Property Directed Equivalence via Abstract Simulation

Grigory Fedyukovich[1,2(✉)], Arie Gurfinkel[3], and Natasha Sharygina[1]

[1] USI, Lugano, Switzerland
{grigory.fedyukovich,natasha.sharygina}@usi.ch
[2] UW, Seattle, USA
grigory@cs.washington.edu
[3] SEI/CMU, Pittsburgh, USA
arie@cmu.com

Abstract. We present a novel approach for automated incremental verification that employs both reusable and relational specifications of software to incrementally verify pairs of programs with possibly nested loops. It analyzes two programs, P - the one already verified, and Q - the one needed to be verified, and proceeds by detecting an abstraction αP of P and a simulation ρ, such that αP simulates Q via ρ. The key idea behind our simulation synthesis is to drive construction of both αP and ρ by the safe inductive invariants of P, thus guaranteeing the property preservations by the results. Finally, our approach allows effective lifting of the safe inductive invariants of P to Q using only αP and ρ. Based on our evaluation, in many cases when the absolute equivalence between programs cannot be proven, our approach is able to establish the *property directed equivalence*, confirming that the program Q is safe.

1 Introduction

Software development is a continuous process that repeatedly iterates between the stages of implementing a program and checking its safety. To satisfy quality standards, a software product should pass through a myriad of intermediate verification stages, each of which assures safety of a particular change against its baseline version. One of the most successful techniques to verify isolated software versions fully automatically and exhaustively is *Model Checking*.

Without detracting from the merits of the recent model checking solutions, there is a demand for new methods to make other steps in the typical "verify-bugfix-verify" workflow automated and exhaustive. In particular, there is a clear

This work is supported in part by the SNSF Fellowship P2T1P2_161971.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense. This material has been approved for public release and unlimited distribution. DM-0001771.

S. Chaudhuri and A. Farzan (Eds.): CAV 2016, Part II, LNCS 9780, pp. 433–453, 2016.
DOI: 10.1007/978-3-319-41540-6_24

need for new techniques that would make the software analysis more efficient by (1) finding a *reusable specification* of an already verified program version to be used while verifying another program version; and (2) finding a *relational specification* between program versions that describes how the versions relate to each other. When discovered, these specifications enable formal analysis of sequences of program versions called *Formal Incremental Verification* (FIV) and can be used for various tasks such as upgrade checking, compositional (modular) verification, change impact calculation, program repair, etc.

Model checkers for the programs with unbounded (and possibly nested) loops reduce the verification tasks to finding safe inductive invariants. Such invariants over-approximate all safe behaviors of the program and constitute so called *proof certificates*. Since the problem of inferring proofs is known to be undecidable in general [29], individual model-checking solutions are not guaranteed to deliver an appropriate invariant. On the other hand, in cases if model checking succeeded, the synthesized proof provides an important reusable specification that comes in handy whenever the program gets modified. In this paper, we address a challenge of migrating proofs across the different program versions.

Simulation is known to be the most general mapping to transfer proofs between program versions [17, 26, 27]. However, discovering a simulation relation is difficult and usually requires a manual guidance. One of the recent promising approaches for the simulation synthesis is SimAbs introduced in [16]. Despite providing a fully automated schema, it is unable to find simulations for pairs of program versions obtained after non-trivial program transformations. We experimented with SimAbs and observed that it discovered precise simulations only in 9 % of cases[1], while in the rest it either provided an *abstract simulation* (i.e., an abstraction of the already verified program version that simulates the precise modified program version) or diverged. In general, abstract simulations are not applicable for migrating proofs, since the delivered abstraction might not preserve all important safety properties of the verified program version.

In this paper, for the task of migrating proofs, we show that the precise simulation relation between program versions is not even needed. Instead, it is enough to deal with abstract simulations, but created in a particular way. It is crucial to ensure that the given invariant is safe for the delivered abstraction. For this reason, we propose to guide the abstraction generation by the proofs. If a simulation for such a *proof-based abstraction* is found then the proof can be lifted directly. We present an algorithm called ASSI (stands for *Abstract Simulation Synthesis with Invariants*) and an algorithm called PDE (stands for *Property Directed Equivalence*) that perform such reasoning completely automatically.

A distinguishing feature of ASSI and PDE is the ability to migrate the invariants through abstractions even if the abstractions do not preserve safety. PDE attempts to lift as much information from the invariant as possible and then strengthens it using a Horn-clause-based unbounded model checker.

We contribute the implementation of ASSI and PDE on the top of the LLVM-based model checker UFO [1] and provide extensive evaluation of

[1] See Sect. 6 for more details.

non-trivial LLVM-optimizations. In the same experimental setting as for SimAbs, ASSI discovered non-trivial abstract simulations in 82 % of cases, that further allowed PDE to migrate the proofs completely (34 %) or at least partially (48 %). Guided by the proofs, ASSI outperformed SimAbs by up to 2000**X**. In other words, it enabled scaling the entire simulation synthesis technology to solve more difficult problems and to do it more efficiently.

To sum up, PDE can be seen as the first technique to effectively exploit both, the *reusable specification* (by means of the proofs) and the *relational specification* (by means of the abstract simulations), to incrementally verify sequences of program versions with non-trivial loop structures and non-trivial transformations. The most important contributions can be classified as follows:

- A concept and a formalization of the PDE framework to incremental verification through abstract simulation and invariants.
- An algorithm ASSI for abstract simulation synthesis, designed to take the proofs into account and consider the proof-based abstractions.
- An LLVM-based evaluation on Software Verification Competition benchmarks that succeeds in establishing the property directed equivalence in many cases when the absolute equivalence between programs cannot be proven. In some other cases PDE was able to lift the proof partially and strengthen it further by means of the model checker UFO.

The rest of the paper is structured as follows. We start with a brief overview of the related methods (Sect. 2) followed by the background of unbounded verification (Sect. 3). Then, we formalize the underlying concepts behind simulation synthesis (Sect. 4) and use them to build the algorithms for ASSI and PDE (Sect. 5). Finally, we outline the evaluation of ASSI and PDE (Sect. 6) and conclude the paper (Sect. 7).

2 Related Work

We aim at checking the *property directed equivalence* (i.e., equivalence of programs with respect to some common property) automatically since it has a direct application in model checking. This is an alternative property to *absolute equivalence* (i.e., equivalence of programs with respect to any possible property) [3,12,19,21,28] that is rare in practice. The first automatic solutions to equivalence checking date back to hardware verification. Based on BDDs and SAT solving, the methods [5,9,30] aim at searching for a counter-example witnessing inequivalence of the two circuits. Most of them exploit structural similarities between the circuits that make them able to scale well with the circuit size. The further application of equivalence checking is to prove validity of compiler optimizations (e.g., [3,28,33]). The basis of most of work on Translation Validation is the idea of guessing a simulation relation between programs. Our algorithm also guesses relations, but before using them for PDE, it formally checks their validity with ASSI and drops those for which the check fails.

A step towards equivalence checking of software was made in [12] that proposed to check equivalence of a Verilog circuit and a C program through encoding and solving a quantifier-free SAT formula. A more recent solution [19] employs Bounded Model Checking [10] (BMC) to establish absolute equivalence between C programs. The method traverses the call graph bottom-up and separately checks whether identity of inputs implies identity of outputs for each pair of matched (e.g., by type annotation) functions, while all the nested calls are abstracted away using the same uninterpreted functions. A similar but language-agnostic approach is implemented in the SymDiff tool [21].

