Analysis of Program Differences with Numerical Abstract Interpretation

David Delmas
Airbus Operations S.A.S. and LIP6
316, route de Bayonne
31060 Toulouse Cedex 9, France
david.delmas@airbus.com

Antoine Miné
Institut universitaire de France
Sorbonne Université, CNRS
Laboratoire d'Informatique de Paris 6 (LIP6)
F-75005 Paris, France
antoine.mine@lip6.fr

1. INTRODUCTION

We present work in progress[1] on the static analysis of software patches. Given two syntactically close versions of a program, our analysis can infer a semantic difference, and prove that both programs compute the same outputs when run on the same inputs. Our method is based on abstract interpretation [1], and parametric in the choice of an abstract domain. At the moment, we focus on numeric properties only, on a toy language. Our method is able to deal with infinite-state programs and unbounded executions, but it is limited to comparing terminating executions, ignoring non terminating ones.

We first present a novel concrete collecting semantics, expressing the behaviors of both programs at the same time. We then show how to leverage classic numeric abstract domains, such as polyhedra [2] or octagons [10], to build an effective static analysis. We also introduce a novel numeric domain to bound differences between the values of the variables in the two programs, which has linear cost, and the right amount of relationality to express useful properties of software patches. We implemented a prototype and experimented on a few small examples from the literature.

In future work, we will consider extensions to non purely numeric programs, towards the analysis of realistic patches.

The paper is organised as follows. Section 2 presents a running example, introducing our syntax and semantics for program patches informally. Section 3 formalises the concrete collecting semantics, and illustrates it on the example. Section 4 describes the abstract semantics, and discusses the choice of numerical abstract domains with respect to the example. Section 5 presents experimental results with a prototype implementation. Section 6 stresses current limitations of the approach. Section 7 discusses related work. Section 8 concludes.

2. RUNNING EXAMPLE

In the following, we sketch our approach to the analysis of semantic differences between two syntactically similar programs P_1 and P_2. We are interested in proving that P_1 and P_2 compute equal outputs when run on equal inputs. P_1 and P_2 are represented together in the syntax of a so-called double program P. Simple programs P_1 and P_2 are referred to as the left and right components of P. Fig. 1 shows the `Unchloop` example, taken from [14], and translated into our syntax of double programs. The $\|$ symbol is used to represent syntactic difference. It is available at expression, condition, and statement levels in our syntax for double programs. For instance at line 3, $c \leftarrow 1 \parallel 0$ means $c \leftarrow 1$ for P_1, and $c \leftarrow 0$ for P_2. In contrast, line 4 means $i \leftarrow 0$ for both P_1 and P_2.

```
1 :   a ← input(−1000, 1000);
2 :   b ← input(−1000, 1000);
3 :   c ← 1 ‖ 0;
4 :   i ← 0;
5 :   while (i < a) {
6 :     c ← c + b;
7 :     i ← i + 1;
8 :   }
9 :   r ← c ‖ c + 1;
10 :  assert_sync(r);
```

Figure 1: `Unchloop` example

Let us describe the example program. Both versions P_1 and P_2 read inputs in the range $[-1000, 1000]$ into `a` and `b` at lines 1 and 2. At ligne 3, the counter `c` is being initialised with value `1` for program P_1, and value `0` for program P_2. Then, both components add `a` times the value of `b` to `c` in a loop. Finally, they both store the result into `r` at line 9: `c` for P_1, `c+1` for P_2. The assertion at line 10 expresses the property we would like to check: if both programs components reach it, then they should have computed equal values for `r`.

We assume here that both programs read the same input value in `a`, and the same input value in `b`. More generally, the semantics $[\![P]\!]$ of a program is parameterized by a (possibly infinite) sequence of input values s, and we wish to prove that, $\forall s : ([\![P_1]\!] s)(r) = ([\![P_2]\!] s)(r)$, *i.e.* the programs have the same result in `r` given the same sequence of input values. The assertion at line 10 of our example is thus valid. Although not presented in the example, our language also supports true non-deterministic random values, that may differ between the executions of both variants.

