Partition-Based Regression Verification

Marcel Böhme
School of Computing
National University of Singapore
marcel@comp.nus.edu.sg

Bruno C. d. S. Oliveira
School of Computing
National University of Singapore
oliveira@comp.nus.edu.sg

Abhik Roychoudhury
School of Computing
National University of Singapore
abhik@comp.nus.edu.sg

***Abstract*—Regression verification (RV) seeks to guarantee the absence of regression errors in a changed program version. This paper presents *Partition-based Regression Verification* (PRV): an approach to RV based on the *gradual* exploration of differential input partitions. A differential input partition is a subset of the common input space of two program versions that serves as a unit of verification. Instead of proving the absence of regression for the complete input space at once, PRV verifies differential partitions in a gradual manner. If the exploration is interrupted, PRV retains *partial verification guarantees* at least for the explored differential partitions. This is crucial in practice as verifying the complete input space can be prohibitively expensive.**

Experiments show that PRV provides a useful alternative to state-of-the-art regression test generation techniques. During the exploration, PRV generates test cases which can expose different behaviour across two program versions. However, while test cases are generally single points in the common input space, PRV can verify entire partitions and moreover give feedback that allows programmers to relate a behavioral difference to those syntactic changes that contribute to this difference.

***Keywords*-Software Verification, Testing and Analysis**

I. Introduction

Software verification seeks to guarantee the absence of errors in a program, but is rather expensive in practice. There are two main reasons: 1) verification requires specifications, which may be difficult to write and maintain; and 2) the verification process can be very time-consuming.

However, there is some hope for a cheap form of *Regression Verification* (RV) [1], [2]. The goal of RV is not to verify the correctness of a program ad absolutum but relative to an earlier version. Thus, RV seeks to guarantee the absence of regression errors. This more modest goal allows RV to avoid separate forms of formal specifications. The previous version serves as sufficient specification for checking whether the changed version is *at least as correct* as the previous version.

Yet, in practice, RV for all inputs is very time-consuming. Godlin and Strichman [1] proposed a decision procedure that takes two program versions and either proves behavioral equivalence (thus the absence of regression) or provides a witness of behavioral difference. The authors report that the verification of non-equivalent versions can take a long time to terminate or run out of memory. In fact, generally proving the equivalence between two programs is an undecidable problem. While the termination of RV provides strong regression guarantees for all inputs, the interruption of the verification procedure (due to time or memory constraints) yields no guarantees at all.

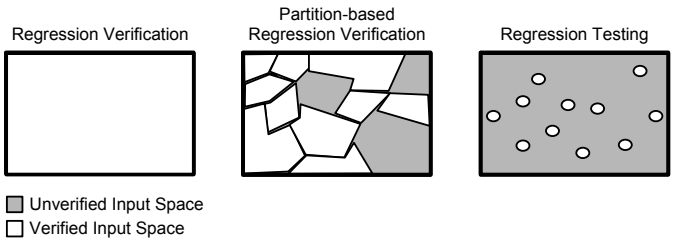

Fig. 1. PRV versus Regression Verification and Regression Testing

This paper presents *Partition-based Regression Verification* (PRV), a *gradual* approach to RV based on the exploration of *differential partitions*. A differential partition is a subset of the common input space of two program versions that serves as unit of verification. Instead of verifying the entire input space at once, PRV allows gradually verifying such partitions one-by-one. As illustrated in Figure 1, PRV shares the advantages of both, Regression Testing (RT) and RV. *Like RV*, if all differential partitions are shown equivalent, then PRV guarantees the absence of regression errors for all inputs. More importantly, PRV allows a form of *partial verification*: if the verification procedure is interrupted, PRV guarantees the absence of regression errors for the explored partitions that are shown equivalent. Thus, *like RT*, PRV allows the gradual checking for regression. However, while RT provides verification guarantees only for the concrete, executed sample inputs, PRV seeks to guarantee the absence of regression for entire input partitions. In practice, this partial verification approach is crucial, as verifying the complete input space can be infeasible due to time or other resource constraints.

Technically, differential partitions are computed using a form of symbolic execution [3] and require *deterministic* program execution. In contrast to other input partitioning techniques [4]–[8], differential partitioning accounts for the inputs of two programs. A differential partition is characterized by a symbolic condition that defines a range (or subset) of valid input for that partition. Input is grouped according to whether it reaches the same syntactic changes and whether it propagates the same differential state to the output. If an input computes the same output in both versions, the respective partition is said to be *equivalence-revealing*. In such case, both versions are soundly guaranteed to compute the same output for all input satisfying the symbolic condition. Otherwise, the respective partition is said to be *difference-revealing*.

```
1  input(i);
2  a = 0; o = 0;
3  i = i; //i=i+1
4  if(i>0)
5    a++;
6  if(a>0)
7    o=i;
8  output(o);
```

(a) Programs P and P'

	Input	Output
P	$i \leq 0$	$o = 0$
	$i > 0$	$o = i$
P'	$i+1 \leq 0$	$o' = 0$
	$i+1 > 0$	$o' = i+1$
PRV	$i < 0$	$o = o'$
	$i = 0$	$o = 0 \wedge o' = i+1$
	$i > 0$	$o = i \wedge o' = i+1$

(b) Differential Partitions

Fig. 2. Running Example (Incomplete Bugfix)

Figure 2 illustrates differential partitions in a concrete example. Program P computes output o based on the values of input i and is changed to P' by substituting line 3 with the commented code. Figure 2.b shows the symbolic output that is computed based on the evaluation of the input variables for both programs. The bottom three rows depict one equivalence-revealing and two difference-revealing input partitions. Note that the analysis of only a single version is insufficient to expose all interesting subsets of input. In particular, a test suite $T \leftarrow \{i = -1, i = 1\}$ covers all paths in both programs and even reveals a difference. However, input $i = 0$ is a missing test case that could represent a regression error. Intuitively, it is interesting because the branch in line 4 is evaluated in different directions in both versions. PRV explores a distinct, difference-revealing partition for this input.

PRV provides an alternative to *regression test generation* techniques [9]–[12]. Upon allowing the continued exploration even of difference-revealing partitions, the developer may (in)formally verify such partitions. The test cases generated for each difference-revealing partition can be checked against the developer's expectation. The program slice, used to compute the partition, can be inspected to determine the changed statements contributing to the difference. The symbolic conditions and summaries (cf. Fig. 2.b) can be further analyzed by tools.

Our initial experience with PRV is very encouraging. For the studied subjects, PRV efficiently exposes regression errors that are not detected by the considered test generation methods.

