Collaborative Verification and Testing with Explicit Assumptions

Maria Christakis, Peter Müller, and Valentin Wüstholz

ETH Zurich, Switzerland
{maria.christakis,peter.mueller,valentin.wuestholz}@inf.ethz.ch

Abstract. Many mainstream static code checkers make a number of compromises to improve automation, performance, and accuracy. These compromises include not checking certain program properties as well as making implicit, unsound assumptions. Consequently, the results of such static checkers do not provide definite guarantees about program correctness, which makes it unclear which properties remain to be tested. We propose a technique for collaborative verification and testing that makes compromises of static checkers explicit such that they can be compensated for by complementary checkers or testing. Our experiments suggest that our technique finds more errors and proves more properties than static checking alone, testing alone, and combinations that do not explicitly document the compromises made by static checkers. Our technique is also useful to obtain small test suites for partially-verified programs.

1 Introduction

Static program checkers are increasingly applied to detect defects in real-world programs. There is a wide variety of such checkers, ranging from relatively simple heuristic tools, over static program analyzers and software model checkers, to verifiers based on automatic theorem proving.

Although effective in detecting software bugs, many practical static checkers make compromises in order to increase automation, improve performance, and reduce both the number of false positives and the annotation overhead. These compromises include not checking certain properties and making implicit, unsound assumptions. For example, HAVOC [1] uses write effect specifications without checking them, Spec# [3] ignores arithmetic overflow and does not consider exceptional control flow, ESC/Java [16] unrolls loops a fixed number of times, and the Code Contracts static checker, Clousot [13], assumes that the arguments to a method call refer to disjoint memory regions, to name a few.

Due to these compromises, static checkers do not provide definite guarantees about the correctness of a program—as soon as a static checker makes a compromise, errors may be missed. This has three detrimental consequences: (1) Static checkers that make such compromises cannot ensure the absence of errors. (2) Even though one would expect static checking to reduce the test effort, it is unclear how to test exactly those properties that have not been soundly verified. In practice, programmers need to test their programs as if no static checking had been applied, which is inefficient. (3) Static checkers cannot be easily integrated to complement each other.

In this paper, we propose a technique that enables the combination of multiple, complementary static checkers and the reinforcement of static checking by automated, specification-based test case generation to check the program executions and properties that have not been soundly verified. Our technique handles sequential programs, properties that can be expressed by contract languages [12,20], and the typical compromises made by abstract interpreters and deductive verifiers. An extension to concurrent programs, more advanced properties (such as temporal properties), and the compromises made by model checkers (such as bounding the number of heap objects) are future work.

Our work is closely related to conditional model checking [6], which is an independently developed line of work. Both approaches make the results of static checking precise by tracking which properties have been verified, and under which assumptions. By documenting all compromises, a static checker becomes sound relatively to its compromises. However, accidental unsoundness, for instance due to bugs in the implementation of the checker, is neither handled nor prevented. Moreover, both approaches promote the collaboration of complementary static checkers and direct the static checking to the properties that have not been soundly verified. A detailed comparison of the two approaches is provided in Sect. 5. The three contributions made by our paper are:

1. It proposes a simple language extension for making many deliberate compromises of static checkers explicit and marking every program assertion as either fully verified, partially verified (that is, verified under certain assumptions), or not verified. This information is expressed via two new constructs whose semantics is defined in terms of assignments and assertions. They are, thus, easy to support by a wide range of static checkers. All assumptions are expressed at the program points where they are made. Therefore, *modular* static checkers may encode their verification results *locally* in the checked module (for instance, locally within a method). This is crucial to allow subsequent checkers to also operate modularly. Moreover, *local* assumptions and verification results are suitable to automatically generate *unit tests* for the module. We demonstrate the effectiveness of our language extension in encoding typical compromises of mainstream static checkers.
2. It presents a technique to automatically generate unit tests from the results of static checkers providing the user with a choice on how much effort to devote to static checking and how much to testing. For example, a user might run an automatic verifier without devoting any effort to making the verification succeed (for instance, without providing auxiliary specifications, such as loop invariants). The verifier may prove some properties correct, and our technique enables the effective testing of all others. Alternatively, a user might try to verify properties about critical components of a program and leave any remaining properties (e.g., about library components) for testing. Consequently, the degree of static checking is configurable and may range from zero to complete.

