Verification-Aided Regression Testing

Fabrizio Pastore,
Leonardo Mariani
University of Milano -
Bicocca. Milan, Italy
{pastore,mariani}
@disco.unimib.it

Antti E. J. Hyvärinen,
Grigory Fedyukovich,
Natasha Sharygina
University of Lugano.
Lugano, Switzerland
{antti.hyvaerinen,
grigory.fedyukovich,
natasha.sharygina}@usi.ch

Stephan Sehestedt
ABB Corporate Research.
Ladenburg, Germany
stephan.sehestedt@de.abb.com

Ali Muhammad
VTT Technical Research
Centre. Tampere, Finland
Ali.Muhammad@vtt.fi

ABSTRACT

In this paper we present *Verification-Aided Regression Testing* (VART), a novel extension of regression testing that uses model checking to increase the fault revealing capability of existing test suites. The key idea in VART is to extend the use of test case executions from the conventional direct fault discovery to the generation of behavioral properties *specific to the upgrade*, by (i) automatically producing properties that are proved to hold for the base version of a program, (ii) automatically identifying and checking on the upgraded program only the properties that, according to the developers' intention, must be preserved by the upgrade, and (iii) reporting the faults and the corresponding counter-examples that are not revealed by the regression tests. Our empirical study on both open source and industrial software systems shows that VART automatically produces properties that increase the effectiveness of testing by automatically detecting faults unnoticed by the existing regression test suites.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification

Keywords

Regression testing, model checking, dynamic analysis

1. INTRODUCTION

A typical software development process produces a sequence of program versions each improving the functionality of the previous version. Upgrades might inadvertently reintroduce or create programming faults that should ideally be detected before the release of the upgraded program. Regression testing aims to detect such *regression faults* early in the development phase. The idea is that developers design and maintain across versions a test suite that can be executed after each program upgrade to reveal faults. The designed test suite typically covers as high number of statements in the code or the specification as is practically possible. However high code coverage does not necessarily imply high fault detection, even in the cases where the fault affects a covered functionality [65]. For instance, if an online shop has a fault in its checkout function that is triggered by the presence of a specific item in the cart, a test suite that ignores that item will never be able to detect the fault.

This paper demonstrates how *bounded model checking* [16, 19, 40, 11] can be efficiently combined with testing and dynamic invariant detection [25, 43, 20] to automatically discover problems that would otherwise go unnoticed by traditional regression testing. The key idea to detect faults missed by regression testing is to verify the upgraded version of a program against a set of properties that: (i) are obtained by monitoring the execution of the regression suite on the base version of the program, (ii) have been verified to hold for the base version, and (iii) are not violated by the regression suite designed for the upgraded version. This key idea is implemented in the *Verification-Aided Regression Testing* (VART) technique, which integrates testing, invariant detection, and model checking into a novel verification technique that is

(1) capable of automatically detecting faults that are not revealed by conventional regression testing;

(2) less expensive to use than conventional model checking, since the properties useful to check the upgrade are automatically generated from the regression tests (see conditions (i) and (ii)) instead of being manually specified by the testers; and

(3) sensitive to the upgrade semantic, since the properties that are intentionally invalidated by the change under test are automatically eliminated (see condition (iii)).

There are techniques that can reveal faults by exploiting statically and dynamically extracted properties [60, 39]. These techniques generate useful results but often produce a high number of false alarms that annoy the users. The false alarms are due to the generation and usage of properties that are not sound but are simply *likely to be true*. Some approaches can eliminate false alarms by generating tests that confirm the discovered faults, but they are limited to faults producing crashes and uncaught exceptions only [50, 21]. VART augments the effectiveness of regression testing

ISSTA '14, July 21-25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07 ...$15.00.

with an automated verification capability that has two key qualities: in addition to crashes and uncaught exceptions *it automatically detects faults that typically require a user-specified oracle to be revealed* (e.g., faults that cause wrong outputs), and *it has an almost negligible risk of producing false positives*.

Other techniques for automated change analysis include Differential Symbolic Execution (DSE) [49] and Behavioral Regression Testing (BERT) [32]. These techniques can detect the modified behaviors but they cannot point at the faults introduced by the upgrade, as VART does.

In the area of model checking, there are several approaches to check upgrades, but they need either manually specified properties [54, 64] or behavioral equivalence between the base and upgraded program version [29, 31]. Contrarily to these approaches VART automatically generates the properties that need to be verified and it is not limited to changes that do not modify the features of the program.

The empirical results obtained with open source and industrial systems confirm that VART can augment the effectiveness of regression test cases by timely and automatically detecting faults that are not revealed by the existing regression test cases and that require manually specified oracles to be addressed with state of the art techniques. VART should thus be considered as a natural complement to conventional regression testing.

The paper is organized as follows. Section 2 provides background information about the techniques used in this paper. Section 3 describes VART. Section 4 introduces a running example. Sections 5, 6, and 7 present the generation of dynamic properties, verified properties, and non-regression properties, respectively. Section 8 presents how VART produces the list of faults introduced with the upgrade by checking non-regression properties. Section 9 presents the empirical results obtained with open source and industrial systems. Section 10 discusses related work, and finally Section 11 provides final remarks.

2. PRELIMINARIES

This section provides some background information about three key technologies used in VART: model checking, regression testing, and invariant detection.

A longstanding challenge in software engineering is to automatically prove non-trivial, semantic properties of computer programs. In its full generality this problem is known to be undecidable, but several procedures, while necessarily either incomplete, non-terminating, or restricted to special cases, have been shown to be extremely efficient in practice [14, 41, 13, 6].

In this paper we use *Bounded Model Checking* [12, 53] (BMC), one of the most successful approaches for purely static software verification, to empower regression testing. The idea in BMC is to represent the software together with the properties to be verified as an instance of the propositional satisfiability problem (SAT). Such a representation captures the software behavior exactly, assuming that all the loop bodies in the software are repeated at most a fixed number of times. This approach has several advantages: the logical formulation is usually very compact compared to traditional model checking, where verification is reduced to a reachability problem in a graph representing the program state space; there are several high-performance SAT solvers [57, 23] that can be used for solving the instances; and the satisfying assignments of an instance can be directly translated to meaningful counterexamples for correctness in the form of fault-inducing executions. Furthermore, it is widely recognized that BMC based approaches are particularly good at quickly finding short counterexamples when they exist, making BMC very appealing for the task at hand.

A *bounded model checker* takes as input a program P, a bound k for loop unrolling, and a set S of properties to be verified against P, and returns for each property s_l in S, expressed as a propositional statement over variables of P at a location l, either

- *verified*, if the executions of P satisfy s_l;
- *unreachable*, if no execution of P reaches l;
- *false*, if there is an execution of P where the property s_l is broken; and
- *unknown*, if the checker is unable, due to memory or time limits, to determine whether s_l holds,

under the assumption that no loop body in the program is repeated more than k times.