The assertion at line 10 is, indeed, validated by our analysis.

3. CONCRETE SEMANTICS

Following the standard approach to abstract interpretation, we developed a concrete collecting semantics for a toy

[1] This work is performed as part of a collaborative partnership between Sorbonnne Université / CNRS (LIP6) and Airbus. This work is partially supported by the European Research Council under the Consolidator Grant Agreement 681393 – MOPSA.

WHILE-like language for double programs. The $\|$ operator may occur anywhere in the parse tree, to denote syntactic differences between the two components of a double program. This operator, however, is not recursive.

Given double program P with variables in $\mathcal{V}$, consider its left (resp. right) component $P_1 = \pi_1(P)$ (resp. $P_2 = \pi_2(P)$), with variables in $V_1 = \{ x_1 \,|\, x \in \mathcal{V} \}$ (resp. $V_2 = \{ x_2 \,|\, x \in \mathcal{V} \}$), where π_1 (resp. π_2) is a projection operator defined by induction on the syntax, keeping only the left (resp. right) side of $\|$ symbols, and renaming variables x to their versions x_1 (resp. x_2) in P_1 (resp. P_2). For instance, $\pi_1(c \leftarrow 1 \,\|\, 0) = c_1 \leftarrow 1$, and $\pi_2(c \leftarrow 1 \,\|\, 0) = c_2 \leftarrow 0$, while $\pi_1(i \leftarrow 0) = i_1 \leftarrow 0$, and $\pi_2(i \leftarrow 0) = i_2 \leftarrow 0$.

P_1 and P_2 are simple programs, with concrete memory states in $\mathcal{E}_1 \overset{\text{def}}{=} \mathcal{V}_1 \to \mathbb{R}$ and $\mathcal{E}_2 \overset{\text{def}}{=} \mathcal{V}_2 \to \mathbb{R}$, respectively. Let $k \in \{1;2\}$. To define the semantics of simple program P_k, we leverage standard, relational, input-output semantics, defined by induction on the syntax, in denotational style : $\mathbb{E}_k[\![e]\!] \in \mathcal{E}_k \to \mathcal{P}(\mathbb{R})$ for non-deterministic expression $e \in \mathit{expr}$, $\mathbb{C}_k[\![c]\!] \in \mathcal{E}_k \to \mathcal{P}(\{\text{true}, \text{false}\})$ for condition $c \in \mathit{cond}$, and $\mathbb{S}_k[\![s]\!] \in \mathcal{P}(\mathcal{E}_k \times \mathcal{E}_k)$ for statement $s \in \mathit{stat}$. $\mathbb{S}_k[\![s]\!]$ describes the relation between input and output states of statement s.

We then lift the semantics $\mathbb{S}_1$ and $\mathbb{S}_2$ to double programs. As P_1 and P_2 have concrete memory states in $\mathcal{E}_1$ and $\mathcal{E}_2$, respectively, P has concrete memory states in $\mathcal{D} \overset{\text{def}}{=} \mathcal{E}_1 \times \mathcal{E}_2$. The semantics of a double statement $s \in \mathit{dstat}$, denoted $\mathbb{D}[\![s]\!] \in \mathcal{P}(\mathcal{D} \times \mathcal{D})$, describes the relation between input and output states of s, which are pairs of states of simple programs. A subset of $\mathbb{D}[\![s]\!]$ is shown on Fig. 2, in relational style, for a subset of the language featuring double statements and expressions, but only simple conditions. It is defined by induction on the syntax, so as to allow for modular, joint analyses of double programs that maintain input-output relations on the variables. Note that $\mathbb{D}$ is parametric in $(\mathbb{S}_1, \mathbb{S}_2)$.

