DISS. ETH NO. 22795

Narrowing the gap between verification and systematic testing

A thesis submitted to attain the degree of

Doctor of sciences of ETH Zurich
(Dr. sc. ETH Zurich)

Presented by

Maria Christakis
Dipl. Electrical and Computer Engineering
National Technical University of Athens, Greece

Born on October 9, 1986
Citizen of Greece

Accepted on the recommendation of

Prof.	Peter Müller	:	examiner
Prof.	Cristian Cadar	:	co-examiner
Dr.	Patrice Godefroid	:	co-examiner
Prof.	Thomas Gross	:	co-examiner

2015

To my parents, Thekla and Yannis, in deepest love and gratitude.

Abstract

This dissertation focuses on narrowing the gap between verification and systematic testing, in two directions: (1) by complementing verification with systematic testing, and (2) by pushing systematic testing toward reaching verification.

In the first direction, we explore how to effectively combine unsound static analysis with systematic testing, so as to guide test generation toward properties that have not been checked (soundly) by a static analyzer. Our combination significantly reduces the test effort while checking more unverified properties. In the process of testing properties that are typically not soundly verified by static analyzers, we identify important limitations in existing test generation tools, with respect to considering sufficient oracles and all factors that affect the outcome of these oracles. Specifically, testing tools that use object invariants as filters on input data do not thoroughly check whether these invariants are actually maintained by the unit under test. Moreover, existing test generation tools ignore the potential interaction of static state with a unit under test. We address these issues by presenting novel techniques based on static analysis and dynamic symbolic execution. Our techniques detected a significant number of errors that were previously missed by existing tools. We also investigate in which ways it is beneficial to complement sound, interactive verification with systematic testing for the properties that have not been verified yet.

In the second direction, we push systematic testing toward verification, in particular, toward proving memory safety of the ANI Windows image parser. This is achieved by concentrating on the main limitations of systematic dynamic test generation, namely, imperfect symbolic execution and path explosion, in the context of this parser. Based on insights from attempting to reach verification of the ANI parser using only systematic testing, we then define IC-Cut, a new compositional search strategy for automatically and dynamically discovering simple function interfaces, where large programs can be effectively decomposed. IC-Cut preserves code coverage and increases bug finding in significantly less exploration time compared to the current search strategy of the dynamic symbolic execution tool SAGE. An additional novelty of IC-Cut is that it can identify which decomposed program units are exhaustively tested and, thus, dynamically verified.

Résumé

Cette thèse s'efforce de combler le fossé qui existe aujourd'hui entre vérification et test systématique en explorant deux axes: d'une part, en complétant la vérification par des tests systématiques, et d'autre part, en repoussant les limites des tests systématiques pour obtenir des garanties proches de celles obtenues par la vérification.

Dans un premier temps, nous étudions l'interaction des analyses statiques non-sûres avec le test systématique ; nous montrons comment orienter la génération de tests de manière à couvrir des propriétés qui n'ont pas été vérifiées (de manière sûre) par l'analyseur statique. Notre technique combinant les deux approches réduit de manière significative les efforts consacrés au test, tout en vérifiant davantage de propriétés. Lorsqu'il s'agit de tester les propriétés qui ne sont généralement pas vérifiées de manière correcte par les analyseurs statiques, nous mettons en exergue d'importantes limitations des outils de génération de tests préexistants ; en particulier, nous soulignons l'importance du choix d'un oracle correct, ainsi que l'importance de prendre en compte tous les facteurs susceptibles d'influencer les résultats de l'oracle. Plus précisément, les outils de test qui se basent sur des invariants d'objets pour filtrer leurs entrées ne vérifient généralement pas que ces invariants sont bel et bien maintenus dans le module présentement soumis aux tests. De plus, les outils actuels de génération de tests ne prennent pas en compte l'interaction entre l'état global du programme et le comportement du module testé. Nous proposons une réponse à ces problèmes, et présentons des techniques novatrices basées sur l'analyse statique et l'exécution symbolique dynamique. Nos techniques ont réussi à identifier un nombre tout à fait significatif d'erreurs qui n'avaient jusqu'alors jamais été décelées par les outils existants. Nous explorons également différents contextes où il peut s'avérer utile de compléter une phase de vérification interactive sûre par du test systématique visant les propriétés qui n'ont pas encore été vérifiées.

Dans un second temps, nous rapprochons le test systématique de la vérification, plus particulièrement, en essayant de prouver l'absence d'erreurs mémoires dans l'analyseur syntaxique d'images Windows ANI. Ce résultat est obtenu en se focalisant sur les limitations fondamentales de la génération systématique de tests dynamiques, à savoir, dans le cas de cet analyseur syntaxique, une exécution symbolique imparfaite et une explosion combina-

toire. Ce que nous avons appris en vérifiant l'analyseur syntaxique ANI, et ce uniquement à l'aide de test systématique, a été mis à profit dans le projet IC-Cut: IC-Cut est une nouvelle procédure de recherche pour le test systématique, qui opère de manière compositionnelle, et permet de découvrir les interfaces simples des fonctions existantes. Il est ainsi possible de décomposer de larges programmes. Comparé à la stratégie de recherche de l'outil SAGE (exécution symbolique dynamique), IC-Cut garde la même surface de code couvert, tout en consacrant beaucoup moins de temps à l'exploration. Un apport supplémentaire d'IC-Cut est qu'il permet d'identifier lesquels des modules ont été testés de manière exhaustive, c'est-à-dire, vérifiés de manière dynamique.

Acknowledgments

I would like to acknowledge:

Peter Müller, my supervisor, who generously shared his time and expertise with me, challenged me to do better, let me be stubborn, and stuck with me through all the twists and turns. Peter, thank you for your sage advice, infinite patience, excellent guidance, good humor, and the life-changing Skype call on Tuesday, May 10, 2011. I hope that our friendship and collaboration continue to grow even stronger in time.

Patrice Godefroid, my internship mentor, whose undivided commitment to research I admire. Patrice, I become better just watching you attack a problem. Thank you for your unlimited energy, sharp feedback, immense help, earnest attention, and the enviable opportunity to work with you.

The co-examiners of my dissertation, Cristian Cadar, Patrice Godefroid, and Thomas Gross, who took the time out of their busy schedules to be on my defense committee and provided many insightful comments and suggestions on this dissertation.

Valentin Wüstholz, my friend and collaborator, who not only is the inventor of free calls, but also knows how to pick wine. Valentin, thank you for your tolerance and moral support, restaurant suggestions, write effects, abstractions, visitors, and debuggers.

Marlies Weissert, who makes the office a magical place. Marlies, thank you for your tremendous assistance, lovely smile, and invaluable fashion tips.

Malte Schwerhoff (or Malticz), my friend and end-of-week coffee buddy.

Dimitar Asenov (or professional Mitko), my favorite IT coordinator.

Lucas Brutschy (or Lucky Luke), my after-lunch coffee buddy.

Milos Novacek, for not hesitating to disclose the secrets of the heap.

Uri Juhasz, for subsidization.

The members of our group, Cédric Favre, Pietro Ferrara, Yannis Kassios, and Alex Summers. I am grateful for our years together and your reassurance when times were tough.

My students, Patrick Emmisberger, Patrick Spettel, Timon Gehr, and David Rohr. Thank you for your fresh ideas and for turning them into code. Patrick Emmisberger, thank you for introducing me to the delicious world of cordon blue.

Denise Spicher, for her keen help, attentive answers to all my questions, and always-pleasant mood.

Everyone at Microsoft Research Redmond. Thank you for numerous discussions, your active encouragement, and warmest welcome. Jonathan Protzenko, thank you for the elegant translation of the abstract of this dissertation into French.

My parents, Thekla and Yannis, to whom I am forever grateful. You did everything more than right, and I would be nowhere without you.

Thank You.

Contents

Algorithms

Figures

Tables

Chapter 1

Introduction

On May 1, 2015, the Federal Aviation Administration published a new airworthiness directive [60] for all The Boeing Company Model 787 airplanes, which requires a repetitive maintenance task for electrical power deactivation. The directive was prompted by a software error, detected during laboratory testing, that causes the electricity generators of a Boeing 787 to shut down every 248 days. This issue, which can arise even in flight, is the consequence of an arithmetic overflow that occurs when incrementing an integer counter over 248 days of continuous power. Although it is expected that such issues should occur, especially since it is standard procedure to reboot flight-control software systems very frequently, it is surprising that this error was discovered so late in the development process.

Software systems are ubiquitous in modern, everyday life. Their robustness and reliability is, therefore, vital in our society and constitutes a primary goal of the computer-science community. Formal verification and automated, systematic testing are two fundamental research areas of computer science, aiming to ensure software correctness and identify issues like the aforementioned overflow as early as possible.

Verification has been studied for approximately five decades and is increasingly applied in industrial software development to detect errors. Verification techniques allow developers to optionally define a set of desired properties that a program is meant to satisfy, and then prove that the program is actually correct, while complying with these properties. So far, verification tools have been so effective in detecting errors in real-world programs that they are increasingly and routinely used in many software development organizations. In fact, there is a wide variety of such tools, targeting mainstream programming languages and ranging from relatively simple heuristic tools, over abstract interpreters and software model checkers, to verifiers based on automatic theorem proving.

Over the last ten years, there has been revived interest in systematic testing, and in particular, in testing techniques that rely on symbolic execution [87], introduced more than three decades ago [24, 23, 68, 22]. Recent

significant advances in constraint satisfiability and the scalability of simultaneous concrete and symbolic executions have brought systematic dynamic test generation to the spotlight, especially due to its ability to achieve high code coverage and detect errors deep in large and complex programs. As a result, dynamic test generation is having a major impact on many research areas of computer science, for instance, on software engineering, security, computer systems, debugging and repair, networks, education, and others.

In light of the above background and observations, this dissertation focuses on narrowing the existing gap between verification and systematic testing, in two directions. In the first direction, we complement verification with systematic testing, to maximize software quality while reducing the test effort. In particular, we precisely define the correctness guarantees that verifiers provide, such that they can be effectively compensated for by dynamic test generation. At the same time, we enhance systematic testing techniques with better oracles, and enable these techniques to consider factors that affect the outcome of such oracles but were previously ignored. This research direction enables the detection of more software errors, earlier in the development process, and with fewer resources. In the second direction, we explore how far systematic testing can be pushed toward reaching verification of real applications. Specifically, we assess to what extent the idea of reaching verification with systematic testing is realistic, in the scope of a particular application domain. This research direction sheds light to the potential of dynamic test generation in ensuring software correctness.

Outline. This chapter is organized as follows. In Sect. 1.1, we give an overview of the basic principles of systematic dynamic test generation. Sects. 1.2 and 1.3 introduce and motivate the two research directions of this dissertation, namely, complementing verification with systematic testing, and pushing systematic testing toward verification.

1.1 Systematic dynamic test generation

We consider a sequential deterministic program P, which is composed of a set of functions (or methods) and takes as input an input vector, that is, multiple input values. The determinism of the program guarantees that running P with the same input vector leads to the same program execution.

We can systematically explore the state space of program P using *systematic dynamic test generation* [70, 20], also called *concolic testing* [128]. Systematic dynamic test generation performs *dynamic symbolic execution*, which consists of repeatedly running a program both concretely and symbolically. The goal is to collect symbolic constraints on inputs, from predicates in branch statements along the execution, and then to infer variants of the previous inputs, using a constraint solver, in order to steer the next execution of the program toward an alternative program path. In this dissertation, we

Algorithm 1.1: Systematic dynamic test generation.

```
1 function EXPLORE(seq<Constraint> prefix)
2   inputs ← SOLVE(prefix)
3   if solution is available then
4     path ← EXECUTECONCRETE(inputs)
5     pathConstraint ← EXECUTESYMBOLIC(path)
6     extension ← pathConstraint[|prefix|...]
7     foreach non-empty prefix p of extension in
8       if p = p'∘ [BRANCH(c)] for some p', c then
9         EXPLORE(prefix ∘ p'∘ [BRANCH(¬c)])
```

use the terms "dynamic test generation" and "dynamic symbolic execution" interchangeably.

Symbolic execution means executing a program with symbolic rather than concrete values. A symbolic variable is, therefore, associated with each value in the input vector, and every constraint is on such symbolic variables. Assignment statements are represented as functions of their (symbolic) arguments, while conditional statements are expressed as constraints on symbolic values. Side-by-side concrete and symbolic executions are performed using a concrete store M and a symbolic store S, which are mappings from memory addresses (where program variables are stored) to concrete and symbolic values, respectively. For a program path w, a *path constraint* (or *path condition*) ϕ_w is a logic formula that characterizes the input values for which the program executes along w. Each symbolic variable appearing in ϕ_w is, thus, a program input. Each constraint is expressed in some theory[1] T decided by a constraint solver, i.e., an automated theorem prover that can return a satisfying assignment for all variables appearing in constraints it proves satisfiable. This assignment drives execution along the desired path w.

Alg. 1.1 [115] is a general algorithm for systematic dynamic test generation. Function EXPLORE explores all program paths whose path constraints start with a given sequence of constraints, which we call *prefix*. Consequently, testing of a program starts by a call to EXPLORE([]). On line 2, we obtain concrete values for the program inputs by solving the constraints in the prefix. For an empty prefix, the inputs are generated randomly. On lines 4 and 5, we run the program with these inputs both concretely and symbolically to generate the path constraint for this execution. On line 6, we obtain the extension of the generated path constraint, that is, the path constraint without the given prefix. For each constraint in the extension that indicates a branching condition BRANCH(c), as opposed to an assumed condition (explained below), we call function EXPLORE recursively to exer-

[1] A theory is a set of logic formulas.

cise the alternative program path for this branch (lines 7–9). The order in which we negate the branching conditions of the extension is imposed by the exploration strategy, for instance, by a depth- or breadth-first strategy.

When assumed conditions, of the form ASSUME(c), appear in a path constraint, they denote invariants of the corresponding execution, for instance, due to a program precondition. If an execution reaches a program point that normally contributes such an assumed condition to the path constraint but $\neg c$ holds at that point, the execution is aborted and ASSUME(c) is added to the path constraint for subsequent explorations. This condition c is never negated.

Note that, on line 2 of Alg. 1.1, the constraint solver might return "unknown", which is a potential outcome for solvers. In this case, we simplify the prefix by replacing a symbolic variable with its concrete value from the parent run of function EXPLORE, and call the solver again. This simplification can be encoded with an assumed condition stating that the symbolic variable is equal to the corresponding concrete value, but might result in missing certain program paths.

Whitebox fuzzing is an application of systematic dynamic test generation for detecting security vulnerabilities. In particular, *whitebox file fuzzing* explores programs that take as input a file, all bytes of which constitute the input vector of the program. SAGE [73] is the first whitebox file fuzzing tool for security testing, which implements systematic dynamic test generation and performs dynamic symbolic execution at the x86 binary level. It is optimized to scale to very large execution traces (billions of x86 instructions) and programs (like Excel) [16]. Note that, throughout this dissertation, we always use the term "dynamic test generation" or "dynamic symbolic execution" to denote the exploration of Alg. 1.1 and its applications.

All program paths in program P can be enumerated by a search algorithm, like Alg. 1.1, that explores all possible branches at conditional statements. The paths w for which ϕ_w is satisfiable are feasible, and are the only ones that can be executed by the actual program provided the solutions to ϕ_w characterize exactly the inputs that drive the program through w. Assuming that the constraint solver used to check the satisfiability of all formulas ϕ_w is sound and complete, this use of symbolic execution for programs with finitely many paths amounts to program verification. Note that, in this dissertation, we use the terms "soundness" and "completeness" to refer to the absence of false negatives and false positives, respectively.

Obviously, testing and symbolically executing *all* feasible program paths is not possible for large programs. Indeed, the number of feasible paths can be exponential in the program size, or even infinite in the presence of loops with an unbounded number of iterations. In practice, this *path explosion* is alleviated using *heuristics* to maximize code coverage as quickly as possible and find bugs faster in a partial search. For instance, SAGE uses a *generational-search strategy* [73], where all constraints in a path constraint

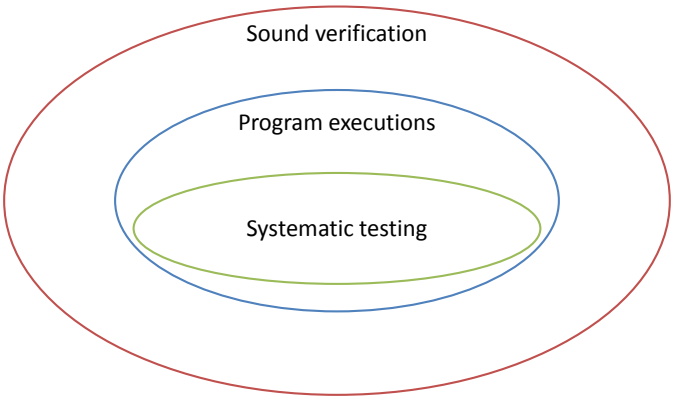

Figure 1.1: Sound verification over-approximates the set of all possible program executions, whereas systematic testing typically under-approximates this set.

are negated one by one (by the Z3 theorem prover [47]) to maximize the number of new tests generated per symbolic execution. This strategy is combined with simple heuristics that guide the exploration toward least covered parts of the search space and prune the search space using *flip count limits* and *constraint subsumption*. Other related industrial-strength tools like Pex [135] use similar techniques.

1.2 Combining verification and systematic testing

It is established that sound verification over-approximates the set of possible program executions, as shown in Fig. 1.1, in order to prove the absence of errors in a program. On the other hand, systematic testing typically under-approximates the set of possible program executions with the purpose of proving the existence of errors in the program [75].

In practice, modern software projects use a variety of techniques to detect program errors, such as testing, code reviews, and static program analysis [82], none of which check all possible executions of a program. They often leave entire paths unverified (for instance, when a test suite does not achieve full path coverage), fail to verify certain properties (such as complex assertions), or verify some paths under assumptions (such as the absence of arithmetic overflow) that might not hold on all executions of the path. Making such assumptions is necessary in code reviews to reduce the complexity of the task; it is also customary in static program analysis to improve the precision, performance, and modularity of the analysis [32], and because some program features elude static checking [106]. That is, most static analyses sacrifice soundness in favor of other important qualities.

Although static analyzers effectively detect software errors, they can-

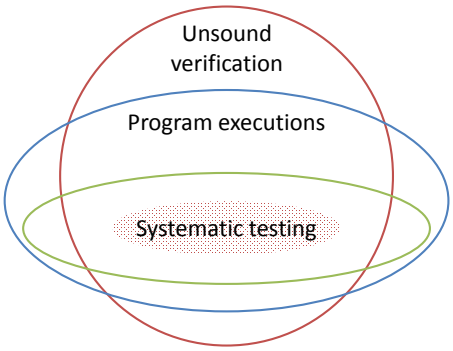

Figure 1.2: In existing work, systematic testing targets only those program executions for which a static verification error has been emitted (shaded area), thus ignoring the executions that unsound verification has missed.

not replace or significantly reduce the test effort. Many practical analyzers for mainstream programming languages make a number of compromises to increase automation, reduce the annotation overhead for the programmer, reduce the number of false positives, and speed up the analysis. These compromises include not checking certain properties (e.g., termination), making implicit assumptions (e.g., arithmetic operations never overflow), and unsoundness (e.g., only considering a fixed number of loop iterations).

Despite these limitations, static analyzers find real errors in real code. However, as a result of these limitations, it is not clear what guarantees a static analysis actually provides about program correctness. It is also not clear how to use systematic testing to check exactly those properties that are not soundly verified by a static analysis. Consequently, software engineers need to test their programs as if no static analysis were applied, which is inefficient, for one, because it requires large test suites.

Until now, various approaches have combined verification and testing [43, 44, 65], but mainly to determine whether a static verification error is spurious (i.e., whether a warning emitted by a verification tool is a false positive). However, these approaches do not take into account that unsound static analyses might generate false negatives and do not address the limitations of verifiers described above. In other words, testing aims to target only those program executions for which a static verification error has been emitted, thus ignoring executions that have not been checked by a static analyzer due to its unsoundness, as shown in Fig. 1.2.

To address this problem, we have developed a technique for combining verification and systematic testing, which guides the latter not only toward those program executions for which a verification error has been emitted, but also toward those executions that unsound verification has missed. The program executions that systematic testing aims to cover with our technique

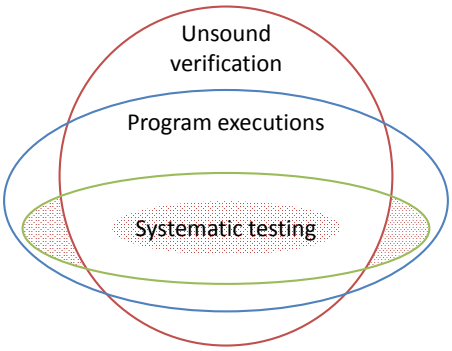

Figure 1.3: In our work, systematic testing targets those program executions for which a verification error has been emitted as well as those that unsound verification has missed (shaded areas).

are depicted by the shaded areas in Fig. 1.3.

In particular, we have proposed a tool architecture that (1) combines multiple, complementary static analyzers that check different properties and make different assumptions, and (2) complements static analysis with systematic dynamic test generation to cover those properties that have not been checked statically [30]. A key originality of this architecture is that it makes explicit which properties have been checked statically and under which assumptions. Therefore, the correctness guarantees provided by static analyzers are documented precisely, and can guide test generation toward those properties that are not verified yet, leading to smaller and more effective test suites.

The three contributions made by our tool architecture are:

1. It makes deliberate compromises of static analyzers explicit, by marking every program assertion as either fully verified, verified under certain assumptions, or not verified at all.