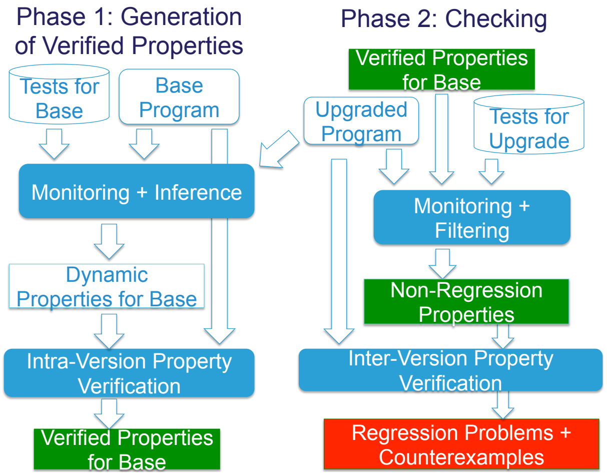
The approach is naturally a compromise between practicality and completeness. As the SAT problem is NP-complete, determining whether s_l holds requires in the worst case exponential time with respect to the size of the SAT instance for all known algorithms. Furthermore, the instances can in some cases grow very large since many operations, such as multiplication, have quadratic encodings in SAT and, for example, the instance grows exponentially in number of nested loops.

Due to numerous optimizations BMC can nevertheless solve many practical problems in reasonable time and memory limits. For example, the size of the resulting SAT instance can be dramatically reduced by slicing off parts of the program that do not affect the validity of the property being checked, and extremely efficient SAT solver implementations which learn the instance structure and use adaptive heuristics [37] rarely suffer from the exponential worst-case behavior in problems emerging from applications. The fact that bounded model checkers only prove correctness of properties for executions not exceeding the bound k is also beneficial in many ways for detecting regressions. In addition to obvious performance benefits our experiments show that in most cases even a single loop iteration is sufficient to indicate a regression between two versions, and a small bound guarantees in a natural way that the reported counterexamples are short. A well-known challenge for BMC is that in practice programs rarely contain manually specified properties. VART answers this challenge by exploiting regression testing and invariant detection to automatically generate the properties that capture the intended behavior of the program.

The main purpose of *regression testing* is to validate that an already tested code has not been broken by an upgrade. To this end regression testing maintains a test suite that can be used to revalidate the software as it evolves [36]. The *regression testing process* consists of adding new tests and, when necessary, modifying the existing tests to validate the software as it evolves.

Dynamic invariant detection exploits software executions, such as the ones produced by executing a regression test suite, to generate likely invariants [25, 43, 20]. A likely invariant is a program property that appears to hold according to the evidence (i.e., the executions) that has been collected so far. Several dynamic invariant detection techniques have been successfully used as part of testing and analysis solutions [52, 21, 38, 62]. In VART we use Daikon [25], a well-known tool that can produce properties in the form of propositional statements over program variables from execution traces.

3. Verification-Aided Regression Testing



Phase 1: Generation of Verified Properties
Phase 2: Checking
Tests for Base
Base Program
Monitoring + Inference
Dynamic Properties for Base
Intra-Version Property Verification
Verified Properties for Base
Verified Properties for Base
Upgraded Program
Tests for Upgrade
Monitoring + Filtering
Non-Regression Properties
Inter-Version Property Verification
Regression Problems + Counterexamples

Figure 1: The VART process.

The VART approach presented in this paper is intended to augment conventional regression testing. Both VART and regression testing are designed to help developers avoid involuntarily creating and reintroducing faults in program upgrades. VART improves regression testing by identifying a list of potential regressions not detected by regression testing, accompanied by the concrete executions demonstrating the regressions. The approach takes as input two versions of a program to be checked: a *base* version and an *upgrade*, the correctness of which is to be determined by VART. In addition the approach uses the two *test suites* that developers usually implement to validate software: one designed to validate the base program and one designed to validate the upgrade. The base test suite is used to derive relevant correctness requirements indicated as *dynamic properties*, while the upgrade test suite is used to identify and eliminate the dynamic properties that are present in the base but are intentionally absent in the upgrade.

The high-level overview of the VART approach is given in Fig. 1. The approach consists of two phases; the first generates the set of *verified properties* from the base program and its regression tests, and the second identifies the regressions of the upgrade and provides the related regressive executions called *counterexamples*. The programmer is then able to use the collected counterexamples for fixing the faults and integrate them to the regression test suite for the upgrade. Once a new upgrade of the program is available, the VART approach can then be re-employed using the previous upgrade as the new base.

The first phase consists of two steps. The first step, labeled *monitoring + inference* in Fig. 1, generates a large number of dynamic properties by observing the behavior of the base program at specific program locations when executing the base test suite. VART generates dynamic properties using an invariant detector such as Daikon [25]. For efficiency reasons the dynamic properties are only generated for the program locations that are likely affected by the change. These locations are identified by comparing the base and upgraded program as discussed in Sec. 5. Since the dynamic properties are heuristically generalized from executions they are potentially imprecise: in particular, some properties might overfit the observed behaviors. For instance, suppose our previous example on a program that implements an on-line shop records the number of bought items in the variable numItems, and satisfies the invariant numItems ≥ 0. If the base test suite for the program only considers cases with less than eight items, the invariant detector might also generate the property numItems < 8.

The second step, *intra-version property verification*, uses a model checker to eliminate the dynamic properties that result from data overfitting and do not hold for the base program in general. The model checker labels each dynamic property either as *verified*, *unreachable*, *false*, or *unknown*. Since VART is intended to be conservative and generate no false alarms, only the properties that the model checker labels as *verified* are included in the set of *verified properties for base*, while the properties labeled as *unreachable*, *false*, and *unknown* are discarded.

The second phase finally determines whether there are verified properties of the base program that are not intentionally invalidated but nevertheless violated in the upgrade. The phase consists of two steps. The *monitoring + filtering* step uses the upgrade test suite to remove the properties that are intentionally broken by the upgrade. Consider again the on-line shop implementation and assume that each item is associated with a function *getType* which in the base version can return only 0 or 1, and the first phase produced a verified dynamic property equivalent to $getType() = 0 \vee getType() = 1$. Now assume that the *getType* implementation in upgrade can also return 2 and the developer has implemented a new test that checks this value. The upgrade test suite, which includes the new test, can then be used to eliminate the corresponding verified property of the base program. Following this intuition VART automatically eliminates the verified properties that are violated during the execution of the upgrade test suite. The properties that are not eliminated in this step are the *non-regression properties* that are expected to hold for the new version of the software, according to the intention of the developers.

The second step, namely *inter-version property verification*, uses model checking to determine whether the non-regression properties hold in the upgrade. Every detected violation and the corresponding counterexample are reported to the developers, who use the output to identify the fault.

Since even the generation of a small number of irrelevant property violations, which are violations not caused by any fault, could be a major obstacle to the adoption of analysis techniques, we have consistently taken design decisions to prevent VART from generating them. As a result there are only two cases in which VART could report a false counterexample: either as a result of a false dynamic property being labeled *verified* by the bounded model checker, or a dynamic property being deliberately invalidated in the upgrade but not checked in the upgrade test suite. When the former occurs the property can be simply dropped. Whereas when the latter occurs the developer can add a new test that violates the property to the upgrade test suite. When re-executing the upgrade test suite VART automatically classifies the false property as outdated and does not include it

```
1  long availableProducts(store* store_data) {
2    list_node* product = store_data→products;
3    long total = 0;
4    while (product ≠ 0) {
5      int avail = isAvailable(product);
6      total += avail;
7      product = product→nxt;
8    }
9    return total;
10 }

11 int isAvailable(list_node* prod) {
12   if (notInitialized(prod)) {
13     return 0;
14   }
15   if (prod→items > 0) {
16     return 1;
17   }
18   return 0;
19 }
```

Figure 2: Sample code: base version

in the set of the non-regression properties. The two cases did not affect our empirical results, in fact only a singe irrelevant property violation has been reported in our experiments.