The semantics for the empty program is the diagonal, identity relation $\Delta_{\mathcal{D}}$. The semantics $\mathbb{D}[\![s_1 \,\|\, s_2]\!]$ for the comparison of two syntactically different statements reverts to the pairing of the simple program semantics of individual simple statements s_1 and s_2. The semantics for assignments of double expressions (different expressions to the same variable) is defined using this construct. The semantics for the sequential composition of statements boils down to the composition of the semantics of individual statements. Note that we use the symbol $⨾$ to denote the left composition of relations: $R_1 ⨾ R_2 \overset{\text{def}}{=} \{ (x, z) \,|\, \exists y : (x, y) \in R_1 \wedge (y, z) \in R_2 \}$. The semantics for selection statements distinguish between cases where both components agree on the value of the controlling expression, and cases where they do not (*a.k.a.* unstable tests). In the latter cases, the semantics is defined by composing the semantics of the projections of the double program on its components. The semantics for (possibly unbounded) iteration statements is defined using the least fixpoint of a function defined similarly.

Coming back to our running example `Unchloop` on Fig. 1, the concrete semantics for the program from line 3 to 9 is displayed on Fig. 3. With the additional assumption that both program components start from equal memory states ($a_1 = a_2 \wedge b_1 = b_2$), ensured by our semantics for the `input` built-in, the two components can be proved to compute equal values for `r`.

$\mathbb{D}[\![\mathit{dstat}]\!] \in \mathcal{P}(\mathcal{D} \times \mathcal{D})$

$$\begin{array}{ll}
\mathbb{D}[\![\textbf{skip}]\!] & \overset{\text{def}}{=} \Delta_{\mathcal{D}} \\
\mathbb{D}[\![s_1 \,\|\, s_2]\!] & \overset{\text{def}}{=} \{ ((i_1, i_2), (o_1, o_2)) \,| \\
 & \quad \forall\ k \in \{1, 2\} : (i_k, o_k) \in \mathbb{S}_k[\![s_k]\!] \ \} \\
\mathbb{D}[\![V \leftarrow e_1 \,\|\, e_2]\!] & \overset{\text{def}}{=} \mathbb{D}[\![V \leftarrow e_1 \,\|\, V \leftarrow e_2]\!] \\
\mathbb{D}[\![s_1;\ s_2]\!] & \overset{\text{def}}{=} \mathbb{D}[\![s_1]\!] ⨾ \mathbb{D}[\![s_2]\!] \\
\mathbb{D}[\![\textbf{if } c \textbf{ then } s_1 \textbf{ else } s_2]\!] & \overset{\text{def}}{=} \mathbb{F}[\![c]\!] ⨾ \mathbb{D}[\![s_1]\!] \cup \mathbb{F}[\![\neg c]\!] ⨾ \mathbb{D}[\![s_2]\!] \\
 & \cup\ \mathbb{F}[\![c \,\|\, \neg c]\!] ⨾ \mathbb{D}_1[\![s_1]\!] ⨾ \mathbb{D}_2[\![s_2]\!] \\
 & \cup\ \mathbb{F}[\![\neg c \,\|\, c]\!] ⨾ \mathbb{D}_1[\![s_2]\!] ⨾ \mathbb{D}_2[\![s_1]\!] \\
\mathbb{D}[\![\textbf{while } c \textbf{ do } s]\!] & \overset{\text{def}}{=} (\text{lfp } H) ⨾ \mathbb{F}[\![\neg c]\!]
\end{array}$$

