\psi_+ \models^{\text{rm}}_{vc} \varphi_-$ that holds when φ_- is an overapproximation of all removed states for a predicate for added states, ψ'_+.*

$$\begin{array}{l}\psi_+ \models^{\text{rm}}_{vc} \varphi_-\ \textit{iff} \\ \quad \textit{forall}\ \sigma_-.\ (\textit{forall}\ \sigma_+.\ \sigma_+ \models \psi_+\ \textit{implies}\ \sigma_- \not\sim_{vc} \sigma_+)\ \textit{implies}\ \sigma_- \models \varphi_-\end{array}$$

$\psi_1 \models^{\text{rm}}_{vc} \varphi_1$ holds if every state satisfying φ_1 has no vc-related state satisfying ψ_1. ψ_1 is "removed model" for φ_1 because the set of states that model ψ'_1 would force any state modeling φ_1 to be removed under the variable correspondence.

Given $\psi_+ \models^{\text{rm}}_{vc} \varphi_-$, for φ_- to overapproximate all removed states with respect to some concrete reachable new states Σ_+, ψ_+ must *underapproximate* Σ_+. To see this, suppose instead that ψ_+ overapproximates Σ_+. Then there may exist a new state, σ_+, such that $\sigma_+ \models \psi_+$ but $\sigma_+ \notin \Sigma_+$. If there is some old state, σ_-, such that $\sigma_- \sim_{vc} \sigma_+$ but $\sigma_- \not\sim_{vc} \sigma'_+$ *forall* $\sigma'_+ \in \Sigma_+$, then σ_- would need not model φ_- even though it is a removed state for Σ_+. This means φ_- getting *stronger* when we *weaken* ψ_+, which has the effect of underapproximating Σ_{rm} if Σ_+ is overapproximated. Thus, if instead ψ_+ underapproximates Σ_+

($forall\ \sigma_+.\ \sigma_+ \models \psi_+\ implies\ \sigma_+ \in \Sigma_+$), then φ_- will always overapproximate Σ_{rm}.

The removed relation predicate from definition 8 quantifies over all possible states, which is not possible in our assertion language. So, we need a way to derive this judgment in our assertion language. To do this, we also require a binary state predicate that holds whenever the variable correspondence does not hold, which we define below, along with a predicate that holds whenever the variable correspondence does hold.

Definition 9 (Variable Correspondence Predicate).

$$\models_{vc} \pi_2 \quad \textit{iff} \quad \textit{forall}\ \sigma_-, \sigma_+.\ \sigma_- \sim_{vc} \sigma_+\ \textit{iff}\ (\sigma_-, \sigma_+) \models \pi_2$$
$$\models_{\neg vc} \pi_2 \quad \textit{iff} \quad \textit{forall}\ \sigma_-, \sigma_+.\ \sigma_- \not\sim_{vc} \sigma_+\ \textit{iff}\ (\sigma_-, \sigma_+) \models \pi_2$$

We now provide a way to derive valid variable correspondence predicates.

$$\textsc{VC-Related}\ \frac{vc = (\overrightarrow{x}, \overrightarrow{y}) \qquad |\overrightarrow{x}| = |\overrightarrow{y}| = n \qquad \pi_2 = \overrightarrow{x}_0 \texttt{ = } \overrightarrow{y}_0 \wedge \ldots \overrightarrow{x}_i \texttt{ = } \overrightarrow{y}_i \wedge \ldots \overrightarrow{x}_n \texttt{ = } \overrightarrow{y}_n}{\vdash_{vc} \pi_2}$$

$$\textsc{Not VC-Related}\ \frac{vc = (\overrightarrow{x}, \overrightarrow{y}) \qquad |\overrightarrow{x}| = |\overrightarrow{y}| = n \qquad \pi_2 = \overrightarrow{x}_0 \texttt{ != } \overrightarrow{y}_0 \wedge \ldots \overrightarrow{x}_i \texttt{ != } \overrightarrow{y}_i \wedge \ldots \overrightarrow{x}_n \texttt{ != } \overrightarrow{y}_n}{\vdash_{\neg vc} \pi_2}$$

$$\textsc{Not VC-Related Length}\ \frac{vc = (\overrightarrow{x}, \overrightarrow{y}) \qquad |\overrightarrow{x}| \neq |\overrightarrow{y}|}{\vdash_{\neg vc} \top}$$