4. RUNNING EXAMPLE

In the next sections we illustrate VART with a running example. The case we consider is a regression fault that affects the function *availableProducts*, which returns the number of products that are available in a struct of type `store`. The C implementation of this function in the base version is shown in Fig. 2. Note that the *availableProducts* function uses the auxiliary function *isAvailable*, which returns 1 if the product is available, and 0 otherwise.

As an upgrade containing a fault we consider a new version of the function *isAvailable* returning −1 for outdated products no longer part of a catalogue. The updated function is shown in Fig. 3, where the new lines 14a–14c have been framed. The considered upgrade is faulty because the *availableProducts* function has not been changed to accommodate the change in *isAvailable*. Thus the resulting total will be wrong when products that are not in a catalogue occur in the list of products passed as input parameter to the *availableProducts* function. In this example we assume that a base test suite has been used to validate the base program and that the base test suite does not reveal the regression problem when executed on the upgrade. To test the upgrade, we assume developers have added the test case shown in Fig. 4 to the upgrade test suite. This test covers the modification in the function *isAvailable*, since the line 14b in the upgraded program is executed by the test, but does not reveal the regression.

Many popular automated techniques would be quite ineffective against this regression fault. Regression testing techniques can be used to select from the existing test cases the ones that cover the change, but they can never reveal this fault unless the regression test suite includes tests that use products not in the catalogue. However, since the use of products that are not in the catalogue produces no differences in the computation for the base version of the software, this is unlikely to happen. Automatic testing techniques would also be ineffective. In fact the failure produced by this fault does not consist of an exception or a crash, but manifests itself in computing of a wrong total, a non-trivial error typically requiring a human written oracle to be recognized. For the same reason, static analysis solutions, like model checking, cannot discover the problem in the upgrade unless suitable assertions are manually provided by the testers. The use of coverage criteria are also not helpful. For instance the test in Fig. 4 already covers the change and, unless particularly complex coverage criteria such as the *context-dependent def-use coverage* [58] are used, there would exist no indicators of inadequacy requiring the design of additional tests. In practice, revealing this fault requires the definition of a new test case, which might or might not be manually implemented by a developer. In this context, VART offers the unique opportunity of *automatically checking the side effects of changes, compensating testing inefficiencies.* In particular, even if the developer does not add a proper test to reveal this fault, VART can automatically reveal it.

```
11  int isAvailable(list_node* prod) {
12    if (notInitialized(prod)) {
13      return 0;
14    }
14a   if (prod→in_catalog==0) {
14b     return −1;
14c   }
15    if (prod→items > 0) {
16      return 1;
17    }
18    return 0;
19  }
```

Figure 3: Sample code: upgraded version

```
1  int testUnavailableProduct() {
2    list_node* prod = createProduct();
3    prod→name = ”MacBook”;
4    prod→in_catalog = 0;
5    assertEquals(−1, isAvailable(prod));
6  }
```

Figure 4: Sample code: test for the upgrade

5. DETECTING DYNAMIC PROPERTIES

The first step of the VART process (see Fig. 1) consists of executing the regression test suite for the base version of the software, monitoring the program behavior, and distilling dynamic properties from the collected data.

Since the VART process attempts to confirm the correctness of an upgrade, the scope of the monitoring is naturally influenced by the changes between the two versions. After identifying the modified functions, VART collects data from certain program statements that are close to the changes but remain still unchanged between the base and the upgraded program, that is, unchanged statements in functions

- that contain changes;
- that call functions that contain changes; and
- that are called by the functions that contain changes.

The rationale is that VART should derive properties that can be checked on the base and upgraded program versions and that capture the impact of the change under analysis. By selecting the unmodified program statements we guarantee that the local variables that occur in these statements exist in both the base and the upgrade, increasing the probability of generating properties that use these variables and that can thus be checked in both program versions. The properties that use variables no more existing on the upgrade are automatically dropped in Phase 2 (see Sec. 7). The selection of program statements that are close to the change (i.e., the modified functions, their callers and callees) increases the likelihood of detecting properties that are influenced by the change while still keeping the size of the monitored area reasonable.

VART intentionally approximates the impact of a change with the simple rules specified above instead of using techniques like impact analysis [7] and program slicing [56] that are more precise but tend to select large portions of programs even for small changes. The selection strategy implemented in VART keeps the cost of monitoring, inference and model checking under control, avoiding scalability issues in practice [48].

The running example contains a single change which is local to the function *isAvailable*. VART selects for monitoring every program statement in the function *isAvailable*, because the statements that occur in the base version of the function also occur in the upgraded version of the function; every program statement in the *availableProducts* function, which is the only function invoking *isAvailable*; and every program statement in the function *notInitialized*, not shown in the example, which is invoked from *isAvailable*.

For each program location selected for monitoring, VART records the value of every variable defined in the scope of that location. After the test suite for the base version of the program has been executed, VART uses Daikon to generate the set of *dynamic properties* from the recorded data.

Consider, for example, the listing in Fig. 2. For the function *availableProducts* a base regression test suite combined with an invariant detector like Daikon would automatically generate the following dynamic properties:

- product $\neq 0$ after execution of line 2;
- avail $= 0 \vee$ avail $= 1$ after execution of line 5;
- product $\neq 0$ after execution of line 5;
- avail $= 0 \vee$ avail $= 1$ after execution of line 6; and
- total ≥ 0 after execution of line 6;

and for the function *isAvailable* the following dynamic properties:

- prod $\neq 0$ after execution of line 15; and
- return $= 0 \vee$ return $= 1$ after execution of line 19.

6. GENERATING VERIFIED PROPERTIES

Dynamic properties may overfit the observed behavior, and thus they may capture the characteristics of the executions rather than the actual behavior of the program. For instance, since in the running example we executed the *availableProducts* function only with parameters that have a non-null pointer to products, invariant detection generated the property product $\neq 0$ for line 2. In the case the *availableProducts* function could also be executed with null store_data→products, this property would capture a characteristic of the test suite rather than a general program property.

A key advantage of VART is that model checking can automatically prune most of the dynamic properties caused by overfitting, leading to a dramatic decrease in the number of false counterexamples reported to the developers. In practice VART produces the list of verified properties by checking each dynamic property against the base version from which the property was derived with a model checker. The verified properties will be exactly the properties that the model checker labels as *verified*.