The problem of checking *non-absolute equivalence* between programs (also referred to as *incremental verification*) was addressed in a number of works, e.g. [2,4,7,13,15,18,24,31,34]. The main motivating idea behind this line of research is the ability of reusing efforts between verification runs, thus achieving performance speedup compared to verification of programs in isolation. eVolCheck [31] extracts the over-approximating function summaries from one program satisfying the given property and then re-checks if summaries still over-approximate function behavior in another program. However, eVolCheck is based on BMC and relies on the user-provided bounds for loops and recursive function calls. Unbounded incremental verifier OptVerify [15] is designed to lift inductive invariants across program transformations using a guessed variable mapping. Contrary to our approach, OptVerify can be applied only to programs sharing the same loop-structure. A similar and generalized approach for CEGAR-based verification was proposed in [7]. It stores the level of abstraction needed to prove safety of one program (e.g., which predicates to use in predicate analysis). The predicates are then transferred and adapted to another program to obtain the initial level of abstraction from which the analysis starts (not from scratch). Note that none of the mentioned techniques relies on automatically derived relational specifications (like a mapping between variables or a type annotation) so they all require re-validating the artifacts migrated from the verified programs. In contrast, the technique presented in this paper benefits from using certified simulation relations between programs, thus confirming that the migrated invariants are always sound.

Alternatively, there are approaches [25,34] to reason not only about differences between behaviors, but also to analyze differences between properties in different programs. The technique called Verification Modulo Versions (VMV) [25] transforms assertions from one program into assumptions for another program. VMV then tries to find (or prove absence of) bugs that are present only in the latter program. The technique called Directed Incremental Symbolic Execution (DISE) [34] is driven by the *change impact* which in fact is the program slice obtained by symbolic execution of the syntactic delta between the programs. The change impact is, however, property-independent, so DISE still requires further analysis whether the requested properties hold or do not in both programs. PDE is also able to calculate the change impact as a side effect of incremental verification. In contrast, our change impact is always property-dependent that would make it potentially useful to identify program locations responsible for the particular property violations.

3 Programs, Abstractions and Proofs

In this paper, we consider "large-block" encoding (LBE) [6] of programs that allows representing complex control-flow graphs compactly. A *program* is a tuple $P = \langle \mathit{Vars}, \mathit{CP}, \mathit{en}, \mathit{err}, E, \tau \rangle$, where Vars is a set of *real* and *boolean* variables, CP is a set of cutpoints (i.e., program locations which represent heads of loops); $\mathit{en}, \mathit{err} \in \mathit{CP}$ are respectively designated locations of the program entry and the error (i.e., the violation of some property of interest); $E \subseteq \mathit{CP} \times \mathit{CP}$ is the cutpoint-relation corresponding to the largest loop-free program fragments, and $\tau : E \to \mathit{Expr}(\mathit{Vars})$ maps each element of E to a formula in first-order logic that encodes a transition relation of the corresponding program fragment. We refer to the graph $\langle \mathit{CP}, E \rangle$ as a *Cutpoint graph* (*CPG*) of the program P.

Throughout the paper, we consider only variables that appear as *source*- and *destination* arguments for the edges E of the program P. In the formulas encoding transition relations τ, the other (local) variables are implicitly existentially quantified. Let $V : E \to 2^{\mathit{Vars}}$ be the function that, given a cutpoint, returns a set of variables live at that cutpoint. We use primed notation for Vars' to distinguish between the source and the destination arguments of each edge. To enable existential quantification over variables in a formula $e \in \mathit{Expr}$, we explicitly declare the variable sets over which e is expressed.

The goal of formal verification is to check whether the location *err* is unreachable by any program behavior starting at the location *en*. One of the most common ways of proving safety of a program is to construct an *inductive invariant* that over-approximates the sets of reachable states in the program, and to prove the unreachability of *err* for the invariant. In the context of LBE, (safe) inductive invariants are represented by a labeling of the cutpoints with logical formulas such that the condition(s) of the following definition hold.

Definition 1. *Given a program P, a mapping $\psi : \mathit{CP} \to \mathit{Expr}(\mathit{Vars})$ is an* inductive invariant *if:*

$$\psi(\mathit{en}) = \top \tag{1}$$

$$\forall (u, v) \in E \,.\, \Big(\psi(u)(\vec{x}) \wedge \tau(u, v)(\vec{x}, \vec{x}') \implies \psi(v)(\vec{x}')\Big) \tag{2}$$

ψ is a proof *(or a* safe *inductive invariant) of P if additionally:*

$$\psi(\mathit{err})(\vec{x}') \implies \bot \tag{3}$$

In (2), $\psi(u)$ is expressed over the source arguments of the *CPG*-edge (u, v), namely $\vec{x}$. In contrast, $\psi(v)$ is expressed over the destination arguments of the *CPG*-edge (u, v), namely $\vec{x}'$. Throughout the paper, we add the following mnemonic notation to emphasize whether (3) holds for an inductive invariant: $|\psi|$ (with vertical bars) to indicate that (1)–(2) hold, but (3) does not, and $\widehat{\psi}$ (with a hat) to indicate that all three conditions hold. If ψ is used without this mnemonic notation then in the current context it does not matter if (3) holds or not.

Since an inductive invariant over-approximates the sets of reachable states for each cutpoint of a program P, it allows more behaviors of P than specified

```
int x = 0;
while ( * ) {
  x++;
}
if ( x < 0 ) {
  error();
}
```

(a) P_0

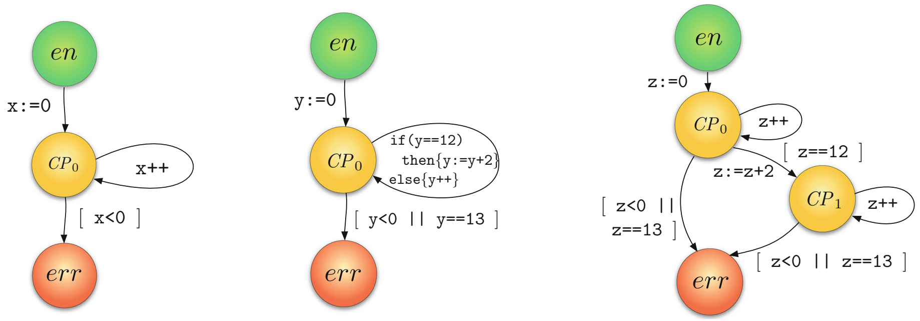
```
int y = 0;
while ( * ) {
  if ( y == 12 ) {
    y = y + 2;
  } else {
    y++;
  }
}
if ( y < 0 ||
      y == 13 ) {
  error();
}
```

(b) Q_0

```
int z = 0;
while ( * && z < 12 ) {
  z++;
}
if ( z == 12 ) {
  z = z + 2;
}
while ( * && z > 12 ) {
  z++;
}
if ( z < 0 || z == 13 ) {
  error();
}
```

(c) Q_1

Fig. 1. Programs P_0 and Q_0 and the loop-splitting optimization of Q_0.



(a) P_0 (b) Q_0 (c) Q_1

Fig. 2. Cutpoint graphs of P_0, Q_0 and Q_1.

$$\tau_{P_0} = \begin{cases} (en, CP_0) \mapsto (x' = 0) \\ (CP_0, CP_0) \mapsto (x' = x + 1) \\ (CP_0, err) \mapsto (x' = x \wedge x' < 0) \end{cases} \qquad \widehat{\psi} = \begin{cases} (en) \mapsto \top \\ (CP_0) \mapsto x \geq 0 \\ (err) \mapsto \bot \end{cases}$$

$$\tau_{\alpha P_0} = \begin{cases} (en, CP_0) \mapsto (x' \geq 0) \\ (CP_0, CP_0) \mapsto (x' \geq x) \\ (CP_0, err) \mapsto (x < 0) \end{cases} \qquad \tau_{\beta P_0} = \begin{cases} (en, CP_0) \mapsto (x' \geq 0) \\ (CP_0, CP_0) \mapsto (x' \geq x) \\ (CP_0, err) \mapsto (x < 0) \vee (x = 13) \end{cases}$$

Fig. 3. Transition relations τ_{P_0}, $\tau_{\alpha P_0}$, $\tau_{\beta P_0}$ and proof $\widehat{\psi}$ of P_0.

by its transition relation τ. It can be used to represent programs that share the *CPG*-structure with P, but have less accurate transition relations. We say that such programs are the *abstractions* of P and describe them formally as follows.