Where `=` *and* `!=` *are the boolean connectives for equality and disequality in our assertion language, respectively.*

Theorem 1 (Derived Variable Correspondence Predicates are Sound). *The proof rules we gave for deriving variable correspondence predicates are sound.*

$$\vdash_{vc} \pi_2\ implies\ \models_{vc} \pi_2 \quad and \quad \vdash_{\neg vc} \pi_2\ implies\ \models_{\neg vc} \pi_2$$

The proof of theorem 1 can be understood by realizing $\vdash_{vc}$ is a direct translation of the variable correspondence from definition 4 to our assertion language.

Now, we can define a proof rule for deriving removed relation predicate judgments, as defined in 8.

Definition 10 (Derivability of Removed Relation Predicates).

$$\frac{vc = (\overrightarrow{x}, \overrightarrow{y}) \qquad \varphi_- = \forall \overrightarrow{y} \cup fv(\psi_+).\ \psi_+ \Rightarrow \pi_2 \qquad \vdash_{\neg vc} \pi_2}{\psi_+ \vdash^{\text{rm}}_{vc} \varphi_-}$$

Here, $\forall \overrightarrow{y} \cup fv(\psi_+)$ universally quantifies over all new variables in vc and all variables that appear free in ψ_+.

In this rule, the universal quantification has the effect of removing the dependence on new program variables from the state(s) predicates in scope. Thus, it can be viewed as transforming ψ_+ from a state predicate to *proposition* that represents all states which ψ_+ held on. This quantification trick also removes the dependence on new state from π_2, transforming it to a predicate on only old states. Therefore, φ_- is a predicate only on old states, as desired. We can see that φ_- represents all possible removed states since any old state that models it will not be related by the variable correspondence to any new state that models ψ_+. For example, consider $\psi_+ := \mathtt{x}_+ > 10$, $vc := (\mathtt{x}_-, \mathtt{x}_+)$. Then, we have $\varphi_- = \forall \mathtt{x}_+.\ \mathtt{x}_+ > 10 \Rightarrow (\mathtt{x}_+ \mathtt{!=}\ \mathtt{x}_+)$, which will holds precisely when $\mathtt{x}_+$ is less than or equal to 10, which correctly characterizes all removed states.

Theorem 2 (Derived Removed Predicates are Sound). *Our proof rule for deriving removed relation predicates from definition 10 is sound with respect to the removed relation predicate judgment defined in 8.*

Once again, the proof of this theorem can be understood by realizing $\vdash^{\mathrm{rm}}_{vc}$ is a translation of $\models^{\mathrm{rm}}_{vc}$ to our assertion language.

Now, we can use $\vdash^{\mathrm{rm}}_{vc}$ to derive safety properties over the removed set of states for some sets of reachable old and new states Σ_+, Σ_-. If we have an old state predicate that is an overapproximation of all reachable old states (Σ_-), and another predicate that is an overapproximation of all possible removed states for Σ_+, then their conjunction will be an overapproximation of Σ_{rm}.

Theorem 3 (Derivability of Removed State Safety Properties). *Given* $\Sigma_{\mathrm{rm}} := \{\sigma_- \in \Sigma_- \mid \sigma_- \not\simeq \Sigma_+\}$*, for any* $\Sigma_-, \Sigma_+, \varphi_-, \psi_+$*, and* θ_- *we have,*

$$
\begin{array}{l}
(\textit{forall } \sigma_-.\ \sigma_- \in \Sigma_- \textit{ implies } \sigma_- \models \varphi_-) \textit{ and} \\
(\textit{forall } \sigma_+.\ \sigma_+ \models \psi_+ \textit{ implies } \sigma_+ \in \Sigma_+) \textit{ and } \psi_+ \vdash^{\mathrm{rm}}_{vc} \theta_- \textit{ implies} \\
\quad \textit{forall } \sigma'_-.\ \sigma'_- \in \Sigma_{\mathrm{rm}} \textit{ implies } \sigma'_- \models \theta_- \wedge \varphi_-
\end{array}
$$

The proof of this theorem directly follows from theorem 2 and the semantics of conjunction, since any σ'_- such that $\sigma'_- \models \theta_- \wedge \varphi_-$ must be both reachable and removed.