To keep the regression checking process reasonably lightweight and practically applicable, the choice was made to limit the scope of model checking on the call trees rooted at the callers of the functions containing the changes. This implies that if the model checker labels a property as *verified*, the property can be verified by the model checker for any execution traversing any of the callers, independently of the context of the callers. The chosen strategy may eliminate properties that hold when they are considered in the entire program but are false when analyzed in a generalized context. This strategy is conservative since the properties that hold in the generalized context are a subset of the properties that hold for the entire program. In the unlikely case that starting the verification from the callers of the modified functions is too expensive, VART further restricts the scope of the checking to the modified functions only. This operation might further eliminate some properties that hold when considering the entire program, but it is conservative because the properties verified in the restricted scope also hold in larger scopes. For similar reasons model checking is not extended to library functions, which are conservatively assumed to change the parameters in an arbitrary way. Again this assumption may lead to the elimination of some valid dynamic properties, but it is conservative as it does not cause false properties to pass this phase.

Returning to the running example, we apply this process to the dynamic properties reported in Sec. 5. Model checking considers every caller of *isAvailable* as entry point of the analysis. In the running example, the only caller is *availableProducts*. Model checking automatically discovers that the dynamic property product $\neq 0$ does not hold and must be eliminated, while the rest of the properties are marked as *verified* and form the set of verified properties.

7. FILTERING VERIFIED PROPERTIES

When a program is upgraded, the verified properties derived from the base program can be used to check if the upgrade preserves the behavior of the base. Some of the properties derived for the base program might be no longer checkable when evaluated in the upgrade, for instance because they refer to variables that have been deleted or renamed. VART automatically identifies and drops these properties by injecting the verified properties into the upgrade, compiling the code, and removing the properties that cause compilation problems.

Once the illegal properties have been dropped, in principle, it is possible to use a model checker to check each of the verified properties against the upgraded program and report violations, and counterexamples, to developers. However, the upgrade may intentionally violate some of the properties that hold for the previous version of the software.

Consider for instance the verified property return $= 0 \vee$ return $= 1$ that holds for the return value of the *isAvailable* function in the running example. The upgrade shown in Fig. 3 intentionally violates this property because it introduces a new value that can be returned. Developers are not interested in collecting such false alarms, but instead want to be informed about any unexpected side effect introduced by the upgrade.

To capture the intention of the change and the expected impact on the verified properties VART exploits the regression test suite that covers the upgrade, including the new test cases implemented by the developers to specifically validate the change. Intuitively the test cases that cover the upgrade entail executions that have been knowingly modified by the developers. VART exploits this fact to eliminate the verified properties that held for the base version but became outdated in the upgrade. In particular, VART executes the regression test suite of the upgraded program and removes from the set of verified properties all the properties that are violated during the execution of the passing tests, that is, all the properties that have been intentionally invalidated by the upgrade.

If we consider the running example, by executing the test shown in Fig. 4 on the upgraded program VART automatically discovers that the property $\text{return} = 0 \vee \text{return} = 1$ does not hold anymore because the value returned by *isAvailable* in the test is -1. Thus the verified property $\text{return} = 0 \vee \text{return} = 1$ is outdated and must be eliminated.

The execution of the test in Fig. 4 also covers the verified property $\text{prod} \neq 0$, which holds after the execution of line 15 and is not broken by the regression test suite[1]. In this case the property is satisfied and it is not discarded.

The subset of verified properties that are not eliminated in this step are the *non-regression properties*, a set of properties that hold for the base version of the software and are not intentionally invalidated by the upgrade. The violations of the non-regression properties indicate the presence of side effects that deserve the attention of the developers.

8. UPGRADE CHECKING

In this final step, VART uses again the model checker, this time to verify the non-regression properties against the upgraded program. The detected violations consist of properties reported either as *false* or *unreachable*. Both false and unreachable properties may indicate the presence of regression faults: *false* properties reveal behaviors unintentionally modified by the upgrade, while *unreachable* properties reveal reachable statements sospiciously turned into unreachable ones by the upgrade.

The violations are reported to the developers together with the counterexamples represented as traces showing how non-regression properties can be actually violated. To facilitate debugging, the counterexamples can also be translated into test cases [8] and the non-regression properties can be instrumented in the code of the upgraded program as assert statements that fail when the tests are executed.

If we check the non-regression properties generated for our running example, VART automatically reveals that the following ones are violated:

1. $\text{avail} = 0 \vee \text{avail} = 1$ after line 5
2. $\text{avail} = 0 \vee \text{avail} = 1$ after line 6
3. $\text{total} \geq 0$ after line 6

This result clearly indicates that the upgraded *isAvailable* function can return a new value that has not been considered in the implementation of *availableProducts* (see the violation of the non-regression properties 1 and 2), and that this new value influences the computation of the total by allowing negative values (see the violation of the non-regression property 3). VART also generates the actual counterexample that demonstrates this case.

Fixing the fault in the running example is straightforward; the fix could simply consist of adding an **if** condition that prevents modifying the variable total when the return value of *isAvailable* is -1.

Although the illustrated case is simple, note that VART automatically identified a regression problem that normally requires a manually specified oracle to be detected.

9. EMPIRICAL EVALUATION

In this section we briefly describe our implementation of the VART approach and present the two empirical studies in support of our key arguments: VART can compensate the lack of thoroughness in regression test suites, and reveal subtle problems when thorough test suites are available. The first study is a controlled experiment that investigates the complementarity between regression testing and VART for a number of faults and program versions while changing the thoroughness of the test suite. In the second study we apply VART to a number of regression faults found in a range of different industrial and open-source applications from the numeric and string manipulation domains to demonstrate that VART can detect faults that have not been revealed with regression testing.

Prototype Tool. We implemented VART for programs written in C. The identification of the program locations that must be monitored, the collection of the runtime data, and the generation of the dynamic properties are implemented on top of the Radar tool [47], a dynamic analysis tool that can generate various behavioral models from runtime data. Radar uses diff to detect changes [2], GDB to record runtime information [3] and Daikon [25] to generate models.

We integrated both the CBMC [30] and the eVolCheck [26] bounded model checkers into VART. CBMC is a general purpose model checker, while eVolCheck includes features to optimize upgrade checking in specific situations. In the experiments we used CBMC when the optimizations implemented in eVolCheck were not effective. We ran model checking with a loop unrolling bound of 5 except for the GREP experiment in Table 3, where the bound was set to 1. Based on our experimentation, these bounds were a reasonable compromise between exactness and efficiency of the tool.

Our tool is freely available at `http://www.lta.disco.unimib.it/tools/vart`.

Controlled Experiment. In principle, the effectiveness of VART depends on the effectiveness of the regression tests that are executed to generate the dynamic properties. If the regression test suite is extremely poor, the dynamic properties generated from the tests will be poor as well. For instance, VART cannot generate properties that hold at program locations not covered by any test. In general, the better the test suite validates the software, the better the properties generated and checked with VART are. In this section we present a study on the complementarity between the thoroughness of the regression test suite and the effectiveness of VART, and we show that VART can compensate test suite inefficiencies even when the test suites do not cover well the change.

[1] when determining changes, VART compares the programs and automatically maps the properties of the base program on the corresponding locations in the upgraded program.

Table 1: The 11 cases of the controlled study.