In summary the *main contributions* of this paper are:

- *A gradual approach to regression verification* that continuously verifies the input space of a program against another version of that program to find regression errors. If the verification procedure is interrupted, PRV guarantees the absence of regression errors for the explored input space that has been shown equivalence-revealing.
- A *differential partitioning* technique, based on symbolic execution, that soundly partitions the input of two versions. The partitioning technique symbolically groups input of the two programs, and creates partitions which either guarantee behavioral equivalence, or expose differences for a certain subset of inputs.
- *An alternative to regression test generation*. The approach can be used to generate test cases for partitions where differences are found. As illustrated by our experimental evaluation, finding such test cases is competitive with state-of-the-art regression test generation techniques.
- The *implementation* and *experimental evaluation* of PRV.

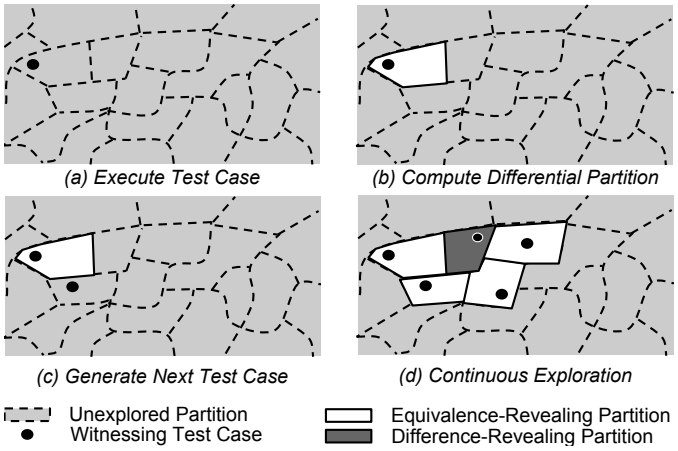

Fig. 3. Exploration of Differential Partitions

II. Partition-based Regression Verification

PRV takes two successive program versions and continuously verifies *differential partitions* (Def. 1). The gradual regression verification can be interrupted at any time. In this case, the regression guarantees are retained for the verified input space. For every partition, PRV generates a concrete sample input that is added to regression test suite T. The programmer can check difference-revealing test cases in T for regression errors and relate an output difference to the set of syntactic changes that contribute to that difference. The intuition of partition-based regression verification is presented in Figure 3, while the detailed procedure is outlined in Algorithm 1. Later, in Theorem 2, we will claim the *exhaustiveness* of this exploration algorithm.

Definition 1 (*Differential Partition*)

A differential partition $d_{P,P'}$ is obtained by partitioning of the common input space of two program versions P, P'. Inputs in any differential partition $d_{P,P'}$ have the following property: either all inputs in $d_{P,P'}$ produce the same output in P and P' (an equivalence-revealing partition), or all inputs in $d_{P,P'}$ produce different outputs in P and P' (a difference-revealing partition).

The exploration starts with a random test case in the queue. Depicted as black dot in Figure 3.a) this random test case t is taken from the queue and executed upon both versions. Test case t is a point in the common input space[1] of both versions, representing concrete knowledge about the differential behavior. In Figure 3.b), input is grouped into a differential partition that yields the same differential behavior as t. This input exercises all those statement instances that are "relevant" to the reachability and propagation of the syntactic program changes exercised by t. Hence, in Algorithm 2 the symbolic condition is computed as a conjunction of the pertinent branch conditions. Later, in Theorem 1, we will claim the *soundness* of this generalization from a test case to a differential partition.

[1]The derivation of the common input space for versions with different input spaces is discussed in [9], e.g., the new version has one more input variable.

Algorithm 1 Partition-based Regression Verification

Input: Versions P and P', Changes C and C'

1: let $queue \leftarrow \emptyset$
2: let $T \leftarrow \emptyset$
3: let $V \leftarrow \emptyset$
4: add $randomInput()$ to $queue$
5: **while** $queue \neq \emptyset$ **do**
6: let $t \leftarrow chooseNextTestcase(queue)$
7: let $condition \leftarrow computeDPartition(t, P, P', C, C')$
8: call $generateAdjacentTestcases(condition, queue)$
9: add t to T
10: add $condition$ to V
11: **end while**

Output: Verified Input Space V, Regression Test Suite T

As depicted in Figure 3.c), the next test case is executed outside of the explored input space. To generate such "adjacent" test cases, the constituent branch conditions are negated one-by-one (cf. Alg. 4), similar to other path exploration techniques. This yields a number of intermediate constraints. If a constraint solver finds a satisfying witness to one of these constraints, then it is added to the queue waiting to be executed.

As depicted in Figure 3.d), after the execution of the next test case from the queue, again, the corresponding differential partition is computed. This procedure repeats until all differential partitions are explored or some (time) budget is exhausted. A *search strategy* would assign some distance or fitness to each constraint and decide the order in which the partitions corresponding to intermediate constraints are explored. This is implemented in the procedure *chooseNextTestcase* (not listed). In particular, PRV takes from the queue in the order they arrive but prioritizes test cases that promise 1) different output, 2) the propagation of already exercised changes and 3) the execution of another set of changes, in that order. Finally, every executed test case is added to the regression test suite T. Each test case is a witness of one differential partition. The set of explored differential partitions V represents the verified input space.

A. Computing Differential Partitions

The computation of the differential partition for a given test case is presented in Algorithm 2. It implements the functionality of procedure *computeDPartition* called in Algorithm 1 and requires determinism - for every execution of the same input on the same program the same output is computed. Also, the deletion of variable assignments (e.g., `x=x++`) in P is represented by dummy-statements (e.g, `x=x`) in P' (cf. [13]).

Upon execution of the test case t on both programs, P and P', the symbolic condition is computed. Input that does not exercise a syntactic change or that does not propagate the differential state to the output is equivalence-revealing. If t does not exercise a changed statement, then PRV employs the *reachability condition* (Def. 3) to group input that does not execute a change for the same "reason". If t exercises at least one changed statement but yields the same output in both versions, then PRV employs the *propagation condition* (Def. 4).

Algorithm 2 - Procedure *computeDPartition*

Input: Input t, Versions P and P', Changes C and C'

1: let trace $\pi \leftarrow execute(t, P)$
2: let trace $\pi' \leftarrow execute(t, P')$
3: let $condition \leftarrow false$
4: **if** not exist an instance of $c' \in C'$ in π' **then**
5: let $condition \leftarrow \bigwedge_{c' \in C'} reach(c', \pi')$
6: **else**
7: let o_i be the instance of output o in π
8: let o'_i be the instance of output o in π'
9: **if** $value(o_i) = value(o'_i)$ **then**
10: let $condition \leftarrow prop(o, \pi, \pi', C, C')$
11: **else**
12: let $condition \leftarrow diff(o, \pi, \pi', C, C')$
13: **end if**
14: **end if**

Output: Condition $condition$

Input that yields different output is difference-revealing. If t yields different output in both program versions, then PRV employs the *difference condition* (Def. 5) to group input that computes different output for the same "reason". These reasons are defined upon the exercised dynamic and static program dependencies, as enunciated in the following.