3. It enables a tool chain that directs the static checking and test case generation to the partially-verified or unverified properties. This leads to more targeted static checking and testing, in particular, smaller and more effective test suites. We implemented our tool chain based on an adaptation of the Dafny verifier [21] and the concolic testing tool Pex [24]. Our experiments suggest that our technique finds more errors and proves more properties than static checking alone, testing alone, and combined static checking and testing without our technique.

Outline. Sect. 2 gives a guided tour to our approach through an example. Sect. 3 explains how we encode the results and compromises of static checkers. We demonstrate the application of our technique to some typical verification scenarios in Sect. 4. We review related work in Sect. 5 and conclude in Sect. 6.

2 Guided Tour

This section gives a guided tour to collaborative verification and testing with explicit assumptions. Through a running example, we discuss the motivation behind the approach and the stages of the tool chain.

Running Example. Let us consider the C# program of Fig. 1 with .NET Code Contracts [12]. Method `foo` takes two `Cell` objects with non-zero values. The intention of the `if` statement is to guarantee that the two values have different signs and therefore, ensure that their product is negative. However, this program violates its postcondition in two cases: (1) The multiplications in the `if` and `return` statements (lines 16 and 20, respectively) might overflow and produce a positive result even if the integers have different signs. (2) In case parameters `c` and `d` reference the same object, the assignment on line 16 changes the sign of both `c.value` and `d.value` and the result is positive.

Checking this program with the Code Contracts static checker, Clousot, detects none of the errors because it ignores arithmetic overflow and uses a heap abstraction that assumes that method arguments are not aliased. A user who is not familiar with the tool's implicit assumptions does not know how to interpret the absence of warnings. Given that errors might be missed, the code has to be tested as if the checker had not run at all.

Running the Code Contracts testing tool, Pex, on method `foo` generates a test case that reveals the aliasing error, but misses the overflow error. Since no branch in the method's control flow depends on whether the multiplications in the `if` and `return` statements overflow, the tool does not generate a test case that exhibits this behavior. So, similarly to the static checker, the absence of errors provides no definite guarantee about program correctness.

Our technique enables collaborative verification and testing by making explicit which properties have been verified, and under which assumptions. The tool chain that we propose is presented in Fig. 2 and consists of two stages: static verification and testing.

```
public class Cell
{
  public int value;

  public static int foo(Cell c, Cell d)
  {
    Contract.Requires(c != null && d != null);
    Contract.Requires(c.value != 0 && d.value != 0);
    Contract.Ensures(Contract.Result<int>() < 0); // verified under a_na, a_ui0, a_ui1

    // assumed c != d as a_na
    if ((0 < c.value && 0 < d.value) || (c.value < 0 && d.value < 0))
    {
      // assumed new BigInteger(-1) * new BigInteger(c.value) ==
      //          new BigInteger(-1 * c.value) as a_ui0
      c.value = (-1) * c.value;
    }
    // assumed new BigInteger(c.value) * new BigInteger(d.value) ==
    //          new BigInteger(c.value * d.value) as a_ui1
    return c.value * d.value;
  }
}
```

Fig. 1. Example program that illustrates the motivation for our technique. The method postcondition is violated if one of the multiplications overflows or if parameters `c` and `d` reference the same object. The comments document the compromises made by a checker that ignores arithmetic overflow and assumes that parameters are not aliased.

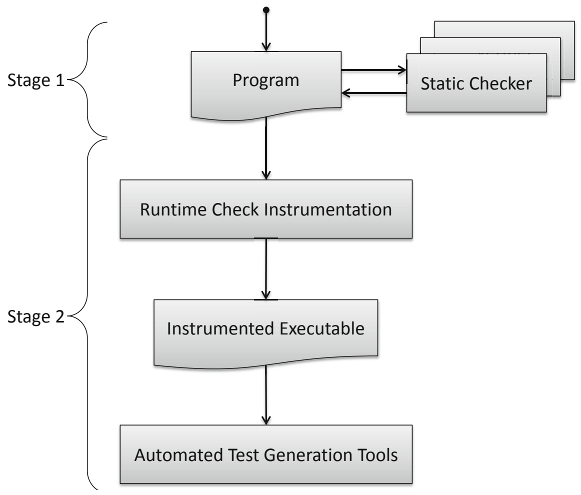
Stage 1
Program
Static Checker
Stage 2
Runtime Check Instrumentation
Instrumented Executable
Automated Test Generation Tools

Fig. 2. The collaborative verification and testing tool chain. Tools are depicted by boxes and programs with specifications by flowchart document symbols.