Num	Fault ID	Base Ver	Upgrade Ver	Change Size
1	v3 DG10	10/1/1999 16:44:13	10/6/1999 1:13:21	483
2	v3 KP2			
3	v3 KP9			
4	v3 DG1	8/13/1999 14:45:21	8/13/1999 15:02:00	944
5	v3 DG8			
6	v3 KP7			
7	v3 DG3	10/7/1999 3:42:57	10/12/1999 4:11:40	185
8	v3 DG2			
9	v4 KP8	1/21/2000 02:22:47	1/25/2000 3:34:23	354
10	v4 DG3			
11	v4 KP6	1/17/2000 0:55:06	1/20/2000 4:43:03	89

We considered Grep [4] as subject program for this study because it is a non trivial program available with a large test suite of 817 test cases[2] that thoroughly covers the program. In this study we want to use such a test suite to extract a number of more realistic and smaller test suites providing different levels of coverage. As set of faults we considered all the Grep faults in the SIR repository [5] that are available for the last three versions of the Grep application and that are revealed by the complete regression test suite. Those amount to a total of 11 faults. *Note that none of these faults causes crashes or generate exceptions, thus they could not be easily revealed with automated techniques*, while VART can address them without requiring any human intervention thanks to the automatically generated non-regression properties.

Each SIR fault has been used to generate a regression fault according to the following steps. We first identified the most recent change in Grep that both is precedent to the version targeted by the SIR fault and affects the statement with the SIR fault. In the study, we used the version before the change as the base version and the version after the change as the upgrade. We then obtained the regression problem by injecting the SIR fault in the upgraded version. We made sure that this process produced a regression problem by checking that there exists at least one test that produced the same result when executed on the base and upgrade Grep versions, but produces a different result when the fault is also injected in the upgrade.

Table 1 reports the 11 regression faults used in this experiment. Column *Num* numbers the 11 regression faults from 1 to 11. Column *Fault ID* reports the SIR identifier of the fault used to generate the regression problem. Columns *Base Ver* and *Upgrade Ver* identify the upgrade used in the study by reporting the dates of two consecutive commits. Note that some faults share the same base version. For example the faults 1, 2, and 3 are injected on different statements modified by the same commit. Column *Change size* specifies the size of the considered change as the number of branches that occur in the modified functions.

We investigate the effectiveness of VART with respect to the effectiveness of regression testing for three relevant test suites: MRT, Cov20 and Cov50. *MRT* is the smallest subset of the original test suite that provides the same branch coverage of the change as the entire test suite. We ordered the test cases in MRT in a greedy fashion, so that the first test case is the one that contributes most to the coverage while the last test case is the one that contributes least to the coverage. Given the thoroughness of MRT, Cov20 and Cov50 represent the cases of smaller test suites that are more likely to be available in the practice. In particular *Cov20* and *Cov50* are the smallest test suites that can be obtained from `MRT` by selecting enough test cases to cover at least 20% and 50% of the branches in the change, respectively. The selection of the test cases from `MRT` follows the ordering of the tests. The number of tests in MRT varies for each case study, ranging from 4 to 47, with an average of 33.4 tests.

Table 2: VART compared to regression testing

Test Suite	Faults		RPV	IPV	Exec. Time (hours)
	Tests	VART			Min\Max\Avg
Cov20	3	5	5	0	6\60\22
Cov50	7	8	2	0	4\18\8.5
MRT	10	10	0	0	4\4\4

We used VART to analyze the 11 regression faults shown in Table 1 while exploiting the 3 test suites defined above (Cov20, Cov50, and MRT), for a total of 33 investigated cases. Table 2 shows the results grouped by test suite. Column *Test Suite* indicates the type of test suite. Column *Faults* reports data about the revealed faults. Columns *Tests* and *VART* indicate the number of faults revealed by the regression test suite and by Verification-Aided Regression Testing, respectively. Columns *RPV* and *IPV* indicate the total number of relevant and irrelevant property violations reported by the model checker. Column *Exec.Time* reports the min, max, and average number of hours needed to analyze each case with a Dell Poweredge Intel Xeon 3.73GHz.

We notice an interesting tradeoff between the number of faults revealed with testing and the ones revealed with VART. Even when the coverage of the change is small, VART has been useful. In fact, when Cov20 is used, regression testing reveals 3 faults, while VART reveals 2 additional faults, for a total of 5 faults revealed. When Cov50 is used, regression testing reveals 7 faults. VART improves the result of regression testing by revealing an additional fault, for a total of 8 faults revealed. Finally, when the minimal test suite with the highest coverage of the change is executed, both regression testing and VART reveal 10 faults. These results confirm the intuition that VART can automatically compensate the inefficiencies of regression test suites by revealing faults that would otherwise remain unrevealed.

In this specific experiment VART did not reveal additional faults when the change is thoroughly tested with *MRT*. However we show in the second experiment that VART not only compensates the inefficiencies of test suites as preliminarily demonstrated by this experiment, but can also reveal subtle problems not revealed by thorough test suites.

VART generated no false alarms. Even when not revealing additional faults, such as for the MRT test suite, VART reported no irrelevant property violations. This is a consequence of the conservativeness of the technique that filters any dynamic property that might generate a false alarm (in this experiment, VART discarded 97% of the dynamic properties in average).

Considering that VART can augment regression testing with automated verification capabilities which can reveal crashing and non-crashing faults with a negligible risk of producing irrelevant property violations, the additional faults

[2] the Grep test suite has been downloaded from the SIR repository: `http://sir.unl.edu/`

Table 3: VART Applied to Regression Problems

Subject		Versions		Test Suite	Dyn.	Non-Reg	RPV	IPV	Execution
App.	Size (LOCS)	Base	Upgrade	Size	Prop	Prop			Time (hours)
VTT	488	10.1	10.2	1000	1045	658	15	0	<1
ABB-1	200	-	-	1600	3278	163	57	1	<1
ABB-2	200	-	-	1600	3278	163	57	1	<1
ABB-3	200	-	-	1600	3278	163	0	1	<1
SORT	4653	20-06-12	02-07-12	427	356	2	1	0	1.5
GREP	590	01-17-01	01-20-01	817	3303	51	3	0	36

revealed in this experiment demonstrate that VART is not only conservative but also effective when relatively good regression test suites are available.

Results about the execution time suggest that the analysis can be feasibly executed overnight. In fact the average execution time varied between 4 and 22.5 hours, depending on the cases that are analyzed. In three cases, when the Cov20 test suite is used, the execution time reached 60 hours. This is due to the generation of a high number of false properties that are filtered out by VART. However, eliminating all these properties required a significant amount of time. When more test cases are used, the number of false properties, as well as the cost of running VART, decreases. Considering that our implementation is not optimized (e.g., properties are checked sequentially instead of being checked all together) and that a better server machine could be used for the analysis, the execution time could likely be narrow done to few hours also for the worst cases.

Detection of Regression Faults. This section presents a study that investigates the effectiveness of VART with several regression faults that have not been revealed by the corresponding regression test suites. The objective is to show that VART can reveal subtle faults overlooked by thorough test suites. To this end, we applied VART to 6 regression problems in industrial and open source systems.