B. Computing Reachability Conditions

Intuitively, an input t does not execute a changed statement c because the conditions of the branch instances s_i upon which c statically control-depends are evaluated in the direction that does not favor the execution of c.

```
1  input(i,j);
2  a = 0; b = 0;
3  if(i>0)
4    a=1;
5  for(c=0; c < j; c++)
6    b += c;
7  if(j>0)
8    if(a>0)
9      //change c
```

Fig. 4. Intuition of Reachability Condition

An example is shown in Figure 4. Input $(0, 1)$ does not execute the changed statement in line 9. Why? Because the branch in line 8 is not evaluated to `true`. This is because the condition in line 7 is evaluated to `true` and the condition in line 3 to `false`. The remainder of this section explains the computation of the reachability condition based on the relevant slice of the branch in line 8.

Definition 2 (*Relevant Slice [14], [15]*)

Given an execution trace π and a statement instance s_i in π, the relevant slice *of s_i in π contains all statement instances r_i in π that are in the transitive closure of dynamic data, control- and potential dependence of s_i.*

A statement instance s_i *potentially depends* [14] on conditional statement instance r_i in path π iff. there exists a variable v used in s_i such that (1) v is not defined between r_i and s_i in π but there exists another path σ from r_i to s_i along which v is defined, and (2) evaluating r_i differently may cause this untraversed path σ to be executed.

Note that relevant slices have a desirable property: If two inputs t_0 and t_1 exercise the same relevant slice computed w.r.t. a statement instance s_i, then the variables used in s_i have the same symbolic values for t_0 and t_1 [7]. Relevant slices are used to define the reachability, propagation, and difference conditions. The property of relevant slices is utilized to prove Theorem 1, establishing that these conditions indeed characterize differential partitions as defined in Definition 1.

Definition 3 (*Reachability Condition*)

The reachability condition, $reach(c, \pi)$, computed over the trace π w.r.t. statement c is the path condition computed over the union of the relevant slices of all instances s_i in π of every statement s that c transitively, statically control-depends on.

If an input t_0 does not exercise statement c, then every input t_1 satisfying $reach(c, \pi(t_0, P))$ does not exercise c. A *path condition* is a quantifier free first order logic formula on program inputs. Any test input satisfying the path condition of a path π is guaranteed to also exercise all statement instances in path π. The negation of a constituent branch condition in the reachability condition computed w.r.t. statement c may change the reachability of c.

C. Computing Propagation Conditions

Intuitively, an input t does not propagate the semantic effect of the exercised changes to the output because certain statement instances N_i upon which the output dynamically depends carry the same values in both versions. On a high level, N_i represents the point where the differential program states converge. Any attempt of negating a branch beyond that point to propagate a difference in program state is futile.

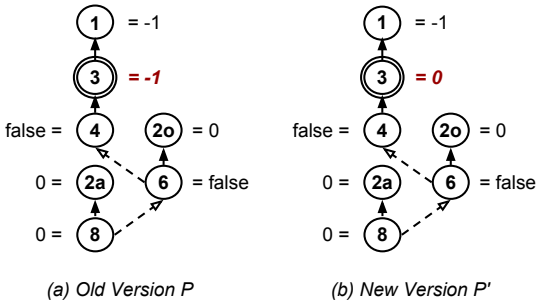

Fig. 5. Intuition of Propagation Condition

Figure 5 shows the dynamic dependency graphs augmented by concrete values and computed for the execution of input (-1) upon the version pairs in Figure 2. The dashed arrows indicate potential dependence while the concrete arrows indicate either dynamic data- or control-dependence. Each node is labeled with the line number of the statement instance it represents. The values for the instance of line 3 are different in both versions. That is, the program state is "infected" after the execution of the change. However, the value of the output in line 8 is the same for both versions. Why?

The semantic effect of the change in line 3 is not propagated to the output in line 8 for the execution of (-1) on P and P' because the branch in line 4 is evaluated in the same direction in both versions even though it dynamically depends on the statement in line 3, which carries different values in both versions. In the remainder of this section, we explain how the instance of line 4 is added to the convergence set N_i and define the propagation condition based on N_i.

As shown in Algorithm 3, both dynamic dependency graphs (DDGs) are computed over the traces π and π' for the execution of input t on both versions P and P'. The DDGs are augmented by potential dependencies and the concrete values for the variables used in every node. Output instances o_i in π and o'_i in π' are aligned and passed into procedure PROPALIGN to compute N_i recursively.

Algorithm 3 Computing Differential State Convergence N_i

Input: Execution Traces π and π', Output Statement o

1: $aDDG \leftarrow augmentedDDG(\pi)$
2: $aDDG' \leftarrow augmentedDDG(\pi')$
3: $(o_i, o'_i) \leftarrow alignableOutput(aDDG, aDDG', o)$
4: $N_i \leftarrow \emptyset$
5: **if** $isChanged(o'_i)$ **then** add (o_i, o'_i) to N_i
6: **else** call PROPALIGN(o_i, o'_i)
7: **procedure** PROPALIGN(s_i, s'_i)
8: $R_i \leftarrow s_i.getDependsOn()$
9: $R'_i \leftarrow s'_i.getDependsOn()$
10: **for all** $r'_i \in R'_i$ **do**
11: **if** $\neg isChanged(r'_i) \wedge \exists r_i \in R_i.\ align(r_i, r'_i) \wedge (value(r_i) = value(r'_i))$ **then**
12: call PROPALIGN(r_i, r'_i)
13: **else**
14: add (s_i, s'_i) to N_i and **return**
15: **end if**
16: **end for**
17: **end procedure**

Output: Statement instances N_i

Assuming that instances s_i and s'_i can be aligned, the tuple (s_i, s'_i) is added to the set N_i if 1) not all of the "subsequent" instances r'_i can be aligned, 2) the values of the variables used in r_i and r'_i are different, or 3) r'_i is a changed statement. This is represented by the intuitively named predicates in line 10. Note, we do not assume that both DDGs can be aligned completely, which would be rather difficult indeed due to the different number of instances every statement can have in both executions. Instead, the alignment begins from the output statement instances (e.g., `return`), which we assume to be alignable, and follow the dependence edges recursively. The instance at which alignment fails is added to N_i. In Figure 5, the instances in line 4 are added to N_i because they depend on the changed statement in line 3 that also has different values.

Definition 4 (*Propagation Condition*)

Let statements C in program P be changed to C' yielding P'. Given traces π and π' for the execution of input t on P and P' and Algorithm 3 computes N_i for π and π' and program output statement o, the propagation condition is defined as $prop(o, \pi, \pi', C, C') \stackrel{def}{=} \forall(n_i, n_i') \in N_i . rsc(n_i, \pi) \wedge rsc(n_i', \pi') \wedge value(n_i) = value(n_i') \wedge \bigwedge_{c \in C} reach(c, \pi) \wedge \bigwedge_{c' \in C'} reach(c', \pi')$.