2. It automatically generates test cases from the results of static analyzers, providing the user with a choice on how much effort to devote to static analysis 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 architecture 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 analysis is configurable and may range from zero to complete.

3. It directs the static analysis and test case generation to the properties

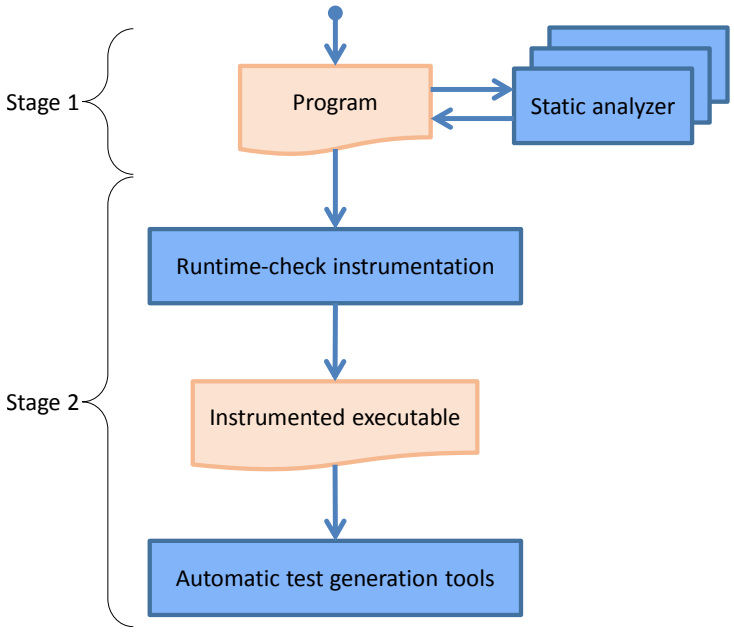

Figure 1.4: Tool architecture for complementing static verification with systematic testing. Tools are depicted by boxes and programs by document symbols.

that have not been (soundly) checked. This leads to more targeted static analysis and testing, in particular, smaller and more effective test suites.

Our tool architecture, which is presented in Fig. 1.4, takes a program containing code, specifications, and all properties that need to be checked for the program, such as division-by-zero errors and null dereferences. For each check in the given program, our architecture records whether it has been soundly verified and under which assumptions. Fig. 1.4 shows that this tool architecture consists of two stages, where stage 1 is static verification and stage 2 is systematic testing.

The static analysis (or verification) stage allows the user to run an arbitrary number (possibly zero) of static analyzers. Each analyzer reads the program, which might also contain results of prior static analysis attempts, and tries to verify any properties that have not already been proven by upstream tools. As previously described, each assertion is marked to be either fully (that is, soundly) verified, verified under certain assumptions, or not verified (that is, not attempted or failed to verify). An analyzer then attempts to prove the assertions that have not been fully verified by upstream tools. Each analyzer records its results in the program, which serves as input to the next downstream tool.

When the static analysis stage is completed, the output program is ei-

ther entirely (and soundly) verified or there still exist unproven checks. The intermediate versions of the program precisely track which properties have been fully verified and which remain to be validated. 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 analysis is limited. Any remaining assertions may then be covered by the subsequent testing stage.

In this second stage, we apply dynamic symbolic execution to automatically generate test cases from the program code, specifications, and the results of static analysis. In particular, the properties that remain to be checked as well as the assumptions made by static analyzers occur, in the form of runtime checks, in an instrumented version of the program. The resulting instrumented program can then be fed to one or more test generation tools. Our instrumentation causes the symbolic execution of these tools to generate the constraints and test data that exercise exactly the properties that have not been statically verified (in a sound way), thus reducing the size of the generated test suites. However, not all specifications can be efficiently checked at runtime (for example, object invariants), and programs may interact with their environment in various ways (for example, through static state). Therefore, in this stage of the architecture, we also enforce strategies for significantly improving the results of systematic testing, in terms of effectively exercising more properties and detecting more errors.

By developing this architecture, we have investigated the following scientific topics.

How to design an annotation language that supports verification and systematic testing

Several annotation languages have been developed since design by contract [109] was first proposed. To name a few, Eiffel [108], the Java Modeling Language (JML) [93], Spec# [12], and Code Contracts [58] for .NET have been designed to enable both static analysis and runtime checking of their annotations. However, these languages cannot support the tool integration of Fig. 1.4. For instance, Eiffel and Code Contracts are not expressive enough for verification purposes, whereas JML and Spec# are tied to a particular verification methodology. Moreover, annotation languages based on separation logic [85, 126] have restricted support for runtime checking [119].

For our purposes, we have designed an annotation language for making deliberate compromises of static analyzers explicit [30] (see Ch. 2). The main virtues of our annotations are that they are (1) simple and easy to support by a wide range of static and dynamic tools [33], (2) expressive, as we have demonstrated by encoding the typical compromises made by deductive verifiers [30] and the .NET abstract interpreter Clousot [59, 32], and (3) well suited for test case generation [30, 33].

What the limitations of mainstream verifiers are and how to make these limitations explicit

Many mainstream static analyzers make compromises in order to increase automation, improve performance, or reduce both the number of false positives and the annotation overhead. For example, HAVOC [5] uses write effect specifications without checking them, Spec# [12] ignores arithmetic overflow and does not consider exceptional control flow, ESC/Java [62] unrolls loops a fixed number of times, the .NET abstract interpreter Clousot [59] uses an unsound heap abstraction, KeY [13] does not have sound support for multi-object invariants, Krakatoa [61] does not handle class invariants and class initialization soundly, and Frama-C [39] uses plug-ins for various analyses with possibly conflicting assumptions, to mention a few.

As long as the compromises of such tools are made explicit, their users should immediately benefit from our tool architecture, which allows these tools to collaborate and be effectively complemented by automatic test case generation.

We have used our annotations to encode typical compromises made by deductive verifiers [30]. We have also encoded most soundness compromises in Clousot, a widely-used, commercial static analyzer. We measured the impact of the unsound assumptions in Clousot on several open-source projects, which constituted the first systematic effort to document and evaluate the sources of unsoundness in an analyzer [32].

How to combine verification and systematic testing to maximize code quality and minimize the test effort

As mentioned above, various existing approaches combine unsound verification and testing [43, 44, 65], primarily to determine whether a static verification error is spurious. Confirming whether a failing verification attempt refers to a real error is also possible in our tool architecture, as shown in Fig. 1.3. The runtime-check 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 stage then uses these assertions to direct test case generation toward the unproven properties. Eventually, the testing tools might generate either a series of successful test cases that will boost the users' confidence about the correctness of their programs or concrete counterexamples that reproduce an error.

To explore how to most effectively combine unsound static analysis and systematic testing with our annotations, we built our tool chain for .NET using the static analyzer Clousot and the dynamic symbolic execution tool Pex. In this setting, we investigated how to best exploit these annotations for guiding test generation toward properties that have not been previously

checked soundly by a static analysis.

In the next chapter, we present a technique for reducing redundancies with static analysis when complementing partial verification results (expressed using our annotations) by automatic test case generation [33]. Our main contribution is a code instrumentation that causes dynamic symbolic execution to abort tests that lead to verified executions, to prune parts of the search space, and to prioritize tests that lead to unverified executions. This instrumentation is based on an efficient static inference, which propagates information that characterizes unverified executions higher up in the control flow, where it may prune the search space more effectively. Compared to directly running Pex on the annotated programs without our instrumentation, our technique produces smaller test suites (by up to 19.2%), covers more unverified executions (by up to 7.1%), and reduces testing time (by up to 52.4%).

We also find it beneficial to complement sound, interactive verification with systematic testing for the properties that have not been verified yet. In Ch. 5, we present Delfy, a dynamic test generation tool for complementing Dafny, a sound and interactive verifier. In that chapter, we explore how to test interesting language and specification constructs of the Dafny programming language, and discuss how testing can save the effort of attempting to verify an incorrect program, debugging spurious verification errors, and writing redundant specifications.

How to generate tests for program properties that are difficult to verify and lie beyond the capabilities of systematic testing

In the second stage of our tool architecture, we attempt to test those properties that have not been soundly verified by upstream static analyzers. Since our ultimate goal is to automatically provide evidence on whether a program is correct, this stage of the architecture must be able to efficiently generate test oracles (in the form of runtime checks) for any unverified, rich properties, and test inputs for thoroughly evaluating these oracles. We investigate how to achieve an attractive trade-off between performance and coverage of test oracles, by using simple static analyses to reduce both the number of oracles that need to be checked as well as the number of test inputs that affect these oracles.

In Ch. 3, we describe and address a limitation of existing testing tools in generating strong enough oracles for a certain rich specification. In particular, automatic test case generation techniques rely on a description of the input data that the unit under test is intended to handle. For heap data structures, such a description is typically expressed as some form of object invariant [109, 101]. If a program may create structures that violate the invariant, the test data generated using the invariant systematically ignores possible inputs and, thus, potentially misses bugs. We address this limita-

tion with a technique that detects violations of object invariants [31]. We describe three scenarios in which traditional invariant checking (as implemented in existing test generation tools) may miss such violations. Based on a set of predefined templates that capture these scenarios, we synthesize parameterized unit tests [136] that are likely to violate invariants, and use dynamic symbolic execution to generate inputs to the synthesized tests. We have implemented our technique as an extension to the dynamic symbolic execution tool Pex [135] and applied it on open-source applications, both for invariants manually written by programmers and invariants automatically inferred by Daikon [57]. In both cases, we detected a significant number of invariant violations.

In Ch. 4, we focus on a second limitation of existing testing tools in generating suitable inputs to a unit under test. Although static state is common in object-oriented programs, automatic test case generators do not take into account the potential interference of static state with a unit under test and may, thus, miss subtle errors. In particular, existing test case generators do not treat static fields as input to the unit under test, and do not control the execution of static initializers. We address these issues by proposing a novel technique in automatic test case generation [27], based on static analysis and dynamic symbolic execution. We have applied this technique on a suite of open-source applications and found errors that go undetected by existing test case generators. Our experiments show that this problem is relevant in real code, indicate which kinds of errors existing tools miss, and demonstrate the effectiveness of our technique.

1.3 Pushing systematic testing toward verification

Systematic dynamic test generation has been implemented in many popular tools over the last decade, such as EXE [21], jCUTE [127], Pex [135], KLEE [19], BitBlaze [130], and Apollo [3], to name a few. Although effective in detecting bugs, these testing tools have never been pushed toward program verification of a large and complex application, i.e., toward proving that the application is free of certain classes of errors. In this second part of the dissertation, we assess to what extent reaching verification with systematic testing is feasible in practice, in the scope of a particular application domain, namely, that of binary image parsers. Specifically, in the scope of this application domain, we assess whether Fig. 1.5 is realistic, that is, whether it is realistic to push systematic testing to cover all possible program executions.

In Ch. 6, we report how we extended systematic dynamic test generation toward program verification of the ANI Windows image parser, written in low-level C [29]. To achieve this, we applied only three core techniques, namely (1) symbolic execution at the x86 binary level, (2) exhaustive pro-

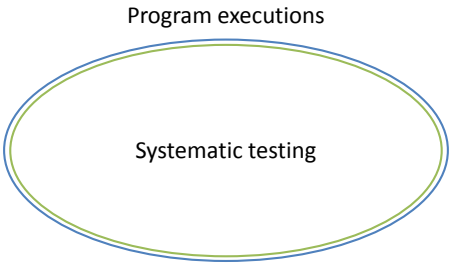

Figure 1.5: Assessing whether it is realistic to push systematic testing to cover all possible program executions.

gram path enumeration and testing, and (3) user-guided program decomposition and summarization. We used SAGE and a very recent tool, named MicroX [67], for executing code fragments in isolation with a custom virtual machine designed for testing purposes. As a result of this work, we are able to prove, for the first time, that a complex Windows image parser is memory safe, i.e., free of any buffer-overflow security vulnerabilities, modulo the soundness of our tools and several additional assumptions, regarding bounding input-dependent loops, fixing a few buffer-overflow bugs, and excluding some code parts that are not memory safe by design. In the process, we also discovered and fixed several limitations in our tools.

In Ch. 6, we limit path explosion in the parser with user-guided program decomposition and summarization. In particular, we decompose the program at only a few function interfaces, which are very simple so that the logic encoding of the summaries remains tractable. Based on these insights, in Ch. 7, we define IC-Cut, short for "Interface-Complexity-based Cut", a compositional search strategy for systematically testing large programs [28]. IC-Cut dynamically detects function interfaces that are simple enough to be cost-effective for summarization. IC-Cut then hierarchically decomposes the program into units defined by such functions and their sub-functions in the call graph. These units are tested independently, their test results are recorded as low-complexity function summaries, and the summaries are reused when testing higher-level functions in the call graph, thus limiting overall path explosion. When the decomposed units are tested exhaustively, they constitute verified components of the program. IC-Cut is run dynamically and on-the-fly during the search, typically refining cuts as the search advances. We have implemented this algorithm as a new search strategy in SAGE, and present detailed experimental results obtained when testing the ANI Windows image parser. Our results show that IC-Cut alleviates path explosion while preserving or even increasing code coverage and bug finding, compared to the current generational-search strategy used in SAGE.

Part I

Complementing verification with systematic testing

Chapter 2

Guiding dynamic symbolic execution toward unverified program executions

In this chapter, we explore how to effectively combine verification with systematic testing to maximize code quality while minimizing the test effort. Dynamic symbolic execution (DSE) [70, 20] systematically explores a large number of program executions and, thus, effectively detects errors missed by other techniques, such as code reviews and unsound static program analysis. However, simply applying DSE in addition to other techniques leads to redundancy, when executions covered by DSE have already been verified using other techniques. In this case, the available testing time is wasted on executions that are known to be correct rather than on exploring previously-unverified executions. This redundancy is especially problematic when DSE is used to complement static analyzers, because static techniques can check a large fraction of all possible program executions and, thus, many or even most of the executions covered by DSE are already verified.

Method `Deposit` in Fig. 2.1 illustrates this problem. A reviewer or static analyzer that checks the implementation under the assumption that the addition on line 5 does not overflow might miss violations of the assertion on line 10. Applying DSE to the method tries to explore six different paths through the method (there are three paths through the conditionals, each combined with two possible outcomes for the assertion), in addition to all the paths through the called methods `ReviewDeposit` and `SuggestInvestment`. Assuming that these two methods are correct, only one of all these paths reveals an error, namely, the path that is taken when `amount` is between zero and 50,000, and `balance` is large enough for the addition on line 5 to overflow. All other generated test cases are redundant because they lead to executions that have already been verified. In particular, if the called methods have complex control flow, DSE might not detect the error because it reaches a timeout before generating the only relevant test case.

To reduce this redundancy, existing work [43, 65, 26] integrates static analyses and DSE; it uses the verification results of a static analysis to prune verified executions from testing. However, existing combinations of

```
1  void Deposit(int amount) {
2    if (amount <= 0 || amount > 50000) {
3      ReviewDeposit(amount);
4    } else {
5      balance = balance + amount;
6      if (balance > 10000) {
7        SuggestInvestment();
8      }
9    }
10   assert balance >= old(balance);
11 }
```

Figure 2.1: A C# example illustrating partial verification results. Techniques that assume that the addition on line 5 does not overflow might miss violations of the assertion on line 10. We use the assertion to make the intended behavior explicit; the old-keyword indicates that an expression is evaluated in the pre-state of the method. `balance` is an integer field declared in the enclosing class. We assume methods `ReviewDeposit` and `SuggestInvestment` to be correct.

static analysis and test case generation do not support analyses that make unsound assumptions. They either require the static analysis to be sound and are, thus, of limited use for most practical analyses, or they ignore the unsoundness of the static analysis and may, therefore, prune executions during DSE that contain errors. In particular, they would miss the error in the example from Fig. 2.1 because the unsound static analysis reports no errors.

In this chapter, we present a novel technique to complement partial verification by automatic test case generation. In contrast to existing work, our technique supports the important case that the verification results are obtained by an unsound (manual or automatic) static code analysis. We use program annotations to make explicit which assertions in a program have already been verified, and under which assumptions. These annotations can be generated automatically by a static analysis [32] or inserted manually, for instance, during a code review. The main technical contribution of this chapter is a code instrumentation of the unit under test that:

- detects redundant test cases early during their execution and aborts them,
- reduces the search space for DSE by pruning paths that have been previously verified, and
- prioritizes test cases that cover unverified executions.

This instrumentation is based on an efficient static inference, which propagates information that characterizes unverified executions higher up in the control flow, where it may prune the search space more effectively. It does not require a specific DSE algorithm and, thus, can be used with a wide range of existing tools.

Our technique works for modular and whole-program verification, and can be used to generate unit or system tests. For concreteness, we present it for modular verification and unit testing. In particular, we have implemented our approach for Microsoft's .NET static checker Clousot [59], a modular static analysis, and the DSE tool Pex [135], a test case generator for unit tests. Our experiments demonstrate that, compared to classical DSE, our approach produces smaller test suites, explores more unverified executions, and reduces testing time.

Outline. This chapter is organized as follows. We give an overview of our approach in Sect. 2.1. Sect. 2.2 explains how we infer the code instrumentation from partial verification results. Our experimental results are presented in Sect. 2.3. We discuss related work in Sect. 2.4.

2.1 Approach

In this section, we present an annotation language for expressing partial verification results, and then illustrate how we use these annotations to guide DSE toward unverified executions. The details of the approach are explained in the next section.

2.1.1 Verification annotations

In order to encode partial verification results, we introduce two kinds of annotations: An *assumed-statement* of the form `assumed` P `as` a expresses that an analysis assumed property P to hold at this point in the code without checking it. The *assumption identifier* a uniquely identifies this statement. In order to record verification results, we use assertions of the form `assert` P `verified` A, which express that property P has been verified under condition A. The *premise* A is a boolean condition over assumption identifiers, each of which is introduced in an assumed-statement. Specifically, it is the conjunction of the identifiers for the assumptions used to verify P, or false if P was not verified. When several verification results are combined (for instance, from a static analysis and a code review), A is the disjunction of the assumptions made during each individual verification. We record verification results for all assertions in the code, including implicit assertions such as a receiver being non-null or an index being within the bounds of an array.

We assume here that a static analyzer records the assumptions it made during the analysis, which assertions it verified, and under which assump-

```
1  void Deposit(int amount) {
2    var a = true;
3    if (amount <= 0 || 50000 < amount) {
4      assume !a;
5      ReviewDeposit(amount);
6    } else {
7      assumed noOverflowAdd(balance, amount) as a;
8      a = a && noOverflowAdd(balance, amount);
9      assume !a;
10     balance = balance + amount;
11     if (10000 < balance) {
12       SuggestInvestment();
13     }
14   }
15   assume !a || balance >= old(balance);
16   assert balance >= old(balance) verified a;
17 }
```

Figure 2.2: The instrumented version of method `Deposit` from Fig. 2.1. The dark boxes show the annotations generated by the static analyzer. The assumed-statement makes explicit that the analyzer assumed that the addition on line 10 does not overflow. The verified-annotation on the assertion on line 16 expresses that the assertion was verified under this (unsound) assumption. The two annotations are connected via the assumption identifier `a`, which uniquely identifies the assumed-statement. The light boxes show the instrumentation that we infer from the annotations and that prunes redundant tests.

tions. We equipped Microsoft's .NET static analyzer Clousot [59] with this functionality [32]. Among other unsound assumptions, Clousot ignores arithmetic overflow and, thus, misses the potential violation of the assertion on line 10 of Fig. 2.1[1]. This partial verification result is expressed by the annotations in the dark boxes of Fig. 2.2 (the light boxes are discussed below). The assumed-statement makes explicit that the addition on line 10 was assumed not to overflow (the predicate `noOverflowAdd` can be encoded as equality of an integer and a long-integer addition); the verified-annotation on the assertion on line 16 expresses that the assertion was verified under this (unsound) assumption.

The meaning of verification annotations can be defined in terms of assign-

[1] Clousot is modular, that is, reasons about a method call using the method's pre- and postcondition; we assume here that the postconditions of `ReviewDeposit` and `SuggestInvestment` state that `balance` is not decreased.

ments and assume-statements, which makes the annotations easy to support by a wide range of static and dynamic tools. For each assumption identifier, we declare a boolean variable, which is initialized to true. For modular analyses, assumption identifiers are local variables initialized at the beginning of the enclosing method (line 2 in Fig. 2.2), whereas for whole-program analyses, assumption identifiers are global variables, for instance, initialized at the beginning of a main method. A statement **assumed** P **as** a is encoded as

$$a \texttt{ = } a \texttt{ \&\& } P;$$

as illustrated on line 8. That is, variable a accumulates the assumed properties for each execution of the assumed-statement. Since assumptions typically depend on the current execution state, this encoding ensures that an assumption is evaluated in the state in which it is made rather than the state in which it is used.

An assertion **assert** P **verified** A is encoded as

$$\begin{aligned} &\textbf{assume } A \Rightarrow P; \\ &\textbf{assert } P; \end{aligned}$$

as illustrated on line 15. The assume-statement expresses that, if condition A holds, then the asserted property P holds as well, which reflects that P was verified under the premise A. Consequently, an assertion is unverified if A is false, the assertion is fully verified if A is true, and otherwise, the assertion is partially verified.

2.1.2 Guiding dynamic symbolic execution

To reduce redundancies with prior analyses of the unit under test, DSE should generate tests that check each assertion **assert** P **verified** A for the case that the premise A does not hold, because P has been verified to hold otherwise. DSE can be guided by adding constraints to path conditions, which will then be satisfied by the generated test inputs. The assume-statement in the encoding of an assertion contributes such a constraint, reflecting that only inputs that violate the premise A may reveal a violation of the asserted property P. However, these assume-statements do not effectively guide DSE, as we explain next.