Stage 1: Collaborative Verification. The static checking (or verification) stage allows the user to run an arbitrary number (possibly zero) of static checkers. Each checker reads the program, which contains the code, the specification, and the results of prior static checking attempts. More precisely, each assertion is marked to be either fully (that is, soundly) verified, partially verified under certain explicit assumptions, or not verified (that is, not attempted or failed to verify). A checker then attempts to prove the assertions that have not been fully verified by upstream tools. For this purpose, it may assume the properties that have already been fully verified. For partially-verified assertions, it is sufficient to show that the assumptions made by a prior checker hold or the assertions hold regardless of the assumptions, which simplifies the verification task. For instance, if the first checker verifies that all assertions hold assuming no arithmetic overflow occurs, then it is sufficient for a second (possibly specialized) checker to confirm this assumption. Each checker records its results in the program that serves as input to the next downstream tool.

The intermediate versions of the program precisely track which properties have been fully verified and which still need validation. This allows developers to stop the static verification cycle at any time, which is important in practice, where the effort that a developer can devote to static checking is limited. Any remaining unverified or partially-verified assertions may then be covered by the subsequent testing stage.

The comments in Fig. 1 illustrate the result of running the Code Contracts static checker on the example. The checker makes implicit assumptions in three places, for the non-aliasing of method arguments (line 11) and the unbounded integer arithmetic (lines 14 and 18). Since some assumptions depend on the current execution state, we document them at the place where they occur rather than where they are used to prove an assertion. We give each assumption a unique identifier (such as a_{na} for the assumption on line 11), which is used to document where this assumption is used. Running the checker verifies the method postcondition under these three assumptions, which we reflect by marking the postcondition as partially verified under assumptions a_{na}, a_{ui0}, and a_{ui1} (line 9). We will show how to formally encode assumptions and verification results in Sect. 3.

Note that the Code Contracts static checker works modularly, that is, it checks each method independently of its clients. Therefore, all assumptions are local to the method being checked; for instance, method `foo` is analyzed independently of any assumptions in its callers. Consequently, the method's verification results are suitable for subsequent modular checkers or unit test generation tools.

Since our example actually contains errors, any subsequent checker will neither be able to fully verify that the assumptions always hold nor that the postcondition holds in case the assumptions do not. Nevertheless, the assumptions document the precise result of the static checker, and we use this information to generate targeted test cases in the subsequent testing stage.

Stage 2: Testing. We apply dynamic symbolic execution [18,23], also called concolic testing [23], to automatically generate parameterized unit tests from the program code, the specification, and the results of static checking.

Concolic testing collects constraints describing the test data that will cause the program to take a particular branch in the execution or violate an assertion[1].

[1] An assertion is viewed as a conditional statement, where one branch throws an exception. A test case generation tool aiming for branch coverage will therefore attempt to generate test data that violates the assertion.

To use this mechanism, we instrument the program with assertions for those properties that have not been fully verified. That is, we assert all properties that have not been verified at all, and for partially-verified properties, we assert that the property holds in case the assumptions made by the static checker do not hold. This way, the properties that remain to be checked as well as the assumptions made by static checkers occur in the instrumented program, which causes the symbolic execution to generate the constraints and test data that exercise these properties.

In our example, the postcondition has been partially verified under three assumptions. The instrumentation therefore introduces an assertion for the property that all three assumptions hold or the original postcondition holds: $a_{na} \wedge a_{ui0} \wedge a_{ui1} \vee$ `c.value * d.value < 0`. Here, we use the assumption identifiers like boolean variables, which are assigned to when the assumption is made (see Sect. 3 for details), and we substitute the call to `Contract.Result` by the expression that method `foo` returns.

Running Pex on the instrumented version of the partially-verified program generates unit tests that reveal both errors, whereas without the instrumentation Pex finds only the aliasing error. A failing unit test is now also generated for the overflow error because the explicit assumptions on lines 14 and 18 of the program create additional branches in the method's control flow graph, thus enriching the constraints that are collected and solved by the testing tool.

