Verifiable Semantic Difference Languages

Thibaut Girka, David Mentré, Yann Régis-Gianas

Verifiable Semantic Difference Languages

Thibaut Girka
Mitsubishi Electric R&D Centre Europe
Rennes, France

David Mentré
Mitsubishi Electric R&D Centre Europe
Rennes, France

Yann Régis-Gianas
Univ Paris Diderot, Sorbonne Paris Cité, IRIF/PPS, UMR 8243 CNRS, PiR2, INRIA Paris-Rocquencourt
Paris, France

ABSTRACT

Program differences are usually represented as textual differences on source code with no regard to its syntax or its semantics. In this paper, we introduce semantic-aware difference languages. A difference denotes a relation between program reduction traces. A difference language for the toy imperative programming language Imp is given as an illustration.

To certify software evolutions, we want to mechanically verify that a difference correctly relates two given programs. *Product programs* and *correlating programs* are effective proof techniques for relational reasoning. A product program simulates, in the same programming language as the compared programs, a well-chosen interleaving of their executions to highlight a specific relation between their reduction traces. While this approach enables the use of readily-available static analysis tools on the product program, it also has limitations: a product program will crash whenever one of the two programs under consideration crashes, thus making it unsuitable to characterize a patch fixing a safety issue.

We replace product programs by *correlating oracles* which need *not* be expressed in the same programming language as the compared programs. This allows designing *correlating oracle languages* specific to certain classes of program changes and capable of relating crashing programs with non-crashing ones. Thanks to oracles, the primitive differences of our difference language on Imp can be assigned a verifiable semantics. Besides, each class of oracles comes with a specific proof scheme which simplifies relational reasoning for a well-specified class of relations. We also prove that our framework is at least as expressive as several Relational Hoare Logic variants by encoding them as correlating oracles, (re)proving soundness of those variants in the process. The entirety of the framework as well as its instantiations have been defined and proved correct using the Coq proof assistant.

ACM Reference Format:
Thibaut Girka, David Mentré, and Yann Régis-Gianas. 2017. Verifiable Semantic Difference Languages. In *Proceedings of PPDP'17, Namur, Belgium, October 9–11, 2017,* 12 pages.
https://doi.org/10.1145/3131851.3131870

PPDP'17, October 9–11, 2017, Namur, Belgium

ACM ISBN 978-1-4503-5291-8/17/10…$15.00
https://doi.org/10.1145/3131851.3131870

1 INTRODUCTION

Let x be a strictly positive natural number. Consider the following two (almost identical) imperative programs:

```
1 s = 0;
2 d = x;
3 while (d > 0) {
4   if (x % d == 0)
5     s = s + d;
6   d = d - 1;
7 }
8 s = s - x
```

```
1 s = 0;
2 d = x - 1;
3 while (d > 0) {
4   if (x % d == 0)
5     s = s + d;
6   d = d - 1;
7 }
8
```

The program P_1 (on the left) stores in s the sum of the proper divisors of the integer x. To that end, it iterates over the integers from x to 1 using the variable d and accumulates in the variable s the values of d that divide x. Finally, the last instruction of P_1 substracts x from s as it was wrongly added during the first iteration of the algorithm (a number is not a proper divisor of itself). Therefore, while P_1 is correct, it can be improved as P_2 (on the right) by two modifications of its source code, described as follows by the standard tool diff [12]:

```
2c2
< d = x;
---
> d = x - 1;
8d7
< s = s - x
```

This patch says that the assignment on Line 2 of P_1 is updated and that the assignment on Line 8 is removed. While the effect of this patch on the source code is obvious, can we characterize its effects on the semantics of P_1, for instance to prove that P_2 is indeed an improvement of P_1? Since textual differences are both low-level and unstructured, they seem inappropriate for such semantic-aware analysis. Even if in the daily practice of software development, textual differences are manually reviewed, applied as patches, composed, compared, merged and versioned, these operations make in general no sense in terms of program behaviors. Hence, the programmer has to manually interpolate the impact of textual changes to understand if these operations really implement the targeted software evolution. This interpolation process is error-prone, especially on large pieces of software.

What if the programmer was given semantically grounded and high-level *difference languages*? Using these languages, the intent behind patches could be expressed, mechanically checked and reviewed in a high-level and semantic-aware setting, paving the way for *certified evolution of software*. For example, a possible difference between P_1 and P_2 could be:

```
1 refactor wrt. store equivalence & optimize
```

Other examples of difference could range from basic program transformations (renaming, independent instruction commutations, ...) to more abstract ones (refactoring, extension or restriction of features, change in resource usage, bug fix, etc).

This paper introduces a general language-agnostic framework in which the semantics of difference languages can be defined and on which mechanically-checked reasoning can be performed. In this framework, differences are interpreted as relations between reduction traces. This interpretation domain includes both semantic equivalence relations (like observational equivalence, simulation, bisimulation, etc) and semantic inequivalence relations (like bugfixing, refinements, extension, etc). It also allows to compare non-terminating or crashing programs, or programs written in distinct programming languages. Finally, this domain is appropriate to interpret intentional differences (resources usage optimizations, policy enforcement, migration to new API versions, etc).

Relational reasoning between programs is notoriously difficult. Fortunately, deductive relational reasoning frameworks like relational Hoare Logic[6], refinement mappings[13] or bisimulations[14] provide proof techniques to derive interesting relations between programs. But, as noticed by Barthe et al. [1], "there is a lack of applicable program logics and tools for relational reasoning." The recent approaches based on product programs[1, 2, 5] and correlating programs[9, 15] have shown their practical strengths to build such missing tools by reducing the verification of relational properties to the verification of a property on a single program.

Roughly speaking, the basic principle of these approaches is to compare two programs using a third one which schedules the instructions of the two compared programs. The idea is to decompose the relational analysis into two steps: (i) find a product program that realizes a specific interleaving of the executions of the compared programs to highlight a specific relation between the configurations of their reduction traces; (ii) reason about the product program to prove that this relation actually holds. While finding the right interleaving requires a deep understanding of the relation between the two programs, the step (ii) is a non-relational proof of program correctness which can be done following standard techniques. For instance, a relevant program relating P_1 and P_2 is:

```
s_1 = 0; s_2 = 0;
d_1 = x; d_2 = x - 1;
if (d_1 > 0 && x % d_1 == 0) s_1 = s_1 + d_1;
if (d_1 > 0) d_1 = d_1 - 1;
while (d_1 > 0 || d_2 > 0) {
  if (d_1 > 0 && x % d_1 == 0) s_1 = s_1 + d_1;
  if (d_2 > 0 && x % d_2 == 0) s_2 = s_2 + d_2;
  if (d_1 > 0) d_1 = d_1 - 1;
  if (d_2 > 0) d_2 = d_2 - 1;
}
s_1 = s_1 - x;
```

The program P_{12} simulates the executions of P_1 and P_2. To maintain these simulations in two disjoint parts of the memory, the variable identifiers from P_1 are postfixed with _1 and those of P_2 with _2. To assert a relation between the configurations of P_1 and P_2, it suffices to mix variables from both sides in the same formula. For instance, proving that the postcondition "s_1 = s_2" holds suffices to prove that P_1 and P_2 produce the same output. The specific scheduling implemented by P_{12} makes this proof straightforward thanks to the loop invariant "d_1 = d_2 ∧ s_1 = s_2 + x". Since these programs are written in the same programming language as the compared programs, existing static analysis or deductive proof systems can be reused to exploit them.

Even though correlating programs and product programs appear as good representations for program differences when it comes to mechanical verification, comparison based on these technique suffers from an important expressiveness defect: comparing two programs with product or correlating programs only makes sense if none of them crashes. To illustrate this point, let us consider the situation where, before getting the correct version P_1 (on the right), a programmer had written an incorrect version P_0 (on the left):

```
s = 0;
d = x;
while (d >= 0) {
  if (x % d == 0)
    s = s + d;
  d = d - 1;
}
s = s - x
```

```
s = 0;
d = x;
while (d > 0) {
  if (x % d == 0)
    s = s + d;
  d = d - 1;
}
s = s - x
```

P_0 crashes because of a division by zero during the last iteration of the while-loop. The program P_1 does not suffer from this problem since its while-loop condition avoids the case where d is zero. The transformation from P_0 to P_1 is a typical *bugfix patch* in the dialect of software development. Because it executes the instructions of the two programs, the correlating or product programs for P_0 and P_1 will crash at some point which prevents us from proving any useful postcondition about it. Generally, if the programming language has poor support to handle internal errors, as it is the case with low-level languages like C, it seems difficult to tailor a product program generator for crashing programs. This expressivity weakness prohibits the reasoning on a large part of software evolution, e.g. bugfixes.

In this paper, we study an interpretation domain for differences inspired by product and correlating programs but which allows for crashing programs to be compared to other programs. Roughly speaking, our idea is based on *correlating oracles*, Coq programs which simulate a parallel execution of the compared programs but *at a meta-level*: a correlating oracle interleaves the execution of two small-step interpreters running the compared programs, not the instructions of the two compared programs. We formally show that this interpretation domain is at least as expressive as Relational Hoare Logic[6] and we use it to interpret the primitive differences of a toy difference language for Imp.