Assume-statements affect DSE in two ways. First, when the execution of a test case encounters an assume-statement whose condition is false, the execution is aborted. Second, when an execution encounters an assume-statement, its condition is added to the symbolic path condition, ensuring that subsequent test cases that share the prefix of the execution path up to the assume-statement, will satisfy the condition. (In other words, assume-statements contribute to the path constraint the assumed conditions, of the form Assume(c), that we discussed in Sect. 1.1.) Therefore, the effect of an

assume-statement is larger the earlier it occurs in the control flow, because early assumptions may abort test cases earlier and share the prefix with more executions.

However, the assume-statements we introduce for assertions do not effectively guide DSE toward unverified executions. Our example (Fig. 2.2) illustrates this problem. First, the conditions of these assume-statements always hold because they soundly express prior partial verification results. That is, they cannot be used to abort test cases—their condition is always true. Nonetheless, since these assume-statements express under which assumptions a subsequent assertion has been verified, they encode which executions are unverified in the context of the entire unit under test. For example, in Fig. 2.2, the assume-statement on line 15 encodes that executions of method `Deposit` for which `!a` holds are unverified. Even if we used this information to abort all test cases that reach line 15 and cover verified executions, we would save almost no execution time.

Second, even if we added this information to the path constraint when line 15 is executed, it would only influence the test cases that share the prefix up to that point, therefore, not including the majority of tests that share only a part of this prefix and target the branches of the conditional statements and called methods. Consequently, in the majority of times when DSE determines the next test case, it will most likely choose inputs that do not overflow, that is, test an execution that has already been verified, since the information about unverified executions is not yet in the path constraint. For these reasons, running DSE on the example from Fig. 2.2 (without lines 4 and 9, which we discuss below) generates the same test cases as if there were no prior verification results.

To address this problem, we propagate constraints that characterize unverified executions higher up in the control flow, where they can be used to effectively prune redundant test cases and to prioritize non-redundant test cases, that is, tests that cover unverified executions.

A test is *redundant* if the premise of each assertion in its execution holds; in this case, all assertions have been verified. In order to detect redundant tests early, we compute, for each program point, a sufficient condition for every execution from this program point onward to be verified. If this condition holds, we can abort the execution. We achieve this behavior by instrumenting the unit under test with assume-statements for the negation of the condition, that is, we assume a necessary condition for the existence of at least one unverified execution from the assume-statement onward. When the assumption evaluates to false during the execution of a test, it aborts the test and introduces a constraint, which implies that at least one premise must be violated, for all other test cases with the same prefix.

The example in Fig. 2.2 has an assertion with premise `a` at the very end. Consider the program points on lines 4 and 9. At both points, `a` is a sufficient condition for the rest of the execution of `Deposit` to be verified.

Since we are interested in test cases that lead to unverified executions, we instrument both program points by assuming the negation, that is, `!a`. With this instrumentation, any test case that enters the outer then-branch is aborted since `a` is always true at this point, which, in particular, prunes the entire exploration of method `ReviewDeposit`. Similarly, any test case that does not lead to an overflow on line 10 is aborted on line 9, which prunes the entire exploration of method `SuggestInvestment`. So, out of all the test cases generated by DSE for the un-instrumented `Deposit` method, only the one that reveals the error remains; all others are either aborted early or pruned.

Since the goal of the instrumentation described so far is to abort or prune redundant test cases, it has to be conservative. Any execution that *may* be unverified cannot be eliminated without potentially missing bugs; hence, we call this instrumentation *may-unverified instrumentation*. If an execution path contains several assertions, which is common because of the implicit assertions for dereferencing, array access, etc., this instrumentation retains any execution in which the premise of *at least one* of these assertions does not hold.

Intuitively, test cases that violate the premise of more than one assertion have a higher chance to detect an assertion violation. To prioritize such test cases, we devise a second instrumentation, called *must-unverified instrumentation*: We compute, for each program point, a sufficient condition for every execution from this program point onward to be *definitely* unverified. If the condition holds, then every execution from the program point onward contains at least one assertion, and the premises of *all* assertions in the execution are false.

When the must-unverified condition is violated, it does not necessarily mean that the subsequent execution is verified and, thus, we cannot abort the test case. Therefore, we instrument the program not by assuming the must-unverified condition, but instead, with a dedicated *tryfirst-statement*. This statement interrupts the execution of the test case and instructs DSE to generate new inputs that satisfy the must-unverified condition, that is, inputs that have a higher chance to detect an assertion violation. The interrupted test case is re-generated later, after the executions that satisfy the must-unverified condition have been explored. This exploration strategy prioritizes test cases that violate all premises over those that violate only some.

Suppose that the `Deposit` method in Fig. 2.2 contained another assertion at the very end that has not been verified, that is, whose premise is false. In this case, the may-unverified instrumentation yields true for all prior program points, since every execution is unverified. In this case, this instrumentation neither aborts nor prunes any test cases. In contrast, the must-unverified instrumentation infers `!a` on line 9. The corresponding tryfirst-statement (not shown in Fig. 2.2) gives priority to executions that

lead to an overflow on line 10. However, it does not prune the others, since they might detect a violation of the unverified second assertion at the end of the method.

The may-unverified and must-unverified instrumentations have complementary strengths. While the former effectively aborts or prunes redundant tests, the latter prioritizes those tests among the non-redundant ones that are more likely to detect an assertion violation. Therefore, our experiments show the best results for the combination of both.

2.2 Condition inference

Our may- and must-unverified conditions reflect whether the premises of assertions further down in the control flow hold. In that sense, they resemble weakest preconditions [49]: a may-unverified condition is the negation of the weakest condition that implies that all premises further down hold; a must-unverified condition is the weakest condition that implies that all premises do not hold. However, existing techniques for precisely computing weakest preconditions have shortcomings that make them unsuitable in our context. For instance, weakest precondition calculi [95] require loop invariants, abstract interpretation [40] may require expensive fixed-point computations in sophisticated abstract domains, predicate abstraction [77, 6] may require numerous invocations of a theorem prover for deriving boolean programs [7] from the original program, and symbolic execution [87] struggles with path explosion, for instance, in the presence of input-dependent loops.

In this section, we present two efficient instrumentations that *over-approximate* may-unverified and must-unverified conditions of a unit under test. For this purpose, we syntactically compute a non-deterministic abstraction of the unit under test. This abstraction is sound, that is, each execution of the concrete program is included in the set of executions of the abstract program. Therefore, a condition that guarantees that all premises hold (or are violated) in the abstract program provides the same guarantee for the concrete program. The may-unverified and must-unverified conditions for the abstract program can be computed efficiently using abstract interpretation, and can then be used to instrument the concrete program.

2.2.1 Abstraction

We abstract a concrete to a boolean program [7], where all boolean variables are assumption identifiers. In the abstract program, all expressions that do not include assumption identifiers are replaced by non-deterministically chosen values, which, in particular, replaces conditional control flow by non-determinism. Moreover, the abstraction removes assertions that have been fully verified, that is, where the premise is the literal `true` or includes `true` as a disjunct.

We present the abstraction for a simple *concrete* programming language with the following statements: assumed-statements, assertions, method calls, conditionals, loops, and assignments. Besides conditional statements and loops with non-deterministic guards, the *abstract* language provides the following statements:

- initialization of assumption identifiers: `var a := true`,
- updates to assumption identifiers: `a := a && *`, where the `*` denotes a non-deterministic (boolean) value,
- assertions: `assert * verified` A, where $A \not\equiv$ `true`, and
- method calls: `call` M_f, where M_f is a fully-qualified method name and the receiver and arguments have been abstracted away.

Note that we desugar assumed-statements into initializations and updates of assumption identifiers, which allows us to treat modular and whole-program analyses uniformly, even though they require a different encoding of assumed-statements (Sect. 2.1.1).

To abstract a program, we recursively apply the following transformations to its statements. These transformations can be considered as an application of predicate abstraction [77], which uses the assumption identifiers as predicates and does not rely on a theorem prover to derive the boolean program:

- an assumption `assumed` P `as a` is rewritten to the assumption identifier initialization `var a := true` (at the appropriate program point, as discussed above) and an update `a := a && *`,
- an assertion `assert` P `verified` A is rewritten to the assertion `assert * verified` A, if A is not trivially `true`, and omitted otherwise,
- a conditional statement `if (`b`)` S_0 `else` S_1 is transformed into `if (*)` S_0' `else` S_1', where S_0' and S_1' are the results of recursively rewriting the statements S_0 and S_1, respectively,
- a loop `while (`b`)` S is transformed into `while (*)` S', where S' is the result of recursively rewriting statement S,
- a method call $r.M$`(...)` is rewritten to `call` M_f, where M_f is the fully-qualified name of M, and
- assignments are omitted.

Fig. 2.3 shows the abstraction of method `Deposit` from Fig. 2.2. The gray boxes (light and dark) show the inferred may-unverified conditions, as we explain in the next subsection.

```
1  method Deposit() {
2    {true}
3    var a := true;
4    {true}
5    if (*) {
6      {!a}
7      call Account.ReviewDeposit;
8      {!a}
9    } else {
10     {true}
11     a := a && *;
12     {!a}
13     if (*) {
14       {!a}
15       call Account.SuggestInvestment;
16       {!a}
17     }
18     {!a}
19   }
20   {!a}
21   assert * verified a;
22   {false}
23 }
```

Figure 2.3: The abstraction of method `Deposit` from Fig. 2.2. The gray boxes (light and dark) show the inferred may-unverified conditions. The conditions that are used for the may-unverified instrumentation are shown in dark gray boxes.

Soundness

The abstraction described above is sound, that is, each execution of the concrete program is included in the set of executions of the corresponding abstract program. The abstraction preserves the control structure of each method, but makes the control flow non-deterministic, which enlarges the set of possible executions. All other occurrences of expressions (in assumed-statements, assertions, and calls) are replaced by non-deterministic values of the appropriate type, which also enlarges the set of possible executions. Once all occurrences of variables have been replaced by non-deterministic values, assignments do not affect program execution and can, thus, be omitted.

2.2.2 May-unverified conditions

A may-unverified condition expresses that some execution from the current program point onward may be unverified. We compute this condition for each program point in two steps. First, we compute the weakest condition at the corresponding program point in the abstract program that implies that all executions are verified. Since the set of executions of the abstract program subsumes the set of concrete executions, this condition also implies that all concrete executions are verified (although for the concrete execution, the computed condition is not necessarily the weakest such condition). Second, we negate the computed condition to obtain a may-unverified condition.

Inference

To compute the weakest condition that implies that all executions from a program point onward are verified, we define a predicate transformer WP on abstract programs. If $WP(S, R)$ holds in a state, then the premise of each assertion in each execution of statement S from that state holds and, if the execution terminates, R holds in the final state. For a modular analysis such as Clousot, calls are encoded via their pre- and postcondition [96] and, thus, do not occur in the abstract program. Defining an inter-procedural WP is, of course, also possible. Thus, we define WP as:

- $WP(\texttt{assert * verified}\ A, R) \equiv A \wedge R$,
- $WP(\texttt{a} := \texttt{true}, R) \equiv R[\texttt{a} := \texttt{true}]$, denoting the substitution of `a` by `true` in R, and
- $WP(\texttt{a} := \texttt{a \&\& *}, R) \equiv R \wedge R[\texttt{a} := \texttt{false}]$.

The semantics of sequential composition, conditionals, and loops is standard [49]. In our implementation, we use backward abstract interpretation to compute the weakest precondition for each program point in terms of a set of cubes (that is, conjunctions of assumption identifiers or their negations). In the presence of loops or recursion, we use a fixed-point computation.

For every program point of the abstract program, the may-unverified condition is the negation of the weakest precondition at that program point

$$\textsc{May}(S) \equiv \neg WP(S, \mathit{true})$$

where S denotes the code fragment after the program point.

The gray boxes in Fig. 2.3 show the may-unverified conditions at each program point (assuming `ReviewDeposit` and `SuggestInvestment` have no preconditions). In the example, the may-unverified inference propagates meaningful information only up until the non-deterministic update

is reached, which corresponds to the assumed-statement. Specifically, on line 10, we infer true because the abstraction loses the information that would be needed to compute a stronger may-unverified condition. So, in return for an efficient condition inference, we miss some opportunities for aborting and pruning redundant tests.

Instrumentation

Since each execution of the concrete program corresponds to an execution of the abstract program, we can instrument the concrete program by adding an `assume` C statement at each program point, where C is the may-unverified condition at the corresponding program point in the abstract program. As we explained in Sect. 2.1.2, these statements abort redundant test cases, and contribute constraints that guide DSE toward unverified executions.

To avoid redundant constraints that would slow down DSE, we omit assume-statements when the may-unverified condition is trivially true or not different from the condition at the previous program point, as well as the `assume false` statement at the end of the unit under test. Therefore, out of all the conditions inferred for the example in Fig. 2.3, we use only the ones on lines 6 and 12 to instrument the program, which leads to the assumptions on lines 4 and 9 of Fig. 2.2 and guides DSE as described in Sect. 2.1.2.

2.2.3 Must-unverified conditions

A must-unverified condition expresses that (1) each execution from the program point onward contains at least one assertion, and (2) on each execution, the premise of each assertion evaluates to false.

Inference

We infer the two properties that are entailed by a must-unverified condition separately, via two predicate transformers $\textsc{Must}_{assert}$ and $\textsc{Must}_{all}$. If $\textsc{Must}_{assert}(S, R)$ holds in a state, then each execution of statement S from that state encounters at least one assertion, or terminates in a state in which R holds. If $\textsc{Must}_{all}(S, R)$ holds in a state, then the premise of each assertion in each execution of statement S from that state does not hold and, if S terminates, R holds. Both transformers yield the weakest condition that has these properties. Consequently, we obtain the weakest must-unverified condition for an abstract statement S as follows:

$$\textsc{Must}(S) \equiv \textsc{Must}_{assert}(S, \mathit{false}) \wedge \textsc{Must}_{all}(S, \mathit{true})$$

$\textsc{Must}_{assert}$ and $\textsc{Must}_{all}$ are defined analogously to WP (see Sect. 2.2.2), except for the treatment of assertions:

$$\textsc{Must}_{assert}(\texttt{assert * verified}\ A, R) \equiv \mathit{true}$$

$$\textsc{Must}_{all}(\texttt{assert * verified}\ A, R) \equiv \neg A \wedge R$$

The definition for $\textsc{Must}_{assert}$ expresses that, at a program point before an assertion, property (1) holds, that is, the remaining execution (from that point on) contains at least one assertion. The definition for $\textsc{Must}_{all}$ expresses that the premise A must evaluate to false, and that R must hold to ensure that the premises of subsequent assertions do not hold either.

Fig. 2.4 shows the abstraction of a variant of `Deposit` from Fig. 2.2 that contains an additional unverified assertion at the end of the method (see Sect. 2.1.2). The gray boxes show the inferred must-unverified conditions, as we explain next. Compared to the may-unverified conditions, the must-unverified conditions are stronger, that is, information is usually propagated further up in the control flow. Whereas the unverified assertion at the end of this example causes the may-unverified conditions to be trivially true, the must-unverified inference obtains conditions that can be used to prioritize test cases.

Instrumentation

To prioritize tests that satisfy their must-unverified conditions, we instrument the concrete program with `tryfirst` C statements, where C is the must-unverified condition at the corresponding program point in the abstract program. This statement causes DSE to prefer test inputs that satisfy condition C. More specifically, when a `tryfirst` C statement is executed for the first time, it adds C to the path condition to force DSE to generate inputs that satisfy condition C. Note, however, that unlike the constraints added by assume-statements, this constraint may be dropped by the DSE to also explore executions where the condition is violated. If during this first execution of the statement condition C is violated, then the test case is interrupted and will be re-generated later, when condition C can no longer be satisfied. So, the tryfirst-statement influences the *order* in which test cases are generated, but never aborts or prunes tests. Nevertheless, the order is important, because DSE is typically applied until certain limits (for instance, on the overall testing time or the number of test cases) are reached. Therefore, exploring non-redundant test cases early increases effectiveness.

Pex supports primitives for expressing `tryfirst` C statements easily, as instrumentation. Alternatively, other tools may encode them by placing additional branches into the code and customizing the search strategy to prefer the branch where C holds.

To avoid wasting time on interrupting tests that will be re-generated later, our implementation enforces an upper bound on the number of interrupts that are allowed per unit under test. When this upper bound is exceeded, all remaining tryfirst-statements have no effect.

As illustrated by lines 4, 6, 8, and 10 in Fig. 2.4, the must-unverified condition at some program points evaluates to false for all executions. Instru-

```
1  method Deposit() {
2    {false}
3    var a := true;
4    {!a}
5    if (*) {
6      {!a}
7      call Account.ReviewDeposit;
8      {!a}
9    } else {
10     {!a}
11     a := a && *;
12     {!a}
13     if (*) {
14       {!a}
15       call Account.SuggestInvestment;
16       {!a}
17     }
18     {!a}
19   }
20   {!a}
21   assert * verified a;
22   {true}
23   assert * verified false;
24   {false}
25 }
```

Figure 2.4: The abstraction of a variant of method `Deposit` from Fig. 2.2 that contains an additional unverified assertion at the end of the method (see Sect. 2.1.2). The gray boxes show the inferred must-unverified conditions. The conditions that are used for the must-unverified instrumentation are shown in dark gray boxes.

menting these points would lead to useless interruption and re-generation of tests. To detect such cases, we apply constant propagation, and do not instrument program points for which the must-unverified conditions are trivially true or false. Moreover, we omit the instrumentation for conditions that are not different from the condition at the previous program point. Therefore, out of all the conditions inferred for the example in Fig. 2.4, we use only the ones on lines 12 and 20 to instrument the program, which prioritize tests that lead to an arithmetic overflow on line 10, as discussed in Sect. 2.1.2.

2.2.4 Combined instrumentation

As we explained in Sect. 2.1.2, the may-unverified instrumentation aborts and prunes redundant tests, while the must-unverified instrumentation prioritizes tests that are more likely to detect an assertion violation. One can, therefore, combine both instrumentations such that DSE (1) attempts to first explore program executions that must be unverified, and (2) falls back on executions that may be unverified when the former is no longer feasible.

The combined instrumentation includes the assume-statements from the may-unverified instrumentation as well as the tryfirst-statements from the must-unverified instrumentation. The tryfirst-statement comes first. Whenever we can determine that the must-unverified and may-unverified conditions at a particular program point are equivalent, we omit the tryfirst-statement, because any interrupted and re-generated test case would be aborted by the subsequent assume-statement anyway.

2.3 Experimental evaluation

In this section, we give an overview of our implementation and present our experimental results. In particular, we show that, compared to dynamic symbolic execution alone, our technique produces smaller test suites, covers more unverified executions, and reduces testing time. We also show which of our instrumentations—may-unverified, must-unverified, or their combination—is the most effective.

2.3.1 Implementation

We have implemented our technique for the .NET static analyzer Clousot [59] and the dynamic symbolic execution tool Pex [135]. Our tool chain consists of four subsequent stages:

1. static analysis and verification-annotation instrumentation,
2. may-unverified and must-unverified instrumentation,
3. runtime checking, and
4. dynamic symbolic execution.

The first stage runs Clousot on a given .NET program, which contains code and optionally specifications expressed in Code Contracts [58], and instruments the sources of unsoundness and verification results of the analyzer, using our verification annotations. For this purpose, we have implemented a wrapper around Clousot, which we call *Inspector-Clousot*, that uses the debug output emitted during the static analysis to instrument the program

(at the binary level). Note that Clousot performs a modular analysis and, thus, the verification annotations are local to the containing methods.

We have elicited a complete list of Clousot's unsound assumptions [32] during two years of work, by studying publications, extensively testing the tool, and having numerous discussions with its designers. We have demonstrated how most of these assumptions can be expressed with our verification annotations. In fact, we have shown that our verification annotations can describe unsound assumptions about sequential programs, expressible by contract languages [58, 93], and typically made by abstract interpreters and deductive verifiers [32, 30]. (An extension to concurrent programs, more advanced properties such as temporal properties, and the unsound assumptions made by model checkers such as bounding the number of heap objects are future work.)

The second stage of the tool chain adds the may-, must-unverified instrumentation, or their combination to the annotated program.

In the third stage, we run the existing Code Contracts binary rewriter to transform any Code Contracts specifications into runtime checks. We then run a second rewriter, which we call *Runtime-Checking-Rewriter*, that transforms all the assumed-statements and assertions of the annotated program into assignments and assumptions, as described in Sect. 2.1.1.

In the final stage, we run Pex on the instrumented code.

2.3.2 Experiments

In the rest of this section, we describe the setup for the evaluation of our technique, and present experiments that evaluate its benefits.

Setup

For our experiments, we used 101 methods (written in C#) from nine open-source projects and from solutions to 13 programming tasks on the Rosetta Code repository. A complete list of the methods used in our evaluation can be found in Appx. A. We selected only methods for which Pex can automatically produce more than one test case (that is, Pex does not require user-provided factories) and at least one successful test case (that is, Pex generates inputs that pass any input validation performed by the method).

In Clousot, we enabled all checks, set the warning level to the maximum, and disabled all inference options. In Pex, we set the maximum number of branches, conditions, and execution tree nodes to 100,000, and the maximum number of concrete runs to 30.