where

$$\begin{array}{ll}
\mathbb{F}[\![c_1 \,\|\, c_2]\!] & \overset{\text{def}}{=} \{ ((\rho_1, \rho_2), (\rho_1, \rho_2)) \,| \\
 & \quad \forall\ k \in \{1, 2\} : \text{true} \in \mathbb{C}_k[\![c_k]\!] \rho_k \ \} \\
\mathbb{F}[\![c]\!] & \overset{\text{def}}{=} \mathbb{F}[\![c \,\|\, c]\!] \\
\mathbb{D}_1[\![s]\!] & \overset{\text{def}}{=} \mathbb{D}[\![\pi_1(s) \,\|\, \textbf{skip}]\!] \\
\mathbb{D}_2[\![s]\!] & \overset{\text{def}}{=} \mathbb{D}[\![\textbf{skip} \,\|\, \pi_2(s)]\!] \\
H(R) & \overset{\text{def}}{=} \Delta_{\mathcal{D}} \cup R ⨾ (\mathbb{F}[\![c]\!] ⨾ \mathbb{D}[\![s]\!] \\
 & \cup\ \mathbb{F}[\![c \,\|\, \neg c]\!] ⨾ \mathbb{D}_1[\![s]\!] \\
 & \cup\ \mathbb{F}[\![\neg c \,\|\, c]\!] ⨾ \mathbb{D}_2[\![s]\!] \,)
\end{array}$$

Figure 2: Denotational concrete semantics of double programs

Unfortunately, our concrete collecting semantics $\mathbb{D}$ is not computable in general. A particular difficulty of the `Unchloop` example is that the input-ouptut relation is non linear: $(a \leq 0 \Rightarrow r = 1) \wedge (a \geq 0 \Rightarrow r = 1 + a \times b)$.

4. ABSTRACT SEMANTICS

We therefore tailor an abstract semantics $\mathbb{D}^\sharp$, suitable for the analysis of program differences. As $\mathcal{D} \approx \mathbb{R}^{|\mathcal{V}_1 \cup \mathcal{V}_2|}$, any numeric abstract domain on pairs of environments may be used. As $\mathbb{D}$ is defined by induction the syntax, the definition for $\mathbb{D}^\sharp$ is straightforward: the abstract semantics need only be defined for the $s_1 \,\|\, s_2$ construct. We define it as $\mathbb{D}^\sharp[\![s_1 \,\|\, s_2]\!] \overset{\text{def}}{=} \mathbb{D}^\sharp_1[\![s_1]\!] ⨾^\sharp \mathbb{D}^\sharp_2[\![s_2]\!]$. This definition is sound, as $\mathbb{D}[\![s_1 \,\|\, s_2]\!] = \mathbb{D}_1[\![s_1]\!] ⨾ \mathbb{D}_2[\![s_2]\!]$ holds in the concrete semantics. For instance, $\mathbb{D}^\sharp[\![c \leftarrow 1 \,\|\, 0]\!] = \mathbb{D}^\sharp[\![c_1 \leftarrow 1]\!] ⨾^\sharp \mathbb{D}^\sharp[\![c_2 \leftarrow 0]\!]$.

Note that the relation between `c` and `i` is non linear in the `Unchloop` example: $c_1 = i_1 \times b_1 + 1$ and $c_2 = i_2 \times b_2$ from line 4 to 9. Thus, a separate analysis of programs P_1 and P_2 would require a non linear abstract domain to compare r_1 and r_2. In contrast, our joint analysis of P_1 and P_2 will be sufficiently precise, even when using linear numeric domains, because the difference between the values of the variables in P_1 and in P_2 remains linear. For instance, the polyhedra domain is able to infer that the invariant $-c_1 + c_2 + 1 = 0$ holds from line 3 to 9, hence $r_1 = r_2$ at line 9, although it is not able to discover any interval for r_1 or r_2. The octagon domain is also able to express these invariants, but its transfer function for assignment is not precise enough to infer them. Indeed, $x \leftarrow a - b$ cannot be exactly abstracted by the domain, and currently proposed transfer functions fall back to plain interval arithmetics in that case, so that the domain cannot exploit the bound it infers on $a - b$ to bound