Contributions. In this paper, we present:

- a formal, general and language-agnostic definition of difference languages for deterministic programming languages (Section 3);
- a toy difference language for the Imp programming language (Section 3.2);
- the definition of an interpretation domain for difference based on correlating oracles (Section 4);
- the encodings of several variants of Relational Hoare Logic as correlating oracles which provides a new proof of their soundness (Section 4.2.6).

All these contributions have been mechanically verified[10] in the Coq proof assistant.

Preliminaries. The formal development of this paper has been conducted in Type Theory but we will sometimes abusively use the vocabulary of mathematics based on Set Theory in our statements and definitions. For instance, the expression "partial function" from A to B, written $A \twoheadrightarrow B$, must be understood as a function of type "$A \to$ option B". The domain of a partial function f is written $\mathrm{dom}(f)$. Similarly, we use the word "subset of A" to denote a predicate of type $A \to \mathbb{P}$, where $\mathbb{P}$ is the type for propositions. When some inductive type A defines a set of first-order terms, its definition will be given by a BNF grammar and we will use A both for the non-terminal and as a type. We also write $\overline{A}$ to range over vectors of A. The length of $\overline{A}$ is written $|\overline{A}|$. Most of the time, functions over A are silently lifted to $\overline{A}$ when the context suffices to deduce how. The notation f^n represents the n-th self-composition of a function f, i.e $\lambda x.f(\cdots(fx))$. Given a potentially infinite nonempty list ℓ, we write $\lceil \ell \rceil$ for the first element of ℓ.

2 OVERVIEW

To present the key ideas of this work, we consider the following two programs P_0 (on the left) and P_3 (on the right):

```
a = 0;
d = x;
while (d >= 0) {
  if (x % d == 0)
    a = a + d;
  d = d - 1;
};
while (a >= 0) {
  d = d + 1;
}
```

```
d = x;
s = 0;
while (d > 0) {
  if (x % d == 0)
    s = s + d;
  d = d - 1;
};
while (s >= 0) {
  d = d + 1;
}
```

Notice that P_0 crashes because of a division by zero during the last iteration of its first while-loop, while the execution of P_3 diverges without crashing.

What could be a relevant semantic difference between P_0 and P_3? Using the difference language presented later in this paper, a difference δ between P_0 and P_3 could be written:

```
fix crash & (
  rename a ↔ s;
  swap assignments at line 1;
  fix off-by-one at line 3
)
```

How should we read this difference? It is a superposition of two differences expressed at two distinct levels of abstraction. The difference on the left-hand-side of the superposition states that the program P_0 crashes while P_3 does not. The right-hand-side of the superposition provides more low-level details about the difference between P_0 and P_3 using a sequence of primitive differences whose composition connects P_0 to P_3. The differences of lines 2 and 3 correspond to basic local program transformations while the difference of line 4 states that an off-by-one crash is fixed at line 3.

What is the formal meaning of this difference? Even though the difference language is designed with the hope of being declarative enough to help the programmer intuitively understand it, it is also equipped with a formal non-ambiguous interpretation. More precisely, each difference represents a relation between reduction traces.

In our example, the interpretation $[\![\text{fix crash}]\!]$ relates all pair of traces $(\mathcal{T}, \mathcal{T}')$ so that $\mathcal{T}$ ends on a stuck configuration while $\mathcal{T}'$ does not. ($\mathcal{T}'$ is either infinite or ends on a valid final configuration.) The interpretation $[\![\text{rename a} \leftrightarrow \text{s}]\!]$ relates every pair of traces $(\mathcal{T}, \mathcal{T}')$ where the configurations of $\mathcal{T}'$ can be obtained from the configurations of $\mathcal{T}$ by renaming a into s. The interpretation $[\![\text{swap assignments at line 1}]\!]$ relates $(\mathcal{T}, \mathcal{T}')$ where $\mathcal{T}$ is the trace of a program P that has two independent assignments at Lines 1 and 2 and where $\mathcal{T}'$ is the trace of P' obtained from P by a swapping of Lines 1 and 2. The interpretation $[\![\text{fix off-by-one at line 3}]\!]$ relates $(\mathcal{T}, \mathcal{T}')$ where $\mathcal{T}$ is the trace of a program P that has a while-loop condition at line 3 whose last iteration provokes a crash and where $\mathcal{T}'$ is the trace of a program P' obtained from P by a modification of Line 3 which removes the crash at last iteration of the while-loop from $\mathcal{T}$ and allows it to continue its evaluation. The superposition operator is interpreted as an intersection between the relations which are the interpretations of both operands. The sequencing operator is interpreted as standard relation composition.

How can we verify that this difference indeed relates P_0 and P_3? Proving the soundness of δ amounts to prove that its interpretation relates the trace $\mathcal{T}(P_0)$ of P_0 and the trace $\mathcal{T}(P_3)$ of P_3:

Proposition 2.1 (δ relates P_0 and P_3).
$\mathcal{T}(P_0)\ [\![\textit{fix crash}]\!]\ \mathcal{T}(P_3)$
$\wedge\ (\exists \mathcal{T}_1\ \mathcal{T}_2,$
$\quad \mathcal{T}(P_0)\ [\![\textit{rename a} \leftrightarrow \textit{s}]\!]\ \mathcal{T}_1$
$\quad \wedge\ \mathcal{T}_1\ [\![\textit{swap assignments at line 1}]\!]\ \mathcal{T}_2$
$\quad \wedge\ \mathcal{T}_2\ [\![\textit{fix off-by-one at line 3}]\!]\ \mathcal{T}(P_3))$

To prove the right-hand-side, we must first find two witnesses for the intermediate traces $\mathcal{T}_1$ and $\mathcal{T}_2$. As the relations of a renaming and a swap assignments can be realized by applying program transformations, these two traces are easily expressed using the following two programs P_1 and P_2 and the relations hold by construction:

```
s = 0;
d = x;
while (d >= 0) {
  if (x % d == 0)
    s = s + d;
  d = d - 1;
};
while (s >= 0) {
  d = d + 1;
}
```

```
d = x;
s = 0;
while (d >= 0) {
  if (x % d == 0)
    s = s + d;
  d = d - 1;
};
while (s >= 0) {
  d = d + 1;
}
```

To conclude the proof of the right-hand-side, one must show that the relation $[\![\text{fix off-by-one at line 3}]\!]$ holds between the trace of P_2 and the trace of P_3. This is the complex part of the proof and where the *correlating oracles* enter the scene.

A possible plan to prove this result would be to analyze P_2 and P_3 independently: in a first part, we would show that P_2 has a while-loop condition on Line 3 and that the last iteration of this loop triggers a crash. In a second part, we would show that P_3 has a modified condition on Line 3 which allows to enter in the final infinite while-loop. Such a proof would be correct but there would

be a lot of redundancy between the two independent proofs about the behaviors of P_2 and P_3. A relational reasoning seems more appropriate to take benefit from the very similar structures of P_2 and P_3.

A product program would hardly help to perform this relational reasoning. Indeed, to show that the only difference between the traces of P_3 and P_2 is the absence of crash at last iteration in the trace of P_3, one is inclined to consider a product program that share the same control-flow in the spirit of the following one:

```
s_2 = 0; s_3 = 0;
d_2 = x; d_3 = 0;
while (d_2 >= 0 || d_3 > 0) {
  if (d_2 >= 0) {
    if (x_2 % d_2 == 0) s_2 = s_2 + d_2;
    d_2 = d_2 - 1
  };
  if (d_3 > 0) {
    if (x_3 % d_3 == 0) s_3 = s_3 + d_3;
    d_3 = d_3 - 1
  }
};
while (s_2 >= 0 || s_3 >= 0) {
  if (s_2 >= 0) d_2 = d_2 + 1;
  if (s_3 >= 0) d_3 = d_3 + 1;
}
```

Unfortunately, this product program crashes during the last iteration of its first while-loop, just like P_2. Hence, it cannot be used to prove something about the evaluation of P_3 after the loop. A product program executing P_3 before P_2 would not help either since the final infinite while-loop of P_3 would prevent the crash of P_2 to be exposed.

In Coq, the relation $[\![$ fix off−by−one at line 3 $]\!]$ is realized by a correlating oracle and this definition allows for a simple proof of this last goal:

Proposition 2.2. $\mathcal{T}(P_2)$ $[\![$ *fix off−by−one at line 3* $]\!]$ $\mathcal{T}(P_3)$

Proof. There exists a concrete state in the oracle language called FixOffByOneCrash which is compatible with the initial states of P_2 and P_3 and such that the invariant of FixOffByOneCrash is verified. By the correctness of this oracle language, the theorem holds. □

To understand this proof, it is necessary to understand (i) what a correlating oracle is and (ii) what kind of proof scheme for relational reasoning an oracle language provides. While those two points will be answered by this paper, the next two paragraphs attempt to give some intuition.

As a first approximation, a correlating oracle can be seen as a *bi-interpreter*, that is a program that interprets two programs at the same time. Just like a product program, an oracle *correlates* the executions of the two programs to help prove that a specific relation holds between their reduction traces. Unlike a product program though, the execution of a correlating oracle is not jeopardized by the crash of one of the compared programs. In addition, a correlating oracle has an internal state which morally contains the states of the two compared programs as well as information to dynamically maintain the desired correlation between the two executions.

We do not define an oracle for each proof but instead a collection of oracle languages. Each oracle language captures a certain form of relational reasoning: it is characterized by a specific correlation strategy and an invariant which holds for each pair of correlated configurations. Each oracle language comes with its proof of correctness which shows that once the invariant holds for two program configurations then the execution of the oracle realizes a relation between (a subset of) the configurations of the two programs. In general, this precise relation between the configurations of the two programs is sufficient to imply that a specific relation between the two program traces exists.