In our experiments, we allowed up to four test interrupts per method under test when these are caused by tryfirst-statements (see Sect. 2.2.3). We experimented with different such bounds (one, two, four, and eight) on 25 methods from the suite of 101 methods. We found that, for an upper

bound of four for the number of allowed interrupts per method, dynamic symbolic execution strikes a good balance between testing time and the number of detected bugs.

We used a machine with a quad-core CPU (Intel Core i7-4770, 3.4 GHz) and 16 GB of RAM for these experiments.

Performance of static analysis and instrumentation

On average, Clousot analyzes each method from our suite in 1.9 seconds. The may-unverified and must-unverified instrumentations are very efficient. On average, they need 22 milliseconds per method, when combined.

Configurations

To evaluate our technique, we use the following configurations:

- UV: *unverified* code.
  Stages 1 and 2 of the tool chain are not run.
- PV: *partially-verified* code.
  Stage 2 of the tool chain is not run.
- MAY: partially-verified code, instrumented with *may*-unverified conditions.
  All stages of the tool chain are run. Stage 2 adds only the may-unverified instrumentation.
- $MUST$: partially-verified code, instrumented with *must*-unverified conditions.
  All stages of the tool chain are run. Stage 2 adds only the must-unverified instrumentation.
- $MAY \times MUST$: partially-verified code, instrumented both with *may*-unverified and *must*-unverified conditions.
  All stages of the tool chain are run. Stage 2 adds the combined may-unverified and must-unverified instrumentation.

Fig. 2.5 shows the tests that each configuration generated for the 101 methods, categorized as non-redundant and failing, or non-redundant and successful, or redundant tests. A failing test terminates abnormally, whereas a successful one terminates normally. However, tests that terminate on exceptions that are explicitly thrown by the method under test, for instance, for parameter validation, are not considered failing. To determine the redundant tests generated by configurations UV, PV, and $MUST$, we ran all tests generated by these configurations against the 101 methods, after having instrumented the methods with the may-unverified conditions. We then counted how many of these tests were aborted. Note that the figure does not

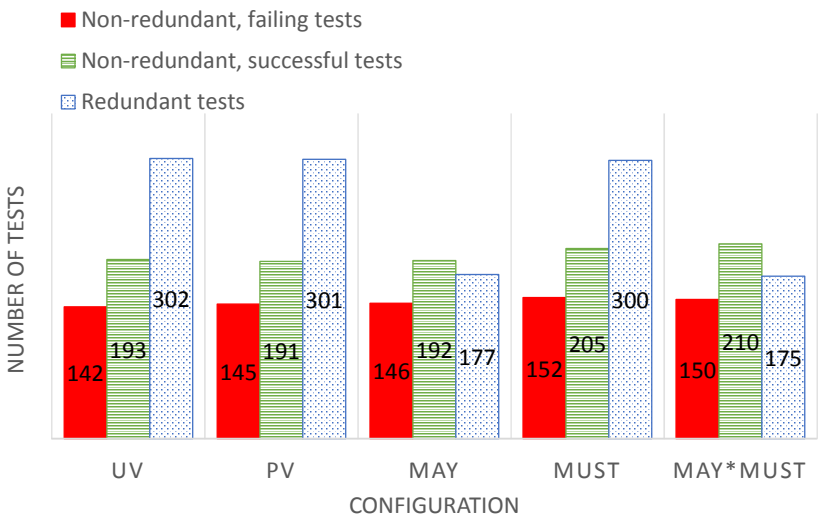

Figure 2.5: The tests generated by each configuration, categorized as non-redundant and failing, or non-redundant and successful, or redundant tests. $MAY \times MUST$ generates 16.1% fewer tests, but 7.1% more non-redundant tests than PV, including five additional failing tests.

include tests that are interrupted because a condition in a tryfirst-statement is violated (since these tests are re-generated—and counted—later).

The results of dynamic symbolic execution alone, of UV, do not significantly differ from those of PV in terms of the total number of tests and the number of non-redundant tests generated for the 101 methods. This confirms that the instrumentation from stage 1 alone, without the may-unverified and must-unverified instrumentation, does not reduce the test effort significantly for partially-verified methods, as we explained in Sect. 2.1.2. For the following experiments, we use configuration PV as the baseline to highlight the benefits of the may-unverified and must-unverified inference.

Smaller test suites

The may-unverified instrumentation causes DSE to abort tests leading to verified executions. By aborting these tests, our technique prunes the verified parts of the search space that would be explored only if these tests were not aborted. As a result, DSE generates smaller test suites.

Fig. 2.5 shows that, in total, MAY generates 19.2% fewer tests and $MAY \times MUST$ generates 16.1% fewer tests than PV. The differences in the total number of tests for configurations without the may-unverified instrumentation are minor.

Fig. 2.6 compares the total number of generated tests (including aborted

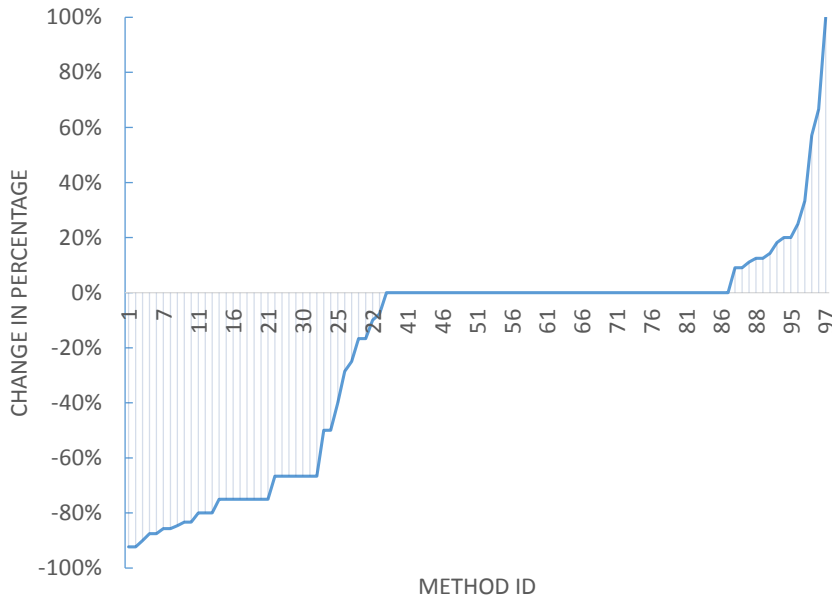

Figure 2.6: Change in total number of tests generated for each of the 101 methods by configuration *MAY* in comparison to *PV* (in percentage). Negative values indicate that *MAY* produces fewer tests.

tests) by *PV* and *MAY* *per method*. For many methods, *MAY* produces fewer tests, as shown by the negative values. However, for some methods, *MAY* generates more tests than *PV*. This happens when pruning verified parts of the search space guides DSE toward executions that are easier to cover within the exploration bounds of Pex (for instance, maximum number of branches or constraint solver timeouts).

More unverified executions

Even though configurations *MAY* and *MAY* × *MUST* generate smaller test suites in comparison to *PV*, they do not generate fewer non-redundant tests, as shown in Fig. 2.5. In other words, they generate at least as many non-redundant tests as *PV*, thus covering at least as many unverified executions.

The must-unverified instrumentation causes DSE to prioritize inputs that lead to unverified executions. In comparison to the may-unverified conditions, the must-unverified conditions are stronger, and their instrumentation is usually added further up in the control flow. As a result, *MUST* and *MAY* × *MUST* can guide dynamic symbolic execution to cover unverified executions earlier, and may allow it to generate more non-redundant tests within the exploration bounds. As shown in Fig. 2.5, configuration *MUST* generates 6.3% more non-redundant tests than *PV* and 5.6% more

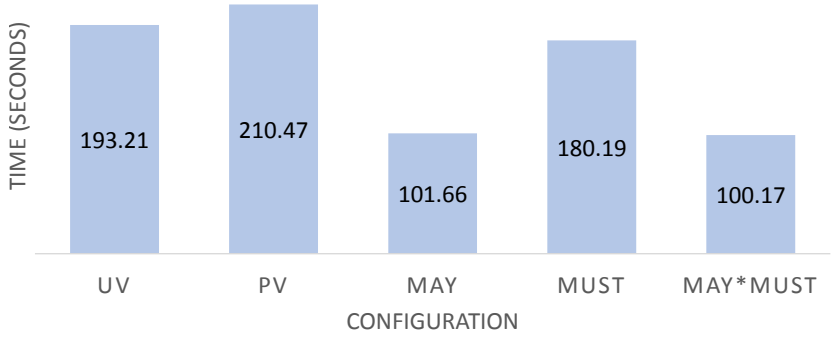

Figure 2.7: Testing time for each configuration. We only considered methods for which all configurations generated the same number of non-redundant tests. $MAY \times MUST$ is 52.4% faster than PV.

than MAY ($MAY \times MUST$ produces 7.1% resp. 6.5% more non-redundant tests). By generating more such tests, we increase the chances of producing more failing tests. In fact, $MUST$ generates 4.8% more failing tests than PV and 4.1% more than MAY ($MAY \times MUST$ produces 3.4% resp. 2.7% more failing tests).

$MUST$ typically generates more non-redundant tests for methods in which Clousot detects errors, that is, for methods with unverified assertions. In such methods, the may-unverified instrumentation is added only after the unverified assertions in the control flow (if the conditions are non-trivial), thus failing to guide dynamic symbolic execution toward unverified executions early on, as discussed in Sect. 2.1.2.

Shorter testing time

We now compare the testing time of the different configurations. For this experiment, we considered only methods for which all configurations generated the same number of non-redundant tests. This is to ensure a fair comparison; for these methods, all configurations achieved the same coverage of unverified executions. This experiment involved 72 out of the 101 methods, and the time it took for each configuration to test these methods is shown in Fig. 2.7. As expected, pruning verified parts of the search space with the may-unverified instrumentation is very effective. In particular, configuration MAY is 51.7% faster and configuration $MAY \times MUST$ is 52.4% faster than PV. Note that Fig. 2.7 does not include the time of the static analysis for two reasons. First, Clousot is just one way of obtaining verification results. Second, the goal of our work is to efficiently complement *existing* verification results with test case generation; we assume that the static analysis is run anyway to achieve a more thorough scrutiny of the

code. Recall that the overhead of the instrumentation is negligible. The differences in performance between configurations without the may-unverified instrumentation are less pronounced.

Even though MAY is overall much faster than PV, there were methods for which the testing time for MAY was longer in comparison to PV. This is the case when constraint solving becomes more difficult due to the inferred conditions. In particular, it might take longer for the constraint solver to prove that an inferred condition at a certain program point is infeasible.

Fewer exploration bounds reached

During its exploration, dynamic symbolic execution may reach bounds that prevent it from covering certain, possibly failing, execution paths. There are four kinds of bounds that were reached during our experiments:

- *max-branches*: maximum number of branches that may be taken along a single execution path;
- *max-stack*: maximum size of the stack, in number of active call frames, at any time during a single execution path;
- *max-runs*: maximum number of runs that will be tried during an exploration (each run uses different inputs, but some runs are not added to the test suite if they do not increase coverage);
- *max-solver-time*: maximum number of seconds that the constraint solver has to find inputs that will cause an execution path to be taken.

Fig. 2.8 shows the exploration bounds in Pex that were reached by each configuration when testing all 101 methods. MAY, $MUST$, and $MAY \times MUST$ reach the max-solver-time bound more often than PV. This is because our instrumentation introduces additional conjuncts in the path conditions, occasionally making constraint solving harder. Nevertheless, configurations MAY and $MAY \times MUST$ overall reach significantly fewer bounds than PV (for instance, the max-stack bound is never reached) by pruning verified parts of the search space. This helps in alleviating an inherent limitation of dynamic symbolic execution by building on results from tools that do not suffer from the same limitation.

Winner configuration

As shown in Fig. 2.5, configuration $MAY \times MUST$ generates the second smallest test suite, containing the largest number of non-redundant tests and the smallest number of redundant tests. This is achieved in the shortest amount of testing time for methods with the same coverage of unverified executions across all configurations (Fig. 2.7), and by reaching the smallest number of exploration bounds (Fig. 2.8).

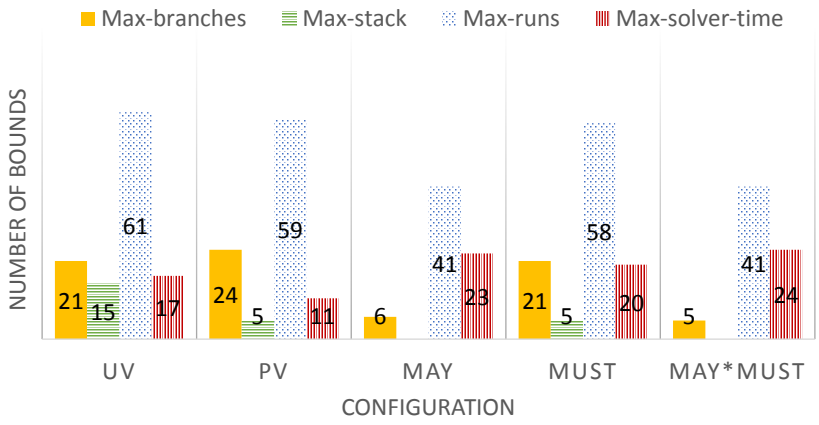

Figure 2.8: The exploration bounds reached by each configuration, grouped into max-branches, max-stack, max-runs, and max-solver-time. *MAY* **and** *MAY* × *MUST* **overall reach fewer bounds than** *PV***.**

Therefore, configuration $MAY \times MUST$ effectively combines the benefits of both the may-unverified and must-unverified instrumentation, to prune parts of the search space that lead only to verified executions as well as to identify and prefer test inputs that lead to unverified executions as soon as possible.

Note that, in practice, these benefits should be independent of the exploration strategy in the underlying dynamic symbolic execution. For methods whose exploration does not reach any bounds, the order in which the tests are generated is obviously not relevant. For the remaining methods, we do not expect an exploration strategy to significantly affect how often our instrumentation is hit because Clousot makes unsound assumptions for various expressions and statements and, thus, assumed-statements are spread across the method body. We have confirmed this expectation by running the $MAY \times MUST$ configuration with different exploration strategies on 20 out of the 101 methods for which exploration bounds were reached. The differences between all strategies (breadth-first, random search, and Pex's default search strategy) were negligible.

Soundness bugs in Clousot

During our experiments, we realized that our verification annotations can also be used to systematically test for soundness issues in static analyzers [45]. This is achieved as follows. Given a piece of code, imagine that configuration *UV* generates a number of failing tests. Now, we instrument the code with the known unsound assumptions made by a static analyzer

and its verification results (stage 1 of the tool chain). We detect a soundness issue if, when running the failing tests against the instrumented code, at least one failing test runs through an assertion `assert` P `verified` A, where $A \not\Rightarrow P$. A soundness issue could either be caused by accidental unsoundness (that is, bugs in the implementation of the analyzer) or by bugs in *Inspector-Clousot* (for instance, missing a source of deliberate unsoundness).

In this way, we found the following three bugs in the implementation of Clousot: (1) unsigned integral types are not always treated correctly, (2) the size of each dimension in a multi-dimensional array is not checked to be non-negative upon construction of the array, and (3) arithmetic overflow is ignored in modulo operations (for instance, `MinValue % -1`). We reported these three bugs to the main developer of Clousot, Francesco Logozzo, who confirmed all of them. The latter two bugs have already been fixed in the latest version of the tool[2].

Threats to validity

We identified the following threats to the validity of our experiments:

- *Sample size*: For our experiments, we used 101 methods from nine C# projects and from solutions to 13 programming tasks.
- *Static analyzer*: For our experiments, we used a modular (as opposed to whole-program) static analyzer, namely, Clousot. Moreover, our experimental results depend on the deliberate sources of unsoundness and verification results of this particular analyzer. Note that there are a few sources of unsoundness in Clousot that our tool chain does not capture [32], for instance, about reflection or unmanaged code.
- *Soundly-analyzed methods*: 23 of the 101 methods contain no assumed-statements. In case Clousot reports no warning, these methods are fully verified and, thus, our may-unverified instrumentation prunes the entire search space. Other code bases could have a smaller fraction of fully-verified methods, leading to less effective pruning.
- *Failing tests*: The failing tests generated by each configuration do not necessarily reveal bugs in the containing methods. This is inherent to unit testing, since methods are tested in isolation rather than in the context of the entire program. However, 50 out of the 101 methods validate their parameters (and for ten methods no parameter validation was necessary), which suggests that programmers did intend to prevent failures in these methods.

[2] `https://github.com/Microsoft/CodeContracts` (revs: `803e34e72061b305c1cde37a886c682129f1ddeb` and `1a3c0fce9f8c761c3c9bb8346291969ed4285cf6`)

2.4 Related work

Many static analyzers that target mainstream programming languages deliberately make unsound assumptions in order to increase automation, improve performance, and reduce the number of false positives and the annotation overhead for the programmer [106]. Examples of such analyzers are HAVOC [5], Spec# [12], and ESC/Java [62]. Our technique can effectively complement these analyzers by dynamic symbolic execution.

Integration of static analysis and test case generation

Various approaches combine static analysis and automatic test case generation, to determine whether an error reported by the static analysis is spurious and to reduce the search space for the test case generator. For example, Check 'n' Crash [43] is an automated defect detection tool that integrates the unsound ESC/Java static checker with the JCrasher [42] testing tool. Check 'n' Crash was later integrated with Daikon [57] in the DSD-Crasher tool [44]. Similarly, DyTa [65] integrates Clousot with Pex. Like our work, all of these approaches use results from the static analysis to guide test case generation toward the errors reported by the static analysis and to prune parts of the search space during testing. However, in contrast to our work, they ignore the unsoundness of the static analysis: each assertion for which the static analysis does not report an error is considered *soundly* verified, even if the analysis makes unsound assumptions. Consequently, these approaches may prune unverified executions, whereas our technique retains all executions that are not fully verified and, therefore, may reveal errors missed by the unsound static analysis.

The SANTE tool [26] uses a sound value analysis (in combination with program slicing) to prune those execution paths that do not lead to unverified assertions. In contrast, our work supports the common case that a static analysis is unsound.

Several analyses combine over- and under-approximations of the set of program executions. Counterexample-guided abstraction refinement (CEGAR) [34] 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. YOGI [75, 121] switches between static analysis and DSE both to prove properties and find bugs, without reporting false positives. Specifically, YOGI uses two different abstract domains, one (not-)may abstraction for proving a property and one must abstraction for disproving a property. The two abstractions are used simultaneously, communicate with each other, and refine each other for either finding a proof or a bug.

To obtain an over-approximation of the set of program executions, these

approaches rely on a sound analysis. In contrast, our work supports the common case that a static analysis is unsound, that is, neither over- nor under-approximates the set of program executions (in other words, the analysis may have both false positives and false negatives). Soundly-verified executions and executions for which the analysis reports an error are handled similarly to work based on over-approximations: we prune soundly-verified executions during test case generation, and use an under-approximation (testing) to find bugs and identify spurious errors among the executions for which the analysis reports an error. The novelty of our work is that we also handle executions that are verified unsoundly, that is, under unsound assumptions. Our annotations make these assumptions explicit (in other words, they express which executions one would have to add to the set of analyzed executions for it to become a sound over-approximation). These executions are then targeted by an under-approximation.

A recent approach [46] starts by running a conditional model checker [14] on a program, and then tests those parts of the state space that were not covered by the model checker (for instance, due to timeouts). More specifically, the model checker produces an output condition, which captures the safe states and is used to produce a residual program that can be subsequently tested. Unlike an instrumented program in our technique, the residual program can be structurally very different from the original program. As a result, its construction can take a significant amount of time, as the authors point out. Furthermore, this approach can characterize assertions only as either fully verified or unverified on a given execution path. It is not clear how to apply this approach in a setting with static analysis tools that are not fully sound [106, 32] without reducing its effectiveness.

Dynamic symbolic execution

Testing and symbolically executing all feasible program paths is not possible in practice. The number of feasible paths can be exponential in the program size, or even infinite in the presence of input-dependent loops.

Existing testing tools based on dynamic symbolic execution alleviate path explosion using search strategies and heuristics, which guide the search toward interesting parts of the program while pruning the search space. These strategies typically optimize properties such as "deeper paths" (in depth-first search), "less-traveled paths" [105], "number of new instructions covered" (in breadth-first search), or "paths specified by the programmer" [129]. For instance, SAGE [73] uses a generational-search strategy with simple heuristics, such as flip count limits and constraint subsumption. Other industrial-strength tools, like Pex, also use similar techniques. As we explained in Sect. 2.3.2, the benefits of our approach are independent of the exploration strategy in the underlying dynamic symbolic execution. Our technique resembles a search strategy in that it optimizes unverified execu-

tions, prunes verified executions, and is guided by verification annotations, instead of properties like the above.

Compositional symbolic execution [66, 2] has been shown to alleviate path explosion. Dynamic state merging [89] and veritesting [4] alleviate path explosion by merging sub-program searches, while RWset [15] prunes searches by dynamically computing variable liveness. By guiding dynamic symbolic execution toward unverified program executions, our technique also alleviates path explosion. In particular, the may-unverified instrumentation causes dynamic symbolic execution to abort tests that lead to verified executions. When aborting these tests, our technique prunes the parts of the search space that would be discovered only if these tests were not aborted. Moreover, since our technique does not require a particular DSE algorithm, it can be combined with any of the above approaches by running them on a program that contains our instrumentation.