From industry we consider two systems developed at VTT and at ABB. The system developed at VTT consists of a motion trajectory controller that is executed by a robotic arm during maintenance tasks in novel experimental nuclear reactors [55]. For the experiment, we selected a known regression problem that does not cause crashes but produces incorrect outputs when activated. This fault is hard to detect with conventional testing and analysis techniques. Since the test cases were not available for this system, we obtained the regression tests by randomly generating 1,000 test inputs for the base program (the inputs of the program consists of 12 numeric variables).

The system developed at ABB consists of the implementation of a protection function that is used in a control system to detect spikes in large-scale power distribution networks. The protection function is available with a suite of 1600 tests. Since it is a very stable function that did not show faults in the last 10 years, we generated the regression faults manually. We exploited this case to investigate if VART can effectively detect configuration problems. We thus obtained the 3 regressions by mutating the definition of the three constants that are used to represent the system state. The mutations have been injected in the most recent upgrade of a portion of the function that uses the mutated constants.

To provide an even more compelling overview of the applicability of the approach we also study VART using two popular open-source string manipulation systems; Sort, which is distributed as part of the GNU Coreutils [1], and Grep [4]. Sort is a utility that can write sorted concatenation of multiple files to the standard output, while Grep searches among multiple files the lines that match a given pattern and prints the matched lines to the standard output.

We selected the most recent regression fault available at the time we made the experiments for both Sort and Grep. In the case of Sort we selected the fault documented at `http://debbugs.gnu.org/cgi/bugreport.cgi?bug=11816#34`. A comment attached to the commit of the bug fix and our manual inspection confirm that this is a regression fault. In the case of Grep we obtained a regression fault by injecting the fault F_DG_1, defined in the SIR repository, in the latest upgrade of Grep. This is the only fault available in the SIR repository that affects the code in the latest upgrade. In the case of Sort, we used the test suite released with the program. In the case of Grep, we used the test suite available on the SIR repository.

In total, we investigated 6 regression faults in a number of diverse systems. Note that the investigated faults are regression faults not revealed by the available test suites.

Table 3 shows the results. Columns *App.* and *Size* indicate the application used in the evaluation and its size. In this empirical study we used non-trivial applications whose size range from 200 locs to about 5,000 locs. Note that applications of these sizes are usually challenging for model checkers. VART, thanks to its approach based on a restricted scope and a conservative analysis as illustrated in Section 6, suitably addressed these cases. Columns *Base* and *Upgrade* indicate the two program versions used in the evaluation. The version is identified by the commit date. The investigated regression problem is always introduced when upgrading the program from the base to the upgrade version. For confidentiality reasons we cannot show the version numbers for the cases based on the ABB system. Column *Test Suite Size* indicates the size of the regression test suite used in the evaluation. Columns *Dyn. Prop* and *Non-Reg Prop* indicate the number of dynamic properties initially generated by VART and the number of non-regression properties that have been actually used to validate the upgrade, respectively. Columns *RPV* and *IPV* indicate the number of the non-regression properties violated by the regression fault and the non-regression properties violated incidentally by the upgrade, respectively. Finally, column *Execution Time* indicates the number of hours necessary for the analysis.

We note that VART generated a significant number of properties (see Dyn. Prop. in Table 3) which efficiently reduce to a handful of verified non-regression properties used to check the upgrade (see Non-Reg Prop. in Table 3). The fact that VART is aggressive in eliminating properties that are not clearly useful to check the upgrade is an important characteristic of VART that is fundamentally conservative and aims at generating no IPVs at the cost of potentially eliminating properties that might be useful in some cases.

Regarding the fault detection ability, this study confirmed the effectiveness of VART, which has been able to automatically detect 5 of the 6 regression faults under investigation (corresponding to the cases with $RPV > 0$ in Table 3), producing at most one irrelevant property violation. In three cases, VART reported a small and useful set of violated properties (see RPV for VTT, Sort and Grep). In two cases, VART reported a significant number of (useful) violated properties (see RPV for ABB-1 and ABB-2). Even if VART reported 57 (useful) violated properties, developers could effectively analyze the output because the violations are well-focused: they refer to 2 program locations only and only a total of 6 variables occur among the 57 properties.

VART, even though it started with a large set of dynamically detected properties, ended up generating a false alarm only for one system. The false alarm actually reveals a changed behavior that was not well tested by the suite made available to us. Overall, small number of generated IPVs is an extremely important result because the generation of a big number of false alarms often represents a barrier to the adoption of a given analysis solution.

VART did not identify the regression problem in one of the cases. In that case, the injected fault impacted on the error handling routines of the program. Since the available regression test suite does not include tests with erroneous inputs, VART cannot generate any assertion related to error handling, and thus could not reveal the regression problem.

VART has been always able to analyze the target systems in a few hours, with the only exception of GREP that needed about 36 hours. This time could be likely narrowed done to few hours using a better implementation and a better server machine for the analysis. Despite the generation of many dynamic properties and the usage of potentially expensive bounded model checking to verify them, the restriction of the scope of the analysis and the abstraction of library calls implemented in VART made the analysis efficient.

When relevant property violations are reported, the identification of the fault is usually straightforward since developers can start debugging from counterexamples and a set of non-regression properties violated by the fault. We demonstrate the effectiveness of the output returned by VART by briefly presenting three of the analyzed faults.

In the VTT system, the analyzed regression fault is an incorrect initialization of the local variable vMax used to compute the speed of a joint. The incorrect initial value causes the speed of the joints, stored in the array jp, to exceed the maximum allowed speed, stored in the array jl. VART detected this fault by identifying that the non-regression properties vMax = jl[0] and jp[0].v $\leq$ jl[0] do not hold in the upgrade. The former property captures the wrong initialization of vMax. The latter property captures the actual speed exceeding the maximum allowed speed. The rest of the violated non-regression properties are related to the algorithmic steps executed in the function with the fault.

In the Sort program, the regression fault is a wrong code that *stream_open* uses as a parameter when invoking function *error*. VART nicely captures this problem by identifying that the upgrade violates the property status = 0 $\vee$ status = 2 defined in the body of function *error*, where status is the name of the first parameter. The counterexample shows that this problem occurs when *error* is invoked from *stream_open*.

In the Grep program, the function *savedir* is expected to set name_size to 1 when *savedir* is invoked with name_size = 0. The upgrade does not execute this operation. VART suitably captures this problem by identifying that the non-regression property name_size $\neq$ 0 is violated exactly after the point where the variable should be set to 1.

The reported cases provide a good evidence that VART not only reveals problems unnoticed by the tests but also provides information about the reason of the failure, usually referring to locations close to the fault that caused the failure, thus making debugging straightforward in several cases. Note that even if the investigated faults did not always cause crashes, VART revealed them thanks to the soundness of the automatically generated non-regression properties.

Threats to Validity. In this paper we provide initial evidence of the effectiveness of VART. Although the results cannot be generalized yet there are various aspects suggesting they could be valid in different scenarios. VART reported either none or a single irrelevant property violation, and has been able to reveal faults unnoticed by regression test suites for systems with different characteristics. We considered both industrial and open source systems, and investigated both programs mostly implementing numerical computations and programs working on strings. These results provide a good, although preliminary, evidence that VART can be a practical solution that should be considered to augment regression testing.