Definition 2. *Given two programs $P = \langle Vars, CP, en, err, E, \tau \rangle$ and $\alpha P = \langle Vars, CP, en, err, E, \tau_\alpha \rangle$, αP is an* abstraction *of P if for some inductive invariant ψ of P,*

$$\forall (u, v) \in E \,.\, \Big(\psi(u)(\vec{x}) \wedge \tau(u, v)(\vec{x}, \vec{x}') \implies \tau_\alpha(u, v)(\vec{x}, \vec{x}') \Big) \qquad (4)$$

The use of an inductive invariant ψ in (4) makes the way of creating abstractions more flexible. Indeed, for each cutpoint $u \in CP$, the formula $\psi(u)$ might bring any additional information about the pre-states at the edge $(u, v) \in E$

learned inductively from the dependent *CPG*-edges. Note that $\tau(u, v)$ might be incomparable (and even inconsistent) with $\psi(u)$.

The simplest way to construct a program abstraction from the given inductive invariant ψ is to assign the transition relation of each *CPG*-edge by the invariant at the post-state of that edge. Thus, an abstraction of P can be constructed directly from ψ, and in rest of the paper we refer to it as to $\alpha^{\psi} P$. The following lemma assures that $\alpha^{\psi} P$ satisfies Definition 2.

Lemma 1. *Given* $P = \langle \mathit{Vars}, \mathit{CP}, \mathit{en}, \mathit{err}, E, \tau \rangle$ *and its invariant* ψ*, let* $\alpha^{\psi} P = \langle \mathit{Vars}, \mathit{CP}, \mathit{en}, \mathit{err}, E, \tau_{\alpha}^{\psi} \rangle$ *be defined as:*

$$\forall (u, v) \in E \,.\, \Big(\tau_{\alpha}^{\psi}(u, v)(\vec{x}, \vec{x}') \triangleq \psi(v)(\vec{x}') \Big) \tag{5}$$

Then $\alpha^{\psi} P$ *is an abstraction of* P*.*

If ψ is not trivial (i.e., $\exists u \in \mathit{CP} \,.\, \psi(u) \neq \top$) and some abstraction αP is as accurate as $\alpha^{\psi} P$ then αP provides a particular interest for incremental verification that is explained in Sect. 5. However, for the sake of completeness of presentation, we must admit that Definition 2 also allows other types of abstractions, abstract transition relation of which does not necessarily satisfy $\psi(v)$ for all post-states at (u, v).

Example 1. Consider a program P_0 shown in Fig. 1(a) that increments an integer counter[2] `x`, initially assigned to 0. The CPG_{P_0} is shown in Fig. 2(a) and consists of $\mathit{CP} = \{\mathit{en}, \mathit{CP}_0, \mathit{err}\}$ and $E = \{(\mathit{en}, \mathit{CP}_0), (\mathit{CP}_0, \mathit{CP}_0), (\mathit{CP}_0, \mathit{err})\}$. Figure 3 shows: (1) transition relation τ_{P_0} labeling each edge in E, (2) the proof $\widehat{\psi}$ labeling each cutpoint in CP, (3–4) transition relations of two abstractions αP_0 and βP_0 respectively. Compared to αP_0, βP_0 allows variable x to be equal to 13 in the cutpoint err. □

4 Simulation Relations in LBE with Invariants

Given a pair of programs $P = \langle \mathit{Vars}_P, \mathit{CP}_P, \mathit{en}_P, \mathit{err}_P,\ E_P, \tau_P \rangle$ and $Q = \langle \mathit{Vars}_Q, \mathit{CP}_Q, \mathit{en}_Q, \mathit{err}_Q, E_Q, \tau_Q \rangle$. A simulation relation between P and Q specifies a matching of every program behavior of Q by some program behavior of P. In LBE, finding simulation relations is the two-steps procedure. First, it requires finding a simulation σ at the level of *CPG*s (further referred to as *CPG-simulation*). Second, it requires finding a simulation ρ at the level of pairs of *CPG*-edges (further referred to as *edge-simulation*).

Definition 3. *Given two programs* P *and* Q*, we say that* CPG_P *simulates* CPG_Q *iff there exists a left-total relation* $\sigma : \mathit{CP}_Q \to \mathit{CP}_P$ *such that:*

$$\begin{aligned} \forall u_Q, v_Q \in \mathit{CP}_Q, u_P \in \mathit{CP}_P \,.\, (u_Q, v_Q) \in E_Q \wedge u_P = \sigma(u_Q) \implies \\ \exists v_P \in \mathit{CP}_P \,.\, (u_P, v_P) \in E_P \wedge v_P = \sigma(v_Q) \end{aligned} \tag{6}$$

[2] Here and later in the paper we assume no arithmetic overflow.

When clear from the context, we omit the subscripts from u_Q, v_Q, etc.

Definition 4. *Program P simulates program Q iff (1) CPG_P simulates CPG_Q via some σ, and (2) for that σ and some inductive invariant ψ of P, there exists a left-total relation $\rho : CP_Q \times CP_P \to Expr(Vars_Q \cup Vars_P)$ such that:*

$$\forall (u,v) \in E_Q \,.\, \Big(\psi\big(\sigma(u)\big)(\vec{y}) \wedge \rho\big(u,\sigma(u)\big)(\vec{x},\vec{y}) \wedge \tau_Q\big(u,v\big)(\vec{x},\vec{x}') \implies \exists \vec{y}' \,.\, \rho\big(v,\sigma(v)\big)(\vec{x}',\vec{y}') \wedge \tau_P\big(\sigma(u),\sigma(v)\big)(\vec{y},\vec{y}')\Big) \tag{7}$$

For each edge (u, v) in (7), the existential quantifier in front of $\vec{y}'$ is served to encode existence of a valuation of the variables in $V'_P\big(\sigma(v)\big)$. In contrast, valuations of the variables $\vec{x}$, $\vec{x}'$, $\vec{y}$ respectively of $V_Q(u)$, $V_Q(v)$ and $V_P\big(\sigma(u)\big)$ are implicitly universally quantified. Thus, for each $\vec{x}$ and $\vec{y}$ matched by $\rho(u, \sigma(u))$ and $\vec{x}'$, there should exists $\vec{y}'$ such that $\vec{x}'$ and $\vec{y}'$ are matched by $\rho(v, \sigma(v))$. Additionally, the pairs $\vec{x}$ and $\vec{x}'$, and $\vec{y}$ and $\vec{y}'$ should belong to valid behaviors corresponding to their transition relations $\tau_Q(u, v)$ and $\tau_P(\sigma(u), \sigma(v))$ respectively. Note that those transition-relation formulas are conjoined with the different sides of the implication, so the validity of the $\forall\exists$-formula means that each behavior of $\tau_Q(u, v)$ is matched by a behavior of $\tau_P(\sigma(u), \sigma(v))$ (but it is still allowed to have unmatched behaviors of $\tau_P(\sigma(u), \sigma(v))$. For this, the simulation relation induced by formulas $\rho(u, \sigma(u))$ and $\rho(v, \sigma(v))$ is required to be left-total.

Whenever for a given pair of programs P and Q, there exists the pair of relations $\langle\sigma, \rho\rangle$ such that P simulates Q, we write $Q \preceq_{\langle\sigma,\rho\rangle} P$, or simply $Q \preceq P$ if $\langle\sigma, \rho\rangle$ are clear from the context.

It is important to note that our definition of simulation relation exploits an inductive invariant ψ of P that over-approximates the sets of reachable states for each cutpoint of P. In particular, for each CPG-edge (u, v) of Q, the condition of Definition 4 restricts the set of pre-states of $\tau_P(\sigma(u), \sigma(v))$ on $\psi(\sigma(u))$. Such restriction is sound, since it does not drop any behavioral information of P that can be potentially useful while constructing and checking a simulation of Q. Furthermore, for each behavior of Q requiring to be matched by some behavior of P, the invariant ψ reduces the search space of this matching.