2.5 Summary and remarks

We have presented a technique for complementing partial verification results by automatic test case generation. Our technique causes dynamic symbolic execution to abort tests that lead to verified executions, consequently pruning parts of the search space, and to prioritize tests that lead to unverified executions. It is applicable to any program with verification annotations, either generated automatically by a (possibly unsound) static analysis or inserted manually, for instance, during a code review. Our work suggests a novel way to combine static analysis and testing in order to maximize software quality, and investigates to what extent unsound static analysis reduces the test effort.

Chapter 3

Synthesizing parameterized unit tests to detect object invariant violations

In Ch. 2, we showed how to guide automatic test case generation toward unverified program executions. By focusing the test effort on unverified parts of the search space, we realized that, similarly to many static analyzers, existing techniques for systematic testing are also unsound, that is, they also produce false negatives. This is the case not only when these techniques fail to exercise certain program executions, but also when the correctness of the exercised executions is evaluated against weak test oracles. In other words, even when a testing tool achieves full code coverage and generates no failing tests, errors might still be missed. In this chapter, we address the problem of weak test oracles with respect to object invariants, as we explain next.

Automatic testing techniques, such as random testing or symbolic execution, rely on a description of the input data that the unit under test (UUT) is intended to handle. Such a description acts as a filter for undesirable input data. It is usually expressed as code in the test driver or as a method precondition that specifies the valid arguments for the method under test. For instance, when testing a method for sorting an array, a test data generator might populate input arrays randomly, and the filter might suppress arrays containing null.

Filtering input data with code in test drivers or with preconditions poses two difficulties when the inputs are heap data structures. First, the same filters must typically be duplicated across many test cases, because many operations on such data structures rely on the same properties of the data structure, for instance, on a field being non-null or on a list being sorted. Second, filters for recursive heap data structures, such as the sortedness of a list, must be expressed with loops or recursion, which are difficult to handle in constraint-based testing approaches such as symbolic execution [74].

To address these difficulties, some test data generators use predicates that express which instances of a data structure are considered valid and should, thus, be used for unit testing. In an object-oriented setting, these predicates are often called *class* or *object invariants* [109, 101]. For example,

in the case of a sorted linked list, the invariant simply states that for each list node, the value of the next node is greater than or equal to its own. Object invariants avoid the shortcomings of other filtering techniques: they are stated only once for each class rather than in each test case or method precondition, and by constraining each instance of the class, they can express properties of recursive structures without introducing loops or recursion.

Invariants may be provided by the programmer in the form of contracts, like in the random testing tool AutoTest [110] for Eiffel and in the dynamic symbolic execution tool Pex [135] for .NET, or by the tester, like in the Korat [17] tool for Java, which exhaustively enumerates data structures that satisfy a given predicate up to a bound. Invariants may also be inferred by tools like the Daikon invariant detector [57], which is used by the symbolic execution tool Symbolic Java PathFinder [125] for obtaining input constraints on a UUT.

Using object invariants to generate test data requires the invariants to accurately describe the data structures a program may create. When an invariant is too weak, i.e., admits more data structures than the program may create, the test case generator may *produce undesirable* inputs, which are however easily detected when inspecting failing tests. A more severe problem occurs when an invariant is too strong, i.e., admits only a subset of the data structures the program might actually create. The test case generator may then *not produce desirable* inputs since they are filtered out due to the overly strong invariant. Consequently, the UUT is executed with a restricted set of inputs, which potentially fail to exercise certain execution paths and may miss bugs.

Too strong invariants occur, for instance, when programmers specify invariants they intend to maintain but fail to do so due to a bug, when they fail to capture all intended program behaviors in the invariant, or when invariants are inferred from program runs that do not exercise all relevant behaviors. Therefore, it is essential that invariants are not only used to filter test inputs, but are also *checked* as part of test oracles, just like method preconditions are checked at call sites. However, in contrast to method preconditions or other filters that are implemented in test drivers, checking object invariants is very difficult, as shown by work on program verification [51, 114, 116, 9, 10]. In particular, it is generally not sufficient to check at the end of each method that the invariant of its receiver is maintained. This traditional approach [109], which is for instance implemented in Pex and AutoTest, may miss invariant violations when programs use common idioms such as direct field updates, inheritance, or aggregate structures (see Sect. 3.1).

We address this issue with a technique for detecting previously missed invariant violations by synthesizing parameterized unit tests (PUTs) [136] that are likely to create *broken objects*, i.e., class instances that do not satisfy their invariants. The synthesis is based on templates that capture

all situations in which traditional invariant checking is insufficient. We use dynamic symbolic execution [70, 20] to find inputs to the synthesized PUTs that actually violate an invariant.

Whenever our approach detects an invariant violation, the programmer has to inspect the situation to decide which of the following three cases applies: (1) The object invariant is stronger than intended. In this case, one should weaken the invariant. (2) The invariant expresses the intended properties, but the program does not maintain it. This case constitutes a bug that should be fixed. (3) The invariant expresses the intended properties and can, in principle, be violated by clients of the class, but the entire program does not exhibit such violations. For instance, the class might provide a setter that violates an invariant when called with a negative argument, but the program does not contain such a call. In such cases, one should nevertheless adapt the implementation of the class to make the invariant *robust* against violations, for future program changes during maintenance and for other clients of the class during code reuse.

The contributions of this chapter are as follows:

- It identifies an important limitation of current test case generation approaches in the treatment of object invariants. In particular, existing approaches that use invariants as filters on input data do not sufficiently check them, if at all.

- It presents a technique that detects invariant violations by synthesizing PUTs based on templates and exploring them via DSE.

- It demonstrates the effectiveness of this technique by implementing it as an extension to Pex and using it on a suite of open-source C# applications that contain invariants provided manually by programmers.

- It investigates how to fix detected invariant violations, that is, by evaluating whether these violations are caused by errors in the code, errors in the invariants, or insufficient robustness of the invariants.

- It applies our technique to invariants that were automatically inferred by Daikon. Our experiments show that the vast majority of these invariants can be violated by our technique and, thus, should be checked before using them to filter test inputs.

- It shows how to use our technique to construct test inputs for auxiliary methods that are intended to be called when the invariant does not hold.

Outline. This chapter is organized as follows. Sect. 3.1 illustrates the situations in which the traditional checks for object invariants are insufficient. Sect. 3.2 gives an overview of our approach. Sect. 3.3 explains how we

select the operations to be applied in a synthesized test, and Sect. 3.4 describes the templates used for the synthesis. We discuss our implementation in Sect. 3.5, present the experimental evaluation in Sect. 3.6, and describe how to use our approach to test auxiliary methods in Sect. 3.7. We review related work in Sect. 3.8.

3.1 Violating object invariants

We present three scenarios in which the traditional approach of checking at the end of each method whether it maintains the invariant of its receiver is insufficient. These scenarios have been identified by work on formal verification and together with a fourth scenario—call-backs, which are not relevant here, as explained in Sect. 3.7—have been shown to cover *all* cases in which traditional invariant checking does not suffice [51]. We assume that invariants are specified explicitly in the code as contracts. However, our technique applies equally to predicates that are provided as separate input to the test case generator, or invariants that have been inferred from program runs (see Sect. 3.6.3).

We illustrate the scenarios using the C# example in Fig. 3.1. A `Person` holds an `Account` and has a `salary`. An `Account` has a `balance`. `Person`'s invariant (lines 5–6) requires that `account` is non-null and the sum of the `account`'s `balance` and the person's `salary` is positive. A `SavingsAccount` is a special `Account` whose `balance` is non-negative (line 26). In each of the following scenarios, we consider an object p of class `Person` that holds an `Account` a.

Direct field updates

In most object-oriented languages, such as C++, C#, and Java, a method may update not only fields of its receiver, but of any object as long as the fields are accessible. For instance, method `Spend2` (which is an alternative implementation of `Spend1`) subtracts `amount` from the `account`'s `balance` through a direct field update, instead of calling method `Withdraw`. Such direct field updates are common among objects of the same class, say, nodes of a list, or of closely connected classes, say, a collection and its iterator. If a is a `SavingsAccount`, method `Spend2` might violate a's invariant by setting `balance` to a negative value. A check of the receiver's invariant at the end of method `Spend2` (here, `Person` object p) does not reveal this violation.

In order to detect violations through direct field updates, one would have to check the invariants of all objects whose fields are assigned to directly. However, these objects are not statically known (for instance, when the direct field update occurs within a loop), which makes it difficult to impose such checks.

```
1  public class Person {
2    Account account;
3    public int salary;
4
5    invariant account != null;
6    invariant 0 < account.balance + salary;
7
8    public void Spend1(int amount) {
9      account.Withdraw(amount);
10   }
11
12   public void Spend2(int amount) {
13     account.balance = account.balance - amount;
14   }
15 }
16
17 public class Account {
18   public int balance;
19
20   public void Withdraw(int amount) {
21     balance = balance - amount;
22   }
23 }
24
25 public class SavingsAccount : Account {
26   invariant 0 <= balance;
27 }
```

Figure 3.1: A C# example on invariant violations. We declare invariants using a special invariant-keyword.

Subclassing

Subclasses may restrict the possible values of a field inherited from a superclass, i.e., they may strengthen the invariant for this field, as shown by class `SavingsAccount`. Methods declared in the superclass are typically tested with instances of the superclass as their receiver; in such cases, only the weaker superclass invariant is being checked. When such methods are inherited by the subclass and called on subclass instances, they may violate the stronger subclass invariant. In our example, in case a is a `SavingsAccount`, calling the inherited method `Withdraw` on a might set `balance` to a negative value and violate the invariant of the subclass.

To detect such violations, one would have to re-test every inherited method whenever a new subclass is declared. Moreover, subclassing makes the invariant checks for direct field updates even more difficult because one would have to consider all subclasses for the objects whose fields are up-

dated. For instance, when introducing `SavingsAccount`, testing `Withdraw` on a subclass instance is not sufficient; one has to also re-test method `Spend2` to detect the invariant violation described in the previous scenario.

Multi-object invariants

Most data structures are implemented as aggregations of several objects. For such aggregate structures, it is common that an object invariant constrains and relates the states of several objects. In our example, the invariant of class `Person` relates the state of a `Person` object to the state of its `Account`. For such *multi-object invariants*, modifying the state of one object might break the invariant of another. For instance, when `Account` a executes method `Withdraw`, it might reduce the `balance` by an amount such that it violates the invariant of `Person` p.

To detect such violations, one would have to check the invariants of all objects that potentially reference a, e.g., the invariants of `Person` objects sharing the account, of collections storing the account, etc. These objects are not statically known and cannot even be approximated without inspecting the entire program, which defeats the purpose of unit testing.

These scenarios demonstrate that the traditional way of checking object invariants may miss violations in common situations, and that the checks cannot be strengthened in any practical way. Therefore, simply including all necessary invariant checks in the test oracle is not feasible; other techniques are required to detect invariant violations.

3.2 Approach

For a given UUT, we synthesize client code in the form of PUTs to detect invariant violations. The synthesis is based on a set of four fixed templates that capture all three scenarios of Sect. 3.1. Each template consists of a sequence of *candidate operations*, i.e., updates of public fields and calls to public methods. These operations are applied to the object whose invariant is under test or, in the case of aggregate structures, its sub-objects. (Since our approach synthesizes client code, it uses public candidate operations. To also synthesize code of possible subclasses, one would analogously include protected fields and methods.) The candidate operations are selected from the UUT based on whether they potentially lead to a violation of the object invariant. Depending on the template, additional restrictions are imposed on the candidate operations, e.g., that they are inherited from a superclass. By instantiating the templates with candidate operations, the synthesized PUTs become snippets of client code that potentially violate the invariant.

Alg. 3.1 shows the general strategy for the PUT synthesis. Function Synthesize takes the class of the object to which candidate operations

Algorithm 3.1: Synthesis of parameterized unit tests.

```
1 function Synthesize(class, inv, len)
2   candOps ← ComputeCandOps(class, inv)
3   puts ← GenFieldCombs(candOps, len)
4   puts ← puts + GenMultiCombs(candOps, len, inv)
5   puts ← puts + GenSubCombs(candOps, len)
6   puts ← puts + GenAllCombs(candOps, len)
7   return AddSpecs(puts)
```

should be applied (*class*), the object invariant under test (*inv*), and the desired length of the PUTs to be synthesized (*len*). The last argument prevents a combinatorial explosion by bounding the number of operations in each synthesized PUT. Synthesize returns a list of PUTs. Each PUT consists of a sequence of candidate operations and additional specifications, such as invariant checks, which are inserted by AddSpecs and explained in Sect. 3.3. The algorithm first determines the set of candidate operations (*candOps*) of *class* that could potentially violate the object invariant *inv*. It then synthesizes the PUTs for each of the three scenarios using the corresponding templates. We discuss the selection of candidate operations as well as these templates in detail in the next sections.

We complement the templates, which cover specific scenarios for violating invariants, by an *exhaustive enumeration* of combinations (of length *len*) of candidate operations. In the algorithm, these combinations are computed by function GenAllCombs. As we will see in Sect. 3.4, this exhaustive exploration is useful for multi-object invariants, where the actual violation may happen by calling a method on a sub-object of an aggregate structure.

For a given UUT, we apply function Synthesize for each class in the unit and the invariant it declares or inherits. We perform this application repeatedly for increasing values of *len*. All operations in the resulting PUTs have arguments that are either parameters of the enclosing PUT or results of preceding operations; all such combinations are tried exhaustively, which, in particular, includes aliasing among the arguments. This makes the PUTs sufficiently general to capture the scenarios of the previous section, i.e., to detect invariant violations caused by these scenarios. We employ dynamic symbolic execution (DSE) [70, 20] to supply the arguments to the PUTs.

3.3 Candidate operations

To synthesize client code that violates object invariants, we select candidate operations from the public fields and methods of the UUT. To reduce the number of synthesized PUTs, we restrict the operations to those that might

```
public class C {
  public int x;
  int y;

  invariant x == 42;

  public void SetX() {
    x = y;
  }

  public void SetY(int v) {
    y = v;
  }
}
```

Figure 3.2: A C# example on selecting candidate operations.

violate a given invariant. Such operations are determined by intersecting the read effect of the invariant with the write effect of the operation. The *read effect* of an invariant is the set of fields read in the invariant. If the invariant contains calls to side-effect free methods, the fields (transitively) read by these methods are also in its read effect. The *write effect* of a method is the set of fields updated during an execution of the method, including updates performed through method calls. The write effect of a field update is the field itself. Note that the effects are sets of (fully-qualified) field names, not concrete instance fields of objects. This allows us to use a simple, whole-program static analysis that conservatively approximates read and write effects without requiring alias information (see Sect. 3.5).

To illustrate these concepts, consider the example in Fig. 3.2. The read effect of the invariant is {`C.x`}, indicating that only the value of `C`'s field `x` determines whether the invariant holds. The write effect of an update to the public field `x` and of method `SetX` is {`C.x`}, while method `SetY` has write effect {`C.y`}. By intersecting these read and write effects, we determine that updates of `x` and calls to `SetX` must be included in the candidate operations.

With these operations, the exhaustive enumeration of sequences of length one (function GenAllCombs in Alg. 3.1) produces the two PUTs in Fig. 3.3. As shown in the figure, each synthesized test expects as argument a non-null object `o` whose invariant holds, applies the synthesized sequence of candidate operations to `o`, and then asserts that `o`'s invariant still holds. We encode the invariant via a side-effect free, boolean method `Invariant`, and use assume-statements to introduce constraints for the symbolic execution. The assume- and assert-statements are inserted into the PUTs by function AddSpecs of Alg. 3.1. The input object `o` is constructed using operations from the UUT, for instance, a suitable constructor. As explained above, the

```
void PUT_0(C o, int v) {
  assume o != null && o.Invariant();
  o.x = v;
  assert o.Invariant();
}

void PUT_1(C o) {
  assume o != null && o.Invariant();
  o.SetX();
  assert o.Invariant();
}
```

Figure 3.3: Parameterized unit tests of length one generated when exhaustively enumerating sequences of candidate operations for the example in Fig. 3.2.

arguments of candidate operations (like the value `v` for the assignment to `o.x` in the first test `PUT_0` of Fig. 3.3) are either parameters of the PUT and supplied later via DSE, or results of preceding operations.

Whether a method call violates an invariant may not only depend on its arguments, but also on the state in which it is called. For instance, a call to `SetX` violates the invariant only if `y` has a value different from 42. Therefore, tests that apply more than one candidate operation must take into account the possible interactions between operations. Consequently, for each candidate operation op_d that might *directly* violate a given object invariant, we compute its read effect and include in the set of candidate operations each operation op_i whose write effect overlaps with this read effect and might, therefore, *indirectly* violate the invariant. To prune the search space, we record that op_i should be executed before op_d. This process iterates until a fixed point is reached.

In our example, method `SetX` has read effect {`C.y`}. As a result, `SetY` is used in the PUTs as a candidate operation that should be called before `SetX`. Therefore, the exhaustive enumeration of sequences of length two includes `PUT_2` of Fig. 3.4. Note that, by assuming `o`'s invariant before the call to `SetX`, we suppress execution paths that have already been tested in a shorter PUT, i.e., paths that violate `o`'s invariant before reaching the final operation.

3.4 Synthesis templates

We now present the templates that capture the three scenarios of Sect. 3.1. Besides other arguments, each template expects an object `r` to which candidate operations are applied, and an object `o` whose invariant is under test. When the templates are used to synthesize an entire test, these two objects

```
void PUT_2(C o, int v) {
  assume o != null && o.Invariant();
  o.SetY(v);
  assume o.Invariant();
  o.SetX();
  assert o.Invariant();
}
```

Figure 3.4: Parameterized unit test of length two generated when exhaustively enumerating sequences of candidate operations for the example in Fig. 3.2.

coincide and we include only one of them in the PUT. The templates are also used to synthesize portions of larger PUTs, and then `r` and `o` may refer to different objects.

3.4.1 Direct field updates

The direct-field-update template tries to violate the invariant of an object `o` by assigning to a field of `r` (or to an element of `r` when `r` is an array). The template has the form shown in Fig. 3.5. It applies a sequence of operations (`Op_0` to `Op_M`) to `r` to create a state in which the subsequent update of `r.f` may violate `o`'s invariant. For instance, if the invariant relates `f` to private fields of the same object, these operations may be method calls that update these private fields. The operations `Op_0` to `Op_M` are selected from the set of candidate operations, and may include a method call or field update more than once. Their arguments as well as the right-hand side `v` of the last field update are either parameters of the template (`a0` to `aN`) or results of preceding operations; all such combinations are tried exhaustively.

The synthesis of PUTs from this template is performed by function GenFieldCombs in Alg. 3.2, which is invoked from Alg. 3.1. Line 3 generates

```
void DFU(r, o, a0..aN) {
  assume r != null;
  assume o != null && o.Invariant();
  Op_0(r, ...);
  ...
  Op_M(r, ...);
  assume o.Invariant();
  r.f = v;
  assert o.Invariant();
}
```

Figure 3.5: The direct-field-update template.

Algorithm 3.2: Synthesis of parameterized unit tests from the direct-field-update template.

```
1 function GenFieldCombs(candOps, len)
2   puts ← []
3   combs ← GenAllCombs(candOps, len − 1)
4   fieldOps ← FieldOps(candOps)
5   foreach comb in combs
6     foreach fieldOp in fieldOps
7       puts ← puts + [comb + [fieldOp]]
8   return puts
```

all possible sequences of length $len - 1$ from the set of candidate operations. Line 4 selects the set *fieldOps* of all field updates from the set of candidate operations, *candOps*. Lines 5–7 append each field update *fieldOp* to each of the sequences of operations computed earlier.

Consider an invocation of the synthesis with this template, where the object to which operations are applied and the object whose invariant is being tested are the same instance of `Person` from Fig. 3.1. The synthesized PUTs of length two include `PUT_DFU` in Fig. 3.6. Symbolically executing this PUT produces input data that causes the assertion of the invariant to fail; for instance, a `Person` object with `salary` 100 and whose `Account` has `balance` 100 for `o`, the value 150 for `a`, and any value less than or equal to 50 for `s`.

3.4.2 Subclassing

The template for the subclassing scenario (shown in Fig. 3.7) aims at breaking the invariant of an object by invoking inherited operations. It exhaustively applies a number of operations to an object of the subclass, including any operations inherited from a superclass, and requires that the last operation is an inherited one (i.e., an update of an inherited field or a call to an

```
void PUT_DFU(Person o, int a, int s) {
  assume o != null && o.Invariant();
  o.Spend1(a);
  assume o.Invariant();
  o.salary = s;
  assert o.Invariant();
}
```

Figure 3.6: Parameterized unit test synthesized with the direct-field-update template.

```
void S(r, o, a0..aN) {
  assume r != null;
  assume o != null && o.Invariant();
  Op_0(r, ...);
  ...
  Op_M(r, ...);
  assume o.Invariant();
  Op_super(r, ...);
  assert o.Invariant();
}
```

Figure 3.7: The subclassing template.

inherited method) to reflect the subclassing scenario described in Sect. 3.1. Like in the template for direct field updates, the first $M+1$ operations (`Op_0` to `Op_M`) construct a state in which the final inherited operation may violate `o`'s invariant, since this final operation was designed to maintain the weaker invariant of a superclass.

This template is useful only when a subclass strengthens the invariant of a superclass with respect to any inherited fields. We identify such subclasses using a simple syntactic check: if the read effect of the invariant declared in the subclass includes inherited fields, we conservatively assume that the invariant is strengthened with respect to those.

The synthesis of PUTs based on this template is performed by function GenSubCombs, which is invoked from Alg. 3.1. GenSubCombs (shown in Alg. 3.3) is analogous to GenFieldCombs (Alg. 3.2) except that, on line 4, it selects the candidate operations that are inherited from a superclass.