$$\begin{array}{llr}
\mathbb{D}[\![Unchloop_{3..9}]\!] = & \{\, s_0, ((a_1, b_1, 1, 0, 1), (a_2, b_2, 0, 0, 1)) & \mid a_1 \leq 0 \wedge a_2 \leq 0 \wedge H_0 \,\} \\
& \cup \ \{\, s_0, ((a_1, b_1, 1 + a_1 \times b_1, a_1, 1 + a_1 \times b_1), (a_2, b_2, 0, 0, 1)) & \mid a_1 > 0 \wedge a_2 \leq 0 \wedge H_0 \,\} \\
& \cup \ \{\, s_0, ((a_1, b_1, 1, 0, 1), (a_2, b_2, a_2 \times b_2, a_2, 1 + a_2 \times b_2)) & \mid a_1 \leq 0 \wedge a_2 > 0 \wedge H_0 \,\} \\
& \cup \ \{\, s_0, ((a_1, b_1, 1 + a_1 \times b_1, a_1, 1 + a_1 \times b_1), (a_2, b_2, a_2 \times b_2, a_2, 1 + a_2 \times b_2)) & \mid a_1 > 0 \wedge a_2 > 0 \wedge H_0 \,\}
\end{array}$$

where

$$s_0 \stackrel{\text{def}}{=} ((a_1, b_1, c_1, i_1, r_1), (a_2, b_2, c_2, i_2, r_2))$$

$$H_0 \stackrel{\text{def}}{=} \forall k \in \{1, 2\} : (b_k, c_k, i_k, r_k) \in \mathbb{R}^4$$

Figure 3: Concrete semantics of the `Unchloop` example

`x`, for efficiency reasons. The interval domain is not able to express the invariants, hence it cannot be used directly for a conclusive analysis.

However, we remark that it is sufficient to bound the differences $x_2 - x_1$ for any variable `x` to express the necessary invariants. Thus, we now design an abstract domain that is specialized to infer these bounds. We therefore introduce the Galois automorphism $(\mathcal{P}(\mathcal{D} \times \mathcal{D}), \subseteq) \xleftarrow[\alpha]{\gamma} (\mathcal{P}(\mathcal{D} \times \mathcal{D}), \subseteq)$ defined by $\alpha(R) \stackrel{\text{def}}{=} \{\, ((i_1, i_2 - i_1), (o_1, o_2 - o_1)) \mid ((i_1, i_2), (o_1, o_2)) \in R \,\}$, and $\gamma(\Delta) \stackrel{\text{def}}{=} \{\, ((\rho, \rho{+}\delta), (\rho', \rho'{+}\delta')) \mid ((\rho, \delta), (\rho', \delta')) \in \Delta \,\}$, and let $\mathbb{\Delta} \stackrel{\text{def}}{=} \alpha \circ \mathbb{D}$. This amounts to changing the representation of states of double program P : variable `x` is represented by its left and right projections (x_1, x_2) in semantics $\mathbb{D}$, and by $(x_1, \delta x)$ in semantics $\mathbb{\Delta}$, where $\delta x \stackrel{\text{def}}{=} x_2 - x_1$. The $\mathbb{\Delta}$ semantics of statements 6 and 9 of the `UnchLoop` example are shown for instance on Fig. 4, before and after simple symbolic simplifications of affine expressions.

Like for $\mathbb{D}$, any numeric domain can be used to abstract $\mathbb{\Delta}$, so that the definition for $\mathbb{\Delta}^\sharp$ is straightforward, by induction on the syntax of double programs. We also define the semantics for the $s_1 \parallel s_2$ construct as $\mathbb{\Delta}^\sharp[\![s_1 \parallel s_2]\!] \stackrel{\text{def}}{=} \mathbb{\Delta}_1^\sharp[\![s_1]\!] \mathbin{⨟}^\sharp \mathbb{\Delta}_2^\sharp[\![s_2]\!]$, where $\mathbb{\Delta}_1[\![s]\!] \stackrel{\text{def}}{=} \mathbb{\Delta}[\![\pi_1(s) \parallel \textbf{skip}]\!]$, and $\mathbb{\Delta}_2[\![s]\!] \stackrel{\text{def}}{=} \mathbb{\Delta}[\![\textbf{skip} \parallel \pi_2(s)]\!]$.