In case the code must be fully verified, an alternative second stage of the tool chain could involve proving the remaining, precisely documented program properties with an interactive theorem prover. The intention then is to prove as many properties as possible automatically and to direct the manual effort towards proving the remaining properties. Yet another alternative is to use the explicit assumptions and partial verification results for targeted code reviews.

3 Verification Results with Explicit Assumptions

To make assumptions and (partially) verified properties explicit in the output of static checkers, we extend the programming and specification language with two new constructs: `assumed` statements and `verified` attributes for assertions. In this section, we present these extensions and define their semantics in terms of their weakest preconditions.

Extensions. An `assumed` statement of the form `assumed` P `as` a records that a checker assumed property P at a given point in the code. P is a predicate of the assertion language, and a is a unique *assumption identifier*, which can be used in `verified` attributes to express that a property has been verified using this assumption. `assumed` statements do not affect the semantics of the program, but they are used to define the semantics of `verified` attributes, as we discuss below. In particular, our `assumed` statements are different from the classical `assume` statements, which express properties that any static checker or testing tool may take for granted and need not check.

In our example of Fig. 1, the assumptions on lines 11, 14, and 18 will be formalized by adding `assumed` statements with the respective predicates. We also allow programmers, in addition to static checkers, to add `assumed` statements in their code, which is for instance useful when they want to verify the code only for certain cases and leave the other cases for testing.

In order to record (partial) verification results, we use assertions of the form `assert` V P, where P is a predicate of the assertion language and V is a set of `verified` attributes. A `verified` attribute has the form `{:verified` A`}`, where A is a set of assumption identifiers, each of which is declared in an `assumed` statement.

When a static checker verifies an assertion, it adds a `verified` attribute to the assertion that lists the assumptions used for its verification. Consequently, an assertion is unverified if it has no `verified` attributes, that is, V is empty (no static checker has verified the assertion). The assertion is fully verified if it has at least one `verified` attribute that has an empty assumption set A (at least one static checker has verified the assertion without making any assumptions). Otherwise, the assertion is partially verified.

Note that it is up to each individual verifier to determine which assumptions it used to verify an assertion. For instance, a verifier based on weakest preconditions could collect all assumptions that are on any path from the start of a method to the assertion; it could try to minimize the set of assumptions using techniques such as slicing to determine which assumptions actually influence the truth of the assertion. In our example, the assertion for the postcondition will be decorated with the attribute `{:verified` $\{a_{na}, a_{ui0}, a_{ui1}\}$`}` to indicate that the static checker used all three assumptions to verify the postcondition.

Semantics. The goal of collaborative verification and testing is to let static checkers and test case generation tools benefit from the (partial) verification results of earlier static checking attempts. This is achieved by defining a semantics for assertions that takes into account what has already been verified. For a fully-verified assertion, a static checker or test case generation tool later in the tool chain does not have to show anything. For partially-verified assertions, it is sufficient if a later tool shows that the assertion holds in case the assumptions made by earlier static checking attempts do not hold. We formalize this intuition as a weakest-precondition semantics.

In the semantics, we introduce a boolean *assumption variable* for each assumption identifier that occurs in an `assumed` statement; all assumption variables are initialized to true. For modular static checking, which checks each method individually, assumption variables are local variables of the method that contains the `assumed` statement. Assumptions of whole-program checking may be encoded via global variables. An `assumed` statement replaces the occurrence of an assumption variable by the assumed property:

$$wp(\texttt{assumed}\ P\ \texttt{as}\ a, R) \equiv R[a := P]$$

where $R[a := P]$ denotes the substitution of a by P in R. This semantics ensures that an assumption is evaluated in the state in which it is made rather than the state in which it is used. Since each assumption variable is initialized to true, every occurrence of an assumption variable in a weakest-precondition computation will eventually be substituted either by the assumed property or by true in those execution paths that do not include an `assumed` statement for that assumption variable.

We define the semantics of assertions as follows:

$$wp(\texttt{assert V } P, R) \equiv \left(\left(\bigvee_{\mathrm{A}\in\mathrm{V}} CA(\mathrm{A})\right) \vee P\right) \wedge (P \Rightarrow R)$$

where $CA(\mathrm{A})$ denotes the conjunction of all assumptions in one `verified` attribute (all assumptions of one static checker). That is, $CA(\mathrm{A}) \equiv \bigwedge_{a\in\mathrm{A}} var(a)$, where $var(a)$ is the assumption variable for the assumption identifier a.