Consider the synthesis with this template, where objects `r` and `o` are the same instance of `SavingsAccount` (Fig. 3.1). This class strengthens the invariant of its superclass `Account` for the inherited field `balance`. The synthesized PUTs of length one include `PUT_S` in Fig. 3.8. The symbolic execution

Algorithm 3.3: Synthesis of parameterized unit tests from the subclassing template.

```
1 function GenSubCombs(candOps, len)
2   puts ← []
3   combs ← GenAllCombs(candOps, len − 1)
4   inheritedOps ← InheritedOps(candOps)
5   foreach comb in combs
6     foreach inheritedOp in inheritedOps
7       puts ← puts + [comb + [inheritedOp]]
8   return puts
```

```
void PUT_S(SavingsAccount o, int a) {
  assume o != null && o.Invariant();
  o.Withdraw(a);
  assert o.Invariant();
}
```

Figure 3.8: Parameterized unit test synthesized with the subclassing template.

of `PUT_S` reveals an invariant violation; for instance, for a `SavingsAccount` object with a `balance` of zero for `o` and any positive value for `a`.

3.4.3 Multi-object invariants

Multi-object invariants describe properties of aggregate structures. The invariant of such a structure may be violated by modifying its sub-objects. For instance, one might be able to violate a `Person`'s invariant by reducing the balance of its account. Such violations are possible when sub-objects of the aggregate structure are not properly encapsulated [116] so that clients are able to obtain references to them: when a client obtains a direct reference to the `Account` sub-object, it can by-pass the `Person` object and modify the account in ways that violate the `Person`'s invariant.

To reflect this observation, we define two templates that allow clients to obtain references to sub-objects of aggregate structures. One template uses *leaking*, i.e., it passes a sub-object from the aggregate structure to its client. The other one uses *capturing*, i.e., it passes an object from the client to the aggregate structure and stores it there as a sub-object. Leaking and capturing are the only ways in which clients may obtain a reference to a sub-object of an aggregate structure.

Leaking

An operation is said to *leak* an object `l` if the following three conditions hold: (1) the operation takes as an argument (or receiver) an object `o` that (directly or transitively) references `l`, (2) the operation returns the reference to `l` or assigns it to shared state, and (3) a field of `l` is dereferenced in `o`'s invariant. We use a static analysis to approximate the operations that might leak a sub-object (see Sect. 3.5). These operations include reading public fields with reference types.

For example, assume that class `Person` from Fig. 3.1 provides a public getter `GetAccount` for field `account`:

```
public Account GetAccount() {
  return account;
}
```

```
void L(r, o, a0..aN) {
  assume r != null;
  assume o != null && o.Invariant();
  Op_0(r, ...);
  ...
  Op_M(r, ...);
  var l = Op_leaking(r, ...);
  ... // operations on leaked object
  assert o.Invariant();
}
```

Figure 3.9: The leaking template.

This method leaks the `account` sub-object of its receiver since (1) its receiver directly references the account, (2) it returns the account, and (3) `account` is dereferenced in the invariant of `Person`. Consequently, this getter enables clients to obtain a reference to the `account` sub-object and violate `Person`'s invariant, for instance, by invoking method `Withdraw` on the account.

In the template for leaking (Fig. 3.9), we first apply a number of operations to create a state in which a sub-object `l` may be leaked via the `Op_leaking` operation. Once the object has been leaked, we try to violate `o`'s invariant by applying operations to the leaked object `l` (indicated by the ellipsis with the comment in Fig. 3.9). To obtain a sequence of operations on `l`, we apply function SYNTHESIZE (Alg. 3.1) recursively with the class of `l` and the invariant of `o`. This recursive call selects candidate operations on `l` that may break `o`'s invariant, e.g., by updating a public field of `l`, or via complex combinations of scenarios, such as repeated leaking. Note that this template attempts to violate `o`'s invariant; whether `l`'s invariant holds is an orthogonal issue that is addressed separately when testing that invariant.

Based on this template, we obtain the PUT in Fig. 3.10 for objects of class `Person` from Fig. 3.1. In this test, method `GetAccount` leaks the

```
void PUT_L(Person o, int a) {
  assume o != null && o.Invariant();
  var l = o.GetAccount();
  // exhaustive enumeration
  assume l != null && o.Invariant();
  l.Withdraw(a);
  assert o.Invariant();
}
```

Figure 3.10: Parameterized unit test synthesized with the leaking template.

`Person`'s `account` object. The recursive application of function SYNTHESIZE determines method `Withdraw` as a candidate operation because its write effect includes `balance`, which is also in the read effect of `Person`'s invariant. `Withdraw` is selected by the exhaustive enumeration (function GENALLCOMBS of Alg. 3.1) and would not be selected by any of the other templates. Symbolically executing this PUT produces input data that causes the assertion of the invariant to fail; for instance, a `Person` object with `salary` 100 and whose `Account` has `balance` 100 for `o`, and a value of at least 200 for `a`.

Capturing

An operation is said to *capture* an object `c` if: (1) the operation takes as arguments two objects `o` and `c` (`o` or `c` could also be the receiver), (2) the operation stores a reference to `c` in a location reachable from `o`, and (3) the field in which `c` is stored is dereferenced in `o`'s invariant. Updating a field `f` of a reference type is considered capturing if `f` is dereferenced in `o`'s invariant.

The template for capturing (shown in Fig. 3.11) is analogous to leaking. In particular, it also uses a recursive application of function SYNTHESIZE to determine the operations to be applied to the captured object. In the common case that the capturing operation is a constructor of object `o`, the template is adjusted as shown in the figure (`Cctor`). This adjustment ensures that `o` is actually created with a constructor that captures `c`, instead

```
void C(r, o, c, a0..aN) {
  assume r != null;
  assume o != null && o.Invariant();
  Op_0(r, ...);
  ...
  Op_M(r, ...);
  Op_capturing(r, c, ...);
  ... // operations on captured 'c'
  assert o.Invariant();
}

void Cctor(c, a0..aN) {
  assume c != null;
  Op_0(c, ...);
  ...
  Op_M(c, ...);
  var o = new ctor(c, ...);
  ... // operations on captured 'c'
  assert o.Invariant();
}
```

Figure 3.11: The capturing templates.

```
void PUT_Cctor(Account c, int b) {
  assume c != null;
  var o = new Person(c);
  // direct field update
  assume c != null && o.Invariant();
  c.balance = b;
  assert o.Invariant();
}
```

Figure 3.12: Parameterized unit test synthesized with the capturing template.

of a constructor selected by the symbolic execution. Note that before the capturing operation, our implementation also allows a number of operations on `c` with the goal of bringing it to a state such that, for instance, the precondition of `Op_capturing` is satisfied or the capturing execution path is taken. We omit such operations here to simplify the presentation.

As an example, assume that class `Person` of Fig. 3.1 declares a constructor that captures an already existing account `a`:

```
public Person(Account a) {
  account = a;
}
```

Then, the `Cctor` template produces the PUT shown in Fig. 3.12. The symbolic execution of `PUT_Cctor` reveals an invariant violation; for instance, for an `Account` object with `balance` 100 for `c` and any non-positive value for `b`.

Recursively applying our technique on leaked or captured objects allows for covering even more complex cases. For example, assume that there is a class `CreditCardAccount` that inherits from `Account`, and that class `Account` has a field of type `CreditCardAccount` and a public setter for this field. In addition, assume that the invariant of class `Person` requires that the account for the credit card always has a non-negative balance. In this case, our technique synthesizes tests such as the one in Fig. 3.13, in which a getter leaks `o`'s account `l`, account `l` is updated to capture a credit card account `a`, and the balance of the credit card account is directly set to an amount `b`. If `b` has a negative value, `o`'s invariant breaks.

Synthesis

The synthesis of PUTs based on the leaking and capturing templates is performed by the GenMultiCombs function in Alg. 3.4. On line 3, a new set of candidate operations, *multiOps*, is created by selecting from *candOps* the operations that leak or capture objects according to the above criteria. Since the synthesis for these templates includes a recursive application of Synthesize, we must split the overall length of the PUT between the operations

```
void PUT(Person o, Account a, int b) {
  // leaking
  assume o != null && o.Invariant();
  var l = o.GetAccount();
  // capturing
  assume l != null && o.Invariant();
  l.SetCreditCardAccount(a);
  // direct field update
  assume a != null && o.Invariant();
  a.balance = b;
  assert o.Invariant();
}
```

Figure 3.13: Complex parameterized unit test synthesized by recursively applying our technique.

occurring before the leaking or capturing operation and the operations on the leaked or captured object occurring after. To explore all possible splits, we generate all combinations of candidate operations of length up to $len - 2$ to be applied before the leaking or capturing operation (lines 4–5). These operations create a state in which the next operation can leak or capture an object. After invoking any such leaking or capturing operation, we recursively apply function SYNTHESIZE of Alg. 3.1 by taking into account the class of the leaked or captured object, *class*, and the original object invariant under test, *inv* (lines 6–8). Therefore, *suffixes* is a list of sequences of operations to be applied to the leaked or captured object. On lines 9–12, we combine the synthesized sub-sequences of lengths i, 1, and $len - 1 - i$.

Algorithm 3.4: Synthesis of parameterized unit tests from the leaking and capturing templates.

```
1  function GenMultiCombs(candOps, len, inv)
2    puts ← []
3    multiOps ← MultiOps(candOps)
4    for i = 0 to len − 2 do
5      prefixes ← GenAllCombs(candOps, i)
6      foreach multiOp in multiOps
7        class ← GetClass(multiOp)
8        suffixes ← Synthesize(class, inv, len − 1 − i)
9        foreach prefix in prefixes
10         foreach suffix in suffixes
11           put ← prefix + [multiOp] + suffix
12           puts ← puts + [put]
13   return puts
```

3.5 Implementation

We have implemented our technique as an extension to Pex [135].

3.5.1 Runtime checks

Pex understands specifications, such as object invariants, written in Code Contracts [58], as long as these specifications are transformed into runtime checks by the Code Contracts binary rewriter. For our purposes, we implemented an additional rewriter, which adds a public method `Invariant` to each class that declares or inherits an invariant. This method takes no arguments, and returns whether its receiver satisfies the invariant declared in the enclosing class as well as the invariants in its superclasses. The `Invariant` methods are called when our technique asserts or assumes the invariant under test in the synthesized PUTs (see Sect. 3.3). Since the synthesized PUTs include all necessary invariant checks, our rewriter turns off any runtime checks for object invariants inserted by the Code Contracts rewriter.

The runtime checks generated for Code Contracts may abort the execution of a synthesized PUT, for instance, when the precondition of one of the operations does not hold. Our implementation permits such cases, but does not report them to the user. As an optimization, one could insert an assumption before each call, that the precondition holds, in order to steer the symbolic execution toward inputs that do not abort the test. We do add such assumptions to express that the receiver of the operation is non-null and array accesses are within the bounds.

3.5.2 Static analysis

Our implementation builds a static call graph for the entire UUT, which includes information about dynamically-bound calls. The call graph is used to compute the read and write effects of all methods in the UUT with a conservative, inter-procedural, control-flow insensitive static analysis on the .NET bytecode. Our effect analysis treats the entire array content as one field and does not distinguish different array elements. The effects determine the candidate operations that might, directly or indirectly, lead to an invariant violation (see Sect. 3.3).

Our effect analysis is extended to also approximate the sets of leaking and capturing operations. When trying to violate the invariant of an object `o` with the leaking template, our analysis considers each candidate operation in the UUT and each potential sub-object `l` of `o`, and checks whether the operation leaks `l` according to the conditions for leaking from Sect. 3.4.3. For efficiency, we coarsely approximate these conditions by working on the level of types rather than objects.

This allows us to use our effect analysis to determine the types of the

potential sub-objects of `o`, that is, of the objects `l` that may satisfy condition 3 from Sect. 3.4.3. If the read effect of `o`'s object invariant contains a field `C.f`, we consider each instance of class `C` as a potential sub-object of `o`. For instance, the read effect of `Person`'s invariant (from Fig. 3.1) is {`Person.account`, `Account.balance`, `Person.salary`} and, therefore, each `Person` or `Account` object is considered to be a potential sub-object.

According to condition 1, a leaking operation takes `o` as an argument or receiver. Again, we identify these operations based on types, that is, based on the signatures of the operations and `o`'s type. For instance, if `o` is of type `Person`, all methods of this class, such as `Spend1`, `Spend2`, and `GetAccount`, are potentially leaking operations.

It remains to check which of these operations satisfy condition 2, that is, return `l` or assign it to shared state. In the former case, we conservatively assume that the method may return any object that is compatible with the method's return type and with the type of at least one field in the method's read effect. For instance, method `GetAccount` has return type `Account` and read effect {`Person.account`} and, thus, may return any `Account` object. If the type of potentially returned objects is compatible with the types of `o`'s sub-objects, the method is considered to be leaking. In our example, the types of sub-objects are `Person` or `Account` and, thus, we treat `GetAccount` as a leaking operation.

The latter case is analogous, but, instead of the return type, uses the types of those public fields of the operation's receiver that are in the write effect of the operation. This approximation may miss certain cases of leaking, for instance, when the leaked object is stored in fields of objects other than the receiver, but it ensures that clients of the operation can access the leaked object because they have a reference to the receiver and may, thus, read its public fields.

Once the leaking operations and the way in which they leak a sub-object are identified, our synthesis generates a PUT for each possible way of accessing the leaked object. For instance, it uses `GetAccount`'s return value to access the leaked object, and applies operations to it with the purpose of breaking `Person`'s invariant (see Fig. 3.10).

We identify capturing operations similarly to leaking operations, but use the write effects of operations, instead of the read effects. The captured object is then accessed via the operation's parameters.

Impact of approximations

The design of our static analysis favors efficiency over precision. In particular, we avoid a precise but costly heap analysis, and instead, obtain rather coarse effect information. As a consequence, our technique might generate irrelevant PUTs and miss invariant violations, as we discuss next.

Our technique may select irrelevant candidate operations due to impre-

cision in our effect analysis. For instance, it cannot distinguish whether a method modifies an existing or initializes a newly-allocated object. Therefore, we might generate PUTs that try to violate an invariant using a method that is effectively side-effect free. Another source of imprecision is the syntactic check that determines whether a subclass strengthens the invariant of a superclass with respect to any inherited fields (Sect. 3.4.2). In both cases, our technique could generate irrelevant PUTs, which may have a negative effect on its performance but does not cause it to miss invariant violations.

On the other hand, invariant violations may be missed when, for instance, our static analysis fails to consider leaking operations that transitively reference the leaked object. Our technique may also miss invariant violations when it fails to consider a method override in an assembly that is not accessible to the static analysis. As a result, the generated PUTs might not explore all possible sequences of operations.

3.5.3 Heuristics and optimizations

To detect invariant violations more efficiently, we carefully chose the order in which the synthesis (Alg. 3.1) applies the templates of Sect. 3.4 and exhaustive enumeration. The templates for direct field updates and multi-object invariants have proven to most effectively detect invariant violations and are, thus, explored first. The exhaustive enumeration comes last as it produces the largest number of PUTs and requires the most effort in DSE.

On a more technical level, a leaking operation applied to an object `o` might declare as its return type a superclass of the leaked reference `l`. In such cases, our implementation automatically down-casts `l` to the subclass declared in `o`'s implementation, before applying any operations to `l`.

Moreover, special care had to be taken for the handling of arrays and generics. For arrays, we defined a set of valid operations, such as retrieving their length and storing a value at an index within their bounds, which could then be used in the synthesized code. In order to handle generic types and methods, our implementation attempts to instantiate them by choosing suitable types from the UUT, if any.

To reduce the number of functionally-equivalent PUTs, we implemented a conservative pruning technique based on operation commutativity. We consider two operations *commutative*, i.e., independent of their order of execution, if and only if each operation's write effect is disjoint from the other operation's read and write effects. Given a PUT, any other PUTs are pruned that differ from the former only in the order of commutative operations.

3.5.4 Object construction

Each synthesized PUT takes as input an object whose invariant holds and that the PUT tries to violate. When generating inputs for the PUT, this

object must satisfy additional constraints in order to execute certain paths. Constructing such objects may require several steps, for instance, creating an empty list and then invoking a method three times to insert three elements. Such constructions lie beyond the automatic capabilities of Pex. Our templates effectively alleviate this issue. If Pex can create any instance whose invariant holds (for instance, an empty list via a suitable constructor), then the sequence of operations in the PUT can be used to bring this instance into a state satisfying all necessary constraints, before applying the final operation to violate its invariant.

3.6 Experimental evaluation

We evaluated our technique for testing object invariants in three ways. First, we applied our technique to several open-source projects that contain object invariants provided manually by programmers, to evaluate whether it detects invariant violations in practice. Second, for one of these projects, we manually fixed all detected invariant violations, to evaluate the causes of these violations. Third, we applied our technique to invariants inferred automatically by Daikon, to evaluate the correctness and robustness of these invariants.

3.6.1 Testing programmer-provided object invariants

We have evaluated the effectiveness of our technique using ten C# applications, which were selected from applications on Bitbucket, CodePlex, and GitHub containing invariants specified with Code Contracts. This section focuses on our experiments with the nine applications for which invariant violations were detected. A brief description of these applications is shown in Tab. 3.1.

Tab. 3.2 summarizes the results of our experiments. The second and third columns show the total number of classes and the number of classes with invariants for each application, respectively. We have tested the robustness of all invariants in these applications. Note that the total number of classes refers only to the classes that were included in the evaluation and not to all classes of each application. We have left out only classes that were defined in dynamic-link libraries (DLLs) containing no object invariants. The two rightmost columns of Tab. 3.2 show the unique and total numbers of invariant violations detected with our technique. The unique number of violations refers to the number of invariants that were violated at least once. The total number of violations refers to the number of distinct PUTs that led to invariant violations, and may include violations of the same object invariant multiple times.

Application	Description
Boogie	Intermediate verification engine[1]
ClueBuddy	GUI application for a board game[2]
Dafny	Programming language and verifier[3]
Draugen	Web application for fishermen[4]
GoalsTracker	Various web applications[5]
Griffin	.NET and jQuery libraries[6]
LoveStudio	IDE for the LÖVE framework[7]
Encore	'World of Warcraft' emulator[8]
YAML	YAML library[9]

Table 3.1: Brief description of the applications in which our technique detected invariant violations.

When running Pex with our technique, we imposed an upper bound of three on the number of operations per PUT, and an upper bound of 300 on the number of synthesized PUTs per object invariant. All unique invariant violations were already detected with two operations per PUT; increasing this bound to four for a number of projects did not uncover any previously undetected invariant violations. On average, 14.7 PUTs were synthesized per second. We then applied Pex to generate input data for the synthesized PUTs, forcing Pex to use only public operations of the UUT (to guarantee that all inputs are constructible in practice). We counted the number of unique invariant violations and of distinct PUTs that led to invariant violations. We imposed a timeout of three minutes for the DSE in Pex to generate inputs for and run the synthesized PUTs. Here, we report the number of invariant violations that were detected within this time limit. Within this time limit, the first invariant violation was detected within 4–47 seconds (and 12.8 seconds on average) for all object invariants in all applications.

The total violations found by our technique may be classified into the following categories, based on the template that was instantiated: 60 due to direct field updates, 41 due to leaking, and 25 due to capturing. The remaining six violations were detected by the exhaustive enumeration. Out of these six invariant violations, five are also detected by Pex without our

[1] `http://boogie.codeplex.com`, rev: `f2ffe18efee7`
[2] `https://github.com/AArnott/ClueBuddy`, rev: `c1b64ae97c01fec249b2212018f589c2d8119b59`
[3] `http://dafny.codeplex.com`, rev: `f2ffe18efee7`
[4] `https://github.com/eriksen/Draugen`, rev: `dfc84bd4dcf232d3cfa6550d737e8382ce7641cb`
[5] `https://code.google.com/p/goalstracker`, rev: `556`
[6] `https://github.com/jgauffin/griffin`, rev: `54ab75d200b516b2a8bd0a1b7cfe1b66f45da6ea`
[7] `https://bitbucket.org/kevinclancy/love-studio`, rev: `7da77fa`
[8] `https://github.com/Trinity-Encore/Encore`, rev: `0538bd611dc1bc81da15c4b10a65ac9d608dafc2`
[9] `http://yaml.codeplex.com`, rev: `96133`

Application	Classes	Classes with invariants	Invariant violations	
			unique	total
Boogie	355	144	21	64
ClueBuddy	44	4	1	2
Dafny	310	113	15	53
Draugen	36	5	3	3
GoalsTracker	63	5	1	1
Griffin	31	3	1	1
LoveStudio	66	7	2	2
Encore	186	30	1	4
YAML	76	6	1	2
Total	1167	317	46	132

Table 3.2: Summary of results for testing programmer-provided object invariants. The second and third columns show the total number of classes and the number of classes with invariants for each application. The two rightmost columns show the unique and total numbers of invariant violations detected with our technique.

technique, i.e., with the traditional approach of checking the invariant of the receiver at the end of a method. The last violation requires a sequence of two method calls and was detected only by our technique. This is because Pex could not generate appropriate input data to the second method such that the invariant check at the end of the method failed. This violation illustrates that the exhaustive enumeration indeed alleviates a known limitation of automatic object creation in Pex [139], as pointed out in the previous section. The object invariants that were violated at least once can be classified into the following categories: 27 invariants were violated at least once due to direct field updates, 24 due to leaking, 17 due to capturing, and five due to the exhaustive enumeration. Note that in these applications we found no subclasses that strengthen the invariant of a superclass with respect to any inherited fields. This is why no invariant violations were detected with the subclassing template.