The reported results depend on the regression test suites used in the evaluation making the nature and the choice of the test suite a threat to the validity of the experiment. In our evaluation we considered multiple test suites: thorough test suites (e.g., the SIR test suite for Grep), real test suites (e.g., the test suite for the Sort program), and random test suites (e.g., the test suite for the VTT program). Since VART revealed faults regardless the kind of test suite used this threat does not appear to be severe.

Our empirical evaluation measures relevant and irrelevant property violations, but classifying a report as relevant or irrelevant might be subjective. In our experiments, VART reported a small number of property violations close and clearly correlated to the faults, as exemplified for some of the cases. Thus, at least for the cases reported in the paper, we do not see any risk of a misclassification.

10. RELATED WORK

VART is related to work in regression testing, anomaly detection, model checking, and change analysis. This section discusses VART with respect to results in these areas.

Regression testing concerns with a multitude of aspects that are relevant when testing upgrades. For instance, it addresses problems like the selection of the test cases that must be re-executed to validate a change [15, 51], the prioritization of those tests [24, 59, 38], and the maintenance of a test suite across program versions [22, 42].

The effectiveness of any regression testing process strongly depends on the set of test cases implemented by the testers. Those tests can only sample a limited number of cases and several faults could be missed by regression testing techniques. Adequacy criteria, such as structural-, specification- or fault-based criteria [65] may mitigate issues related to incompleteness of test suites but do not solve the problem.

VART is a technique that augments the effectiveness of the regression testing process thanks to its ability of automatically discovering non-regression properties and detecting violations of those properties across program versions. In our empirical results VART automatically discovered regression faults not revealed by the regression test suites in 2 out of 3 configurations studied for the Grep program and in 5 out of 6 regression faults studied in 4 different applications.

The idea of learning behavioral models and properties, either statically or dynamically, and then revealing anomalous behaviors by checking the learnt models and properties, either statically or dynamically, has been already investigated in *anomaly detection* techniques.

The rationale exploited in anomaly detection is that failures are not frequent and thus the behaviors that violate the common patterns might point at incorrect executions. Techniques investigated a number of combinations. For instance, Wasylkowski and Zeller statically derived and checked computational tree logic formulas that capture method preconditions [60]. BCT dynamically derives and checked finite state models and Boolean properties that capture components' behaviors [39]. Jadet statically identifies patterns of method invocations and discovers anomalous sequence of method calls [61]. Radar dynamically generates and checks program properties and control-flow models that capture the behavior of software units, such as program functions [48].

These techniques can identify a number of faults, but they also produce several false alarms that hinder their applicability. This is mainly due to the *oracle problem*, i.e., these techniques cannot distinguish between failing and passing runs. To overcome the oracle problem and eliminate the false alarms, some techniques report only the problems that can be confirmed by generating a test [50, 21]. This solution limits the faults that can be revealed to problems that produce crashes and uncaught exceptions. On the contrary, VART suitably integrates static and dynamic analysis to produce properties that are provably true and uses heuristics to remove outdated properties when analyzing upgrades. Those mechanisms almost eliminate irrelevant property violations. Moreover, even if limited to regression faults, VART can automatically identify problems beyond crashes and exceptions, that typically require an oracle to be detected.

The idea of confirming automatically generated properties has been explored in other settings, such as using static analysis to confirm dynamic properties [44], using the crowd to confirm assert statements [46], and jointly using mutants and testing to confirm dynamic properties [28]. The approach closer to VART, regarding property generation, is the one by Nimmer and Ernst who used static analysis to confirm properties generated for small-sized programs [45, 44]. They investigated this integration to automatically suggest developers the annotations that can be added into programs. VART exploits a similar idea on larger programs to generate the verified properties, which are used to obtain the non-regression properties that are checked on the upgrades.

Model checkers are widely used to statically verify safety properties of software programs [18, 17, 10, 9, 53, 34]. Those properties are represented as expressions over program variables in the control flow of the program. Such properties typically require knowledge of the program functionality and must be written by the developer, although some tools, such as those based on CProver [17, 53], support automatic generation of simple properties to be checked: pointer dereferencing, division by zero, array bounds checks and a few more. The tool Houdini [27] offers various static heuristics to guess program invariants to be checked using a model checker. These invariants are based on a random choice and therefore not necessarily useful. The tool Spacer [34] generates safe inductive invariants from the proof of program correctness. Since this tool performs analysis with respect to already specified properties, the newly generated invariants will also depend on such properties. The approach taken in VART is novel in the sense that the properties are generalized from executions and employed in the context of regression testing.

Previous works studied how to use model checking to verify whether properties are preserved by software upgrades. Using the efforts spent on checking the previous version, eVolCheck [26] tends to localize the verification only to the modified parts of the program. Assertions have been used to compare the behavior of programs also in other contexts, such as discovering interleaving bugs [33], and checking relative correctness specifications [35]. Unlike our approach, these approaches lack the ability to automatically capture the intention of the change.

Program slicing [56] and impact analysis [7] can be used to assess the *impact of a change*. Regression faults that invalidate program assertions can be addressed with program slicing combined with symbolic executions [63]. Other sophisticated techniques, such as DSE [49] and BERT [32], can be used to determine the changed behaviors between two program versions. In particular DSE identifies behavioral differences at the level of function summaries, while BERT identifies differences on the outputs that can be produced by two program versions. These techniques can detect behavioral differences but cannot distinguish between intended changes and regression problems, like VART does.

11. CONCLUSION

Regression testing is a well-established solution for the validation of changes and upgrades. The effectiveness of regression testing is strongly dependent on the completeness of the test suite that is executed to validate the changes. Complete regression test suites are hard to design, and ineffective regression test suites cause several erroneous upgrades to be discovered late, when fixing faults is expensive.

This paper presents *Verification-Aided Regression Testing* (VART), an extension of regression testing that in addition to exploiting the tests to reveal faults uses behavioral properties, derived from test executions, to automatically discover additional faults not revealed by the regression tests.

VART originally combines testing, invariant detection, and model checking to obtain the unique capability of (1) automatically producing properties proved to hold for the base version of a program, (2) automatically identifying and checking on the upgraded program only the properties that, according to the developers' intention, must be preserved by the upgrade, and (3) reporting faults that are not revealed by tests and corresponding counter-examples that show how to activate the faults. Note that this process can automatically reveal faults that require manually specified assertions to be detected with testing and model-checking.

Results indicate that VART can both compensate the inefficiencies of regression test suites and augment the effectiveness of thorough test suites revealing subtle faults overlooked by the tests.

12. REFERENCES

[1] Coreutils. http://www.gnu.org/software/coreutils/.

[2] Diff utils. http://www.gnu.org/software/diffutils/.

[3] GDB. http://sources.redhat.com/gdb/.

[4] Grep. http://www.gnu.org/software/grep/.

[5] Software-artifact infrastructure repository. http://sir.unl.edu.

[6] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy abstraction with interpolants for arrays. In *proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 7180 of *LNCS*. Springer, 2012.

[7] R. Arnold and S. Bohner. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Press, 1996.