Nonetheless, we add a particular case for the simple assignment $V \leftarrow e$, to gain both precision and efficiency through simple symbolic simplifications. That particular case is displayed on Fig. 5. Under some conditions on the expression `e`, we say `e` is "differentiable", and update the δV component of variable `V` in the abstract from the δx components of all program variables `x`, independently of their x_1 components. We call an expression differentiable if it is an affine expression of the form $\mu + \sum_{x \in \mathcal{V}} \lambda_x x$, or a deterministic expression where all variables `x` are such that $\delta x = 0$, or a so-called "synchronised" input expression.

We say an input expression is synchronised when P_1 and P_2 have called **input** equal numbers of times. For instance, after statement $a \leftarrow \textbf{input}(-1000, 1000)$ at line 1 of the `Unchloop` example, we have $a \in [-1000, 1000]$, but $\delta a = 0$. This last property stems from the fact that both programs evaluate the same input value stream at the same index. To infer this property of inputs, we developed a very simple dedicated domain. The domain associates a counter to each input instruction, that counts the number of times the instruction has been called in each program since the start of the executions. If the domain can proove that these counts are equal when reaching some statement $x \leftarrow \textbf{input}(m, M)$, for some variable `x` and some constants `m` and `M`, then it adds the contraint $\delta x = 0$. Otherwise, $\delta x \in [m - M, M - m]$. As the counters are numeric quantities, the analysis can delegate infering their equality to numeric abstract domains.

If the expression `e` is differentiable, we update the δV component of variable `V` in the abstract with the result of our differentiation function on `e`. This function operates like mathematical differentiation on affine expressions. Otherwise, if `e` is differentiable but not affine, then our differentiation function returns 0, as `e` is guaranteed to evaluate to equal values in P_1 and P_2.

To further enhance precision on some examples, we generalize slightly this particular case to double assignments $V \leftarrow e_1 \parallel e_2$, when expressions e_1 and e_2 are found syntactically equal, modulo some semantics preserving transformations, such as associativity, commutativity, and distributivity.

5. EVALUATION

We implemented a prototype abstract interpreter for the semantics $\mathbb{D}^\sharp$ and $\mathbb{\Delta}^\sharp$ of the toy language introduced with the `Unchloop` example of Fig. 1. It is about 2,000 lines of OCaml source code, and uses the Apron [7] library to experiment with the polyhedra and octagon abstract domains. We compare results on small examples selected from other authors' benchmarks [14, 11, 12]. These references deal with C programs directly, while we encode their benchmarks in our toy language. In addition, these references not only prove equivalences, but also characterise differences, while we focus on equivalence for now. We therefore selected benchmarks relevant to equivalence only, except for the so-called "Fig. 2" example of [14], which we modified slightly to restore equivalence of terminating executions: see Fig.6.

Table 7 summarises the results of our analysis. Our results are comparable with those of the original authors, with significant speedups – several orders of magnitude with respect to [14]. All experiments were conducted on a Intel® Core-i7™ processor.

6. LIMITATIONS

Our analysis is based on abstractions of the concrete collecting semantics $\mathbb{D}$, which relates pairs of terminating executions of components of a double program. It is suitable to prove a number of properties, including that two terminating programs starting from equal initial states will produce equal outputs, a notion called partial equivalence in [3]. In contrast, an analysis based on this collecting semantics will fail to report differences between pairs of executions where at least one of the component does not terminate. For instance, in the example on Fig.6. our analysis does not report any difference between P_1 and P_2, although P_1 terminates on input $x = 2$, and P_2 does not.