3 DIFFERENCE LANGUAGES

3.1 General definitions

3.1.1 Programming languages. In our Coq library, the general definitions about difference languages (as well as the ones about oracle languages) can be instantiated for any deterministic programming language. A programming language is implemented in Coq as a record of the following signature:

Definition 3.1. A *programming language definition* is a 7-uple $(\mathcal{R}, \mathcal{G}, \mathcal{S}, \mathcal{E}, \mathcal{F}, \mathcal{P}, \mathcal{I})$ such that:

- $\mathcal{P}$ is the type for *program abstract syntax trees*
- $\mathcal{G}$ is the type for the *static evaluation environment*
- $\mathcal{S}$ is the type for the *dynamic evaluation environment*
- $\mathcal{R}$ is the type of the *final result* of program evaluation
- $\mathcal{F}$ is the *result extraction* function of type $\mathcal{S} \twoheadrightarrow \mathcal{R}$
- $\mathcal{I}$ is an *initialisation function* of type $\mathcal{P} \to \mathcal{G} \times \mathcal{S}$
- $\mathcal{E}$ is the *evaluation function* of type $\mathcal{G} \to \mathcal{S} \twoheadrightarrow \mathcal{S}$

and it must also fulfills the following requirements:

- $\forall g : \mathcal{G}, \forall s : \mathcal{S}, \mathcal{E}\ g\ s$ is undefined if $\mathcal{F}\ s$ is defined.
- $\mathcal{S}$ and $\mathcal{G}$ must be equipped with decidable equivalences.

A programming language is therefore defined as usual by a syntax, the inhabitants of type $\mathcal{P}$, and a small-step operational semantics specified by an evaluation function $\mathcal{E}$ which transforms a dynamic state of type $\mathcal{S}$ with additional information stored in a global static environment of type $\mathcal{G}$. This function $\mathcal{E}$ is partial: some dynamic states cannot be reduced, either because they are stuck or because they are final, in which case a final result of type $\mathcal{R}$ can be extracted. The programming language is deterministic because for each program there is a unique initial state returned by $\mathcal{I}$ and because $\mathcal{E}$ returns at most one result.

Let $\mathcal{L}$ be a programming language definition. For each type identifier T, we generally use the corresponding lowercase letter t to range over the inhabitants of this type. Thus, we will write p to range over $\mathcal{P}$, g to range over $\mathcal{G}$ and s to range over $\mathcal{S}$. We now introduce generic standard definitions about operational semantics.

Definition 3.2. The type for *configurations* of $\mathcal{L}$ is $C = \mathcal{G} \times \mathcal{S}$.

A decidable equivalence over configurations is immediately built on top of the decidable equivalences over $\mathcal{G}$ and $\mathcal{S}$.

Definition 3.3. The trace of a program p is coinductively defined as the following potentially infinite nonempty list of configurations:

$$
\begin{array}{rcll}
\mathcal{T}(p) & = & \mathcal{T}^\star\ (\mathcal{I}(p)) & \\
\mathcal{T}^\star\ (g,s) & = & (g,s) & \text{if } \mathcal{E}\ g\ s \text{ is undefined} \\
\mathcal{T}^\star\ (g,s) & = & (g,s) \cdot \mathcal{T}^\star\ (g, \mathcal{E}\ g\ s) & \text{otherwise}
\end{array}
$$

Let us now formally introduce standard notions about traces.

Definition 3.4. The trace $\mathcal{T}(p)$ is *crashed* (resp. *converged*) if:

- $\mathcal{T}(p)$ is finite, and
- if s is the final state of $\mathcal{T}(p)$, then $\mathcal{F}(s)$ is undefined (resp. defined).

Definition 3.5. The set of *reduction traces* T of $\mathcal{L}$ is $\{\mathcal{T}(p) \mid p \in \mathcal{P}\}$ and the set R of *relations over reduction traces* is defined as the binary predicates over T, i.e. the inhabitants of $\mathrm{T} \to \mathrm{T} \to \mathbb{P}$.

Amongst the relations between reduction traces, we are particularly interested in γ-*correlations*, because we will see that several interesting differences are naturally specified by this kind of relations.

Definition 3.6. Let γ be a relation over configurations. A relation ρ in R is *a γ-correlation* if it relates pair of traces from which two dense subsequences of configurations are pointwise related by γ i.e. $\mathcal{T}_1 \rho \mathcal{T}_2$ implies that $\mathcal{T}_1 \overset{\gamma}{\sim} \mathcal{T}_2$, where $\overset{\gamma}{\sim}$ is coinductively defined as:

$$\textsc{Step}\ \frac{|\bar{c}_1| + |\bar{c}_2| > 0 \qquad \lceil \bar{c}_1 \cdot \mathcal{T}_1 \rceil \gamma \lceil \bar{c}_2 \cdot \mathcal{T}_2 \rceil \qquad \mathcal{T}_1 \overset{\gamma}{\sim} \mathcal{T}_2}{\bar{c}_1 \cdot \mathcal{T}_1 \overset{\gamma}{\sim} \bar{c}_2 \cdot \mathcal{T}_2} \qquad \textsc{Stop}\ \frac{c_1 \gamma c_2}{c_1 \overset{\gamma}{\sim} c_2}$$

This definition of γ-correlation deserves several remarks. First, for $\mathcal{T}_1 \overset{\gamma}{\sim} \mathcal{T}_2$ to hold, the head configurations of $\mathcal{T}_1$ and $\mathcal{T}_2$ must be related by γ. If each trace contains only one configuration, the γ-correlation holds. Otherwise, there must exist two strict prefixes $\bar{c}_1$ and $\bar{c}_2$ of $\mathcal{T}_1$ and $\mathcal{T}_2$ that can be skipped to get to another pair of γ-correlated traces. Second, the hypothesis $|\bar{c}_1| + |\bar{c}_2| > 0$ forces one of the prefixes to not be empty. This condition ensures that a correlation between two *finite* traces covers both traces. It might be surprising that we do not force both prefixes not to be empty. This is actually necessary to be able to express Lamport's and Abadi's *stuttering*[13]. In a word, the stuttering mechanism allows the γ-correlation to relate a configuration of one trace to several configurations of the other trace. That way, a γ-correlation can denote a local desynchronization between the two compared traces. We come back on this notion in Section 4.2. A γ-correlation is said to be (locally) *lockstep* if $|\bar{c}_1| = 1$ and $|\bar{c}_2| = 1$ for a given instantiation of the rule (Step).

As a final remark, notice that our notion of γ-correlation does not require the relation to cover infinitely many configurations from infinite traces: if $\mathcal{T}_2$ is infinite, a γ-correlation may choose not to skip configurations from $\mathcal{T}_1$ by consuming an infinite number of configurations from $\mathcal{T}_2$ (taking $|\bar{c}_1| = 0$ and $|\bar{c}_2| > 0$ to satisfy the first hypothesis of the rule (Step)). An additional *fairness* condition could be added in the definition of γ-correlations to avoid that case: typically, we could force the hypotheses $|\bar{c}_1| > 0$ and $|\bar{c}_2| > 0$ to alternate infinitely often. Yet, we will show in Section 6 that fairness is not always appropriate for our purpose.

3.1.2 *Syntax and semantics of difference languages.*

Definition 3.7. A *difference language definition* for a programming language $\mathcal{L}$ is a pair $(\mathcal{D}, [\![\bullet]\!])$ where

- $\mathcal{D}$ is the type of difference abstract syntax trees;
- $[\![\bullet]\!]$ is the interpretation function of type $\mathcal{D} \to \mathrm{R}$.

We write δ for a difference ranging over $\mathcal{D}$.

Definition 3.8. δ is *sound* for p_1 and p_2 if $\mathcal{T}(p_1)\ [\![\delta]\!]\ \mathcal{T}(p_2)$.

3.1.3 *Generic differences.*

We introduce several families of differences that can be generically derived from any programming language definition $\mathcal{L}$.

Definition 3.9. Let γ be an equivalence relation over configurations. A difference δ is a *refactoring* with respect to γ if for every pair of programs (p_1, p_2) for which δ is sound, $\mathcal{T}(p_1)$ and $\mathcal{T}(p_2)$ are γ-correlated.

With that definition of refactoring, we expect to capture meaning-preserving transformations applied to converging, diverging and stuck programs. For instance, if a γ-correlation ρ only relates the final configurations of converging programs with respect to the equivalence of stores, then ρ corresponds to the usual observational equivalence. With that notion, one can also capture transformations that preserve bugs or transformations that turn an infinite computation into another infinite computation which shares with it an infinite number of correspondences. In general, this notion of parameterized refactoring allows for the specification of "points of interest" which must be equivalent in the source program and its refactored version.

Definition 3.10. A difference δ is an *optimization* (reducing execution time) if for every pair of programs (p_1, p_2) for which δ is sound, $\mathcal{T}(p_1)$ and $\mathcal{T}(p_2)$ are finite and the length of $\mathcal{T}(p_2)$ is smaller than the length of $\mathcal{T}(p_1)$.

Definition 3.11. A difference δ is a *crash fix* if for every pair of programs (p_1, p_2) for which δ is sound, $\mathcal{T}(p_1)$ is crashed and $\mathcal{T}(p_2)$ is not crashed.