[8] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *proceedings of the International Conference on Software Engineering*, 2004.

[9] D. Beyer, T. A. Henzinger., R. Jhala, and R. Majumdar. The software model checker Blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 9:505–525, 2007.

[10] D. Beyer and M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *proceedings of the International Conference on Computer Aided Verification*, LNCS, pages 184–190, 2011.

[11] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking Using SAT Procedures instead of BDDs. In *proceedings of the annual Design Automation Conference*, 1999.

[12] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *proceedings of the International Conference Tools and Algorithms for Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[13] A. R. Bradley. SAT-based model checking without unrolling. In *proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretatio*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[14] S. Chaki, E. M. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, 25(2–3):129–166, 2004.

[15] Y. Chen, D. Rosemblum, and K. Vo. TestTube: A system for selective regression testing. In *proceedings of the International Conference on Software Engineering*, 1994.

[16] E. Clarke and A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, 1981.

[17] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *proceedings of the International Conference Tools and Algorithms for Construction and Analysis of Systems.*

[18] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based Predicate Abstraction for ANSI-C. In *proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, 2005.

[19] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[20] J. Cobb, J. Jones., G. Kapfhammer, and M. J. Harrold. Dynamic invariant detection for relational databases. In *proceedings of the Ninth International Workshop on Dynamic Analysis*, 2011.

[21] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodologies*, 17(2):8:1–8:37, May 2008.

[22] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. On test repair using symbolic execution. In *proceedings of the International Conference on Automated Software Engineering*, 2009.

[23] N. Eén and N. Sörensson. An extensible SAT-solver. In *proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[24] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, Feb. 2002.

[25] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[26] G. Fedyukovich, O. Sery, and N. Sharygina. eVolCheck: Incremental Upgrade Checker for C. In *proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *LNCS*, 2013.

[27] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *proceedings of the International Symposium of Formal Methods Europe*, 2001.

[28] M. Gabel and Z. Su. Testing mined specifications. In *proceedings of the International Symposium on the Foundations of Software Engineering*, 2012.

[29] B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification & Reliability*, 23(3):241–258, 2013.

[30] S. V. Group. The cbmc homepage. http://www.cprover.org/cbmc/.

[31] R. H. Hardin, R. P. Kurshan, K. L. McMillan, J. A. Reeds, and N. J. A. Sloane. Efficient regression verification. In *proceedings on the International Workshop on Event Systems*, 1996.

[32] W. Jin, A. Orso, and T. Xie. Automated behavioral regression testing. In *proceedings of the Third International Conference on Software Testing, Verification and Validation (ICST)*, 2010.

[33] S. Joshi, S. Lahiri, and A. Lal. Underspecified harnesses and interleaved bugs. In *proceedings of the annual Symposium on Principles of Programming Languages*, 2012.

[34] A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke. Automatic abstraction in SMT-based unbounded software model checking. In *proceedings of the International Conference on Computer Aided Verification*, 2013.

[35] S. K. Lahiri, K. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Foundations of Software Engineering*, 2013.

[36] H. Leung and L. White. Insights into regression testing. In *proceedings of the International Conference on Software Maintenance*, 1989.

[37] Y. S. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An efficient SAT solver. In *proceedings of the International Conference on Theory and Applications of Satisfiability Testing, Revised Selected Papers*, volume 3542 of *LNCS*, pages 360–375. Springer, 2005.

[38] L. Mariani, S. Papagiannakis, and M. Pezzé. Compatibility and regression testing of COTS-component-based software. In *proceedings of the International Conference on Software Engineering*, 2007.

[39] L. Mariani, F. Pastore, and M. Pezzé. Dynamic analysis for diagnosing integration faults. *IEEE Transactions on Software Engineering*, 37(4):486–508, 2011.

[40] K. McMillan. *Symbolic Model Checking. An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992.

[41] K. L. McMillan. Lazy abstraction with interpolants. In *proceedings of the International Conference on Computer Aided Verification*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[42] M. Mirzaaghaei, F. Pastore, and M. Pezzé. Supporting test suite evolution through test case adaptation. In *proceedings of the International Conference on Software Testing, Verification, and Validation*, 2012.

[43] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *proceedings of the International Conference on Software Engineering*, 2012.

[44] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. *Electronic Notes on Theoretical Computer Science*, 55(2):255–276, 2001.

[45] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *proceedings of the International Symposium on Software Testing and Analysis*, 2002.

[46] F. Pastore, L. Mariani, and G. Fraser. Crowdoracles: Can the crowd solve the oracle problem. In *proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2013.

[47] F. Pastore, L. Mariani, and A. Goffi. RADAR: a tool for debugging regression problems in C/C++ software. In *proceedings of the International Conference on Software Engineering - Tool Demo Track*, 2013.

[48] F. Pastore, L. Mariani, A. Goffi, M. Oriol, and M. Wahler. Dynamic analysis of upgrades in C/C++ software. In *proceedings of the International Symposium on Software Reliability Engineering*, 2012.

[49] S. Person, M. Dwyer, S. Elbaum, and C. Păsăreanu. Differential symbolic execution. In *proceedings of the International Symposium on Foundations of Software Engineering*, 2008.

[50] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking api protocol conformance with mined multi-object specifications. In *proceedings of the International Conference on Software Engineering*, 2012.

[51] G. Rothermel and M. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodologies*, 6(2):173–210, Apr. 1997.

[52] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *proceedings of the International Symposium on Software Testing and Analysis*, 2009.

[53] O. Sery, G. Fedyukovich, and N. Sharygina. FunFrog: Bounded model checking with interpolation-based function summarization. In *proceedings of the International Symposium on Automated Technology for Verification and Analysis*, volume 7561 of *LNCS*, pages 203–207. Springer, 2012.

[54] O. Sery, G. Fedyukovich, and N. Sharygina. Incremental upgrade checking by means of interpolation-based function summaries. In *proceedings of the International Conference on Formal Methods in Computer-Aided Design*. IEEE, 2012.

[55] Y. Shimomura. The present status and future prospects of the iter project. *Journal of Nuclear Materials*, 329-333(1):5–11, 2004.

[56] J. Silva. A vocabulary of program slicing-based techniques. *ACM Computing Survey*, 44(3):12:1–12:41, 2012.

[57] J. P. M. Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[58] A. L. Souter and L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE Transactions on Software Engineering*, 29(11):1005–1018, Nov. 2003.

[59] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Timeaware test suite pioritization. In *proceedings of the International Symposium on Software Testing and Analysis*, 2006.

[60] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *proceedings of the International Conference on Automated Software Engineering*, 2009.

[61] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *proceedings of the joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*, 2007.

[62] T. Xie and D. Notkin. Tool-assisted unit test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 13(3):345–371, July 2006.

[63] G. Yang, S. Khurshid, S. Person, and N. Rungta. Property differencing for incremental checking. In *proceedings of the International Conference on Software Engineering*, 2014.

[64] J. Yi, D. Qi, S. Tan, and A. Roychoudhury. Expressing and checking intended changes via software change contracts. In *proceedings of the International Symposium on Software Testing and Analysis*, 2013.

[65] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.