The first conjunct in the weakest precondition expresses that in order to fully verify the assertion, it is sufficient to show that all assumptions made by one of the checkers actually hold or that the asserted property P holds anyway. The disjunction weakens the assertions and therefore, lets tools benefit from the partial results of prior static checks. Note that in the special case that one of the `verified` attributes has an empty assumption set A, $CA(\mathrm{A})$ is true, and the first conjunct of the weakest precondition trivially holds (that is, the assertion has been fully verified and nothing remains to be checked). Since the first conjunct of the weakest precondition ensures that assertion P is verified, the second conjunct requires only that postcondition R is verified under the assumption that P holds.

In our example, the weakest precondition of the partially-verified postcondition is $a_{na} \wedge a_{ui0} \wedge a_{ui1} \vee$ `c.value * d.value < 0`. As we explained in Sect. 2, we use this condition to instrument the program for the test case generation.

When multiple static checkers (partially) verify an assertion, we record each of their results in a separate `verified` attribute. However, these attributes are not a mere accumulation of the results of independent static checking attempts. Due to the above semantics, the property to be verified typically becomes weaker with each checking attempt. Therefore, many properties can eventually be fully verified, without making any further assumptions. The remaining ones can be tested or verified interactively.

4 Examples

For the evaluation of our tool chain and the underlying technique, we used the Dafny language and verifier, and the testing tool Pex. Dafny is an imperative, class-based programming language with built-in specification constructs to support sound static verification. For our purposes, we extended the Dafny language with the `assumed` statements and `verified` attributes of Sect. 3, and changed the Dafny verifier to simulate common compromises made by mainstream static checkers. For the instrumentation phase of the architecture, we extended the existing Dafny-to-C# compiler to generate runtime checks, expressed as Code Contracts, for program properties that have not been fully verified.

In this section, we demonstrate how common compromises may be encoded with our language extensions and subsequently tested. We apply our technique to three verification scenarios and show that it finds more errors than the architecture's constituent tools alone and achieves small, targeted test suites.

4.1 Encoding of Common Compromises

To simulate common compromises, we implemented three variants of the Dafny verifier: (1) ignoring arithmetic overflow, like e.g. Spec#, (2) unrolling loops a fixed number of times, like e.g. ESC/Java, and (3) using write effect specifications without checking them, like e.g. HAVOC.

Unbounded Integers. A common compromise of static checkers is to ignore overflow in bounded integer arithmetic, as in the case of ESC/Java and Spec#. To model this behavior in Dafny, which uses unbounded integers, we adapted the verifier to add explicit assumptions about unbounded integer arithmetic and modified the compiler to use bounded (32-bit) integers.

We use `BigInteger` to express that a static checker that ignores arithmetic overflow considers bounded integer expressions in the code to be equivalent to their mathematical counterparts. For instance, the assumption that the expression `c.value * d.value` from Fig. 1 does not lead to an overflow is expressed as:

```
assumed new BigInteger(c.value) * new BigInteger(d.value) ==
        new BigInteger(c.value * d.value) as a_ui1;
```

Loop Unrolling. To avoid the annotation overhead of loop invariants, some static checkers unroll loops a fixed number of times. For instance, ESC/Java unrolls loops 1.5 times by default: first, the condition of the loop is evaluated and in case it holds, the loop body is checked once; then, the loop condition is evaluated again after assuming its negation. As a result, the code following the loop is checked under the assumption that the loop iterates at most once.

This compromise cannot be modeled using explicit assumptions alone. For this reason, we implemented a variant of the Dafny verifier that transforms loops as shown in Fig. 3. After unrolling the loop once, an explicit assumption is added which states that the loop condition does not hold. Assertions following the `assumed` statement are verified under this assumption. Note that the loop is still part of the transformed program so that the original semantics is preserved for downstream static checkers, which might not make the same compromise, and testing tools.

Write Effects. Another compromise made by static tools, such as HAVOC and ESC/Java, involves assuming write effect specifications without checking them. We encode this compromise by simply leaving all the required checks unverified, that is, by not marking them with a `verified` attribute.

Original loop.

```
while (C) {
  B
}
```

Transformed loop.

```
if (C) {
  B
}
assumed ¬C as a;
while (C) {
  B
}
```

Fig. 3. Loop transformation and explicit assumption about loop unrolling. The loop is unrolled 1.5 times.