Every input satisfying the same propagation condition does not propagate the effects of the exercised changes for the same reason. Hence, Definition 4 is a conjunction of five conditions. The *necessary conditions 1) and 2)* leverage the property of relevant slices. Note, $rsc(n_i, \pi)$ is the path condition computed over the relevant slice of statement instance n_i in trace π. Every input exercising the same relevant slice w.r.t. n_i, compute the same symbolic value for n_i. The negation of a constituent branch condition may change the computation of n_i and thus enable propagation. The *necessary condition 3)* captures that the symbolic values for the alignable instances in N_i are the same. The negation of such an equivalence condition may enable propagation. The *necessary condition 4) and 5)* captures that those changes (not) exercised by the test case t are also (not) exercised by other input satisfying the same propagation condition. The negation of a constituent branch condition may enable the reachability of other changes.

D. Computing Difference Conditions

Intuitively, input t computes different output because it exercises a certain set of statement instances in P that contribute to computing the symbolic output of P and another set of statement instances in P' that contribute to computing the symbolic output of P'.

Definition 5 (*Difference Condition*)

Let statements C be changed to C'. Given instances o_i of output statement o in trace π and o_i' of o in trace π', the difference condition is defined as $diff(o, \pi, \pi', C, C') \stackrel{def}{=} rsc(o_i, \pi) \wedge rsc(o_i', \pi') \wedge value(o_i) \neq value(o_i') \wedge \bigwedge_{c \in C} reach(c, \pi) \wedge \bigwedge_{c' \in C'} reach(c', \pi')$.

Every input satisfying the same difference condition propagates the semantic effect of the exercised changes for the same reason. To achieve this property, Definition 5 is a conjunction of five necessary conditions. The *necessary conditions 1) and 2)* leverage the property of relevant slices. Every input exercising the same relevant slice w.r.t. o_i, compute the same symbolic value for o_i. The negation of a constituent branch condition may change the computation of o_i and thus disable propagation. The *necessary condition 3)* captures that the symbolic output values are different in both versions. The negation of this condition may disable propagation. The *necessary conditions 4) and 5)* capture that those changes (not) in π or π' are also (not) exercised by other input satisfying the same difference condition. The negation of a constituent branch condition may enable the reachability of other changes.

A set of changed statements C_t *semantically interferes* for the execution of input t on both program versions, if t yields different output in P and P' and every $c \in C_t$ contributes to computing the output. Thione et al. [16] approximate semantic interference based on static data- and control-dependence. It can be used to understand the origin of regression.

Interestingly, every changed statement in the relevant slice of o_i' contributes in computing o_i' and therefore semantically interferes. This allows the developer to inspect the set of changes responsible for an observed semantic difference.

E. Generating Adjacent Test Cases

Algorithm 4 generates "adjacent" test cases from the provided symbolic condition and adds those to the *queue*. It implements *generateAdjacentTestcases* called in Algorithm 1.

Algorithm 4 - GENERATEADJACENTTESTCASES

Input: Condition *cond*, Queue *queue*

1: let $cond = (\psi_0' \wedge \ldots \wedge \psi_m') \wedge (\psi_0 \wedge \ldots \wedge \psi_n) \wedge (\upsilon_0 \wedge \ldots \wedge \upsilon_k)$
2: **for all** υ_i in $[\upsilon_0, .., \upsilon_k]$ **do**
3: $\quad constr \leftarrow (\psi_0' \wedge \ldots \wedge \psi_m') \wedge (\psi_0 \wedge \ldots \wedge \psi_n) \wedge \neg \upsilon_i$
4: $\quad$ **if** exists t^+ that satisfies $constr$ **then**
5: $\quad\quad$ add t^+ to $queue$
6: $\quad$ **end if**
7: **end for**
8: let $(\varphi_0 \wedge \ldots \wedge \varphi_n) \leftarrow reorder(\psi_0 \wedge \ldots \wedge \psi_n)$
9: **for all** i from n to 0 **do**
10: $\quad constr \leftarrow (\psi_0' \wedge \ldots \wedge \psi_m') \wedge (\varphi_0 \wedge \ldots \wedge \varphi_{i-1} \wedge \neg \varphi_i)$
11: $\quad$ **if** exists t^+ that satisfies $constr$ **then**
12: $\quad\quad$ add t^+ to $queue$
13: $\quad$ **end if**
14: **end for**
15: let $(\varphi_0' \wedge \ldots \wedge \varphi_m') \leftarrow reorder(\psi_0' \wedge \ldots \wedge \psi_m')$
16: **for all** i from m to 0 **do**
17: $\quad constr \leftarrow \varphi_0' \wedge \ldots \wedge \varphi_{i-1}' \wedge \neg \varphi_i'$
18: $\quad$ **if** exists t^+ that satisfies $constr$ **then**
19: $\quad\quad$ add t^+ to $queue$
20: $\quad$ **end if**
21: **end for**

Output: Queue *queue*

The symbolic condition is composed of branch conditions $(\psi_0' \wedge \ldots \wedge \psi_m')$ in P', branch conditions $(\psi_0 \wedge \ldots \wedge \psi_n)$ in P, and equivalence conditions υ of the form $value(s_i) = value(s_i')$ or $value(s_i) \neq value(s_i')$ (cf. line 1). First, the constituent equivalence conditions υ_0 to υ_k are negated one-by-one (lines 2-7). If there exists a solution to the computed constraint, it is added to the queue. Second, if some branch conditions are removed from a path condition, the remaining branch conditions have to be *reordered* before negation (lines 8-9). Otherwise, the exploration algorithm ceases to be exhaustive (cf. [7]). Hence, the branch conditions $(\psi_0 \wedge \ldots \wedge \psi_m)$ in P are reordered as follows: If a branch instance b is in the relevant slice of branch instance b_k, then the branch condition of b is placed before the branch

condition of b_k. Otherwise, the branch condition of b is placed after the branch condition of b_k. The reordered branch conditions in P are negated one-by-one and conjoined with $(\psi'_0 \wedge \ldots \wedge \psi'_m)$ in P' (line 10-15). If there exists a solution to the computed constraint, it is added to the queue. Lastly, the branch conditions in P' are reordered and negated one-by-one (lines 16-23). Again, if there exists a solution to the computed constraint, it is added to the queue.

F. Theorems

In the following, we postulate the soundness of Algorithm 2 that computes the differential partition for a given test case and the exhaustiveness of Algorithm 1 that explores differential partitions. For the lack of space, the detailed proof has been moved to the corresponding technical report [17]. In practice, the absence of regression errors can be guaranteed for all inputs to the same extent as symbolic execution can guarantee the absence of program errors (see e.g., [18]). Specifically, we assume deterministic program execution.