3.2 A difference language for Imp

3.2.1 *Imp programming language, syntax and semantics.*

The type $\mathcal{P}_{\text{Imp}}$ for programs in Imp is exactly the type for commands C, whose definition is in Figure 1. Commands include a **skip** statement, an assignment of an integer computed from an arithmetic expression, a sequencing operator, a branching statement and a while-loop. Using the **assert** keyword, the programmer can in addition claim that a given boolean expression evaluates to **true** at some point of the program. Expressions are divided into arithmetic and boolean expressions. They enjoy a standard syntax.

The operational semantics of Imp needs no global static environment $\mathcal{G}_{\text{Imp}}$ (which is therefore defined as unit) but a dynamic state $\mathcal{S}_{\text{Imp}}$ which is made of the configurations c specified by Figure 2. A configuration is composed of a store M and a continuation κ, whose definitions are standard. The configurations are also the results of evaluation $\mathcal{R}_{\text{Imp}}$ and the function $\mathcal{F}_{\text{Imp}}$ is defined as follows:

$$\mathcal{F}_{\text{Imp}}\,(M, \kappa) = \begin{cases} (M, \kappa) & \text{if } \kappa = \textbf{halt} \\ \textbf{undefined} & \text{otherwise} \end{cases}$$

The initialization function $\mathcal{I}_{\text{Imp}}$ is

$$\mathcal{I}_{\text{Imp}}\, C = (\bullet, C \cdot \textbf{halt})$$

The evaluation function $\mathcal{E}_{\text{Imp}}$ is defined as $\mathcal{E}_{\text{Imp}}\ ()\ c = \mathcal{I}_{\text{step}}\, c$ where $\mathcal{I}_{\text{step}}$ is specified by cases on the shape of the current continuation in Figure 3. These small-step evaluation rules are standard and they make use of two standard big-step evaluation judgments for arithmetic and boolean expressions, which are omitted.

		COMMANDS	
C	::=	**skip**	Inaction
	|	$x = e$	Assignment
	|	$C; C$	Sequence
	|	**if** (b) C **else** C	Conditional
	|	**while** (b) C	Loop
	|	**assert** (b)	Assertion
		COMMAND CONTEXTS	
$\mathbb{C}$	::=	$[]$	Context hole
	|	$\mathbb{C}; C$	On the left of sequence
	|	$C; \mathbb{C}$	On the right of sequence
	|	**if** (b) $\mathbb{C}$ **else** C	In the then-branch
	|	**if** (b) C **else** $\mathbb{C}$	In the else-branch
	|	**while** (b) $\mathbb{C}$	In the loop body
		ARITHMETIC EXPRESSIONS	
e	::=	n	Integer literal
	|	x	Read
	|	$e \oplus e$	Arithmetic operation
$\oplus$	::=	$+ \mid - \mid * \mid / \mid \%$	Arithmetic operators
		BOOLEAN EXPRESSIONS	
b	::=	**true**	True
	|	**false**	False
	|	$!b$	Negation
	|	$b \;\&\&\; b$	Conjunction
	|	$b \;\|\|\; b$	Disjunction
	|	$e = e$	Equality test
	|	$e \leq e$	Inequality test

Figure 1: Syntax of IMP.

		CONFIGURATIONS	
c	::=	(κ, M)	Configuration
		STORES	
M	::=	$\bullet$	Empty store
	|	$M[x := n]$	Bind
		CONTINUATIONS	
κ	::=	**halt**	End
	|	$C \cdot \kappa$	Next command

Figure 2: IMP configurations.

3.2.2 ΔIMP, a difference language for IMP.

Syntax. We now introduce ΔIMP, a toy difference language for IMP. The syntax for ΔIMP is described in Figure 4. A difference δ can be either a primitive difference δ_p, a composition of two differences $\delta; \delta$ or a superposition of two differences $\delta \,\&\, \delta$.

The syntax for primitive differences enumerates a collection of *builtin* differences. This choice of primitives is *adhoc* and there is no guarantee that they match all the software changes that can happen in a real software development. Finding a more complete set of primitives is left as future work and discussed in Section 6.

Nevertheless, we have defined four categories of changes, depending on their level of abstraction. The two categories named *syntactic refactorings* and *syntactic changes* contain primitive differences that can be expressed as program transformations. The next two categories, *extensional changes* and *abstract changes* relate two programs by exploiting a proof that a specific relation holds between their configurations during the evaluation.

Syntactic refactorings are program transformations that preserve the semantics of the source program for a given notion of program equivalence. They include (i) any renaming with respect to a bijection between their variable identifiers, (ii) any swap between two consecutive independent assignments found at a program point characterized by a context $\mathbb{C}$ and (iii) any swap between the branches of a conditional statement provided that the condition of this statement is negated in the target program.

Syntactic changes are program transformations that modify the meaning of the source program. The difference **fix off-by-one at** $\mathbb{C}$ is a program transformation that applies to a while-loop whose last iteration crashes and that modifies its condition to avoid this last buggy iteration. The difference **fix with defensive condition at** $\mathbb{C}$ is a local program transformation that inserts a conditional statement at a source program location characterized by $\mathbb{C}$ which precedes a statement C that triggers a crash. This conditional statement makes the evaluation of the target program avoid the evaluation of the statement C. The difference **change values of** $\overline{x}$ is a program transformation which modifies the assignments of any variable in $\overline{x}$ in the source program provided that this variable has no influence on the control flow.

Extensional changes are modifications of the source program that are not necessarily instances of a program transformation but for which a proof can be exploited to show that a specific relation holds between the two programs' traces. The difference **ensure equivalence at** $\mathbb{C}$ relates two equivalent subprograms located at $\mathbb{C}$ in the two programs. This difference exploits a proof of extensional equivalence to show that the target program is a refactoring of the source program. The difference **assume** $\{P\}$**ensure**$\{Q\}$ relates two programs for which a relational Hoare logic proof validates the precondition P and the postcondition Q. As said in the introduction, by referring to the variables of both programs in P and Q, such a proof establishes that a specific relation holds between the reduction traces of two *a priori* inequivalent programs.

The category of abstract changes corresponds to the generic differences that we introduced in Section 3.1.3.

Semantics. As said in the Section 2, the interpretation of a difference is a relation between traces. The interpretation of composite differences is straightforward : the interpretation of a sequence is a

$$
\begin{array}{rcll}
\mathcal{I}_{\text{step}}(\textbf{skip} \cdot \kappa, M) & = & (\kappa, M) & \\
\mathcal{I}_{\text{step}}(x = e \cdot \kappa, M) & = & (\kappa, M[x := n]) & \text{where } M \vdash e \Downarrow n \\
\mathcal{I}_{\text{step}}((C_1; C_2) \cdot \kappa, M) & = & (C_1 \cdot (C_2 \cdot \kappa), M) & \\
\mathcal{I}_{\text{step}}(\textbf{if } (b)\ C_1 \textbf{ else } C_2 \cdot \kappa, M) & = & (C_1 \cdot \kappa, M) & \text{where } M \vdash b \Downarrow \textbf{true} \\
\mathcal{I}_{\text{step}}(\textbf{if } (b)\ C_1 \textbf{ else } C_2 \cdot \kappa, M) & = & (C_2 \cdot \kappa, M) & \text{where } M \vdash b \Downarrow \textbf{false} \\
\mathcal{I}_{\text{step}}(\textbf{while } (b)\ C \cdot \kappa, M) & = & (C \cdot (\textbf{while } (b)\ C \cdot \kappa), M) & \text{where } M \vdash b \Downarrow \textbf{true} \\
\mathcal{I}_{\text{step}}(\textbf{while } (b)\ C \cdot \kappa, M) & = & (\kappa, M) & \text{where } M \vdash b \Downarrow \textbf{false} \\
\mathcal{I}_{\text{step}}(\textbf{assert } (b) \cdot \kappa, M) & = & (\kappa, M) & \text{where } M \vdash b \Downarrow \textbf{true}
\end{array}
$$

Figure 3: Small-step operational semantics of Imp.

Composite differences

$$
\begin{array}{rcll}
\delta & ::= & \delta_p & \text{Primitive} \\
 & | & \delta; \delta & \text{Composition} \\
 & | & \delta \,\&\, \delta & \text{Superposition}
\end{array}
$$

Primitive differences

$$
\begin{array}{rcl}
\delta_p & ::= & \underline{\textit{Syntactic refactorings}} \\
 & | & \textbf{rename } \overline{x \leftrightarrow y} \\
 & | & \textbf{swap assign at } \mathbb{C} \\
 & | & \textbf{swap branches at } \mathbb{C} \\
 & & \underline{\textit{Syntactic changes}} \\
 & | & \textbf{fix off-by-one at } \mathbb{C} \\
 & | & \textbf{fix with defensive condition at } \mathbb{C} \\
 & | & \textbf{change values of } \overline{x} \\
 & & \underline{\textit{Extensional changes}} \\
 & | & \textbf{ensure equivalence at } \mathbb{C} \\
 & | & \textbf{assume } \{P\}\textbf{ensure}\{Q\} \\
 & & \underline{\textit{Abstract changes}} \\
 & | & \textbf{refactor with respect to } \gamma \\
 & | & \textbf{crash fix} \\
 & | & \textbf{optimize}
\end{array}
$$

Figure 4: Syntax of ΔImp.

relation composition and the interpretation of a superposition is an intersection of relations.

$$
\begin{array}{rcl}
[\![\delta]\!] & \in & \mathrm{R} \\
[\![\delta_1; \delta_2]\!] & = & \lambda \mathcal{T}_1 \mathcal{T}_2.\ \exists \mathcal{T}_3.\ \mathcal{T}_1 [\![\delta_1]\!] \mathcal{T}_3 \wedge \mathcal{T}_3 [\![\delta_2]\!] \mathcal{T}_2 \\
[\![\delta_1 \,\&\, \delta_2]\!] & = & \lambda \mathcal{T}_1 \mathcal{T}_2.\ \mathcal{T}_1 [\![\delta_1]\!] \mathcal{T}_2 \wedge \mathcal{T}_1 [\![\delta_2]\!] \mathcal{T}_2
\end{array}
$$

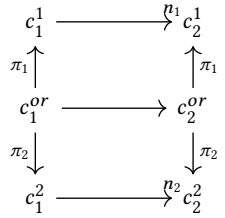
The interpretation of the primitive differences of the category of abstract changes have been defined in Section 3.1.3. For the other categories, we interpret each primitive difference using a specific language of correlating oracles described in the next Section 4.