4.2 Improved Defect Detection

Having shown how `assumed` statements may be used to encode common compromises of static checkers, we will now discuss two scenarios in which Pex exploits explicit assumptions made by upstream static checkers to find more errors than any of these tools alone.

Scenario 1: Overflow Errors. The method of Fig. 4 computes the sum of squares $\sum_{i=\texttt{from}}^{\texttt{to}} i^2$, where `from` and `to` are input parameters. When we run the version of the Dafny verifier that ignores arithmetic overflow on this method, no verification errors are reported and the invariant is partially verified under explicit assumptions about unbounded integer arithmetic. For instance, an `assumed` statement with predicate

```
new BigInteger(i) + new BigInteger(1) == new BigInteger(i + 1)
```

is added before line 10. Running Pex on the original method, where the invariant has been translated into two Code Contracts assertions (one before the loop and one at the end of the loop body), generates five failing unit tests in all of which the invariant is violated before the loop due to an overflow. However, when we run Pex on the partially-verified program produced by the verifier, an additional failing unit test is generated revealing a new error: the invariant is not preserved by the loop due to an overflow in the loop body.

In analyzing these results, we notice that without the explicit assumptions Pex is not able to craft appropriate input values for the method parameters such that

```
 0 static method SumOfSquares(from: int, to: int) returns (r: int)
 1   requires from ≤ to;
 2 {
 3   r := from * from;
 4   var i := from + 1;
 5   while (i ≤ to)
 6     invariant from * from ≤ r;
 7     decreases to - i;
 8   {
 9     r := r + i * i;
10     i := i + 1;
11   }
12 }
```

Fig. 4. Method that computes the sum of squares $\sum_{i=\texttt{from}}^{\texttt{to}} i^2$. The loop invariant is violated in case an integer overflow occurs before the loop or in the loop body.

the invariant preservation error also be revealed. This is because after a bounded number of loop iterations the constraints imposed by the invariant become too complex for the underlying constraint solver to solve under certain time limits, if at all. However, the explicit assumptions added by the verifier create new branches in the method's control flow graph which Pex tries to explore. It is these branches that enrich the tool's path constraints and guide it in picking input values that reveal the remaining error.

Scenario 2: Aliasing Errors. In this scenario, we consider an object hierarchy in which class `Student` and interface `ITeacher` both inherit from interface `IAcademicPerson`, and class `TeachingAssistant` inherits both from class `Student` and interface `ITeacher`. Interface `IAcademicPerson` declares a method `Evaluate` for giving an evaluation grade to an academic person, and a method `Evaluations` for getting all the evaluation grades given to an academic person. Method `EvaluateTeacher` of Fig. 5 takes a student and a rating for the teacher that is associated with the student, and ensures that evaluating the teacher does not affect the student's evaluation grades. The postcondition may be violated when a teaching assistant that is their own teacher is passed to the method. Clousot misses this error because of its heap abstraction, which assumes that certain forms of aliasing do not occur. Pex is also unable to generate a failing unit test because no constraint forces it to generate an object structure that would reveal the error. However, with the explicit assumption shown on line 7 of Fig. 5, Pex does produce a unit test revealing this error.

```
1 public static void EvaluateTeacher(Student s, char rating)
2 {
3   Contract.Requires(s != null && s.Teacher() != null && "ABCDF".Contains(rating));
4   Contract.Ensures(s.Evaluations() ==           // verified under a_nse
5                    Contract.OldValue<string>(s.Evaluations()));
6
7   // assumed s.Teacher() != s as a_nse
8   s.Teacher().Evaluate(rating);
9 }
```

Fig. 5. Method for the evaluation of a student's teacher. The postcondition may be violated when a teaching assistant that is their own teacher is passed to the method.

4.3 Small Test Suites

In addition to finding more errors, our technique is also useful in obtaining small, targeted test suites for partially-verified programs as methods that are fully verified need not be tested. To illustrate this, we developed a `List` class with a number of common list operations: the constructor of the list and methods `Length`, `Equals`, `ContainsElement`, `Head`, `Tail`, `LastElement`, `Prepend`, `Append`, `Concatenate`, `Clone`, and `ReverseInPlace`[2]. This implementation is written in Dafny, consists of about 270 lines of code, and may be found at the URL `http://www.pm.inf.ethz.ch/publications/FM12/List.dfy`.