Theorem 1 (*Sound Generalization*)

Given statements C in program P are changed to C' yielding P', every input satisfying the condition computed by Algorithm 2 for input t is in the same differential partition as t.

Informally, the differential behavior of a point in the common input space is soundly generalized to the set of points in the same differential partition. In particular, let Algorithm 2 compute the symbolic condition Φ for a test case t. If t is equivalence-revealing, then every input satisfying Φ is equivalence-revealing. Similarly, if t is difference-revealing, then every input satisfying Φ is difference-revealing.

Theorem 2 (*Exhaustive Exploration*)

If there exists an input t_0 that computes different values for the output o in versions P and P' and Algorithm 1 terminates with regression test suite T, then there exists a test case $t \in T$ so that t_0 satisfies $diff(o, \pi(t, P), \pi(t, P'), C, C')$.

Informally, if the verification procedure terminates then all differential partitions have been explored. The respective proof leverages the exhaustiveness of the exploration based on relevant slices as shown in [7].

III. Empirical Study

Our experiments evaluate the relative efficiency of PRV and discuss practicability based on our experience. The experiments do not prove the scalability of PRV. In fact, PRV suffers from the same limitations as symbolic execution. Similarly, it can benefit from relevant optimizations such as domain reduction [19], [20], parallelization [21], and better search strategies [22], [23].

A. Setup and Infrastructure

PRV has been implemented into our dynamic backward slicing tool JSlice [24]. The differential partitions are explored in a breadth-first manner starting from the same initial input within the time bound of five minutes, unless stated otherwise. Every version of the same subject uses the same test driver to construct necessary input objects, strings, or arrays from the input integers that come as solution to a first-order logic formula from the Z3-constraint solver [25]. The subject programs are analyzed on a desktop computer with an Intel 3GHz quad-core processor and 4GB of memory.

B. Subject Programs

The subjects summarized in Figure 6 are chosen according to two criteria: 1) they represent a variety of evolving programs and 2) are discussed in related work (which allows the comparison with our own experimental results). There are 83 versions of programs ranging from 20 to almost 5000 lines of code (LoC). Some versions are derived by seeding faults, called mutants, of the original versions. Some are real versions that were committed to a version control system.

Subject	Reference	Classes	Functions	LoC	Versions
Min	[26]	1	1	20	5
Tcas	[1], [13], [27], [28]	1	8	166	21
Replace	[12], [27]	1	21	564	33
Siena	[12]	6	107	1529	7+11
Apache CLI		22	183	4966	6
	Total	*30*	*320*	*7245*	*83*

Fig. 6. Subject Programs

We compare the empirical results of the references discussing regression verification [1] and regression test generation [12], [13], [27], [28]. Note, there are no empirical results available for the regression test generation techniques [10], [11], [29] and differential symbolic execution [30].

Min [26] is a short function introduced to discuss the problem of equivalent mutants. An equivalent mutant is a simple syntactic change to a program that yields no semantic difference. *Tcas* is the traffic collision avoidance system. This well-studied program is available in the SIR [31] with several versions that contain seeded faults. We chose the first 20 changed versions. *Replace* performs pattern matching and substitution and is available in the SIR with 32 versions that contain seeded faults. *Siena* is an event notification architecture. Note, there are 7 versions available in the SIR and for every version there exist between one and four faulty versions (in total 11 mutants).

Revision	Submission	Developer's Submission Comment
129800	15.08.2002	bug. no 11680 resolved
129803	18.08.2002	bug #11457: implemented fix [..]
129843	14.11.2002	added fix for Rob's problem [..]
129849	19.11.2002	some bug fixes submitted by Rob [..]
538031	15.05.2007	Applying Brian Egge's fix from CLI-13
667565	13.06.2008	Restored CLI 1.0 behavior (CLI-137)

Fig. 7. Apache CLI Revisions (http://commons.apache.org/cli/)

Apache CLI is an open source command line interpreter. We retrieved the six revisions from the version control system (branches/cli-1.x/src) that are presented in Figure 7 along with the submission date and comments and the unique identifiers.

All programs are tested as whole programs, except for Apache CLI. In this case, the the command line component was tested for regression. The first three programs all have a main method. For Siena, `encode` and `decode` in the class `SENP` serve as main methods. For Apache CLI, `addOption` and `getOptionValue` in the class `CommandLine` serve as testing hooks.

C. Research Questions

*RQ1: How efficiently does PRV find the first input that exposes **semantic difference**?*
Empirical studies in the discussed related work are concerned with finding the first difference-revealing input as witness of semantic difference. We compare the efficiency of PRV to the efficiency reported in related work.

*RQ2: How efficiently does PRV find the first input that exposes **software regression**?*
Not every difference-revealing input exposes software regression. In fact, after syntactic changes to the program, semantic changes may be anticipated in the form of progression. For instance, when a buggy program is fixed input failing in the buggy version is supposed to pass in the fixed version. To classify a semantic change as regression, we have to define correctness. As often in reality, we assume the *absence* of formal specifications. In this scenario, the developer checks the generated difference-revealing test cases informally against her expectation. If she observes regression, the developer can relate the regression-revealing test cases to the changes that semantically interfere.

RQ3: How practical is PRV in an example usage scenario?
The subject Apache CLI shall be used to evaluate PRV in a practical usage scenario. PRV generates difference-revealing test cases within the bound of 20 minutes for every version pair. We classify the generated test cases (e.g. regression-revealing) and compare the (informal) measure of regression and progression to the submission comments in Figure 7.

IV. Results and Analysis

RQ1: Efficiency - Semantic Difference

We measure two aspects when searching for the first difference-revealing input as shown in Table I. The first seven rows show the *average time* to find a difference-revealing input per subject. If for a version pair none of the approaches finds a difference-revealing test case within five minutes, then it does not contribute to the calculation of the average time. The *mutation score* depicts the fraction of versions for which a difference-revealing input can be found within five minutes. To gather results for the symbolic execution of the changed version P', we implemented a DART-like [6] and eXpress-like [12] path-exploration technique (Columns 3-4) into JSlice. The DART-like technique explores all paths in P' while the eXpress-like technique prunes all paths that do not exercise a changed statement in P'. The results for the exploration of the differential behavior of both versions, P and P', are gathered using PRV (Column 2).