4 CORRELATING ORACLES

Correlating oracles provide generic relational reasoning principles. As said in the overview, they reduce the proof that a relation holds between two programs' reduction traces to a proof that an invariant holds between the initial configurations of these programs. In this section, we give general definitions about correlating oracles and we present a collection of correlating oracles languages on Imp.

4.1 General definitions

Roughly speaking, a correlating oracle is a Coq program which simulates a specific parallel execution of two compared programs in such way that it highlights a stream of pairs of configurations which verify a given relation γ. As shown in the next diagram, these pairs of configurations are retrieved from the oracle state using two projections π_1 and π_2. Each execution step of the oracle may correspond to zero or more steps in each of the two programs (but it must not be steady on both sides). By iterating this process, the complete (possibly infinite) reduction trace of the oracle is meant to realize a γ-correlation between the two compared programs.



Of course, for this to be true, the γ-correlation must really hold between the two compared program traces, which is in general wrong for an arbitrary pair of program configurations. Hence, an additional ingredient is needed for this story to make sense: the *oracle invariant*. Each oracle language comes with a declared invariant about its internal state. Once this invariant is proved to hold for an internal oracle state initialized for a given pair of program configurations, the oracle definition guarantees that the oracle execution realizes a γ-correlation between the two programs traces.

To understand how this proof principle works in more technical terms, let us now introduce the formal definition of oracle languages.

Definition 4.1 (Oracle language definition). Given two language definitions $\mathcal{L}_1$ and $\mathcal{L}_2$, an *oracle language definition between* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ is a 6-uple $(\mathbb{O}, \mathbb{G}, \mathbb{S}, \pi_1, \pi_2, \mathbb{I})$ such that:

- $\mathbb{O}$ is a type for dynamic states;
- $\mathbb{G}$ is a type for static states;
- $\mathbb{S}$ is an interpretation function of type $\mathbb{G} \times \mathbb{O} \twoheadrightarrow \mathbb{O}$;
- π_1 (resp. π_2) is a projection function of type $\mathbb{O} \to C_{\mathcal{L}_1}$ (resp. $\mathbb{O} \to C_{\mathcal{L}_2}$);
- $\mathbb{I}$ is an invariant of type $\mathbb{G} \times \mathbb{O} \to \mathbb{P}$;

with the following additional requirements:

(R0) $\mathbb{S}$ is $\mathbb{I}$-safe i.e.

$$\forall g : \mathbb{G}, \forall o : \mathbb{O}, \mathbb{I}(g, o) \Rightarrow (g, o) \in \mathrm{dom}(\mathbb{S})$$

(R1) $\mathbb{I}$ is preserved by $\mathbb{S}$ i.e. $\forall g : \mathbb{G}, \forall o : \mathbb{O}$,

$$\mathbb{I}(g, o) \Rightarrow \mathbb{I}(g, \mathbb{S}(g, o))$$

(R2) $\mathbb{S}$ predicts well i.e. $\forall g : \mathbb{G}, \forall o : \mathbb{O}$,

$$\mathbb{I}(g, o) \Rightarrow \begin{cases} \exists n_1 n_2, \begin{cases} \pi_1(o') & = & \mathcal{E}^{n_1}_{\mathcal{L}_1}(\pi_1(o)) \\ \pi_2(o') & = & \mathcal{E}^{n_2}_{\mathcal{L}_2}(\pi_2(o)) \\ n_1 + n_2 & > & 0 \end{cases} \\ \text{or } \pi_1(o') \text{ is final } \wedge\, \pi_2(o') \text{ is final} \end{cases}$$

where $o' = \mathbb{S}(g, o)$.

In addition, we need the following definition:

Definition 4.2. An oracle state s is *initialized for a static state* g *and two configurations* c_1 *and* c_2, which we write $c_1 \leftarrow o \rightarrow c_2$, if $\mathbb{I}(g,\ o)$ holds, $\pi_1(o) = c_1$ and $\pi_2(o) = c_2$.

Let us explain the requirements one by one. The requirement (R0) guarantees that the invariant is sufficient for the interpretation function to evaluate with no error. The requirement (R1) ensures that $\mathbb{I}$ is indeed an invariant of the oracle interpretation function $\mathbb{S}$. Combined, these two requirements make sure that the evaluation of the oracle never goes wrong. The requirement (R2) is the most complex one. It ensures that the (potentially infinite) sequence of configuration pairs obtained by projection of the oracle internal state realizes some γ-correlation. To that end, (R2) imposes that the projected configurations are actual configurations of the two compared programs and that these configurations respect their order of appearance in the reduction traces. The subcases of (R2) cover the different behaviors of $\mathbb{S}$ with respect to the termination of each program: $\mathbb{S}$ must progress inside at least one non-terminated trace and only if both traces are finite and traversed, $\mathbb{S}$ is allowed not to progress. Thus, in the case of two finite traces $\mathcal{T}_1$ and $\mathcal{T}_2$, the requirement (R2) implies that the function $\mathbb{S}$ covers both traces in fewer than $|\mathcal{T}_1| + |\mathcal{T}_2|$ iterations:

Theorem 4.3 (Finite traces are covered). *Let O be an oracle language definition. For every pair of finite traces $\mathcal{T}_1$ and $\mathcal{T}_2$, for every dynamic state s such that $\lceil \mathcal{T}_1 \rceil \leftarrow s \rightarrow \lceil \mathcal{T}_2 \rceil$, if s' is $\mathbb{S}^{|\mathcal{T}_1|+|\mathcal{T}_2|}(s)$, then $\pi_1(s') \in dom(\mathcal{F})$ and $\pi_2(s') \in dom(\mathcal{F})$*

One might wonder why we do not require some sort of fairness property on oracles to be able to cover infinite traces too. Although that kind of property can indeed be shown for some specific oracle languages, we cannot require fairness in the general case because such requirement would rule out some oracles that we would want to define. It is still an open question to find a fairness property that would be adequate in the general case. This point is discussed in Section 6.

Theorem 4.4 (An oracle realizes a γ-correlation). *Let O be an oracle language definition. For every pair of finite traces $\mathcal{T}_1$ and $\mathcal{T}_2$, for every dynamic state o such that $\lceil \mathcal{T}_1 \rceil \leftarrow o \rightarrow \lceil \mathcal{T}_2 \rceil$, then $\mathcal{T}_1 \overset{\gamma}{\sim} \mathcal{T}_2$ where $c_1 \gamma c_2$ iff $\exists o : \mathbb{O}, \exists g : \mathbb{G}, \mathbb{I}(g, s) \wedge \pi_1(o) = c_1 \wedge \pi_2(o) = c_2$*

4.2 Correlating oracle languages on Imp

By lack of space, we cannot present all our oracle languages on Imp formally so we sketch their definitions, focusing on their most subtle aspects. The interested reader can take a look at the Coq development to get fully formal definitions for these oracle languages.

4.2.1 Renaming. The oracle language for *renamings* is the simplest of our oracle languages: given a bijection ϕ between two sets of identifiers, an oracle of this language realizes a γ-correlation where $c_1 \gamma c_2$ iff $c_2 = \phi(c_1)$. A renaming is a refactoring with respect to the equivalence relation $\lambda c_1 c_2.\exists \phi, c_2 = \phi(c_1)$.

The dynamic state $\mathbb{O}$ of this oracle only contains a pair (c_1, c_2) which corresponds to the current configurations of the two compared traces. The static state $\mathbb{G}$ contains the bijection ϕ. The interpretation function $\mathbb{S}$ follows a lockstep correlation strategy:

$$\mathbb{S}(\phi, (c_1, c_2)) = \begin{cases} (\mathcal{E}(c_1), \mathcal{E}(c_2)) & \\ (c_1, c_2) & (c_1, c_2) \notin \mathrm{dom}(\mathcal{E})^2 \end{cases}$$

The projections are defined as $\pi_1(c_1, c_2) = c_1$ and $\pi_2(c_1, c_2) = c_2$. The invariant $\mathbb{I}$ is $\lambda(c_1, c_2).\phi(c_1) = c_2$. The proof that the requirements hold is immediate.