Simulations are used to lift the proofs between programs. In fact, if the error location err_P is proven unreachable in P, and P simulates Q, then the error location err_Q is unreachable in Q. Interestingly, this fact can be further propagated to the level of inductive invariants [22,26] making the following lemma hold:

Lemma 2. *Given programs P and Q, let ψ be a (safe) inductive invariant of P and $Q \preceq_{\langle\sigma,\rho\rangle} P$. Consider a mapping $\varphi : CP_Q \to Expr(Vars_Q)$ defined for each $u \in CP_Q$ such that:*

$$\varphi(u)(\vec{x}) \triangleq \exists \vec{y} \,.\, \rho\big(u,\sigma(u)\big)(\vec{x},\vec{y}) \wedge \psi\big(\sigma(u)\big)(\vec{y}) \tag{8}$$

Then φ is a (safe) inductive invariant of Q.

$$\tau_{Q_0} = \begin{cases} (en, CP_0) \mapsto (y' = 0) \\ (CP_0, CP_0) \mapsto \big((y = 12 \wedge y' = y + 2) \vee (y \neq 12 \wedge y' = y + 1)\big) \\ (CP_0, err) \mapsto \big((y' = y) \wedge (y' < 0 \vee y' = 13)\big) \end{cases}$$

$$\sigma = \begin{cases} en \mapsto en \\ CP_0 \mapsto CP_0 \\ err \mapsto err \end{cases} \qquad \rho = \begin{cases} (en, \sigma(en)) \mapsto \top \\ (CP_0, \sigma(CP_0)) \mapsto (x = y) \\ (err, \sigma(err)) \mapsto (x = y) \end{cases}$$

$$|\varphi| = \begin{cases} en \mapsto \top \\ CP_0 \mapsto \exists x \,.\, (x = y \wedge x \geq 0) \\ err \mapsto \exists x \,.\, (x = y \wedge x = 13) \end{cases} \qquad \widehat{\varphi} = \begin{cases} en \mapsto \top \\ CP_0 \mapsto (y \geq 0 \wedge y \neq 13) \\ err \mapsto \bot \end{cases}$$

Fig. 4. Simulation relation between Q_0 and βP_0, and lifted invariants.

$$\tau_{Q_1} = \begin{cases} (en, CP_0) \mapsto (z' = 0) \\ (CP_0, CP_0) \mapsto (z < 12 \wedge z' = z + 1) \\ (CP_0, CP_1) \mapsto (z = 12 \wedge z' = z + 2) \\ (CP_1, CP_1) \mapsto (z > 12 \wedge z' = z + 1) \\ (CP_0, err) \mapsto \big((z' = z) \wedge (z' < 0 \vee z' = 13)\big) \end{cases} \qquad \sigma = \begin{cases} en \mapsto en \\ CP_0 \mapsto CP_0 \\ CP_1 \mapsto CP_0 \\ err \mapsto err \end{cases}$$

$$\rho = \begin{cases} (en, \sigma(en)) \mapsto \top \\ (CP_0, \sigma(CP_0)) \mapsto (y = z) \\ (CP_0, \sigma(CP_0)) \mapsto (y = z) \\ (err, \sigma(err)) \mapsto (y = z) \end{cases} \qquad \widehat{\pi} = \begin{cases} en \mapsto \top \\ CP_0 \mapsto (z \geq 0 \wedge z \neq 13) \\ CP_1 \mapsto (z \geq 0 \wedge z \neq 13) \\ err \mapsto \bot \end{cases}$$

Fig. 5. Simulation relation between Q_1 and Q_0, and lifted invariants.

Example 2. Suppose that P_0 (shown in Fig. 1(a)) evolved to a "lucky" program Q_0 (shown in Figs. 1(b), 2(b)) such that the counter jumps over the value "13": the new variable `y` appeared instead of `x`, and the program fragment corresponding to the looping edge (CP_0, CP_0) is replaced by `if (y==12) then {y=y+2} else {y++}`. More importantly, the property to hold in Q_0 is stronger than the one in P_1: in addition to be positive, y is restricted to be not equal to 13. CPG_{P_0} and CPG_{Q_0} are identical. Note that $Q_0 \not\preceq P_0$, $Q_0 \not\preceq \alpha P_0$, but $Q_0 \preceq \beta P_0$. Figure 4 shows: (1) transition relation τ_{Q_0}, (2) CPG-simulation between Q_0 and P_0 via the *identity* relation σ; (3) edge-simulation between Q_0 and βP_0 via ρ, (4) lifted inductive (but not safe) invariant $|\varphi|$ labeling each cutpoint in CP of Q_0, and (5) proof $\widehat{\varphi}$ of Q_0 obtained from $|\varphi|$ by some strengthening procedure. □

In order to obtain the inductive invariant $|\varphi|$ for Example 2, we first need to weaken $\widehat{\psi}$ (as in Example 1) to be an inductive invariant of βP_0. Weakening can be done, e.g., by replacing the labeling $\widehat{\psi}(err)$ by a formula $x = 13$. Then $|\varphi|$ can be strengthened to $\widehat{\varphi}$ using an induction-based model checker to become safe.

Example 3. Consider a loop-splitting optimization Q_1 of Q_0 (shown in Figs. 1(c) and 2(c) respectively) produced by inserting an `if`-conditional out of the `while`-loop and a renaming of `y` to `z`. Thus, an extra loop (and an extra cutpoint CP_1) appeared in Q_1, but both loops were simplified to contain only an increment `z++`. Note that $Q_1 \preceq Q_0$. Figure 5 shows: (1) transition relation τ_{Q_1}, (2) CPG-simulation between Q_1 and Q_0 via σ, (3) edge-simulation between for Q_1 and Q_0 via ρ, and (4) lifted inductive (and safe) invariant $\widehat{\pi}$ of Q_1. □

In the next section we elaborate on the way of computing simulation relations and lifting proofs, the results of which were demonstrated in Examples 2 and 3.

5 Property Directed Equivalence

Our key result is a new technique that exploits both, the *reusable specification* and the *relational specification*, to incrementally verify pairs of programs. We instantiate reusable specifications by the safe inductive invariants, and relational specification by the simulation relations. To the best of our knowledge, PDE is unique in a sense that all the competitors in the scope of FIV operate either by reusable or by relational specifications, but not by both.

Given an abstraction αP of P and a proof $\widehat{\psi}$ of P, we say that αP is $\widehat{\psi}$*-safe* iff $\widehat{\psi}$ is also a proof of αP. Not every abstraction of P is $\widehat{\psi}$-safe, but there might exist several $\widehat{\psi}$-safe abstractions of P of different precision, and the most precise one of those is P itself. Formally, it is reflected in the following definition.

Definition 5. *Given a program P and a proof $\widehat{\psi}$, an abstraction $\alpha P = \langle Vars, CP, en, err, E, \tau_\alpha \rangle$ of P is $\widehat{\psi}$-safe iff the following holds:*

$$\forall (u,v) \in E \, . \, \widehat{\psi}(u)(\vec{x}) \wedge \tau_\alpha(u,v)(\vec{x},\vec{x}') \implies \widehat{\psi}(v)(\vec{x}') \tag{9}$$

Definition 6. *Programs P and Q are $\widehat{\psi}$-equivalent iff there exists another program R such that $P \preceq R$, $Q \preceq R$, and $\widehat{\psi}$ is a proof of R.*

Note that Definition 6 allows R to be either P or Q, in cases when $\widehat{\psi}$ is a proof of P or Q, respectively. Similarly, R is allowed to be an abstraction of P or Q.

Example 4. Programs P_0 and Q_0 (shown in Fig. 1(a) and (b) respectively) are not $\widehat{\psi}$-equivalent, since we cannot find a $\widehat{\psi}$-safe abstraction of P_0 (αP_0 is $\widehat{\psi}$-safe, but $Q_0 \not\preceq \alpha P_0$, and βP_0 is not $\widehat{\psi}$-safe). Contrary to them, Q_0 and Q_1 (shown in Fig. 1(b) and (c) respectively) are $\widehat{\psi}$-equivalent, since we have shown in Example 3 that $Q_1 \preceq Q_0$. □

The FIV-problem for P, Q and $\widehat{\psi}$ can be formulated as establishing a $\widehat{\psi}$-equivalence between P and Q. In this paper, we want to provide not only a generic, but also an efficient solution to the FIV-problem. One crucial obstacle on the way towards efficiency is that the simulation synthesis in general requires more efforts for solving than needed to verify Q from scratch. However, PDE does not require Q to be simulated by the precise program P via some total $\langle \sigma, \rho \rangle$. Instead, PDE aims at finding a $\widehat{\psi}$-safe abstraction αP that simulates Q via some abstract $\langle \sigma, \rho_\alpha \rangle$. Detecting ρ_α is expected to be easier than detecting ρ and to have more chances to converge.