Now that we have shown how to derive a safety property over the removed states, we turn our attention to deriving a safety property for related states. We follow the same approach, but now all predicates can be overapproximations of reachable states since a safety property of Σ_{rel} is a 2-safety property.

Definition 11 (Related Predicate).

$$
\begin{array}{l}
\varphi_-, \varphi_+ \models^{\mathrm{rel}}_{vc} \phi_{\mathrm{rel}} \quad \textit{iff} \\
\quad \forall \sigma_-, \sigma_+.\ (\sigma_- \models \varphi_- \textit{ and } \sigma_+ \models \varphi_+ \textit{ and } \sigma_- \sim_{vc} \sigma_+) \textit{ implies } (\sigma_-, \sigma_+) \models \phi_{\mathrm{rel}}
\end{array}
$$

$$
\frac{\phi_{\mathrm{rel}} = \varphi_- \wedge \varphi_+ \wedge \pi_2 \qquad \vdash_{vc} \pi_2}{\varphi_-, \varphi_+ \vdash^{\mathrm{rel}}_{vc} \phi_{\mathrm{rel}}}
$$

In definition 11, we follow a similar strategy as we have discussed for removed states. If $\varphi_-, \varphi_+ \vdash_{vc}^{\text{rel}} \phi_{\text{rel}}$, then any pair of states that models ϕ_{rel} must be related by the variable correspondence and model φ_- and φ_+, respectively. Then, if φ_- and φ_+ overapproximates reachable old and new states, ϕ_{rel} must overapproximate related states.

Theorem 4 (Related Predicate is Sound). *If φ_- is an overapproximation of Σ_-, φ_+ is an overapproximation of Σ_+, and $\varphi_-, \varphi_+ \vdash_{vc}^{\text{rel}} \phi_{\text{rel}}$, then ϕ_{rel} is an overapproximation of Σ_{rel} induced by $\sim_{vc}$, Σ_-, and Σ_+.*

Using definitions 10, 11, and one for $\vdash_{vc}^{\text{add}}$ that is the same as $\vdash_{vc}^{rm}$ but for added states, we can define how to derive a dassert triple.

Definition 12 (A Proof System for Dassert Triples).

$$\frac{\vdash \{\phi_-\}\ s_-\ \{\varphi_-\} \quad \vdash \{\phi_+\}\ s_+\ \{\varphi_+\} \quad \vdash [\phi_-]\ s_-\ [\psi_-] \quad \vdash [\phi_+]\ s_+\ [\psi_+] \quad \varphi_-, \varphi_+ \vdash_{vc}^{\text{rel}} \phi_{\text{rel}} \quad \psi_+ \vdash_{vc}^{\text{rm}} \theta_- \quad \psi_- \vdash_{vc}^{\text{rm}} \theta_+}{\vdash_{vc} \{\phi_-, \phi_+\}\ s_- \pm s_+\ \{\theta_- \wedge \varphi_-, \phi_{\text{rel}}, \theta_+ \wedge \varphi_+\}}$$

We also provide a rule of consequence.

$$\frac{\vdash_{vc} \{\phi_-, \phi_+\}\ s_- \pm s_+\ \{\phi'_{\text{rm}}, \phi'_{\text{rel}}, \phi'_{\text{add}}\} \quad \phi'_{\text{rm}} \Rightarrow \phi_{\text{rm}} \quad \phi'_{\text{rel}} \Rightarrow \phi_{\text{rel}} \quad \phi'_{\text{add}} \Rightarrow \phi_{\text{add}}}{\vdash_{vc} \{\phi_-, \phi_+\}\ s_- \pm s_+\ \{\phi_{\text{rm}}, \phi_{\text{rel}}, \phi_{\text{add}}\}}$$