4.2.2 Assignments and branches swapping. The oracle language for *independent assignments swapping* realize a γ-refactoring where γ is the equivalence of stores. The correlation steps it realizes are lockstep except when their evaluations reach the program point denoted by a given context $\mathbb{C}$. At that point, the intermediate configurations of the two programs' reductions are not γ-correlated since their stores may temporarily differ.

The dynamic state $\mathbb{O}$ contains both configurations but also some contextual information to determine if the programs' evaluations have reached the context $\mathbb{C}$. In that case, the evaluation function $\mathbb{S}$ performs several reduction steps in each trace to fully evaluate the original command of the first program and the transformed command in the second program. If one of these evaluations gets stuck, the oracle starts to stutter on the crashed configuration of the corresponding trace. Otherwise, the lockstep strategy is followed again until $\mathbb{C}$ is reached another time.

This time, the invariant of the oracle must not only imply the relation γ but it must also maintain the internal consistency of the oracle: if the oracle has not reached the context $\mathbb{C}$, the invariant simply states that the two current configurations are related by γ and that the contextual information is consistent with the current program continuations; if the oracle has reached a stuck state, the invariant ensures that the two configurations are indeed stuck. The oracle language for *Branches swapping* is defined similarly.

4.2.3 Modification of output values. Given a set of modified variables $\overline{x}$, an oracle in the oracle language for *modifications of*

$$\textsc{Control-Skip}\ \frac{}{\textbf{skip} \equiv_{\overline{x}} \textbf{skip}} \qquad \textsc{Control-Seq}\ \frac{C_1^1 \equiv_{\overline{x}} C_1^2 \qquad C_2^1 \equiv_{\overline{x}} C_2^2}{C_1^1; C_2^1 \equiv_{\overline{x}} C_1^2; C_2^2}$$

$$\textsc{Control-If}\ \frac{C_1^1 \equiv_{\overline{x}} C_1^2 \qquad C_2^1 \equiv_{\overline{x}} C_2^2 \qquad \forall\, y,\, y \in vars(b) \Rightarrow y \notin \overline{x}}{\textbf{if}\ (b)\ C_1^1\ \textbf{else}\ C_2^1 \equiv_{\overline{x}} \textbf{if}\ (b)\ C_1^2\ \textbf{else}\ C_2^2}$$

$$\textsc{Control-While}\ \frac{C^1 \equiv_{\overline{x}} C^2 \qquad \forall\, y,\, y \in vars(b) \Rightarrow y \notin \overline{x}}{\textbf{while}\ (b)\ C^1 \equiv_{\overline{x}} \textbf{while}\ (b)\ C^2}$$

$$\textsc{Control-Assert}\ \frac{\forall\, y,\, y \in vars(b) \Rightarrow y \notin \overline{x}}{\textbf{assert}\ (b) \equiv_{\overline{x}} \textbf{assert}\ (b)} \qquad \textsc{Control-Same-Assign}\ \frac{\forall\, y,\, y \in vars(e) \Rightarrow y \notin \overline{x}}{x = e \equiv_{\overline{x}} x = e}$$

$$\textsc{Control-Diff-Assign}\ \frac{y \in \overline{x} \qquad e_1 \text{ and } e_2 \text{ have the same variables and divisor expressions}}{y = e_1 \equiv_{\overline{x}} y = e_2}$$

$$\textsc{Control-Halt}\ \frac{}{\textbf{halt} \equiv_{\overline{x}} \textbf{halt}} \qquad \textsc{Control-Cmd}\ \frac{C_1 \equiv_{\overline{x}} C_2}{C_1 \cdot \kappa_1 \equiv_{\overline{x}} C_2 \cdot \kappa_2}$$

Figure 5: Sharing the same control with distinct output.

output values realizes a lockstep γ-correlation where γ relates configurations whose stores only differ on the values of $\overline{x}$.

For the sake of simplicity, the invariant is stronger than γ: it relates configurations whose stores indeed differ only on values but whose continuations may differ only on the assignments of modified variables. In other words, two programs related by these oracles share the same control-flow which is not influenced by modified variables. Modified assignments must feature the same set of variable identifiers and divisor expressions as the original assignments. This syntactic criterion is sufficient to avoid different crashing behaviors between the two compared programs. The judgment "$C_1 \equiv_{\overline{x}} C_2$" defined in Figure 5 captures this notion and is used in the invariant.

4.2.4 Defensive guard and off-by-one crash fix. Given a context $\mathbb{C}$ denoting a source code location where a crash happens after a bounded number of reduction steps in a program P_1, the oracle for the *insertion of a defensive guard* is meant to capture a common defensive programming pattern to fix a bug. This oracle relates P_1 with a program P_2 which behaves like P_1 until $\mathbb{C}$ is reached. After this point, P_2 differs from P_1 : while P_1 executes a command C_1 which triggers a crash, P_2 executes a command **if** (b) C_1 **else** C_2 where b evaluates to **false** to execute C_2 instead of C_1. Hence, if C_2 does not contain itself a command which crashes and if the other commands in the continuation do not crash, the relation realized by the oracle is a crash fix.

The dynamic state of this oracle has two successive forms: (i) until the execution reaches the context $\mathbb{C}$, the oracle follows a lockstep correlation strategy and therefore only contains the two configurations and some contextual information to track the current execution point; (ii) when the context $\mathbb{C}$ is reached, the oracle checks if the guard b evaluates to **true**, and in that case, the oracle continues with the same form of dynamic state to follow a lockstep correlation strategy; otherwise if the guard evaluates to **false**, the oracle executes P_1 until the crash happens and starts to stutter on this stuck configuration while continuing the execution of P_2.

The invariant of this oracle essentially ensures that the guard is indeed **false** when the execution of the command of P_1 at the context $\mathbb{C}$ crashes. It also guarantees that the two programs coincide outside $\mathbb{C}$. The oracle language for *corrections of off-by-one crashes* follows a similar structure.

4.2.5 Replacement by an equivalent subprogram. An oracle for *a replacement by an equivalent subprogram* realizes a γ-correlation where γ is the equivalence over memory stores. The correlation is lockstep until the executions reach a given context $\mathbb{C}$ and then, the two distinct commands C_1 and C_2 at this program point are skipped by the correlation to move to a pair of configurations from which the lockstep correlation continues.

The static environment of this oracle contains a proof of equivalence that C_1 and C_2 are terminating and observationally equivalent. As in the previous examples, the dynamic state contains both configurations and contextual information to determine when the context $\mathbb{C}$ is reached. The termination of both commands allows the oracle to execute them in one silent step.

4.2.6 Minimal Relational Hoare Logic. Minimal Relational Hoare Logic (MRHL) is a program logic to reason about two terminating programs executed in lockstep. The rules of Figure 6 define a judgment $\vdash \{P\}\, C_1 \sim C_2\, \{Q\}$ which establishes that under a precondition that can mix the variables of C_1 and C_2, if the commands C_1 and C_2 converge, then the resulting stores satisfy the postcondition Q. These rules have roughly the same shape as standard Hoare Logic and require both programs to have exactly the same control structure.

The oracle for a proof Π of MRHL realizes a lockstep γ-correlation where γ relates pair of stores that satisfy an MRHL judgment appearing in Π. Intuitively, an oracle encoding Minimal RHL should relate the lockstep execution of the two programs under consideration with the RHL proof of some bi-property on those programs. This is done by maintaining a list of RHL proof terms relating each pair of statements from both programs' continuations. In some sense, we are giving a small-step operational semantics to the RHL proof, in a way that is compatible with the semantics of the pair of programs to be related. The following diagram illustrates the shape of the dynamic state of the oracle and the two projections π_1 and π_2:

$$\begin{array}{c} (C_1^1 \cdot \kappa_1, M_1) \\ \pi_1 \uparrow \\ ((C_1^1 \cdot \kappa_1, M_1), (C_1^2 \cdot \kappa_2, M_2), M, P, \vdash \{P\}\, C_1^1 \sim C_1^2\, \{Q\} \cdot \overline{\Pi}) \\ \pi_2 \downarrow \\ (C_1^2 \cdot \kappa_2, M_2) \end{array}$$

In that oracle language, the static state of an oracle is a formula Q corresponding to the post-condition of the MRHL judgment, while

$$\frac{}{\vdash \{P\}\ \textbf{skip}\ \sim\ \textbf{skip}\ \{P\}}\ \text{R-Skip}$$

$$\frac{P \Rightarrow b_1 \wedge b_2}{\vdash \{P\}\ \textbf{assert}\ (b_1)\ \sim\ \textbf{assert}\ (b_2)\ \{P\}}\ \text{R-Assert}$$

$$\frac{}{\vdash \{P[e_2/x_2][e_1/x_1]\}\ x_1 = e_1\ \sim\ x_2 = e_2\ \{P\}}\ \text{R-Assign}$$

$$\frac{\vdash \{P\}\ C_1^1\ \sim\ C_1^2\ \{Q\} \qquad \vdash \{Q\}\ C_2^1\ \sim\ C_2^2\ \{R\}}{\vdash \{P\}\ C_1^1; C_2^1\ \sim\ C_1^2; C_2^2\ \{R\}}\ \text{R-Seq}$$

$$\frac{\begin{array}{c}\vdash \{P \wedge b_1\}\ C_1^1\ \sim\ C_1^2\ \{Q\} \\ \vdash \{P \wedge \neg b_1\}\ C_2^1\ \sim\ C_2^2\ \{Q\} \qquad P \Rightarrow b_1 = b_2\end{array}}{\vdash \{P\}\ \textbf{if}\ (b_1)\ C_1^1\ \textbf{else}\ C_2^1\ \sim\ \textbf{if}\ (b_2)\ C_1^2\ \textbf{else}\ C_2^2\ \{Q\}}\ \text{R-If}$$

$$\frac{\vdash \{P \wedge b_1\}\ C_1\ \sim\ C_2\ \{P\} \qquad P \Rightarrow b_1 = b_2}{\vdash \{P\}\ \textbf{while}\ (b_1)\ C_1\ \sim\ \textbf{while}\ (b_2)\ C_2\ \{P \wedge \neg b_1\}}\ \text{R-While}$$

$$\frac{\vdash \{P\}\ C_1\ \sim\ C_2\ \{Q\} \qquad P' \Rightarrow P \qquad Q \Rightarrow Q'}{\vdash \{P'\}\ C_1\ \sim\ C_2\ \{Q'\}}\ \text{R-Sub}$$