TABLE I
First Witness of Semantic Difference

	P, P'	only P'	
	PRV	**DART**-like	**eXpress**-like
***Average Time** in sec*			
Min (4 Mutants)	0.4	0.3 (-25%)	0.3 (-25%)
Tcas (20 Mutants)	5.6	20.9 (+273%)	20.7 (+270%)
Replace (32 Mutants)	22.8	130.5 (+472%)	60.1 (+164%)
Siena (11 Mutants)	30.7	66.2 (+116%)	40.4 (+32%)
Siena (7 Versions)	14.0	18.3 (+31%)	12.7 (-9%)
Apache CLI (6 Versions)	57.8	38.9 (-33%)	45.1 (-22%)
***Mutation Score** - fraction of versions shown semantically different*			
Min (4 Mutants)	0.75	0.50 (-33%)	0.50 (-33%)
Tcas (20 Mutants)	1.00	0.56 (-44%)	0.56 (-44%)
Replace (32 Mutants)	0.76	0.56 (-26%)	0.63 (-17%)
Siena (11 Mutants)	0.82	0.73 (-11%)	0.73 (-11%)
Siena (7 Versions)	0.67	0.67 ($\pm$0%)	0.67 ($\pm$0%)
Apache CLI (6 Versions)	1.00	1.00 ($\pm$0%)	1.00 ($\pm$0%)

Answer to RQ1. For the analyzed subjects, *PRV generates a difference-revealing test case on average for 21% more version pairs in 41% less time*, than the eXpress-like approach that analyzes only the changed version P'. For the subtle, seeded faults PRV can find a difference-revealing test case more efficiently. In particular, Tcas is fully analyzed within the time bound by all approaches but only PRV can find a difference-revealing test case for every mutant supporting the motivation illustrated in Figure 2. In general, PRV's relative efficiency is better for the first four subjects containing subtle, seeded faults. This efficiency reduces as the changes become more complex in the latter two subjects. This can be attributed to the increased number of changed statements correlating with an increased probability to reveal a difference (for random input). However, not every difference-revealing test case is also regression-revealing as analyzed in RQ2.

Compared to DART, our eXpress-like implementation has a similar relative efficiency than eXpress in [28] and [12]. The authors compare full path exploration (Pex) to pruning paths that do not execute a changed statement (Pex-eXpress). For Siena, Replace, and the chosen mutants of Tcas, the authors report an improvement in terms of time of 29%, 57%, and 13%, respectively. For these subjects, we see a similar improvement of 37%, 54%, and 16% of the eXpress-like approach over the DART-like approach, respectively.

Mutation Score	Matrix [13]	SHOM [27]	PRV [this]
Tcas	62.7%	62%	100%
Replace	-	72%	76%

Fig. 8. PRV mutation scores vs SHOM and Matrix

Santelices et al. [13] and Harman et al. [27] report the mutation score for the test generation tools Matrix and SHOM, respectively. Note that many of the subjects used by these authors and us are different. However, Tcas was used to evaluate Matrix and SHOM, while Replace was also used to evaluate SHOM. As shown in Figure 8, PRV compares favourably for the commonly evaluated subjects, Tcas and Replace. The last row shows the average mutation score for the subjects evaluated in the respective references. In contrast to these

search-based techniques, PRV avoids searching for difference-revealing test cases within the already explored input space. This may help explaining the observed improvements.

Godlin et al. [1] evaluate the implementation of regression verification using randomly generated programs and Tcas. It takes many hours or the system runs out of memory when analyzing non-equivalent programs. Offutt et al. [26] discuss the problem of equivalent mutants using subject `Min`. PRV guarantees equivalence for Mutant 3 and provides a witness for the other non-equivalent mutants in less than a second.

RQ2: Efficiency - Software Regression

In practice, not every difference-revealing test case reveals software regression. A difference-revealing test case can be checked formally or informally against the programmer's expectation. In the latter case the programmer looks at the output of difference-revealing test cases in both programs and may know whether the test case reveals regression. Table II presents the two aspects measured to find the first regression-revealing input. The first seven rows show the *average time* to find a regression-revealing input per subject. If for a version pair none of the approaches finds a regression-revealing test case within the time bound, then it does not contribute to the calculation of the average time. The *mutation score* depicts the fraction of versions for which a regression-revealing input can be found within 20 minutes for Apache CLI and five minutes for the other subjects.

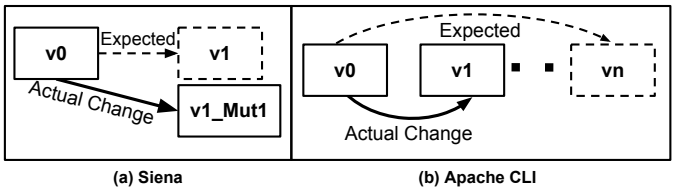

Fig. 9. How to Measure Regression?

How do we measure regression? For Siena, we simulate an incomplete bug fix [32] from one version to the next. An example is shown in Figure 9.a). The programmer fixes Siena.v0 which is expected to behave like Siena.v1. Instead, he introduces another bug yielding Siena.v1_Mut1 – a version of Siena.v1 that also contains seeded faults. For Apache CLI, there are no seeded faults available. But we can capture the programmer's idea of the expected behavior to be in the last revision 667565 which remains unchanged for the last four years. This allows us to measure the regression of intermediate revisions w.r.t. the last revision. For Min, Tcas, and Replace every difference-revealing test case also reveals a regression.

Answer to RQ2: For the analyzed subjects, *PRV generates a regression-revealing test case on average for 48% more version pairs in 63% less time* than the eXpress-like approach that analyzes only the changed version P'. The improvement of efficiency over finding a single difference-revealing input (cf. Table I) may be attributed to the subtleness of regression faults. As an instance of this subtleness, consider the program versions in Figure 2. The programmer expects that the new

TABLE II
FIRST WITNESS OF SOFTWARE REGRESSION

	P, P'	only P'	
	PRV	**DART**-like	**eXpress**-like
***Average Time** in sec*			
Min (4 Mutants)	0.4	0.3 (-25%)	0.4 (-25%)
Tcas (20 Mutants)	5.6	20.9 (+273%)	20.7 (+270%)
Replace (32 Mutants)	22.8	130.5 (+472%)	60.1 (+164%)
Siena (11 Faulty Versions)	17.6	50.4 (+186%)	44.1 (+151%)
Apache CLI (6 Versions)	141.3	259.6 (+84%)	263.9 (+87%)
***Mutation Score** - fraction of versions exposed as regression*			
Min (4 Mutants)	0.75	0.50 (-33%)	0.50 (-33%)
Tcas (20 Mutants)	1.00	0.56 (-44%)	0.56 (-44%)
Replace (32 Mutants)	0.76	0.56 (-26%)	0.63 (-17%)
Siena (11 Faulty Versions)	0.55	0.45 (-17%)	0.45 (-17%)
Apache CLI (6 Versions)	0.40	0.20 (-50%)	0.20 (-50%)

version computes output $o = i + 1$ instead of $o = i$ for input $i > 0$. Otherwise, the behavior shall remain unchanged. Thus, a regression test generation tool may determine progression for almost 50% of the input ($i > 0$) but because a branch is evaluated in different directions for $i = 0$, there exists regression only for one input. Unintendedly, this input computes different output in the changed version, too. Even the generation of an input for every path in the changed program, like for eXpress, may not produce this test case. In contrast, differential partitions can capture such subtle differences.