As opposed to [11, 12], who develop algorithms to automate the construction of a correlating program $P_1 \bowtie P_2$,

$$
\begin{aligned}
\Delta[\![c \leftarrow c + b]\!] &= \{\, (s_1, (((a_1, b_1, c_1 + b_1, i_1, r_1), ((a_1 + \delta a) - a_1, (b_1 + \delta b) - b_1,\\
&\qquad ((c_1 + \delta c) + (b_1 + \delta b)) - (c_1 + b_1), (i_1 + \delta i) - i_1, (r_1 + \delta r) - r_1))) \mid H_1 \,\}\\
&= \{\, (s_1, ((a_1, b_1, c_1 + b_1, i_1, r_1), (\delta a, \delta b, \delta c + \delta b, \delta i, \delta r))) \mid H_1 \,\}
\end{aligned}
$$

$$
\begin{aligned}
\Delta[\![r \leftarrow c \parallel c + 1]\!] &= \{\, (s_1, ((a_1, b_1, c_1, i_1, c_1), ((a_1 + \delta a) - a_1, (b_1 + \delta b) - b_1, ((c_1 + \delta c) - c_1, (i_1 + \delta i) - i_1, (c_1 + \delta c + 1) - c_1))) \mid H_1 \,\}\\
&= \{\, (s_1, ((a_1, b_1, c_1, i_1, c_1), (\delta a, \delta b, \delta c, \delta i, \delta c + 1))) \mid H_1 \,\}
\end{aligned}
$$

where

$$
\begin{aligned}
s_1 &\stackrel{\text{def}}{=} ((a_1, b_1, c_1, i_1, r_1), (\delta a, \delta b, \delta c, \delta i, \delta r))\\
H_1 &\stackrel{\text{def}}{=} ((a_1, b_1, c_1, i_1, r_1), (\delta a, \delta b, \delta c, \delta i, \delta r)) \in \mathbb{R}^{10}
\end{aligned}
$$

Figure 4: Examples of Δ semantics

$$
\Delta^\sharp[\![V \leftarrow e]\!] \stackrel{\text{def}}{=} \Delta_1^\sharp[\![V \leftarrow e]\!] \fatsemi^\sharp \left(\begin{array}{ll}
\Delta^\sharp[\![\delta V \leftarrow 0]\!] & \textbf{if} \left(\begin{array}{lll} & (\exists a, b : e = \textbf{input}(a, b) & \wedge\ \text{input\_is\_sync}()) \\ \vee & (\text{is\_deterministic}(e) & \wedge\ \forall x \in \text{Vars}(e) : \delta x = 0) \end{array} \right) \\
\Delta^\sharp[\![\delta V \leftarrow \sum_{x \in \mathcal{V}} \lambda_x \delta x]\!] & \textbf{if } \exists (\mu, (\lambda_x)_{x \in \mathcal{V}}) \in \mathbb{R}^{|\mathcal{V}|+1} \quad : \quad e = \mu + \sum_{x \in \mathcal{V}} \lambda_x x \\
\Delta_2^\sharp[\![V \leftarrow e]\!] & \textbf{otherwise}
\end{array} \right)
$$

Figure 5: Symbolic simplifications in $\Delta^\sharp$

```
x ← input(−100, 100);
if (x < 0) x ← −1;
else {
    if (x ≥ 2 || x ≥ 4) {} // x > 4 in [14]
    else {
        while (i = 2) x ← 2;
        x ← 3;
    }
}
assert_sync(x); // x = 2 ignored
```

Figure 6: Modified `Fig.2` example (from [14])

on which to run the static analysis, we assume for now the joint representation of P_1 and P_2 given, as part of a double program in our toy language.

7. RELATED WORK

[6] pioneered the field of semantic differencing between two versions of a procedure by comparing dependencies between input and output variables. Symbolic execution methods [13, 14] have proposed analysis techniques for programs with small state space and bounded loops, which may support modular regression verification. The RVT [3] and SymDiff [8] combine two versions of the same program, with equality constraints on their inputs, and compile equivalence properties into verification conditions to be checked by SMT solvers.