Theorem 1. *Given programs P, αP and Q, let $\widehat{\psi}$ be a proof of P, and αP be a $\widehat{\psi}$-safe abstraction of P. If $Q \preceq \alpha P$ then P and Q are $\widehat{\psi}$-equivalent.*

The tie that binds the abstraction and the simulation in Theorem 1 is the proof $\widehat{\psi}$. In practice, synthesis of αP and $\langle \sigma, \rho_\alpha \rangle$ benefits from the guidance by $\widehat{\psi}$. Furthermore, when discovered, $\langle \sigma, \rho_\alpha \rangle$ is directly used to migrate $\widehat{\psi}$ from P to Q. In the rest of the section, we elaborate on these routines in more detail.

5.1 Simulation Synthesis

In our previous work [16], we presented SimAbs, the first algorithm to synthesize simulation relations completely automatically. Given programs P and Q, SimAbs attempts to deliver a total simulation relation[3] $\langle \sigma, \rho \rangle$ between P and Q such that $Q \preceq_{\langle \sigma, \rho \rangle} P$. If such *concrete* simulation cannot be found, SimAbs iteratively performs abstraction-refinement reasoning to detect an *abstract simulation*, i.e., an abstraction αP of P that simulates Q via some $\langle \sigma, \rho_\alpha \rangle$.

However, the results of SimAbs are not always useful for PDE, since it does not provide any guarantees of the strength and the property preservation of αP. In particular, SimAbs can unadvisedly abstract away some important details of P, so αP becomes not $\widehat{\psi}$-safe, and Theorem 1 becomes inapplicable. In this section, we present a novel algorithm ASSI that guides the simulation discovery by the invariants of P. Furthermore, ASSI supports a more general case when $CPG_Q \preceq_\sigma CPG_P$, and σ is not necessarily the identity relation. We outline ASSI and highlight its distinguishing features in Algorithm 1.

Synthesizing a CPG-simulation. The algorithm starts (lines 2–9) with synthesizing a CPG-simulation σ. It maintains a temporary graph G which is expected to be equivalent to CPG_P and by the end of the algorithm to become a supergraph of CPG_Q. Thus, G is initiated by CPG_P, and in the first iteration, ASSI checks whether G is a supergraph of CPG_Q. If the check succeeds, CPG_P simulates CPG_Q via identity, and ASSI directly proceeds to synthesizing ρ.

If G is not a supergraph of CPG_Q then ASSI attempts to *grow* G by introducing redundant nodes and edges, thus ensuring that G remains equivalent to CPG_P. The method CloneLoops has a relatively straightforward meaning: it finds a node in G with a looping edge (u, u), creates a new node u' and an edge (u', u'). Finally, it clones all the outgoing edges of u: (u, v) to (u', v) and all incoming edges of u: (v, u) to (v, u'). The information that u' obtains after copying u is book-kept and further used to recurrently create σ.

Checking whether one graph is a supergraph of another one is reduced to checking CPG-simulation via the identity relation and in turn to checking validity of the $\forall\exists$-formula (6). However, in the worst-case scenario, the procedure of cloning loops may keep iterating forever. Therefore, in practice, it makes sense to bound the iterations either by using some maximal number of cloned loops or by a timeout (method CanGrow).

The algorithm gets another challenge when Q has multiple loops that would require cloning different loops in a branch-and-bound manner. In general, it may lead to establishing multiple possible CPG-simulations, and each of those could be used to further establish its own edge-simulation ρ. To simplify presentation, we do not consider this case in the paper and assume that it is a rather engineering question of enhancing ASSI with backtracking to support multiple simulations.

[3] σ is limited to be the identity relation in the original algorithm.

Algorithm 1. ASSI (Q, P, ψ)

```
   Input: programs $Q$ and $P$, inductive invariant $\psi$ of $P$
   Output: abstraction $\alpha P$, relation $\langle\sigma, \rho_\alpha\rangle$ such that $Q \preceq_{\langle\sigma,\rho_\alpha\rangle} \alpha P$
   Data: universal abstraction $\mathbb{U}$, temporary graph $G$
 1 $G \leftarrow CPG_P$;
 2 while $(\top)$ do                                   ▷ Synthesize $\sigma$
 3     if (isSupergraph($G$, $CPG_Q$)) then            ▷ Wait until $G$ is big enough
 4         $\sigma \leftarrow$GrowingHistory($CPG_P$, $G$);  ▷ Restore all changes in $G$ since $CPG_P$
 5         break;                                      ▷ And go to line 10
 6     if (CanGrow($G$)) then                          ▷ If $G$ does not cover $CPG_Q$
 7         $G \leftarrow$ CloneLoops($G$);             ▷ Try growing $G$ by cloning loops
 8     else                                            ▷ Until no more loops can be cloned
 9         return $\mathbb{U}, \varnothing$;           ▷ Or a timeout is exceeded
10 while $(\top)$ do                                   ▷ Use $\sigma$ to synthesize $\rho$
11     $\rho \leftarrow$ Guess($P, Q, \sigma$);        ▷ Guess some relation $\rho$ over variables at each cutpoint
12     if $(Q \preceq_{\langle\sigma,\rho\rangle} P)$ then  ▷ Use $\rho$ and $\sigma$ in (7) and check $\forall\exists$-validity
13         return $P, \langle\rho, \sigma\rangle$;     ▷ If $\rho$ is an edge-simulation then the algorithm terminates
14     else                                            ▷ If not, iteratively replace $P$ by some of its abstractions:
15         $P \leftarrow$ Abstract($P, \psi$);         ▷ Try $\alpha^{\psi}$ first, then $\alpha^{\psi\exists}$
```

Algorithm 2. PDE $(P, Q, \widehat{\psi})$

```
Input: Programs $P$ and $Q$, proof $\widehat{\psi}$ of $P$
Output: Verification result $res \in \{$Safe, Buggy$\}$, proof $\widehat{\varphi}$ of $Q$
$\alpha P, \langle\sigma, \rho_\alpha\rangle \leftarrow$ ASSI $(Q, P, \widehat{\psi})$;      ▷ Find $\alpha P$ such that $Q \preceq \alpha P$
if (isPsiSafe($\alpha P, \widehat{\psi}$)) then                          ▷ Check if (9) holds
    return $\langle$Safe, $\exists\langle\sigma, \rho_\alpha\rangle\widehat{\psi}\rangle$;   ▷ If $\alpha P$ is $\widehat{\psi}$-safe then lift the proof completely (Theorem 1)
else                                                       ▷ If not, attempt to lift the proof partially
    $|\psi| \leftarrow$ Weaken($\widehat{\psi}$);                  ▷ Weaken $\widehat{\psi}$ to $|\psi|$ such that $|\psi|$ is inductive for $\alpha P$
    $|\varphi| \leftarrow \exists\langle\sigma, \rho_\alpha\rangle|\psi|$;        ▷ Lift $|\psi|$ to $|\varphi|$ such that $|\varphi|$ is inductive for $Q$ (Lemma 2)
    return Verify($Q, |\varphi|$);                      ▷ Strengthen $|\varphi|$ to become safe for $Q$ (if possible)
```

Synthesizing an edge-simulation. The further reasoning of ASSI (lines 10–15) proceeds by finding an edge-simulation ρ. Note that at this point σ is already synthesized, and each pair of *CPG*-edges is fixed due to valid $\forall\exists$-formula (6). Now, ASSI has to iteratively decide validity of another set of $\forall\exists$-formulas (7). Each such formula requires a guessed relation ρ over *live* variables at each pair of cutpoints matched by σ. ASSI makes a guess based on similarity (ideally, equality) of the variable names.