RQ3: Practicability - Usage Scenario: Apache CLI

Apache CLI is used to evaluate PRV in a practical usage scenario. PRV generates difference-revealing test cases within the bound of 20 minutes for every version pair. A developer checks these test cases for regression and relates the regression-revealing test cases to the changes that semantically interfere. The check is automated in our experiment as illustrated in Figure 9.b). The expected behavior of CLI is captured by the last revision (667565) which has not changed in the last four years and is released in CLI1.1. This allows us to measure progression and regression w.r.t. to the expected behavior.

The first column in Table III shows the revision pairs, the earlier versus the later revision. The second column presents the total number of tests generated by PRV followed by the number of equivalence- and difference-revealing test cases, respectively. The percentage of difference-revealing test cases (Column 4) witnessing progression (**%Progr**), regression (**%Regr**), and the computation of output that has changed but still does not behave as expected (**%Chan**) are shown in columns 5, 6, and 7, respectively.

TABLE III
EXPLORATION OF DIFFERENTIAL BEHAVIOR IN LIMITED TIME

	Subject and Versions	**#Test**	**#Equ**	**Difference Revealing**			
				#Diff	**%Progr**	**%Regr**	**%Chan**
CLI (20min)	r129800-r129803	788	748	40	0%	0%	100%
	r129803-r129843	835	809	26	65%	0%	35%
	r129843-r129849	721	639	82	82%	1%	17%
	r129849-r538031	509	485	24	0%	***88%***	13%
	r538031-r667565	536	455	81	100%	0%	0%
	Average				**49%**	**18%**	**33%**

Answer to RQ3: For the evolution of Apache CLI over six years, tests generated as witnesses of differential behavior of two successive versions suggest an *average progression of 49%, regression of 18% and intermediate semantic changes of 33%* towards the latest revision. The interested reader may compare the results in Table III to the developer's notes in Figure 7. The behavior of CLI generally experiences progression from version r129800 to r129849 when suddenly the behavior regresses with the change to r538031. In fact, while trying to fix bug CLI-13[2], the developer introduces bug CLI-137[3]. This is a clear regression bug which is witnessed by 88% of the difference-revealing test cases generated by PRV. However, it takes two months to report and twelve to fix bug CLI-137 and commit it as revision r667565. In contrast, PRV generates the first regression-revealing test case for r538031 in 88 seconds among the first five generated difference-revealing test cases.

V. Threats to Validity

The main threat to *internal validity* is the correctness of our implementation of PRV into JSlice. We tried to mitigate this threat by using the same implementation to gather results for the DART-like and eXpress-like approaches. In practice, any implementation of PRV can guarantee the absence of regression errors to the same extent as symbolic execution can guarantee the absence of program errors (see e.g., [18]). Our particular implementation could be faulty, so that it may not report a witness of behavioral difference if one exists. On the other hand, a reported witness of behavioral difference is indeed a witness of behavioral difference. This is inherent to the approach, as the generated test cases are concretely (and symbolically) executed on both programs.

The main threat to *external validity* is the generalization of our results. The limited choice and number of subjects does not suggest generalizability and serve mainly as comparison to related work and give an idea about the practicability of PRV.

VI. Related Work

Regression Verification (RV) is the problem of deciding whether a changed program is *at least as correct* as a previous version. One line of work takes an earlier version as a program specification of the new version [1], [2], [33]. The authors argue if both versions are semantically equivalent, then there is no software regression. Yet, not every difference is a regression. For instance, a bug-fix yields anticipated behavioral difference. Another line of work requires an explicit specification, and builds on the full verification of an earlier version. Subsequently, only the changed behavior of the following versions need to be checked incrementally [34], [35]. In general, RV can take a long time to terminate. When the search is interrupted, no intermediate guarantees can be reported. In contrast, the interruption of PRV can guarantee the absence of regression at least for the explored equivalence-revealing partitions. Moreover, the difference-revealing test cases can be "informally" checked for further regression by developers.

[2]https://issues.apache.org/jira/browse/CLI-13

[3]https://issues.apache.org/jira/browse/CLI-137

Differential Symbolic Execution (DSE) [30] is a general approach to compute program differences while PRV is a specialized approach tailored to RV. Specifically, DSE computes the differences based on two types of program summaries. The *symbolic summaries* in Figure 2 on the right (P and P') precisely characterize the behavior of the program versions on the left. The *abstract summaries* in Figure 10 over-approximate the behavior for the same versions. Exploiting the syntactic similarity of both versions, the behavior of common code blocks can be represented by uninterpreted functions.

	Input	**Output**	Regression Test Case t
$P\ (=\Delta_{\langle P,P'\rangle})$	*true*	$o = o(0,i)$	Value for i satisfying
$P'\ (=\Delta_{\langle P',P\rangle})$	*true*	$o' = o(0,i+1)$	$o(0,i) \neq o(0,i+1)$

Fig. 10. Program Deltas (Δ) and Abstract Summaries (cp. Fig.2)

RV based on program summaries is either less scalable or infeasible. While symbolic summaries may be used for RV, the differences are computed as an expensive cross-product of (incomplete) summaries. On the other hand, PRV is based on differential partitions that account for the common input space of both versions. Furthermore, PRV yields coarser partitions. For instance, if input does not reach a change already implies both programs compute the same output. Abstract summaries, on the other hand, remove information required for RV. The interested reader may verify in Figure 10 that, if the delta contains uninterpreted functions, a concrete difference-revealing test case cannot be generated. In this example, each delta accounts for a single partition ($true$), while PRV distinguishes two difference-revealing and one equivalence-revealing partition. For each partition, PRV generates a witnessing test case.

Regression Test Generation (RTG) is the problem of constructing sample input that can expose software regression. Classically, test cases are generated towards the coverage of the program's behavior [4], [6]. The hope, when the program is changed and behavior regresses, is that at least one test case fails in the new version. Further, when the program is changed, test cases are generated towards the coverage of program elements that are affected by the syntactic changes [11], [12], [36]. These test cases *augment* an existing test suite that was coverage-adequate for the earlier version. However, the analysis of a single program may be insufficient to generate a regression-revealing test case (cf. Fig. 2). Instead, some research directly aims at generating difference-revealing test cases. Syntactic approaches seek to *reach* a change, *infect* the program state, and *propagate* it to the output [9], [10], [13], [27]. However, the number of possibly semantically interfering sets of changes is exponential to the number of overall syntactic changes. Harman et al. [27] note the testing of every subset would be prohibitively expensive. Even for a single subset, the search for a difference-revealing input may not terminate. In contrast, the number of changes is unimportant to PRV, à priori. It groups input, depending on whether it reaches and propagates the same set of changes, on demand during exploration. More importantly, PRV can guarantee the absence of regression not only for a singular point in the common input space, but for an entire partition.