[2] `ReverseInPlace` is the only method that is implemented iteratively.

In order to simulate a realistic usage scenario of our tool chain, we decided to spend no more than two hours on attempting to soundly verify the code. By the end of that time frame, we had not managed to complete the proof of the `ReverseInPlace` method and were obliged to add `assumed` statements in methods `Equals` and `ContainsElement`.

To evaluate the effectiveness of our technique in achieving small test suites, we compared the size of the suite that was generated by running Pex alone on the list implementation to the number of unit tests that were produced with collaborative verification and testing. For the verification stage of the tool chain, we employed the following four variants of the Dafny verifier: sound verification (S), verification with unbounded integer arithmetic (UIA), verification with loop unrolling (LU), and verification with unbounded integer arithmetic and loop unrolling (UIA & LU). Table 1 shows the percentage by which the size of the test suite was reduced using our technique, and the methods that were still tested (that is, had not been fully verified) in each of the aforementioned verification attempts.

Table 1. Effectiveness of our technique in achieving small test suites

Verification	Test Reduction	Tested Methods
S	66%	`Equals`, `ContainsElement`, `ReverseInPlace`
UIA	58%	`Length`, `Equals`, `ContainsElement`, `ReverseInPlace`
LU	65%	`Equals`, `ContainsElement`, `ReverseInPlace`
UIA & LU	58%	`Length`, `Equals`, `ContainsElement`, `ReverseInPlace`

5 Related Work

Many automatic static checkers that target mainstream programming languages make compromises to improve performance and reduce the number of false positives and the annotation overhead. We already mentioned some of the compromises made by HAVOC, Spec#, ESC/Java, and the Code Contracts static checker. In addition to those, KeY [4] does not soundly support multi-object invariants, Krakatoa [14] does not handle class invariants and class initialization soundly, and Frama-C [7] uses plug-ins for various analyses with possibly conflicting assumptions. Our technique would allow these tools to collaborate and be effectively complemented by automatic test case generation.

Integration of Checkers. The work most closely related to ours is conditional model checking (CMC) [6], which combines complementary model checkers to improve performance and state-space coverage. A conditional model checker takes as input the program and specification to be verified as well as a condition that describes the states that have already been checked, and it produces another such condition to encode the results of the verification. The focus of CMC is on encoding the typical limitations of model checkers, such as space-out and time-out, but it can also encode compromises such as assuming that no arithmetic overflow occurs. Beyer et al. performed a detailed experimental evaluation that demonstrates the benefits of making assumptions and partial verification results explicit, which is in line with our findings. Despite these similarities, there are significant technical differences between CMC and our approach. First, as is common in model checking, CMC is presented as a whole-program analysis, and the resulting condition may contain assumptions about the whole program. For instance, the verification of a method may depend on assumptions made in its callers. By contrast, we have demonstrated how to integrate modular static analyzers, such as Clousot, and deductive verifiers, such as Dafny and Spec#. Second, although Beyer et al. mention test case generation as a possible application of CMC, they do not explain how to generate test cases from the conditions. Since these conditions may include non-local assumptions, they might be used to generate *system* tests, whereas the generation of *unit* tests seems challenging. However, test case generation tools based on constraint solving (such as symbolic execution and concolic testing) do not scale well to the large execution paths that occur in system tests. By contrast, we have demonstrated how to use concolic testing to generate unit tests from our local assumptions and verification results.

A common form of tool integration is to support static checkers with inference tools, such as Houdini [15] for ESC/Java or Daikon [11] for the Java PathFinder [19] tool. Such combinations either assume that the inference is sound and thus, do not handle the compromises addressed in our work, or they verify every property that has been inferred, which is overly conservative and increases the verification effort. Our technique enables a more effective tool integration by making all design compromises explicit.

Integration of Verification and Testing. Various approaches combine verification and testing mainly to determine whether a static verification error is spurious. Check 'n' Crash [9] is an automated defect detection tool that integrates the ESC/Java static checker with the JCrasher [8] testing tool in order to decide whether errors emitted by the static checker truly exist. Check 'n' Crash was later integrated with Daikon in the DSD-Crasher tool [10]. DyTa [17] integrates the Code Contracts static checker with Pex to reduce the number of spurious errors compared to static verification alone and perform more efficiently compared to dynamic test generation alone. Confirming whether a failing verification attempt refers to a real error is also possible in our technique: The instrumentation phase of the architecture introduces assertions for each property that has not been statically verified (which includes the case of a failing verification attempt). The testing phase then uses these assertions to direct test case generation towards the unproved properties. Eventually, the testing tools might generate either a series of successful test cases that will boost the user's confidence about the correctness of their programs or concrete counterexamples that reproduce an error.