In cases when there is an invalid formula among (7), ASSI attempts to lower the precision of P by abstracting some (preferably, minimal amount of) details away. Intuitively, the goal is to weaken τ_P which is placed on the right-hand-side of the $\forall\exists$-formula such that the formula becomes valid. In general, ASSI is parametrized by the method Abstract which performs such weakening automatically.

Due to an infinite number of possible abstractions, an arbitrary chosen implementation of Abstract might not converge. The most distinguishing feature of ASSI compared to SimAbs, is that it guides the whole process of edge-simulation synthesis by invariants. That is, an invariant ψ is not only plugged into the formulas (7), but also used to create the abstraction $\alpha^{\psi}P$ (defined in Lemma 1). In practice, $\alpha^{\psi}P$ dramatically weakens τ_P, and ASSI earns much higher number of valid $\forall\exists$-formulas than SimAbs earns from the existential abstraction $\alpha^{\exists}P$ [11]. The latter simply treats some program variables nondeterministically and does not change the transition relation itself.

Since depending on the semantic delta between P and Q, the simulation check between Q and $\alpha^{\psi}P$ is still not guaranteed to succeed. In such cases, the other variants of Abstract (e.g., the existential abstraction $\alpha^{\exists}$) might come in handy. In our implementation, we apply $\alpha^{\exists}$ to the result of a previous application of $\alpha^{\widehat{\psi}}$, thus delivering *not* $\widehat{\psi}$-safe abstractions of P (denoted $\alpha^{\widehat{\psi}\exists}P$).

Contrary to SimAbs, ASSI lacks a so called *Skolem-based refinement* that would attempt strengthening of abstract simulations, but still would not guarantee any success. But since ASSI is designed to deal with PDE, a necessary strengthening is performed on the level of proof generation. In the next Sect. 5.2, we show how weak abstractions could be still useful for lifting proofs partially.

5.2 Lifting Proofs Completely and Partially

The main practical importance of PDE is that it allows lifting a proof $\widehat{\psi}$ of program P directly to a proof $\widehat{\varphi}$ of program Q if there exists a $\widehat{\psi}$-safe abstraction of P that simulates Q (recall Theorem 1). In such a case, no additional analysis of Q is required unless one wants to eliminate existential quantifiers from the adapted proof. However, if the conditions of Theorem 1 are not met, we are still interested in accelerating the verification process for Q. In particular, if the detected abstraction αP of P is not $\widehat{\psi}$-safe, we still may be able to lift some (not safe, but) inductive invariant to be further strengthened by a Horn-clause-based model checker.

We address the problem of verifying Q using P and $\widehat{\psi}$. Our solution is outlined in Algorithm 2. PDE proceeds as follows. First (line 1) it invokes the two-steps procedure of ASSI: (1) obtaining a relation σ between cutpoints of P and Q via iterative growing of CPG_P and checking validity of the implication (6); (2) discovering an abstraction αP of P and a relation ρ_α such that $Q \preceq_{\langle \sigma, \rho_\alpha \rangle} \alpha P$.

The discovered abstraction αP is then checked for being $\widehat{\psi}$-safe (line 2). This is done by deciding validity of a set of implications (9) for each edge of CPG_P. If this check succeeds then the simulation relation ρ_α discovered by means of ASSI is combined with $\widehat{\psi}$ using existential quantification to obtain an inductive invariant $\exists\langle \sigma, \rho_\alpha \rangle \widehat{\psi}$ (a shortcut for the invariant defined in Lemma 2). By Theorem 1, $\exists\langle \sigma, \rho_\alpha \rangle \widehat{\psi}$ is also a *safe* inductive invariant, which entails that the program Q is safe.

If the abstraction αP delivered by ASSI is not $\widehat{\psi}$-safe then ρ_α cannot be directly used to lift invariants. But since $P \preceq \alpha P$ via the identity relation, $\widehat{\psi}$

can be weakened to become an inductive invariant $|\psi|$ of αP (line 5). Method Weaken can implement different methods including simple generation of the strongest postcondition (as in Example 5), or a counter-example-guided inductive weakening (method MkInd [15]) that constructs an inductive invariant out of a set of conjunctions of candidate formulas (also referred to as *lemmas*).

MkInd performs inductive weakening using an incremental SMT solver. MkInd assumes that each candidate invariant is represented by a conjunction of lemmas, and the weakening by itself is performed by dropping some lemmas from this conjunction. MkInd iterates over the set of *CPG*-edges in the Weak Topological Ordering [8] (in which inner loops are traversed before outer loops). For each edge, MkInd checks whether $\widehat{\psi}$ is inductive (i.e., formula (2) holds). If the check for some edge (u, v) fails, MkInd uses a counter-example provided by the SMT solver to identify all lemmas to be dropped from $\widehat{\psi}(v)$. Afterwards, the check is propagated to all the *CPG*-edges (v, w) outgoing from v. Effectiveness of MkInd requires the sets of candidate invariants to contain many small lemmas.

Example 5. Consider programs P_0 and Q_0 (shown in Fig. 1(a) and (b) respectively). Suppose, P_0 is verified and has a proof $\widehat{\psi}$ (shown in Fig. 3). Let us show how PDE operates in order to derive the proof $\widehat{\varphi}$ of Q_0 (envisioned in Fig. 4). First, PDE invokes ASSI to iteratively abstract P_0, e.g., to αP_0 and to βP_0 (both shown in Fig. 3) and check whether the abstraction simulates Q_0: the former does not, but the latter does. Second, PDE confirms that βP_0 is not $\widehat{\psi}$-safe and thus attempts to lift the proof partially.

The next step is to do Weaken and to obtain the inductive invariant $|\psi|$ of βP_0. For this, PDE exploits the efforts spent on checking that βP_0 is not $\widehat{\psi}$-safe. In particular, for all *CPG*-edges for which that check succeeded, the invariants at the pre- and post-states remain the same as specified by $\widehat{\psi}$ (i.e., $\widehat{\psi}(CP_0) = |\psi|(CP_0)$). But $\widehat{\psi}$ is broken for the edge (CP_0, err), i.e., the implication $(x \geq 0) \wedge \big((x < 0) \vee (x = 13)\big) \implies \bot$ is invalid. This means that $\bot$ is too strong to label *err* in $|\psi|$, and a weaker formula should be discovered. Following MkInd, the labeling $|\psi|$ of cutpoint *err* would be aggressively assigned to $\top$, which would in turn require re-verification of Q_0 from scratch.

Alternatively, it can be assigned to $|\psi|(\mathit{err}) = (x = 13)$, which is the strongest postcondition for $\tau_{\beta P_0}(CP_0, \mathit{err}) = (x < 0) \vee (x = 13)$ and precondition $|\psi|(CP_0) = (x \geq 0)$. It is easy to see that $|\psi|$ constitutes an inductive invariant $|\psi|$ of βP_0. Finally, $|\psi|$ is lifted to become inductive invariant $|\varphi|$ (shown in Fig. 4) of Q_0 using the already established abstract simulation ρ_α. □

The last bit in PDE is done by method Verify (line 7). At that stage, the partially lifted invariant $\exists\langle\sigma, \rho_\alpha\rangle|\psi|$ needs to be strengthened to finally become a proof of Q. Method Verify can exploit an off-the-shelf model checker as long as it is able to fight against the following two challenges. First, the model checker should deal with induction and avoid re-verifying Q from scratch. Second, the model checker should deal with existentially quantified variables of P and avoid expensive quantifier elimination.

5.3 Finding Inductive Invariants Without Quantifier Elimination

In this section, we focus on method Verify that strengthens an inductive invariant $\exists\langle\sigma,\rho_\alpha\rangle\psi$ for PDE. The key idea is based on the fact that adding invariants to the transition relation does not affect any behaviors of the program. In a nutshell, the invariants are extra constraints about pre- and post-states of each *CPG*-edge $(u,v)\in E$.