The only aspects of this proof rule that we have not discussed are the incorrectness and Hoare logic triples. Throughout this section, we have referenced that we require overapproximations and underapproximations of the reachable states. If $[\phi_-]\ s_-\ [\psi_-]$ is a derivable incorrectness logic triple, it follow from the soundness of incorrectness logic that ψ_- overapproximates the set of states reachable from the set of states satisfying ϕ_-. More precisely, if Σ_- is the set of states satisfying ϕ_-, and $\Sigma_- \vdash s \Downarrow^{\{\}} \Sigma'_-$, then $forall\ \sigma'_-.\ \sigma'_- \models \psi_-$ *implies* $\sigma'_- \in \Sigma'_-$. This is what we require for underapproximations, and the same holds for derivable Hoare logic triples and overapproximations. Something to note for the consequence rule is that we cannot weaken or strength the preconditions, since they are serving as preconditions in both Hoare logic triples and incorrectness logic triples.

Theorem 5 (Dassert Proof System is Sound). *The rules we presented for deriving dassert triples in definition 12 is sound with respect to the validity of overapproximate dassert triples defined in 7.*

$$\vdash_{vc} \{\phi_-, \phi_+\}\ s_- \pm s_+\ \{\phi_{\text{rm}}, \phi_{\text{rel}}, \phi_{\text{add}}\}\ \textit{iff}\ \models_{vc} \{\phi_-, \phi_+\}\ s_- \pm s_+\ \{\phi_{\text{rm}}, \phi_{\text{rel}}, \phi_{\text{add}}\}$$

Theorem 5 follows from the soundness of Hoare logic and incorrectness logic, along with theorems 3 and 4. This proof system is not relatively complete. Showing this would require that any valid related state safety assertion, ϕ_{rel}, can be

formulated with our related predicate derivation, $\varphi_-, \varphi_+ \vdash^{\text{rel}}_{vc} \phi_{\text{rel}}$. This means the binary state predicate ϕ_{rel} must be expressible as the conjunction of two single state predicates and the variable correspondence predicate, which is not always possible in our assertion language. We believe an extension to this system with a relational precondition could be formulated to allow for relative completeness, but we leave this as future work.

5 Implementation

We mechanized our collecting semantics from definition 6 and the proof system from definition 12 in the Lean proof assistant [13]. One major difference between our Lean formalism and the formalism in this paper is we did not define an assertion language in Lean. Instead, we used Lean itself as the assertion language, which means assertions are functions on states, and we can quantify over all possible states within them. This simplified our proofs since we did not need to prove our assertion language constructs were sound with respect to their denotations which are theorems 1 and 2 In our mechanization, we proved the soundness of our proof system, which is theorem 5 with a more expressive assertion language.

6 Related Works

Our work to establish a revision-specific collecting semantics and proof system is inspired by a line of work that can broadly characterized as "semantic program differencing". These works use symbolic execution [18], game semantics [14], abstract interpretation [16,17], bounded model checking [6], and other techniques [12,11,2,7] to prove specific properties about program differences and equivalences. The properties these works establish include equivalence checking [8], regression verification [6], and refinement [19]. We instead focused on providing a framework that is expressible enough to specify a variety of these properties in an intuitive manor.

Some existing program logics [1,5,21] can establish relational program properties that require an overapproximation of one program and an underapproximation of the other, but not arbitrary relational program properties. Hyper Hoare logic [20] is expressive enough to reason about arbitrary properties on sets of states, but only for one program (*i.e.* it is not relational). The techniques used in our proof theory are similar to, and inspired by, [4] and [20].

References

1. Antonopoulos, T., Koskinen, E., Le, T.C., Nagasamudram, R., Naumann, D.A., Ngo, M.: An Algebra of Alignment for Relational Verification. Proceedings of the ACM on Programming Languages **7**(POPL), 573–603 (Jan 2023). `https://doi.org/10.1145/3571213`, `https://dl.acm.org/doi/10.1145/3571213`