Figure 6: Minimal Relational Hoare Logic

the dynamic state is a tuple $(c_1, c_2, M, P, \overline{\Pi})$ where c_1 is the configuration c_1 of the left program, c_2 is the configuration of the right program, M is the disjoint union of the stores of c_1 and c_2, P is a formula corresponding to the precondition of the next proof term, and a list $\overline{\Pi}$ of proof terms. The transition function $\mathbb{S}$ makes use of a recursive function ψ defined in Figure 7 that gives a small-step operational semantics to the proof term.

The invariant to be maintained states that (i) the left and right configurations are separable—that is they do not have variables in common—, (ii) that the union store is indeed the disjoint union of the two configurations' stores, (iii) that it satisfies the precondition of the first proof of the list of proofs, (iv) that this list of proof terms indeed corresponds to the programs under consideration, and (v) that each postcondition in the proof list implies the precondition of the next proof (or the target post-condition for the last proof item). In particular, on a final configuration, the invariant states that the union store satisfies the final postcondition. Together with the facts that this oracle language cannot represent crashing programs, and that an oracle is required to make progress, this fact allows us to prove the soundness of Minimal RHL.

4.2.7 Core Relational Hoare Logic with self-composition. Core Relational Hoare Logic (CRHL) is an extension of Minimal Relational Hoare Logic in which additional rules permit the independent execution of basic instructions and conditionals of one program without forcing lockstep execution, thus significantly enhancing the expressive power of the program logic.

CRHL extends MRHL with rules to reason about only one of the programs at a time. The rules for the left program are:

$$\frac{\vdash \{P\}\ \textbf{skip}\ \sim\ C\ \{Q\} \qquad P \Rightarrow b}{\vdash \{P\}\ \textbf{assert}\ (b)\ \sim\ C\ \{Q\}}\ \text{R-Assert-L}$$

$$\frac{\vdash \{P\}\ \textbf{skip}\ \sim\ C\ \{Q\}}{\vdash \{P[e/x]\}\ x = e\ \sim\ C\ \{Q\}}\ \text{R-Assign-L}$$

$$\frac{\vdash \{P \wedge b\}\ C_1\ \sim\ C\ \{Q\} \qquad \vdash \{P \wedge \neg b\}\ C_2\ \sim\ C\ \{Q\}}{\vdash \{P\}\ \textbf{if}\ (b)\ C_1\ \textbf{else}\ C_2\ \sim\ C\ \{Q\}}\ \text{R-If-L}$$

$$\frac{\vdash \{P\}\ C_1\ \sim\ \textbf{skip}\ \{Q\} \qquad \vdash \{Q\}\ C_2\ \sim\ C\ \{R\}}{\vdash \{P\}\ C_1; C_2\ \sim\ C\ \{R\}}\ \text{R-Seq-Skip-L}$$

It should be noted that those are not exactly the same rules as presented in the original paper[3] because the (R-Assign-) rule of this paper seems unsound to us and some rules were missing.

Those rules are further extended with self-composition:

$$\frac{\vdash \{P\}\ C_1; C_2\ \{Q\}}{\vdash \{P\}\ C_1\ \sim\ C_2\ \{Q\}}\ \text{R-SelfComp}$$

Encoding Core RHL with an oracle language. CRHL is trickier to encode in our framework than MRHL. Indeed, the main difficulty is that some rules may either refer to actual sub-statements or an "imaginary" **skip** statement used in CRHL to encode the absence of statements: (R-Assign-L), (R-Assert-L) and (R-Seq-Skip-R),as well as their symmetric counterparts, introduce an "imaginary" **skip** in place of the right (respectively left) program to encode an empty sub-program. Therefore, any rule applicable to a **skip** on either side may now refer to either an actual or an "imaginary" **skip**. Those rules are (R-Skip), (R-Sub), (R-Assign-L), (R-Assign-R), (R-Assert-L), (R-Assert-R), (R-If-L), and (R-If-R), and may need to be handled differently whether they apply to "imaginary" **skip** or not. In addition, (R-Sub) and (R-Skip) may be applied to imaginary **skip** commands in both sides at the same time, in which case they do not represent any computation in the compared programs. Such a tree does not appear in the state of an oracle: it is "unfolded" within a single oracle step instead.

All of this implies that the list of proof terms have to be annotated to distinguish between proofs related two actual sub-programs and proofs for which one command is an "imaginary" **skip**. This is further complicated by self-composition, which is handled by two additional markers distinguishing between a single Hoare proof-term of the left program and one of the right program.

Once these adjustments made, the invariant has exactly the same shape as for MRHL, and the proof of soundness of CRHL with self-composition follows MRHL's closely.

4.2.8 Extended RHL. Extended Relational Hoare Logic is the final variant of RHL presented by Barthe et al[3]. It allows to replace any sub-program by some other extensionally equivalent program. While we have no reason to doubt this can also be encoded in our framework, the unsufficiently formal presentation of the original paper, together with the fact this new rule is inherently extensional, decided us not to formally prove it. Intuitively, this would probably involve the same kind of reasoning as in the oracle language for replacements of equivalent programs.

$$\psi(M, \frac{}{\vdash \{P\}\ \mathbf{skip} \sim \mathbf{skip}\ \{P\}}\ \text{R-Skip}, \overline{\Pi}) = (M, P, \overline{\Pi})$$

$$\psi(M, \frac{}{\vdash \{P'\}\ x_1 = e_1 \sim x_2 = e_2\ \{P\}}\ \text{R-Assign}, \overline{\Pi}) = (M[x_1 := n_1][x_2 := n_2], P, \overline{\Pi}) \quad \text{where } M \vdash e_1 \Downarrow n_1 \text{ and } M \vdash e_2 \Downarrow n_2$$

$$\psi(M, \frac{P \Rightarrow b_1 \wedge b_2}{\vdash \{P\}\ \mathbf{assert}\ (b_1) \sim \mathbf{assert}\ (b_2)\ \{P\}}\ \text{R-Assert}, \overline{\Pi}) = (M, P, \overline{\Pi})$$

$$\psi(M, \frac{\Pi_1 \quad \Pi_2}{\vdash \{P\}\ C_1^1; C_2^1 \sim C_1^2; C_2^2\ \{Q\}}\ \text{R-Seq}, \overline{\Pi}) = (M, P, \Pi_1 \cdot \Pi_2 \cdot \overline{\Pi})$$

$$\psi(M, \frac{\Pi_1 \quad \Pi_2 \quad P \Rightarrow b_1 = b_2}{\vdash \{P\}\ \mathbf{if}\ (b_1)\ C_1^1\ \mathbf{else}\ C_2^1 \sim \mathbf{if}\ (b_2)\ C_1^2\ \mathbf{else}\ C_2^2\ \{Q\}}\ \text{R-If}, \overline{\Pi}) = (M, P \wedge b_1, \Pi_1 \cdot \overline{\Pi}) \quad \text{where } M \vdash b_1 \Downarrow \mathbf{true}$$

$$\psi(M, \frac{\Pi_1 \quad \Pi_2 \quad P \Rightarrow b_1 = b_2}{\vdash \{P\}\ \mathbf{if}\ (b_1)\ C_1^1\ \mathbf{else}\ C_2^1 \sim \mathbf{if}\ (b_2)\ C_1^2\ \mathbf{else}\ C_2^2\ \{Q\}}\ \text{R-If}, \overline{\Pi}) = (M, P \wedge \neg b_1, \Pi_2 \cdot \overline{\Pi}) \quad \text{where } M \vdash b_1 \Downarrow \mathbf{false}$$

$$\psi(M, \frac{\Pi_1 \quad P \Rightarrow b_1 = b_2}{\vdash \{P\}\ \mathbf{while}\ (b_1)\ C_1 \sim \mathbf{while}\ (b_2)\ C_2\ \{P \wedge \neg b_1\}}\ \text{R-While}, \overline{\Pi}) = (M, P \wedge b_1, \Pi_1 \cdot \overline{\Pi}) \quad \text{where } M \vdash b_1 \Downarrow \mathbf{true}$$

$$\psi(M, \frac{\Pi_1 \quad P \Rightarrow b_1 = b_2}{\vdash \{P\}\ \mathbf{while}\ (b_1)\ C_1 \sim \mathbf{while}\ (b_2)\ C_2\ \{P \wedge \neg b_1\}}\ \text{R-While}, \overline{\Pi}) = (M, P \wedge \neg b_1, \overline{\Pi}) \quad \text{where } M \vdash b_1 \Downarrow \mathbf{false}$$

$$\psi(M, \frac{\Pi_1 \quad P' \Rightarrow P \quad Q \Rightarrow Q'}{\vdash \{P'\}\ C_1 \sim C_2\ \{Q'\}}\ \text{R-Sub}, \overline{\Pi}) = \psi(M, \Pi_1, \overline{\Pi})$$

Figure 7: Small-step operational semantics of Minimal RHL proofs.