VII. Conclusion

Regression testing is probably the most widely used testing technique. In theory, an adequate regression test suite covers many *semantic elements* in a program, that is, different functionalities. The hope is, when the program is changed and existing functionality stops working as expected, at least one test case fails. In practice, the adequacy of a regression test suite is measured in terms of covered *syntactic elements* in a program, that is, code coverage. For instance, a test suite is 100% branch coverage-adequate if it exercises every branch.

However, code-coverage may not properly quantify the capability of a test suite to reveal regression errors, as Böhme motivates in [37]. While syntactic coverage is a practical approximation of semantic coverage, it remains unclear whether a Coverage-adequate Test Suite (CaTS) performs significantly better in terms of revealing errors than the execution of random input [38], [39]. Weyuker et al. [4] argue that any (failing) test case in a CaTS may not be representative of (the observed) errors. Moreover, even if *all* CaTS tests execute successfully, that shall not inspire confidence in correctness [5].

We propose the exploration of differential partitions to group input with the same "differential behavior". As such, each generated test case becomes significant and representative of a larger set of inputs. Once a test case is executed on both versions and exposes a difference, PRV can *soundly* generalize to those inputs that are also difference-revealing. This allows to assess the adequacy of a test suite not in terms of the covered code elements but the covered *input space*. PRV shall inspire confidence in the absence of regression at least for input in the covered input space - establishing a form of "comfort zone". Experiments show that PRV exposes regression errors that are not detected by other regression test generation methods.

In summary, differential partitions enable a gradual and partial form of regression verification. Differential partitions exist as a unit of verification in the common input space of two program versions and are checked one after another. When the verification process is interrupted, PRV retains regression guarantees for the explored input space. This is crucial as verifying the complete input space is prohibitively expensive.

Acknowledgment

This work was partially supported by Singapore's Ministry of Education research grant MOE2010-T2-2-073.

References

[1] B. Godlin and O. Strichman, "Regression verification: proving the equivalence of similar programs," *Softw. Test., Verif. Reliab.*, pp. 1–18, March 2012. [Online]. Available: http://dx.doi.org/10.1002/stvr.1472

[2] S. Chaki, A. Gurfinkel, and O. Strichman, "Regression verification for multi-threaded programs," in *VMCAI*, 2012, pp. 119–135.

[3] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[4] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. Softw. Eng.*, vol. 17, no. 7, pp. 703–711, Jul. 1991.

[5] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence (program testing)," *IEEE Trans. Softw. Eng.*, vol. 16, no. 12, pp. 1402–1411, Dec. 1990.

[6] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI*, 2005, pp. 213–223.

[7] D. Qi, H. D. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," in *ESEC/SIGSOFT FSE*, 2011.

[8] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler, "Conditional model checking: a technique to pass information between verifiers," in *FSE*, 2012, pp. 57:1–57:11.

[9] B. Korel and A. M. Al-Yami, "Automated regression test generation," in *ISSTA*, 1998, pp. 143–152.

[10] D. Qi, A. Roychoudhury, and Z. Liang, "Test generation to expose changes in evolving programs," in *ASE*, 2010, pp. 397–406.

[11] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *PLDI*, 2011, pp. 504–515.

[12] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux, "express: guided path exploration for efficient regression test generation," in *ISSTA'11*.

[13] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *ASE*, 2008.

[14] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London, "Incremental regression testing," in *ICSM*, 1993, pp. 348 – 357.

[15] T. Gyimóthy, A. Beszédes, and I. Forgács, "An efficient relevant slicing method for debugging," in *ESEC/SIGSOFT FSE*, 1999, pp. 303–321.

[16] G. L. Thione and D. E. Perry, "Parallel changes: Detecting semantic interferences," in *COMPSAC*, 2005, pp. 47–56.

[17] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, "Partition-based regression verification," National University of Singapore, Tech. Rep. TRB8/12, '12, http://www.comp.nus.edu.sg/~mboehme/PRV-TR12.pdf.

[18] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons, "Proofs from tests," in *ISSTA '08*, pp. 3–14.

[19] P. Boonstoppel, C. Cadar, and D. R. Engler, "Rwset: Attacking path explosion in constraint-based test generation," in *TACAS*, 2008, p. 351.

[20] M. Delahaye, B. Botella, and A. Gotlieb, "Explanation-based generalization of infeasible path," in *ICST*, 2010, pp. 215 –224.

[21] M. Staats and C. S. Pasareanu, "Parallel symbolic execution for structural test generation," in *ISSTA*, 2010, pp. 183–194.

[22] J. Malburg and G. Fraser, "Combining search-based and constraint-based testing," in *ASE*, 2011, pp. 436–439.

[23] P. McMinn, M. Harman, D. Binkley, and P. Tonella, "The species per path approach to searchbased test data generation," in *ISSTA*, 2006, pp. 13–24.

[24] T. Wang and A. Roychoudhury, "Using compressed bytecode traces for slicing Java programs," in *ICSE*, 2004, pp. 512–521.

[25] L. M. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, 2008, pp. 337–340.

[26] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Softw. Test., Verif. Reliab.*, vol. 7, pp. 165–192, 1997.

[27] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *ESEC/SIGSOFT FSE*, 2011, pp. 212–222.

[28] K. Taneja, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Guided path exploration for regression test generation," in *ICSE Companion*, 2009, pp. 311–314.

[29] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," in *ICST*, 2010, pp. 137–146.

[30] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu, "Differential symbolic execution," in *SIGSOFT FSE*, 2008, pp. 226–237.

[31] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empir. Softw. Eng.*, vol. 10, no. 4, pp. 405–435, 2005.

[32] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. N. Bairavasundaram, "How do fixes become bugs?" in *SIGSOFT FSE*, 2011.

[33] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: A language-agnostic semantic diff tool for imperative programs," in *CAV*, 2012, pp. 712–717.

[34] G. Yang, M. B. Dwyer, and G. Rothermel, "Regression model checking," in *ICSM*, 2009, pp. 115–124.

[35] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan, "Incremental state-space exploration for programs with dynamically allocated data," in *ICSE*, 2008, pp. 291–300.

[36] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: techniques and tradeoffs," in *FSE '10*, pp. 257–266.

[37] M. Böhme, "Software regression as change of input partitioning," in *ICSE*, 2012, pp. 1523–1526.

[38] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Trans. Softw. Eng.*, vol. 10, no. 4, pp. 438 –444, July 1984.

[39] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 258–277, 2012.