A perhaps more precise approach towards the same direction as the aforementioned tools is counterexample-guided abstraction refinement (CEGAR) [2,5].

CEGAR exploits the abstract counterexample trace of a failing proof attempt to suggest a concrete trace that might reveal a real error. If, however, the abstract trace refers to a spurious error, the abstraction is refined in such a way that subsequent verification attempts will not reproduce the infeasible abstract trace. More recently, YOGI [22], a tool for checking properties of C programs, was developed to refine CEGAR with concolic execution. Such techniques, if regarded as tool chains, address the issue of program correctness from the opposite direction than we do: they use concrete traces to refine static over-approximations, whereas, in our work, combinations of potential under-approximations made by different static checkers are checked by the testing tools. If, on the other hand, these techniques are regarded as single tools, they could also be integrated in our architecture.

6 Conclusion

We have presented a technique for collaborative verification and testing that makes compromises of static checkers explicit with a simple language extension. In our approach, the verification results give definite answers about program correctness allowing for the integration of multiple, complementary static checkers and the generation of more effective unit test suites. Our experiments suggest that our technique finds more errors and proves more properties than verification alone, testing alone, and combined verification and testing without the explicit assumptions. As future work, we plan to implement our technique for Spec# and the Code Contracts static checker and to use them for experiments on large code bases. We expect such experiments to shed light on the impact of some design compromises and suggest guidelines for the effective use of static checkers in industrial projects.

Acknowledgments. We would like to thank Alexander Summers and the anonymous reviewers for their helpful comments.

References

1. Ball, T., Hackett, B., Lahiri, S.K., Qadeer, S., Vanegue, J.: Towards Scalable Modular Checking of User-Defined Properties. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 1–24. Springer, Heidelberg (2010)
2. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: POPL, pp. 1–3. ACM (2002)
3. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: The Spec# experience. CACM 54, 81–91 (2011)
4. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
5. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST: Applications to software engineering. STTT 9, 505–525 (2007)
6. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking. CoRR, abs/1109.6926 (2011)

7. Correnson, L., Cuoq, P., Kirchner, F., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C User Manual (2011), `http://frama-c.com//support.html`
8. Csallner, C., Smaragdakis, Y.: JCrasher: An automatic robustness tester for Java. SPE 34, 1025–1050 (2004)
9. Csallner, C., Smaragdakis, Y.: Check 'n' Crash: Combining static checking and testing. In: ICSE, pp. 422–431. ACM (2005)
10. Csallner, C., Smaragdakis, Y., Xie, T.: DSD-Crasher: A hybrid analysis tool for bug finding. TOSEM 17, 1–37 (2008)
11. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69, 35–45 (2007)
12. Fähndrich, M., Barnett, M., Logozzo, F.: Embedded contract languages. In: SAC, pp. 2103–2110. ACM (2010)
13. Fähndrich, M., Logozzo, F.: Static Contract Checking with Abstract Interpretation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 10–30. Springer, Heidelberg (2011)
14. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
15. Flanagan, C., Leino, K.R.M.: Houdini, an Annotation Assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
16. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI, pp. 234–245. ACM (2002)
17. Ge, X., Taneja, K., Xie, T., Tillmann, N.: DyTa: Dynamic symbolic execution guided with static verification results. In: ICSE, pp. 992–994. ACM (2011)
18. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: PLDI, pp. 213–223. ACM (2005)
19. Havelund, K., Pressburger, T.: Model checking JAVA programs using JAVA PathFinder. STTT 2, 366–381 (2000)
20. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML Reference Manual (2011), `http://www.jmlspecs.org/`
21. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
22. Nori, A.V., Rajamani, S.K., Tetali, S., Thakur, A.V.: The YOGI Project: Software Property Checking via Static Analysis and Testing. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 178–181. Springer, Heidelberg (2009)
23. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: ESEC, pp. 263–272. ACM (2005)
24. Tillmann, N., de Halleux, J.: Pex–White Box Test Generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)