Given $(u,v)\in E$ and $\tau : E \to \mathit{Expr}(V(u)\cup V'(v))$, let $\widehat{\tau} : E \to \mathit{Expr}(V(u)\cup V'(v))$ denote the relation constrained by the invariants $\exists\langle\sigma,\rho_\alpha\rangle\psi$, i.e.:

$$\begin{aligned}\widehat{\tau}(u,v) = {} & \exists\vec{y}\,.\,\rho_\alpha\big(u,\sigma(u)\big)(\vec{x},\vec{y})\ \wedge\ \psi(u)(\vec{y})\ \wedge \tau(u,v)(\vec{x},\vec{x}')\wedge \\ & \exists\vec{y}'\,.\,\rho_\alpha\big(v,\sigma(v)\big)(\vec{x}',\vec{y}')\ \wedge\ \psi(v)(\vec{y}')\end{aligned} \tag{10}$$

It is easy to see that a program $\widehat{Q} = \langle \mathit{Vars}, \mathit{CP}, \mathit{en}, \mathit{err}, E, \widehat{\tau}\rangle$ is equivalent to program $Q = \langle \mathit{Vars}, \mathit{CP}, \mathit{en}, \mathit{err}, E, \tau\rangle$, and the proof $\widehat{\psi}$ of Q is sufficient for $\widehat{Q}$. However, the opposite is not true, i.e., a proof $\widehat{\psi}$ of $\widehat{Q}$ might not be sufficient for Q.

Theorem 2. *Given Q and inductive invariant $\exists\langle\sigma,\rho_\alpha\rangle\psi$ of Q, let $\widehat{Q}$ be as in (10). If $\widehat{\varphi}$ is the proof of $\widehat{Q}$ then $\varphi = \widehat{\varphi}\wedge\exists\langle\sigma,\rho_\alpha\rangle\psi$ is the proof of Q.*

Method Verify reduces the task of obtaining $\widehat{\varphi}$ to solving a system of Constrained Horn Clauses [20]. This system consists of the rules that have a form (1), (2) or (3). The quantifier elimination is done lazily inside the solving engine. Note that such a model-checking approach is also applicable in cases when $\exists\langle\sigma,\rho_\alpha\rangle\psi$ is not only inductive, but also safe. If so, a constant mapping $\widehat{\varphi}(u,v) = \top$ for any (u,v) is a solution for the Horn system, and solving terminates immediately.

5.4 Calculating the Change Impact

In case when PDE (and in turn Verify) cannot prove safety (i.e., fails to strengthen the inductive invariant), it generates a so called *change impact* – an indication whether the change of the code in a particular edge of the CPG_P broke the proof. Change impact can be calculated cheaply as a by-product of checking whether an abstraction αP of P is $\widehat{\psi}$-safe for the proof $\widehat{\psi}$.

Definition 7. *Given P, Q, a proof $\widehat{\psi}$ of P and abstraction αP of P such that $Q \preceq_{\langle\sigma,\rho_\alpha\rangle} \alpha P$, the change impact δ of program Q is a mapping $\delta : E_Q \to \{\top,\bot\}$ such that for each $(u,v)\in E_Q$:*

$$\delta(u,v) \equiv \begin{cases} \top & \text{if } \widehat{\psi}\big(\sigma(u)\big)(\vec{x})\wedge\tau_\alpha\big(\sigma(u),\sigma(v)\big)(\vec{x},\vec{x}') \implies \widehat{\psi}\big(\sigma(v)\big)(\vec{x}') \\ \bot & \text{else} \end{cases} \tag{11}$$

If calculated this way, the change impact is precise enough to indicate all *CPG*-edges that are responsible for a property violation. Together with a counter-example witnessing the property violation, the change impact is a step towards shrinking the search space of a possible bugfix.

Let us denote the set of edges $\Delta = \{(u,v) \in E_Q \mid \delta(u,v) = \bot\}$. In order to fix the given bug, the encoding τ of some of the *CPG*-edges from Δ must be rewritten, but the encoding τ of the edges in $E_Q \setminus \Delta$ can remain unchanged. In other words, program Q can be used to create a *partial program* Q_Δ that preserves the encoding of the edges $E_Q \setminus \Delta$ and contains *holes* to represent the absence of the encoding of Δ. Then, such a partial program Q_Δ is given as input to a program synthesizer, such as Sketch [32] to automatically find instantiations of the holes. In our future work we plan to integrate an automatic program repairer with PDE.

6 Evaluating ASSI and PDE

We built PDE on the top of the model checker UFO [1] and the simulation synthesizer SimAbs. UFO relies on LLVM to create verification conditions for the input programs that involves inlining procedures, lowering memory to registers, extracting a *CPG*-representation. UFO synthesizes a proof by running the PDR engine [23] implemented in Z3 [14].

We evaluate ASSI and PDE for benchmarks from SVCOMP[4]. We focus our attention only on safe programs, i.e., those for which it is possible to generate a proof $\widehat{\psi}$. We further consider a program transformation from an original version P to a transformed version Q. Finally, we use $\widehat{\psi}$ to (1) find an abstraction αP of P that simulates Q via some ρ_α, (2) check whether αP is $\widehat{\psi}$-safe, and (3) use $\widehat{\psi}$ and ρ_α to incrementally verify Q.

One of the essential applications of discovering simulation relations is proving correctness of program optimizations. The users often perform optimizations and refactoring manually, and usually end up with the semantically different programs. Similarly, optimizations are performed *silently*, by compilers. PDE is insensitive to the source of program transformation and could be applicable to both types of optimizations. In our experiments, we focused on compiler optimizations, as a larger base of benchmarks.

For evaluation, we used two non-trivial LLVM-optimizations: `indvars` and `licm`. The `indvars` (stands for *Canonicalize Induction Variables*) transforms the loops to have a single canonical induction variable initially assigned to zero and being incremented by one. The `licm` (stands for *Loop Invariant Code Motion*) detects loop invariants and moves them outside of the loop body. Note that the combination of these optimizations is aggressive, and does not necessarily preserve the loop structure of the given program. We considered 115 programs, for which UFO is able to discover a proof $\widehat{\psi}$ within a given timeout of 700 s, and the correspondent LLVM-optimizations. The size of the programs ranges from 91 to 2904 lines of code, and the syntactic delta between versions ranges from 3 to 345 instructions.

[4] Software Verification Competition, http://sv-comp.sosy-lab.org/.

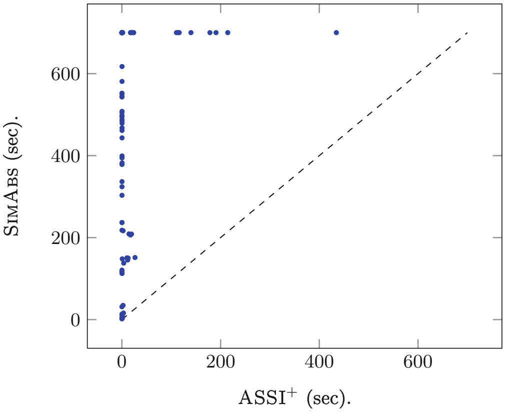


Fig. 6. Simulation synthesis by ASSI$^+$ compared to SimAbs.

Evaluating ASSI. We compared our novel algorithm ASSI$^+$ with SimAbs (Fig. 6). In our experiment[5], SimAbs delivers precise simulations only in 13 cases, and this result can be interpreted as the *absolute equivalence* between the original and the optimized programs. In 38 more cases, SimAbs ended up with an abstraction of P that simulates Q In the remaining 64 cases, SimAbs exceeds the timeout or diverged.

In contrast to SimAbs, ASSI$^+$ adds $\widehat{\psi}$ to the low-level $\forall\exists$-formulas and manipulates directly with the $\widehat{\psi}$-safe abstraction $\alpha^{\widehat{\psi}}P$ of P. These two improvements made the low-level $\forall\exists$-formulas smaller, as encoding of the original transition relation of P got replaced by more compact formulas representing $\widehat{\psi}$. All 115 experiments terminated. There were 39 $\widehat{\psi}$-safe abstractions (i.e., that are used to adapt the proof completely); 55 weaker abstractions (i.e., that are used to adapt the proof at least partially), and only 21 abstractions were trivial (i.e., too weak to adapt any invariant). Performance-wise, ASSI$^+$ was in order of magnitude faster than SimAbs, and in some cases outperformed its competitor by 2000**X**.