The DIZY [11, 12] tool leverages numerical abstract interpretation to establish equivalence under abstraction. Our work is thus similar. Our main contribution so far is a novel concrete collecting semantics by induction on the syntax, and a novel numeric domain to bound differences between the values of the variables in the two programs.

The Fluctuat [9, 4] static analyser compares the real and floating-point semantics of numeric programs to bound errors in floating-point computations. The authors use the zonotope abstract domain to bound the difference between real and floating-point values, which is similar to our $\Delta^\sharp$ abstraction. Like in our concrete semantics $\mathbb{D}^\sharp$, they also address unstable test analysis [5].

8. CONCLUSION

We presented work in progress on the static analysis of software patches. Our method is based on abstract interpretation, and parametric in the choice of an abstract domain. We presented a novel concrete collecting semantics, expressing the behaviors of two syntactically close versions of a program at the same time. We also introduced a novel numeric domain to bound differences between the values of the variables in the two programs, which has linear cost. We implemented a prototype and experimented on a few small examples from the literature. In future work, we will consider extensions to non purely numeric programs, towards the analysis of realistic patches. We will also consider other abstract domains, such as zonotopes.

9. REFERENCES

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, Jan. 1977.

[2] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL'78*, pages 84–97. ACM, 1978.

[3] B. Godlin and O. Strichman. Regression verification. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 466–471, New York, NY, USA, 2009. ACM.

[4] E. Goubault and S. Putot. Static analysis of finite precision computations. In R. Jhala and D. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 232–247, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] E. Goubault and S. Putot. Robustness analysis of finite precision implementations. In C.-c. Shan, editor, *Programming Languages and Systems*, pages 50–57, Cham, 2013. Springer International Publishing.

[6] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pages 243–252, Washington, DC, USA, 1994. IEEE Computer Society.

[7] B. Jeannet and A. Miné. Apron: A library of

Benchmark	LOC	Origin	$\mathbb{D}^\sharp$(polyhedra)		$\mathbb{D}^\sharp$(octagon)		$\Delta^\sharp$(interval)	
			time (ms)		time (ms)		time (ms)	
Comp	13	[14]	14	✓	18	✗	2	✗
Const	9	[14]	7	✓	17	✓	1	✓
Fig. 2	14	[14]	4	✓	5	✓	1	✓
LoopMult	14	[14]	20	✓	56	✗	1	✗
LoopSub	15	[14]	19	✓	53	✗	2	✗
UnchLoop	13	[14]	15	✓	36	✗	2	✓
copy	37	[11]	102	✓	60	✓	2	✗
remove	19	[11]	6	✓	18	✓	2	✗
seq	41	[11, 12]	75	✓	500	✗	2	✗
sign	12	[11]	6	✓	8	✗	2	✗
sum	14	[11]	13	✓	19	✓	2	✓

Figure 7: Benchmarks

numerical abstract domains for static analysis. In *Proc. of the 21th Int. Conf. on Computer Aided Verification (CAV'09)*, volume 5643 of *LNCS*, pages 661–667. Springer, June 2009.

[8] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification*, pages 712–717, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[9] M. Martel. Propagation of roundoff errors in finite precision computations: A semantics approach. In D. Le Métayer, editor, *Programming Languages and Systems*, pages 194–208, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[10] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[11] N. Partush and E. Yahav. Abstract semantic differencing for numerical programs. In F. Logozzo and M. Fähndrich, editors, *Static Analysis*, pages 238–258, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[12] N. Partush and E. Yahav. Abstract semantic differencing via speculative correlation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 811–828, New York, NY, USA, 2014. ACM.

[13] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA, 2008. ACM.

[14] A. Trostanetski, O. Grumberg, and D. Kroening. Modular demand-driven analysis of semantic difference for program versions. In F. Ranzato, editor, *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, volume 10422 of *Lecture Notes in Computer Science*, pages 405–427. Springer, 2017.