An example of an invariant violation detected by our technique in LoveStudio is shown in Fig. 3.14. A `StackPanel` object has a `LuaStackFrame` array, and its invariant holds if all array elements are non-null. In the PUT, method `SetFrames` captures `a0` depending on the value of `a1`. The last operation of the test assigns a `LuaStackFrame` object to the array at a valid index `a2`. In case `a3` is null, `o`'s invariant is violated.

The invariant violations detected by our technique indicate overly strong invariants in the sense that they may be violated by possible clients of a UUT. These clients are not necessarily present in the given program and,

```
void PUT(StackPanel o, LuaStackFrame[] a0,
         bool a1, int a2, LuaStackFrame a3) {
  assume o != null && o.Invariant();
  o.SetFrames(a0, a1);
  assume a0 != null && o.Invariant();
  assume 0 <= a2 && a2 < a0.Length;
  a0[a2] = a3;
  assert o.Invariant();
}
```

Figure 3.14: Synthesized parameterized unit test that reveals an invariant violation in LoveStudio.

thus, the violations do not necessarily reveal bugs. This behavior is to be expected for unit testing, where each unit is tested independently of the rest of the program. Nevertheless, the detected violations do indicate robustness issues that might lead to bugs during maintenance or when classes are reused as libraries.

We have manually inspected all detected invariant violations. Violations detected with the direct-field-update template reveal design flaws; in the inspected code, these violations could be fixed by making fields private and providing setters that maintain the invariants. Violations due to leaking or capturing could be fixed either by cloning the leaked or captured objects, or by using immutable types in the interfaces of the classes whose invariants are under test. The largest number of invariant violations found with the leaking and capturing templates was detected in the Boogie and Dafny applications, which declare several multi-object invariants in their code. Both Boogie and Dafny were originally written in Spec# [12], whose verification methodology restricts modifications of leaked or captured objects using object ownership [99]. Note, however, that only a small fraction of Boogie was actually verified with Spec#, and both Boogie and Dafny have changed substantially since ported to C#. We discussed the detected invariant violations in Boogie and Dafny with the lead developer, Rustan Leino. All of them indicate robustness issues that should be addressed by either weakening the invariants or changing the design of the code, as we discuss in the next subsection.

We have also examined the object invariants that were not violated. In most cases, these invariants expressed very simple properties, such as non-nullness or ranges of integer values, which were also specified as constructor preconditions or referred to read-only fields that were properly initialized in the constructors. Another common case involved automatically-generated setters for C# properties to which Code Contracts add appropriate preconditions that ensure the object invariant is not violated by the setters.

3.6.2 Fixing violated object invariants

As we have discussed at the beginning of this chapter, a test that violates an invariant could indicate an error in the code, an overly strong invariant, or an invariant that is not robust against future program changes and other clients.

To gain more insight into how a developer would use feedback from our tool, we inspected and fixed all violated invariants for one of the applications in Tab. 3.1. We chose Boogie (revision `31e2170a4d1b`, which is a more recent revision than the one used in Sect. 3.6.1) because: (1) it contains a large number of classes with invariants and has been actively developed for many years, (2) our tool was able to violate a significant number of these invariants, and (3) there are several widely-used client applications that use Boogie as a library (e.g., the Chalice [102], Dafny [98], Spec# [12], and Viper [86] verifiers), which makes it especially important that Boogie's invariants are robust against these clients.

We asked a student to propose changes to Boogie based on concrete test cases that led to violated invariants and were produced by running Pex on PUTs generated by our tool. The student had not used our tool before and, more importantly, was not familiar with the Boogie code base. All proposed changes were reviewed by independent Boogie developers, to confirm that they were sensible and correctly reflected the intended software design (e.g., by not restricting the API in undesirable ways). The changes were accepted and then merged into the Boogie repository.

We classified the proposed changes into three different categories:

1. *Invariants*: ones that weakened or removed overly strong invariants;

2. *Encapsulation*: ones that mainly affected the visibility (incl. qualifiers such as `readonly`) of a class or class member to encapsulate the state an invariant depends on, for instance, by making a public field private and possibly introducing corresponding getters and setters that preserve the invariant;

3. *Code*: ones that changed the code in more involved ways, for instance, by copying a returned object to avoid leaking.

The above categories are ordered roughly according to how much work seems to be typically involved in implementing such changes and how much they may affect clients. For instance, invariant changes are usually relatively straightforward and do not affect clients, while code changes can be complex and may require changes to the API.

The student provided changes for 32 classes involving 57 invariant clauses (that is, conjuncts) that our tool was able to violate. Many of those invariant clauses expressed relatively simple properties, such as non-nullness of

reference fields or numeric ranges for integer fields. Since many of the involved fields were public, the direct-field-update template turned out to be very effective in breaking them, and 34 of those invariant clauses were made robust by strengthening encapsulation. Somewhat surprisingly, all other 23 changes were code changes. That is, none of the violated clauses was considered to be overly strong, likely because Boogie is a mature application that has used such invariants from the start to document its design. Many of the code changes involved invariant clauses that expressed non-nullness of elements in a collection (e.g., arrays, lists, or dictionaries), and were detected using the leaking and capturing templates. The changes made use of different, well-known techniques for preserving such invariants (e.g., copying an object to avoid leaking, or exposing it as an immutable object). There was one invariant that was violated using the subclassing template; in this case, the invariant clause was moved to the superclass that declared the constrained field, because it also applied to instances of the superclass.

Our experiment is encouraging because it suggests that the detected invariant violations can be fixed with a rather low effort, and we were pleased that the Boogie developers considered these fixes to improve the design of the tool, by making it more robust against future changes and additional clients.

3.6.3 Testing inferred object invariants

Some tools, such as Symbolic Java PathFinder [125], use dynamic invariant inference to obtain input constraints on a UUT. To evaluate the suitability of our technique for testing the robustness of such invariants, we used the Daikon invariant detector [57] and its .NET front end Celeriac[10] to infer object invariants for one of the applications from Tab. 3.1. Given an execution trace, Daikon infers invariants by observing the values that the program computed during its execution. Daikon then reports properties that are true over the observed executions and expressible in the built-in set of invariant templates of the tool. Celeriac dynamically instruments .NET programs to produce a Daikon-compatible execution trace. For this experiment, we chose ClueBuddy[11], a GUI application for the 'Clue' board game, because it has a comprehensive test suite that produces traces of a suitable size.

To produce the execution trace that we then passed to Daikon, we ran the existing test suite of ClueBuddy, which contains both unit and system tests. Based on this execution trace, Daikon inferred 288 potential object-invariant clauses (conjuncts) for 13 of the 44 classes. Only two of these invariant clauses corresponded to ones that were also provided by the developers. We subsequently removed the following clauses:

[10] `https://code.google.com/p/daikon-dot-net-front-end`

[11] `https://github.com/AArnott/ClueBuddy`, rev: `c1b64ae97c01fec249b2212018f589c2d8119b59`

– clauses about the runtime types of objects, such as

```
this.GetType() == typeof(ClueBuddy.Weapon)
```

which are better expressed through the type system;

– clauses (directly or indirectly) relating a private field with the return value of a getter for this field, e.g.,

```
this.place.Name == this.Place.Name
```

because such constraints should rather be expressed as postconditions of getters;

– equivalent occurrences of a clause, such as the property

```
this.player.Name != null
```

of the field `player` and the property

```
this.Player.Name != null
```

of the getter `Player`;

– clauses reported as either semantically incorrect or always true by the C# compiler, e.g.,

```
this.Rules != null
```

where `this.Rules` is a struct.

After this manual filtering, there were 105 object-invariant clauses for 13 classes.

We tested these invariant clauses with our technique. We imposed an upper bound of three on the number of operations per PUT, and an upper bound of 1,000 on the number of synthesized PUTs per object invariant. When applying Pex to generate input data for the synthesized PUTs, we imposed a timeout of 10,000 seconds and an upper bound of 100,000 symbolic branches for the DSE.

Tab. 3.3 summarizes the results. The second column shows the number of object-invariant clauses (per class) that were inferred by Daikon and not filtered by us. Unlike for the programmer-provided invariants from Sect. 3.6.1, we found that a significant number of invariants threw exceptions during their evaluation (e.g., a null-dereference in a clause `0 <= a.Length`). We, therefore, decided to distinguish between invariant clauses that were violated and ones that were ill formed, i.e., for which we found cases where their evaluation would throw an exception. The third and fourth columns of the table show the number of clauses that were found to be violated or ill formed with our technique. Note that the sets of violated and ill-formed

Class	Inferred clauses	Violated clauses	Ill-formed clauses	Rejected clauses	
`CannotDisprove`	9	4	7	9	(100%)
`Card`	3	2	1	3	(100%)
`Clue`	4	3	3	4	(100%)
`Disproved`	10	5	8	10	(100%)
`Game`	25	19	1	20	(80%)
`GameVariety`	17	14	11	17	(100%)
`Node`	6	4	0	4	(67%)
`Place`	5	4	2	5	(100%)
`Player`	6	4	1	5	(83%)
`SpyCard`	6	5	4	6	(100%)
`Suspect`	3	2	1	3	(100%)
`Suspicion`	7	3	0	3	(43%)
`Weapon`	4	3	2	4	(100%)
Total	105	72	41	93	(89%)

Table 3.3: Summary of results for testing inferred object invariants. The second column shows the number of inferred invariant clauses per class. The third and fourth columns show the number of clauses that were found to be violated or ill formed with our technique. The last column shows the total number of clauses that were found not to be robust (because they were violated or ill formed).

clauses are not necessarily disjoint for a given class. For instance, the invariant clause `0 <= a.Length` about an array `a` might be classified both as ill formed, when not preceded by the clause `a != null`, and as violated when it evaluates to false without throwing an exception. The last column of Tab. 3.3 shows the total number of clauses that were found not to be robust (because they were violated or ill formed). In total, we rejected 93/105 (or 89%) of the inferred invariant clauses. This high percentage indicates the effectiveness of our technique in refuting invariants inferred from program runs that possibly do not exercise all relevant paths.

To provide an overview of the kinds of invariant clauses that were inferred, Tab. 3.4 further categorizes the inferred and rejected clauses based on the properties they specify, i.e., nullness of an expression (third column), comparison of boolean expressions (fourth column), comparison of numerical expressions (fifth column), and all others (sixth column), e.g., clauses involving calls to side-effect free methods. The four rightmost columns of the table show, for each category of clauses, the number of rejected clauses over the number of inferred clauses. Note that most inferred invariant clauses (44/105 or 42%) express comparisons with null and are relatively more robust than all other kinds of clauses. As shown in the table, Daikon inferred

Class	Inferred clauses	Null comparisons	Boolean comparisons	Numeric comparisons	Other
`CannotDisprove`	9	8/8	0/0	1/1	0/0
`Card`	3	0/0	3/3	0/0	0/0
`Clue`	4	3/3	0/0	1/1	0/0
`Disproved`	10	9/9	0/0	1/1	0/0
`Game`	25	2/3	8/10	6/8	4/4
`GameVariety`	17	3/3	3/3	6/6	5/5
`Node`	6	2/4	1/1	1/1	0/0
`Place`	5	0/0	3/3	0/0	2/2
`Player`	6	1/2	3/3	1/1	0/0
`SpyCard`	6	5/5	0/0	1/1	0/0
`Suspect`	3	0/0	3/3	0/0	0/0
`Suspicion`	7	3/7	0/0	0/0	0/0
`Weapon`	4	0/0	3/3	0/0	1/1
Total	105	36/44	27/29	18/20	12/12

Table 3.4: Categorization of inferred and rejected invariant clauses based on the properties they specify. The four rightmost columns of the table show, for each category of clauses, the number of rejected clauses over the number of inferred clauses.

a significant number of invariant clauses of each category, and our technique is effective in refuting clauses of all categories.

Since there were no public fields, the direct-field-update template was not used in detecting any of the 72 violated invariant clauses. However, all other templates turned out to be used. This demonstrates that our approach is more effective in detecting invariant violations than the traditional approach, which would only find a subset of the violations detected by the exhaustive enumeration. Leaking was used 26 times, capturing 19 times, subclassing five times, and the exhaustive enumeration was used 51 times, five out of which required a sequence length of two. Note that several clauses were violated using multiple templates. Out of the ones that were detected by a single template, leaking was used eight times, capturing once, and the exhaustive enumeration 42 times. All but one of the invariant clauses that were violated using the capturing template were also violated using the leaking template. Manual inspection showed that many of the captured objects are assigned to public properties, which may also leak said objects.

Our experiment suggests that many dynamically-inferred invariants are not robust. It is beneficial to apply our technique to reject such invariants or to flag them for manual inspection before using them to filter test inputs.

3.7 Testing auxiliary methods

Auxiliary methods are methods that are intended to operate correctly even if they are called on an object whose invariant does not hold. For instance,

a balanced-tree data structure might call a `rebalance` method in a state in which the tree is not balanced in order to re-establish the invariant.

Even if auxiliary methods do not require invariants to hold, they are typically not intended to operate on arbitrary inputs. For instance, `rebalance` would be designed to operate on an unbalanced tree, but not on an arbitrary graph. Therefore, the test inputs to auxiliary methods should neither be constrained by the object invariant (which would miss relevant executions) nor be arbitrary (which would include inputs the method has not been designed for). Instead, the inputs should include objects that could actually be created by operations of the data structure.

When attempting to violate object invariants, our technique creates exactly such objects. So in addition to reporting the invariant violations to the user, one can use the generated objects as inputs for testing auxiliary methods. That is, our technique can complement any existing approach for generating complex input objects, to achieve higher coverage and detect more bugs in auxiliary methods.

This is useful when it *is* possible to synthesize client code that violates invariants. However, many auxiliary methods (such as `rebalance` above) are invoked in intermediate states, in which invariants are temporarily broken. That is, `rebalance` is supposed to work on unbalanced trees even if there is no way for a client to construct such a tree. This scenario is a special case of a *call-back* (or re-entrant call): a method breaks the invariant of its receiver and then calls another method that, either directly or indirectly, calls back into the receiver when its invariant does not hold.

The example in Fig. 3.15 illustrates this scenario. Method `L` breaks the invariant of its receiver and then, via a call to `M`, calls the auxiliary method `N`. To generate test inputs for auxiliary method `N`, we have specified a designated call-back template, which is shown in Fig. 3.16.

The call-back template applies a number of operations to object `r`, restricting the last of these operations to be a call to a method `Op_callback`, which breaks `o`'s invariant and then starts a call chain as described above. For the above example, our technique would instantiate the call-back template such that method `L` is called last in a synthesized PUT. As a result, `N` will be tested with a receiver whose invariant does not hold (because `b` is true), but not with arbitrary inputs (because `i` is non-negative).

The algorithm for synthesizing PUTs from the call-back template is analogous to GenFieldCombs (Alg. 3.2) and GenSubCombs (Alg. 3.3) except that, on line 4, it selects `Op_callback` methods. These can be over-approximated by simply inspecting the static call graph of the UUT, which is built by our implementation and includes information about dynamically-bound calls.

An interesting side effect of the call-back template is that it improves error reporting for call-back scenarios. Assume that method `L` in Fig. 3.15 did not re-establish the invariant by setting `b` to false. Traditional invariant

```
class C {
  private int i;
  private bool b;

  invariant 0 <= i && !b;

  public void L(D d) {
    b = true;
    d.M(this);
    b = false;
  }

  public void N() {
    ...
  }
}

class D {
  public void M(C c) {
    ...
    c.N();
    ...
  }
}
```

Figure 3.15: An example of the call-back scenario.

checking, as implemented in Pex, is sufficient to detect such problems while testing `L`. It attempts to generate inputs for method `L` that violate the assertions in `L` and all methods it calls, in particular, the check of the receiver's invariant at the end of `N`. A drawback of this approach is that traditional invariant checking blames method `N` for the invariant violation although, arguably, `L` should not have started the call chain while its receiver's invari-

```
void CB(r, o, a0..aN) {
  assume r != null;
  assume o != null && o.Invariant();
  Op_0(r, ...);
  ...
  Op_M(r, ...);
  assume o.Invariant();
  Op_callback(r, ...);
  assert o.Invariant();
}
```

Figure 3.16: The call-back template.

ant was violated. Our call-back template blames `L` for any failure of the assertion of the invariant at the end of the PUT.

3.8 Related work

Our approach to testing object invariants is inspired by static verification techniques [51]. Poetzsch-Heffter [123] pointed out that the traditional way of checking the invariant of the receiver at the end of each method is insufficient. The checks he proposed are sufficient for sound verification, but not suitable as unit test oracles since they make heavy use of universal quantification. Some modular verification techniques for object invariants [99, 116] handle the challenges mentioned in Sect. 3.1, but require annotation overhead that does not seem acceptable for testing.

There are several test case generators for object-oriented programs that rely on invariants but miss the violations presented here. AutoTest [110], a random testing tool for Eiffel, follows the traditional approach of checking the invariant of the receiver at the end of each method. Pex [135] follows the same approach, but asserts the invariant of the receiver only at the end of public methods. Korat [17] and Symbolic Java PathFinder [125] do not check object invariants of the UUT at all; they use invariants only to filter test inputs. Given an invariant, Korat generates all non-isomorphic object structures, up to a small bound, for which the invariant holds. These structures are subsequently passed as input to the UUT. Symbolic Java PathFinder, built on top of the Java PathFinder model checking tool [138], obtains input constraints through user-provided or inferred specifications. All such tools may miss bugs when object invariants are violated and would, thus, benefit from our technique.

Besides testing tools, there are also static techniques to check invariants. Clousot [59] follows the traditional approach of checking the invariant of the receiver at the end of each method [32] and, thus, misses violations due to the scenarios presented here. Spec# [99] is a program verifier that handles all of these scenarios by using an elaborate methodology for enforcing invariants. However, this methodology requires a significant number of user-provided annotations, and cannot handle certain correct programs since they do not conform to the methodology of the tool.

Invariants can also be validated manually by asking programmers to review user-provided or inferred invariants. For instance, Polikarpova et al. [124] compared contracts (including invariants) that were provided by programmers with ones that were inferred using Daikon. Their results show that roughly a third of the inferred contracts were considered to be incorrect or irrelevant, while our approach rejected 89% of the inferred invariant clauses for the application in Sect. 3.6.3. This might suggest that an automatic technique is more thorough in detecting invariants that are not robust,

although the difference could also be attributed to including different benchmarks in the evaluation.

Work on synthesizing method call sequences to generate complex input data is complementary to ours. In fact, such approaches could be applied in place of the object construction mechanism in Pex to generate input objects for our PUTs. In certain cases, this might reduce the length of the synthesized tests since fewer candidate operations may be required to generate the same objects. These approaches include a combination of bounded exhaustive search and symbolic execution [140], feedback-directed random testing [122], a combination of feedback-directed random testing with concolic testing [64, 50], evolutionary testing [137], an integration of evolutionary and concolic testing [84], and source code mining [133]. Moreover, Palus [143] combines dynamic inference, static analysis, and guided random test generation to automatically create legal and behaviorally-diverse method call sequences. In contrast to existing work, our technique synthesizes code that specifically targets violations of object invariants. This allows for a significantly smaller search space restricted to three known scenarios in which invariant violations may occur.

The work on method call synthesis that is most closely related to ours is Seeker [134], an extension to Pex that combines static and dynamic analyses to construct input objects for a UUT. More specifically, Seeker attempts to cover branches that are missed by Pex. For this purpose, it statically identifies which branches, called pre-targets, should be covered first so that a missed branch is also covered. Seeker then explores the pre-targets dynamically, and eliminates those that do not actually lead to the missed branch. As a result, a feedback loop is created between the static and dynamic analyses. Even though this approach does not rely on object invariants, the negation of an invariant could be regarded as a branch to be covered. However, some of the scenarios of Sect. 3.1 are not captured by Seeker's static analysis. For example, a multi-object invariant violation involves leaking or capturing parts of an object's representation, and might not necessarily involve a sequence of missed branches. The same holds for subclassing. In addition, Seeker cannot generate input objects by calling certain methods multiple times, which may be crucial in detecting invariant violations.

3.9 Summary and remarks

We have presented a technique for detecting object invariant violations by synthesizing PUTs. Given one or more classes under test, our technique uses a set of templates to synthesize snippets of client code. We then symbolically execute the synthesized code to generate inputs that might lead to invariant violations. As a result, our technique can reveal critical defects in the UUT, which go undetected by existing testing tools. We have demonstrated the

effectiveness of our implementation by testing both programmer-provided and inferred invariants. Our experiments found a large number of invariants that can be violated.

The PUTs synthesized by our technique represent possible clients of the UUT, although not necessarily clients that are present in a given program. Therefore, the detected invariant violations might not actually manifest themselves in the program, which is to be expected for unit testing. Nevertheless, invariant violations indicate that the public interface of the UUT is not sufficiently robust. We evaluated the causes of invariant violations and fixed them for the Boogie project.

The main motivation for our work is the use of invariants to generate test inputs. By developing this technique for sufficiently testing object invariants as part of the oracle, we bring systematic testing one step closer to verification. When invariants are not soundly checked by a static analysis, our technique provides a principled alternative for subsequently evaluating their correctness. However, our technique also has other applications. For instance, one might test invariants *before* attempting to verify them statically.

Possible directions for future work are to improve the precision of the static effect analysis in order to reduce the number of generated PUTs. It would also be interesting to generalize our technique to handle static class invariants [100], which constrain static fields.