One can observe an interesting phenomenon that despite ASSI$^+$ never delivers concrete simulations, it is in general more precise than SimAbs. It can be explained by the fact that ASSI$^+$ is able to safely ignore some details of P that can break simulation synthesis in SimAbs. It is important to note that in cases when αP is trivial, ASSI$^+$ does not produce big overhead. In our experiments, the running time for such scenarios is less than 10 s.

Evaluating PDE. We compared the performance of PDE with the performance of the model checker UFO that verifies the optimized program from scratch (shown in the upper chart of Fig. 7). Provided with the proof and the abstract simulation, PDE outperformed UFO in 90 out of 115 cases. In the remaining cases, the performed optimizations dramatically simplified the program so it became easier to verify the optimized program from scratch.

[5] Full results are available at http://www.inf.usi.ch/phd/fedyukovich/niagara.

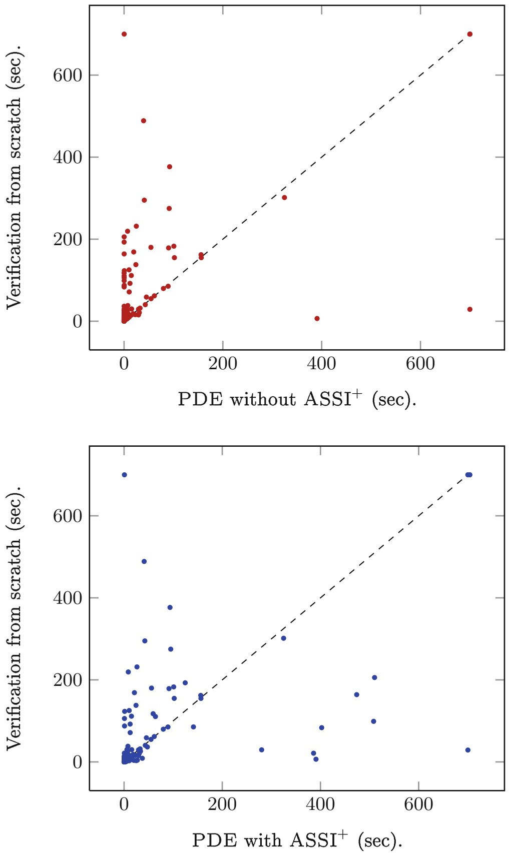


Fig. 7. Verification by PDE (with and without ASSI$^+$) compared to UFO.

Both simulation discovery and incremental verification (ASSI + PDE) were faster than UFO in 60 cases (shown in the lower chart of Fig. 7). This includes 1 case in which UFO exceeded timeout (i.e., PDE solved the problems that cannot be solved by UFO). In future, as a possible performance improvement, we may run ASSI + PDE and UFO in parallel, and terminate both processes whenever one of them returned a result. Thus, we can exploit benefits of incremental and non-incremental verification at the same time.

To summarize our case studies, we must mention that being an SMT-based framework, PDE currently supports only Linear Rational Arithmetic that makes it difficult to evaluate programs handling arrays, floating point arithmetic, bit-vectors and so on. PDE shown its potential to be the first working framework that is able to connect reusable and relational specifications of the versioned software, and we envision multiple improvements of its workflow in future.

7 Conclusion

In this paper, we formalized the concept of PDE that allows migrating safe inductive invariants across program transformations. We presented an algorithm ASSI for simulation relation synthesis with invariants and an algorithm PDE to address the FIV-problem using ASSI. We evaluated ASSI and PDE on the benchmarks from SVCOMP and LLVM-optimizations. It confirmed that in many cases when the absolute equivalence between programs cannot be proven, our approach is able to establish the property directed equivalence. In cases when the proof can be lifted only partially, our approach allows its further strengthening by means of a Horn-clause-based model checker.

References

1. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: a framework for abstraction- and interpolation-based software verification. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 672–678. Springer, Heidelberg (2012)
2. Backes, J., Person, S., Rungta, N., Tkachuk, O.: Regression verification using impact summaries. In: Bartocci, E., Ramakrishnan, C.R. (eds.) SPIN 2013. LNCS, vol. 7976, pp. 99–116. Springer, Heidelberg (2013)
3. Barrett, C.W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.D.: TVOC: a translation validator for optimizing compilers. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 291–295. Springer, Heidelberg (2005)
4. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011)
5. Berman, C.L., Trevillyan, L.H.: Functional comparison of logic designs for VLSI circuits. In: ICCAD, pp. 456–459. IEEE (1989)
6. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD, pp. 25–32. IEEE (2009)
7. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: ESEC/FSE, pp. 389–399. ACM (2013)
8. Bourdoncle, F.A.: Efficient chaotic iteration strategies with widenings. In: Bjørner, D., Broy, M., Pottosin, I.V. (eds.) Formal Methods in Programming and Their Applications. LNCS, vol. 735, pp. 128–141. Springer, Heidelberg (1993)
9. Burch, J.R., Singhal, V.: Tight integration of combinational verification methods. In: ICCAD, pp. 570–576. IEEE (1998)
10. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
11. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**(5), 1512–1542 (1994)
12. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: DAC, pp. 368–371. ACM (2003)

13. Conway, C.L., Namjoshi, K.S., Dams, D.R., Edwards, S.A.: Incremental algorithms for inter-procedural analysis of safety properties. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 449–461. Springer, Heidelberg (2005)
14. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
15. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Incremental verification of compiler optimizations. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 300–306. Springer, Heidelberg (2014)
16. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Automated discovery of simulation between programs. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR-20 2015. LNCS, vol. 9450, pp. 606–621. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48899-7_42
17. Gjomemo, R., Namjoshi, K.S., Phung, P.H., Venkatakrishnan, V.N., Zuck, L.D.: From verification to optimizations. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 300–317. Springer, Heidelberg (2015)
18. Godefroid, P., Lahiri, S.K., Rubio-González, C.: Statically validating must summaries for incremental compositional dynamic test generation. In: Yahav, E. (ed.) Static Analysis. LNCS, vol. 6887, pp. 112–128. Springer, Heidelberg (2011)
19. Godlin, B., Strichman, O.: Regression verification. In: DAC, pp. 466–471. ACM (2009)
20. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012)
21. Kawaguchi, M., Lahiri, S.K., Rebelo, H.: Conditional equivalence. Technical report MSR-TR-2010-119, Microsoft Research (2010)
22. Kesten, Y., Pnueli, A.: Control and data abstraction: the cornerstones of practical formal verification. STTT **2**(4), 328–342 (2000)
23. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 17–34. Springer, Heidelberg (2014)
24. Leino, K.R.M., Wüstholz, V.: Fine-grained caching of verification results. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 380–397. Springer, Heidelberg (2015)
25. Logozzo, F., Lahiri, S.K., Fähndrich, M., Blackshear, S.: Verification modulo versions: towards usable verification. In: PLDI, p. 32. ACM (2014)
26. Namjoshi, K.S.: Lifting temporal proofs through abstractions. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 174–188. Springer, Heidelberg (2002)
27. Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. LNCS, vol. 7935, pp. 304–323. Springer, Heidelberg (2013)
28. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI, pp. 83–94. ACM (2000)
29. Padon, O., Immerman, N., Shoham, S., Karbyshev, A., Sagiv, M.: Decidability of inferring inductive invariants. In: POPL, pp. 217–231. ACM (2016)
30. Paruthi, V., Kuehlmann, A.: Equivalence checking combining a structural SAT-solver, BDDs, and simulation. In: ICCD, pp. 459–464. IEEE (2000)

31. Sery, O., Fedyukovich, G., Sharygina, N.: Incremental upgrade checking by means of interpolation-based function summaries. In: FMCAD, pp. 114–121. IEEE (2012)
32. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS, pp. 404–415. ACM (2006)
33. Tristan, J., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for LLVM. In: PLDI, pp. 295–305. ACM (2011)
34. Yang, G., Khurshid, S., Person, S., Rungta, N.: Property differencing for incremental checking. In: ICSE, pp. 1059–1070. ACM (2014)