5 RELATED WORK

Comparison of inequivalent programs. Even the notion of program equivalence itself is subject to many variations depending on what is observed about their evaluation: definitional, observational, intentional, experimental, bisimulation-based equivalences and many more have been intensively studied. The existence of encodings of these equivalences as oracle languages is left as future work. On the contrary, few general frameworks deal with inequivalent program comparison. Amongst them, refinement mappings[13] and minimal relational Hoare logic[6] capture the strict subset of difference languages whose predictions skip no instruction. They are too concrete to relate two different sorting algorithms for instance. Mutual summaries[11] abstract away procedure implementations leading to more abstract comparison of procedures. Nonetheless, thanks to their dynamic state, oracles can dynamically choose a specific comparison at each call site while mutual summaries cannot. As said in the introduction, product programs[1, 2, 5] are effective representations of correlations which allow to reduce a relational analysis to a standard proof of program. The idea of using oracles draws inspiration from product programs but oracles handle nontermination and crash, which seems difficult to achieve, if not possible, with product programs. A recent extension of product programs, namely probabilistic product programs[4], conducts relational reasoning in programming languages enjoying some form of nondeterminism while our framework is limited to deterministic languages.

Semantic patches and refactoring tools. Coccinelle's semantics patches[7] are source code transformations specified in a language called SmPL and designed to perform collateral evolutions in system code. As source code transformations, these patches work at the syntactic level but they try to abstract away as many details as possible to augment their applicability: the process that matches the source code with the patch takes the program control-flow into account thanks to temporal logic formulas. Our approach is more general because we can model differences between programs that cannot be expressed by static source code transformations. Indeed, the oracle can exploit the dynamic information stored in its internal state to generate predictions that are context-dependent.

The implementation of refactoring tools is an active research topic in software engineering. Except for a renaming refactoring tool based on CompCert[8], none of these tools is mechanically certified and unfortunately, none is exempt from bugs as shown by a recent study[20].

Differential static analysis. Differential static analysis[19, 21] typically ensure the preservation of some properties through the evolution of software[17, 18] or try to infer relations between two close versions of the same program[9, 15, 16]. In this work, we do not focus on the automatic verification of differences but on the design of an expressive framework to manually reason about them in a proof assistant. *Differential Symbolic Execution*[16] is close to our approach since it characterizes program differences generated by summaries-directed symbolic bi-interpreters. DSE exploits functional deltas and partition-effects deltas: these deltas are differences written in a low-level difference language expressed by execution paths and variable mappings. Other primitive differences may help turn low-level differences into higher-level ones. Consider:

```
if (x > 0)
  return 1;
else if (x < 0)
  return -1;
else return 0;
```

```
if (y > 0)
  return 1;
else if (y <= 0)
  return -1;
```

Thanks to rename x ↔ y, DSE could identify more paths:

```
rename x ↔ y;
P1: (x == 0), RETURN=0, P2: (y == 0), RETURN=-1
```

6 LIMITATIONS AND FUTURE WORK

Covering of infinite traces by an oracle. In general, we do not require oracles to be fair (even if in practice, our oracles for Imp are). Roughly speaking, a fair oracle never stutters an infinite number of times on the same configuration (unless this configuration is final). If we required fairness, some interesting program evolutions would not be representable using oracles. For instance, to interpret a difference to fix an unwanted infinite loop in a program that is nonterminating by design (typically a server), an oracle forced to progress infinitely often in the unwanted infinite loop would also be constrained to stutter an infinite number of times on a configuration replacing the infinite loop in the second program. Such an oracle would therefore not be fair. Finding a notion of fairness compatible with correlation is therefore left to future work.

Non deterministic programming languages. Most of our definitions generalize smoothly to non deterministic programming languages. In that setting, a difference δ is sound for p_1 and p_2 if its interpretation relates any trace of p_1 to any trace of p_2. However, the design of oracle step functions $\mathbb{S}$ to realize such a relation seems complex to us because the correlation strategy should be devised for any nondeterministic choice made by each program. Maybe prophecy variables introduced by Lamport and Abadi[13] or some form of speculative execution, i.e. a piece of information about the future of the compared programs evaluations, will help us tackle this limitation in the future.

Less ad hoc differences. The toy difference language ΔImp for Imp we presented in this paper has a rather *ad hoc* and limited set of difference primitives. Anyway, the completeness of the difference primitives offered by a difference language seems inherently impossible to achieve since its designer cannot predict all the relations that will interest developers in the future. However, the development of ΔImp leads us to think that interesting combinators exist to actually program differences. For instance, all the definitions of the oracle languages interpreting local program transformation share the same mechanism to determine if the location of the transformation is reached by the oracle. That mechanism could be generalized as a combinator to compose several independent of such differences as a single one. Another important step to invent less *ad hoc* differences is to apply our framework to a realistic programming language and to try to design differences which formally capture the evolution of a real world software development.

Towards a theory of version control systems. Version control systems usually provide an operation to *merge* concurrent changes made on the same version of a source tree. This operation is purely textual: it assumes that if two disjoint fragments of source code have been changed concurrently, then the union of these changes can be used to obtain a version of the program which includes both changes. Unfortunately, even if two changes are textually disjoint, that does mean that they are compatible. A good starting point for a theory of control version systems could the following specification for a semantic-aware merge operation.

Definition 6.1 (Sound merge between concurrent changes). Given three programs P_0, P_1 and P_2 and two differences δ_1 and δ_2 such that $P_0[\![\delta_1]\!]P_1$ and $P_0[\![\delta_2]\!]P_2$, a program P_3 is *a valid merge between* P_1 *and* P_2 if $P_0[\![\delta_1 \,\&\, \delta_2]\!]P_3$.

REFERENCES

[1] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational verification using product programs. In *International Symposium on Formal Methods*. Springer, 200–214.

[2] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2013. Beyond 2-safety: Asymmetric product programs for relational program verification. In *International Symposium on Logical Foundations of Computer Science*. Springer, 29–43.

[3] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2016. Product programs and relational program logics. *J. Log. Algebr. Meth. Program.* 85, 5 (2016), 847–859. DOI: https://doi.org/10.1016/j.jlamp.2016.05.004

[4] Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2017. Coupling proofs are probabilistic product programs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 161–174.

[5] Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2017. Coupling proofs are probabilistic product programs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 161–174.

[6] Nick Benton. 2005. Simple Relational Correctness Proofs for Static Analyses and Program Transformations (Revised, Long Version). (2005).

[7] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L Lawall, and Gilles Muller. 2009. A foundation for flow-based program matching: using temporal logic and model checking. In *Acm Sigplan Notices*, Vol. 44-1. ACM, 114–126.

[8] Julien Cohen. 2016. Renaming Global Variables in C Mechanically Proved Correct. In *Proceedings of the Fourth International Workshop on Verification and Program Transformation, VPT@ETAPS 2016, Eindhoven, The Netherlands, 2nd April 2016*. 50–64. DOI: https://doi.org/10.4204/EPTCS.216.3

[9] Thibaut Girka, David Mentré, and Yann Régis-Gianas. 2015. A Mechanically Checked Generation of Correlating Programs Directed by Structured Syntactic Differences. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 64–79.

[10] Thibaut Girka, David Mentré, and Yann Régis-Gianas. 2017. Coq development. https://www.irif.fr/~thib/oracles/. (2017).

[11] Chris Hawblitzel, Ming Kawaguchi, Shuvendu Lahiri, and Henrique Rebelo. 2011. *Mutual Summaries: Unifying Program Comparison Techniques*. Technical Report.

[12] James Wayne Hunt. *An algorithm for differential file comparison.*

[13] Leslie Lamport and Martin Abadi. 1988. The existence of refinement mappings, In Proceedings of the 3rd Annual Symposium on Logic in Computer Science. 165–175.

[14] David Park. 1981. Concurrency and automata on infinite sequences. In *Theoretical computer science*. Springer, 167–183.

[15] Nimrod Partush and Eran Yahav. 2014. Abstract semantic differencing via speculative correlation. In *ACM SIGPLAN Notices*, Vol. 49-10. ACM, 811–828.

[16] Suzette Person, Matthew B Dwyer, Sebastian Elbaum, and Corina S Păsăreanu. 2008. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 226–237.

[17] Chris Hawblitzel Shuvendu Lahiri, Kenneth McMillan. 2013. Differential Assertion Checking. ACM.

[18] Ming Kawaguchi Henrique Rebelo Shuvendu Lahiri, Chris Hawblitzel. 2012. SymDiff: A language-agnostic semantic diff tool for imperative programs. Springer.

[19] Tony Hoare Shuvendu Lahiri, Kapil Vaswani. 2010. Differential Static Analysis: Opportunities, Applications, and Challenges. Association for Computing Machinery, Inc.

[20] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. 2013. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering* 39, 2 (2013), 147–162.

[21] Ofer Strichman and Benny Godlin. 2005. Regression Verification-a practical way to verify programs. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 496–501.