2. Apiwattanapong, T., Orso, A., Harrold, M.: A differencing algorithm for object-oriented programs. In: Proceedings. 19th International Conference on Automated Software Engineering, 2004. pp. 2–13. IEEE, Linz, Austria (2004). https://doi.org/10.1109/ASE.2004.1342719, http://ieeexplore.ieee.org/document/1342719/
3. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. SIGPLAN Not. **39**(1), 14–25 (Jan 2004). https://doi.org/10.1145/982962.964003, https://dl.acm.org/doi/10.1145/982962.964003
4. Dardinier, T., Müller, P.: Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties (extended version) (Apr 2024), http://arxiv.org/abs/2301.10037, arXiv:2301.10037
5. Dickerson, R., Ye, Q., Zhang, M., Delaware, B.: RHLE: Modular Deductive Verification of Relational $$\forall \exists $$ Properties. pp. 67–87 (Nov 2022). https://doi.org/10.1007/978-3-031-21037-2_4
6. Godlin, B., Strichman, O.: Regression verification. In: Proceedings of the 46th Annual Design Automation Conference. pp. 466–471. DAC '09, Association for Computing Machinery, New York, NY, USA (Jul 2009). https://doi.org/10.1145/1629911.1630034, https://dl.acm.org/doi/10.1145/1629911.1630034
7. Gyori, A., Lahiri, S.K., Partush, N.: Interprocedural Semantic Change-Impact Analysis using Equivalence Relations (Sep 2016). https://doi.org/10.48550/arXiv.1609.08734, http://arxiv.org/abs/1609.08734, arXiv:1609.08734 [cs]
8. Hawblitzel, C., Kawaguchi, M., Lahiri, S.K., Rebêlo, H.: Towards Modularly Comparing Programs Using Automated Theorem Provers. In: Bonacina, M.P. (ed.) Automated Deduction – CADE-24. pp. 282–299. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_20
9. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (Oct 1969). https://doi.org/10.1145/363235.363259, https://dl.acm.org/doi/10.1145/363235.363259
10. Jackson, Ladd: Semantic Diff: a tool for summarizing the effects of modifications. In: Proceedings 1994 International Conference on Software Maintenance. pp. 243–252 (Sep 1994). https://doi.org/10.1109/ICSM.1994.336770, https://ieeexplore.ieee.org/document/336770/?arnumber=336770
11. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification, vol. 7358, pp. 712–717. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_54, http://link.springer.com/10.1007/978-3-642-31424-7_54, series Title: Lecture Notes in Computer Science
12. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 345–355. ACM, Saint Petersburg Russia (Aug 2013). https://doi.org/10.1145/2491411.2491452, https://dl.acm.org/doi/10.1145/2491411.2491452
13. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean Theorem Prover (System Description). In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction - CADE-25. pp. 378–388. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_26

14. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: Game Semantic Analysis of Equivalence in IMJ. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) Automated Technology for Verification and Analysis. pp. 411–428. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-24953-7_30
15. O'Hearn, P.W.: Incorrectness logic. Proceedings of the ACM on Programming Languages **4**(POPL), 1–32 (Jan 2020). https://doi.org/10.1145/3371078, https://dl.acm.org/doi/10.1145/3371078
16. Partush, N., Yahav, E.: Abstract Semantic Differencing for Numerical Programs. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Logozzo, F., Fähndrich, M. (eds.) Static Analysis, vol. 7935, pp. 238–258. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_14, http://link.springer.com/10.1007/978-3-642-38856-9_14, series Title: Lecture Notes in Computer Science
17. Partush, N., Yahav, E.: Abstract semantic differencing via speculative correlation. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. pp. 811–828. ACM, Portland Oregon USA (Oct 2014). https://doi.org/10.1145/2660193.2660245, https://dl.acm.org/doi/10.1145/2660193.2660245
18. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. pp. 226–237. SIGSOFT '08/FSE-16, Association for Computing Machinery, New York, NY, USA (Nov 2008). https://doi.org/10.1145/1453101.1453131, https://dl.acm.org/doi/10.1145/1453101.1453131
19. Roever, W.P., Engelhardt, K.: Data refinement: model-oriented proof methods and their comparison (Jan 2008)
20. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 57–69. PLDI '16, Association for Computing Machinery, New York, NY, USA (Jun 2016). https://doi.org/10.1145/2908080.2908092, https://dl.acm.org/doi/10.1145/2908080.2908092
21. Wu, S., Wu, X., Cao, Q.: Encode the $\forall\exists$ Relational Hoare Logic into Standard Hoare Logic (Aug 2025). https://doi.org/10.1145/3763138, http://arxiv.org/abs/2504.17444, arXiv:2504.17444 [cs]