Chapter 4

Dynamic test generation with static fields and initializers

In Ch. 3, we showed how to generate test oracles for checking a certain class of rich specifications, namely, object invariants, which might have not been previously verified by a static analysis. In this chapter, we investigate how to thoroughly check existing oracles in a program under test by efficiently generating appropriate input data to the program. We specifically focus on program inputs that have so far not been taken into account by existing testing tools, namely, static fields and the potentially non-deterministic execution of static initializers.

In object-oriented programming, data stored in static fields is common and potentially shared across the entire program. In case developers choose to initialize a static field to a value different from the default value of its declared type, they typically write initialization code. The initialization code is executed by the runtime environment at *some* time prior to the first use of the static field. The time at which the initialization code is executed depends on the programming language and may be chosen non-deterministically, which makes the semantics of the initialization code non-trivial, even to experienced developers.

In C#, initialization code has the form of a static initializer, which may be inline or explicit. The C# code below shows the difference: field `f0` is initialized with an inline static initializer, and field `f1` with an explicit static initializer.

```
class C {
  // inline
  static int f0 = 19;
  static int f1;

  // explicit
  static C() {
    f1 = 23;
  }
}
```

If any static initializer exists, inline or explicit, the C# compiler always generates an explicit initializer. This compiler-generated explicit initializer first initializes the static fields of the class that are assigned their initial value with inline initializers, and then incorporates the code of the original explicit initializer (if any) written by the developer, as shown below for class `C`.

```
// compiler-generated
static C() {
  f0 = 19;
  f1 = 23;
}
```

However, the semantics of the compiler-generated static initializer depends on whether the developer has indeed written an explicit initializer. If this is the case, the compiler-generated initializer has *precise* semantics: the body of the initializer is executed (*triggered*) exactly on the first access to any (non-inherited) member of the class (that is, static field, static method, or instance constructor). Otherwise, the compiler-generated initializer has *before-field-init* semantics: the body of the initializer is executed no later than the first access to any (non-inherited) static field of the class [53]. This means that the initializer could be triggered by the runtime environment at any point prior to the first static-field access.

In Java, static (initialization) blocks are the equivalent of explicit static initializers with precise semantics in C# [76]. In C++, static initialization occurs before the program entry point in the order in which the static fields are defined in a single translation unit. However, when linking multiple translation units, the order of initialization between the translation units is undefined [52].

Even though static state is common in object-oriented programs and the semantics of static initializers is non-trivial, automatic test case generators do not take into account the potential interference of static state with a unit under test. They may, thus, miss subtle errors. In particular, existing test case generators do not solve the following issues:

1. *Static fields as input*: When a class is initialized before the execution of the unit under test, the values of its static fields are part of the state and should, thus, be treated as inputs to the unit under test. Existing tools fail to do that and may miss bugs when the unit under test depends on the values stored in static fields (for instance, to determine control flow or evaluate assertions).

2. *Initialization and uninitialization*: Existing testing tools do not control whether static initializers are executed before or during the execution of the unit under test. The point at which the initializer is executed may affect the test outcome since it may affect the values of static fields

and any other variables assigned to by the static initializer. Ignoring this issue may cause bugs to be missed. A related issue is that existing tools do not undo the effect of a static initializer between different executions of the unit under test such that the order of executing tests may affect their outcomes [142].

3. *Eager initialization*: For static initializers with before-field-init semantics, a testing tool should not only control whether the initializer is run before or during test execution; in the latter case, it also needs to explore all possible program points at which initialization of a class may be triggered (non-deterministically).

4. *Initialization dependencies*: The previous issues are further complicated by the fact that the order of executing static initializers may affect the resulting state due to their side effects. Therefore, a testing tool needs to consider all relevant execution orders so as not to miss bugs.

We address these issues by designing and implementing a novel technique in automatic test case generation based on dynamic symbolic execution [70, 20] and static analysis. Our technique treats static fields as input to the unit under test and systematically controls the execution of static initializers. The dynamic symbolic execution collects constraints describing the static-field inputs that will cause the unit under test to take a particular branch in the execution or violate an assertion. It also explores the different program points at which a static initializer might be triggered. The static analysis improves performance by pruning program points at which the execution of a static initializer does not lead to any new behaviors of the unit under test.

We have implemented [55] our technique as an extension to the testing tool Pex [135] for .NET. We have applied it on a suite of open-source applications and found errors that go undetected by existing test case generators. Our results show that this problem is relevant in real code, indicate which kinds of errors existing techniques miss, and demonstrate the effectiveness of our technique.

Outline. This chapter is organized as follows. Sect. 4.1 explains how we explore static input state where all relevant classes are initialized. Sects. 4.2 and 4.3 show how we handle static initializers with precise and before-field-init semantics, respectively. Sect. 4.4 demonstrates the effectiveness of our technique by applying it on a suite of open-source applications, and Sect. 4.5 reviews related work.

4.1 Static fields as input

In this section, we address the issue of treating static fields of *initialized* classes as input to the unit under test. The case that a class is not yet

```
1  public class C {
2    public static int F;
3
4    static C() {
5      F = 0;
6    }
7
8    public static void M() {
9      F++;
10     if (F == 2) abort;
11   }
12 }
```

Figure 4.1: A C# method accessing static state. For all branches to be covered, dynamic symbolic execution must treat static field `F` as an input to method `M` and collect constraints on its value.

initialized is discussed in the next two sections.

The example in Fig. 4.1 illustrates the issue. Existing automatic test case generators do not treat static field `F` of class `C` as input to method `M`. In particular, testing tools based on dynamic symbolic execution generate only one unit test for method `M` since there are no branches on a method parameter or receiver of `M`. Given that the body of method `M` contains a branch on static field `F` (line 10), these tools achieve low code coverage of `M` and potentially miss bugs, like the abort-statement in this example.

Dynamic symbolic execution

To address this issue, we treat static fields as inputs to the method under test and assign to them symbolic variables. This causes the dynamic symbolic execution to collect constraints on the static fields, and use these constraints to generate inputs that force the execution to explore all branches in the code.

Treating *all* static fields of a program as inputs is not practical. It is also not modular and defeats the purpose of unit testing. Therefore, we determine at runtime which static fields are read during the execution of a unit test and treat only those as inputs to the unit under test.

We implement this approach in a function DSE(UUT, IC), which performs dynamic symbolic execution of the unit under test UUT. IC is the set of classes that have been initialized *before* the execution of the unit under test. For all other classes, initialization may be triggered *during* the execution of the generated unit tests. The DSE function treats the static fields of all classes in the IC set as symbolic inputs. It returns the set TC of classes whose initialization is triggered before or during the execution of

the generated unit tests, that is, TC is a superset of IC. The static fields of the classes in TC include all static fields that are read by the unit tests. We call the DSE function repeatedly to ensure that the static fields of all of these classes are treated as inputs to the unit under test. The precise algorithm for this exploration as well as more details of the DSE function are described in the next section.

Consider the dynamic symbolic execution DSE(`M`, {}) of method `M` from Fig. 4.1. This dynamic symbolic execution generates one unit test that calls method `M`. The execution of this unit test triggers the initialization of class `C` due to the access to static field `F` (line 9). Therefore, function DSE returns the singleton set {`C`}. As a result, our exploration algorithm will call DSE(`M`, {`C`}). This second dynamic symbolic execution treats static field `F` as a symbolic input to method `M` and collects constraints on its value. For instance, assuming that the first unit test of the second dynamic symbolic execution executes `M` in a state where `F` is zero, the conditional statement on line 10 introduces the symbolic constraint $\neg(F + 1 = 2)$, where F is the symbolic variable associated with `F`. The dynamic symbolic execution subsequently negates and solves the symbolic constraints on `M`'s inputs. Consequently, a second unit test is generated that first assigns the value one to field `F` and then calls `M`. The second unit test now reaches the abort-statement and reveals the bug. We will see in the next section that, even though the second call to DSE is the one that explores the unit under test for different values of static field `F`, the first call to DSE is also important; besides determining which static fields should be treated symbolically, it is also crucial in handling uninitialized classes.

Discussion of correctness guarantees

As usual with the automatic generation of unit tests, the generated static-field inputs might not occur in any actual execution of the entire program; to avoid false positives, developers may write specifications (preconditions or invariants) that further constrain the possible values of these inputs.

Determining at runtime which static fields to treat as inputs depends on the soundness of the underlying dynamic symbolic execution engine of the DSE function. For example, when the dynamic symbolic execution engine cannot generate inputs that satisfy a particular constraint, or when all paths in a unit under test cannot be explored in a reasonable amount of time, we might fail to consider as inputs all static fields read by the unit under test.

4.2 Initialization with precise semantics

In the previous section, we addressed the issue of treating static fields of initialized classes as input to the unit under test. In this section, we explain how our technique (1) controls the execution of static initializers and

(2) explores executions that trigger static initializers. Here, we consider only static initializers with precise semantics; initializers with before-field-init semantics are discussed in the next section.

4.2.1 Controlling initialization

In order to explore the interaction between a unit under test and static initializers, we must be able to control for each unit test which classes are initialized before its execution and which ones are not. This could be achieved by restarting the runtime environment (virtual machine) before each execution of a unit test and then triggering the initialization of certain classes. To avoid the high performance overhead of this naïve approach, we instrument the unit under test such that the execution *simulates* the effects of triggering an initializer and restarting the runtime environment.

Initialization

We insert calls to the dynamic symbolic execution engine at all points in the entire program where a static initializer could be triggered according to its semantics. For static initializers with precise semantics, we insert instrumentation calls to the dynamic symbolic execution engine on the first access to any (non-inherited) member of their class. Where to insert these instrumentation calls is determined using the inter-procedural control-flow graph of the unit under test. This means that we might insert an instrumentation call at a point in the code where, along certain execution paths, the corresponding class has already been initialized. Note that each .NET bytecode instruction triggers at most one static initializer; therefore, there is at most one instrumentation call at each program point.

For an exploration DSE(UUT, IC), the instrumentation calls in UUT have the following effect. If the instrumentation call is made for a class C that is in the IC set, then C has already been initialized before executing UUT and, thus, the instrumentation call has no effect. Otherwise, if this is the first instrumentation call for C in the execution of this unit test, then we use reflection to explicitly invoke C's static initializer. That is, we execute the static initializer no matter if the runtime environment has initialized C during the execution of a previous unit test or not. Moreover, we add class C to the TC set of classes returned by function DSE. If the same unit test has already initialized C during its execution, the instrumentation call has no effect.

In method `M` from Fig. 4.1, we add instrumentation calls for class `C` before the two accesses to static field `F`, that is, between lines 8 and 9 and between lines 9 and 10. (Our implementation omits the second instrumentation call in this example, but this is not always possible for methods with more interesting control flow.) Consider again the exploration DSE(`M`, {}).

During the execution of the generated unit test, the instrumentation call at the first access to static field `F` calls `C`'s static initializer such that the unit test continues with `F` = 0. The instrumentation call for the second access to `F` has no effect since this unit test has already initialized class `C`. DSE returns the set {`C`}, as described above.

Note that an explicit call to a static initializer is itself an access to a class member and, thus, causes the runtime environment to trigger another call to the same initializer. To prevent the initializer from executing twice (and thereby duplicating its side effects), we instrument each static initializer such that its body is skipped on the first call, as shown below.

```
static C() {
  if (/* this is the first call */)
    return;
  // body of original static initializer
}
```

This instrumentation decouples the execution of a unit test from the initialization behavior of the runtime environment. Static initializers triggered by the runtime environment have no effect and, thus, do not actually initialize the classes, whereas our explicit calls to static initializers initialize the classes even in cases where the runtime environment considers them to already be initialized.

Uninitialization

To avoid the overhead of restarting the runtime environment after each execution of a unit test, we simulate the effect of a restart through code instrumentation. Since our technique does not depend on the behavior of the runtime environment to control class initialization, we do not have to actually uninitialize classes. It is sufficient to reset the static fields of all classes initialized by the unit under test to the default values of their declared types after each execution of a unit test. Therefore, the next execution of the static initializer during the execution of the next unit test behaves as if it ran on an uninitialized class.

Existing automatic test case generators (with the exception of the random testing tool JCrasher [42] for Java) do not reset static fields to their initial values between test runs. For code like in Fig. 4.1, Pex emits a warning that the unit under test might not leave the dynamic symbolic execution engine in a clean state. Therefore, the deterministic re-execution of the generated unit tests is not guaranteed. In fact, the Pex documentation suggests that the tester should mock all interactions of the unit under test with static state. However, this requires the tester to be aware of these interactions and renders Pex significantly less automatic.

4.2.2 Dynamic symbolic execution

The core idea of our exploration is as follows. Assume that we knew the set *classes* of all classes whose initialization may be triggered by executing the unit under test UUT. For each subset $IC \subseteq classes$, we perform dynamic symbolic execution of UUT such that the classes in IC are initialized before executing UUT and their static fields are symbolic inputs. The classes in $classes \setminus IC$ are not initialized (that is, their initializers may be triggered when executing a unit test). We can then explore all possible initialization behaviors of UUT by testing it for each possible partition of *classes* into initialized and uninitialized classes.

Algorithm

Alg. 4.1 is a dynamic symbolic execution algorithm that implements this core idea, but also needs to handle the fact that the set of relevant classes is not known upfront, but determined during the execution. Function EXPLORE takes as argument a unit under test UUT, which has been instrumented as described above. Local variable *classes* is the set of relevant classes determined so far, while local variable *explored* is the set of sets of classes that have been treated as initialized in the exploration so far; that is, *explored* keeps track of the partitions that have been explored. As long as there is a partition that has not been explored (that is, a subset IC of *classes* that is not in *explored*), the algorithm picks any such subset and calls the dynamic symbolic execution function DSE, where classes in IC are initialized and their static fields are treated symbolically. If this function detects any classes that are initialized during the dynamic symbolic execution, they are added to *classes*. The EXPLORE function terminates when all possible subsets of the relevant classes have been explored.

Algorithm 4.1: Dynamic symbolic execution for exploring the interactions of a unit under test with static state.

```
1 function EXPLORE(UUT)
2   classes ← {}
3   explored ← {}
4   while ∃IC ⊆ classes · IC ∉ explored do
5     IC ← choose({IC | IC ⊆ classes ∧ IC ∉ explored})
6     TC ← DSE(UUT, IC)
7     classes ← classes ∪ TC
8     explored ← explored ∪ {IC}
```

Initialization dependencies

Alg. 4.1 enumerates all combinations of initialized and uninitialized classes in the input state of the method under test, that is, all possible partitions of *classes* into IC and $\mathit{classes} \setminus IC$. This includes combinations that cannot occur in any actual execution. If the static initializer of a class E triggers the static initializer of a class D, then there is no input state in which E is initialized, but D is not. To avoid such situations and, thus, false positives during testing, we trigger the static initializers of all classes in IC before invoking the method under test. In the above example, this ensures that both E and D are initialized in the input state of the method under test, and D's initializer is not triggered during the execution of the method. Since the outcome of running several static initializers may depend on the order in which they are triggered, we explore all orders among dependent static initializers. We determine dependent initializers with a static analysis that approximates their read and write effects. This analysis is described in more detail in the following section.

Triggering the static initializers of the classes in IC happens at the beginning of the set-up code that precedes the invocation of the method under test in every generated unit test. This set-up code is also responsible for creating the inputs for the method under test, for instance, for allocating objects that will be passed as method arguments. Therefore, the set-up code may itself trigger static initializers, for instance, when a constructor reads a static field. To handle the dependencies between set-up code and initialization, we treat set-up code as a regular part of the unit test (like the method under test itself), that is, apply the same instrumentation and explore all possible execution paths during dynamic symbolic execution.

Handling dependencies between static initializers is particularly useful in C++, where static initialization happens before the program entry point. When linking multiple translation units, the order of initialization between the translation units is undefined. By exploring all orders of execution of dependent initializers, developers can determine dependencies that may crash a program before its entry point is even reached.

Example

The example in Fig. 4.2 illustrates our approach. The assertion on line 13 fails only if `N` is executed in a state in which class `D` is initialized (such that the if-statement may be executed), the static field `G` is negative (such that the if-statement will be executed and `E`'s initialization will be triggered), and class `E` is not initialized (such that its static initializer will affect the value of `G`).

We will now explain how Alg. 4.1 reveals such subtle bugs. In the first iteration, IC is the empty set, that is, no class is considered to be initialized.

```
1  public class D {
2    public static int G;
3
4    static D() {
5      G = 0;
6    }
7
8    public static void N() {
9      if (G < 0) {
10       E.H++;
11       G = -G;
12     }
13     assert 0 <= G;
14   }
15 }
16
17 public class E {
18   public static int H;
19
20   static E() {
21     H = 0;
22     D.G = 1;
23   }
24 }
```

Figure 4.2: A C# example illustrating the treatment of static initializers with precise semantics. We use the assert-keyword to denote Code Contracts [58] assertions. The assertion on line 13 fails only if `N` is called in a state where `D` is initialized, but `E` is not.

Therefore, when the DSE function executes method `N`, class `D` is initialized right before line 9. Consequently, static field `G` is zero, the if-statement is skipped, and the assertion holds. DSE returns the set {`D`}.

In the second iteration, *IC* will be {`D`}, that is, the static initializer of class `D` is triggered by the set-up code, and static field `G` is treated symbolically. Since there are no constraints on the value of `G` yet, the dynamic symbolic execution executes method `N` with an arbitrary value for `G`, say, zero. (Note that when there are no constraints on a static-field input, our implementation does not assign an arbitrary value to the static field, but retains the value assigned to the field by the static initializer of its class.) This unit test passes and produces the constraint $G < 0$ for the next unit test, where G is the symbolic variable associated with `G`. For any such value of `G`, the unit test will now enter the if-statement and initialize class `E` before the access to `E`'s static field `H`. This initialization assigns one to `G` such that the subsequent negation makes the assertion fail, and the bug is detected.

The call to the DSE function in the second iteration returns {D, E}.

The two remaining iterations of Alg. 4.1 cover the cases in which *IC* is {E} or {D, E}. The former case illustrates how we handle initialization dependencies. The static initializer of class E accesses static field G of class D. Therefore, when E's initializer is called by the set-up code of the generated unit test, D's initializer is also triggered. (Recall that the set-up code and all static initializers are instrumented like the method under test). This avoids executing N in the impossible situation where E is initialized, but D is not. The rest of this iteration is analogous to the first iteration, that is, class D gets initialized (this time while executing the set-up code), the if-statement is skipped, and the assertion holds.

Finally, for $IC = \{\texttt{D}, \texttt{E}\}$, all relevant classes are initialized. The dynamic symbolic execution will choose negative and non-negative values for G. The assertion holds in either case.

Discussion of correctness guarantees

Similarly to the previous section, our approach might generate unit tests for executions that do not occur in the entire program. For instance, in the context of the entire program, the static initialization of a particular class could always be triggered before the execution of a unit under test, and never during its execution. To avoid false positives, developers could write a new kind of specifications (e.g., in the form of preconditions) that express which classes are required to be initialized before the execution of a unit under test.

Determining at runtime the set of all classes whose initialization may be triggered by executing the unit under test depends on the soundness of the underlying dynamic symbolic execution engine of the DSE function, as discussed in the previous section for treating static fields as inputs.

Applications

Alg. 4.1 can be implemented in any testing tool based on dynamic symbolic execution. We have implemented it in Pex, whose existing dynamic symbolic execution engine is invoked by our DSE function. Alg. 4.1 could also be implemented in jCUTE [127] for testing how static fields and static blocks in Java interact with a unit under test. Moreover, this algorithm can be adjusted to perform all dynamic tasks statically for testing tools based on static symbolic execution. For instance, Symbolic Java PathFinder [125] could then be extended to take static state into account.

By treating static fields symbolically, our technique gives meaning to specifications that refer to static fields, like assertions or preconditions. For example, an assertion about the value of a static field is now treated as a branch by the symbolic execution. One could also support preconditions

that express which classes are required to be initialized before the execution of a method.

As part of the integration with unit testing frameworks, like NUnit for .NET and JUnit for Java, many automatic test case generators support defining set-up methods for a unit under test. Such methods allow testers to initialize and reset static fields manually. Since set-up methods might express preconditions on static fields (in the form of code), we extended our technique not to override the functionality of these methods. That is, when a set-up method assigns to a static field of a class C, we do not trigger the initialization of class C and do not treat its static fields symbolically. We do, however, reset the values of all static fields in class C after each execution of a unit test such that the next execution of the set-up method starts in a fresh state.

This technique could also be used in existing unit testing frameworks for detecting whether a set-up method allows for any static fields to retain their values between runs of the unit under test. This is achieved by detecting which static fields are modified in the unit under test but have not been manually set up. If such fields exist, an appropriate warning could be emitted by the framework.

4.3 Initialization with before-field-init semantics

The technique presented in the previous section handles static initializers with precise semantics. Static initializers with before-field-init semantics, which may be triggered at any point before the first access to a static field of the class, impose two additional challenges. First, they introduce nondeterminism because the static initializer of any given class may be triggered at various points in the unit under test. Second, in addition to the classes that have to be initialized in order to execute the unit under test, the runtime environment could in principle choose to trigger any other static initializer with before-field-init semantics, even initializers of classes that are completely unrelated with the unit under test. In this section, we describe how we solve these challenges. Our solution uses a static analysis to determine the program points at which the execution of a static initializer with before-field-init semantics may affect the behavior of the unit under test. Then, we use a modified dynamic symbolic execution function to explore each of these possibilities.

As the running example of this section, consider method `P` in Fig. 4.3. The static initializer of class `D` has before-field-init semantics and must be executed before the access to field `D.Fd` on line 10. If the initializer runs on line 5 or 9, then the assertion on line 8 succeeds. If, however, the initializer runs on line 7, the assertion fails because the value of field `C.Fc` has been incremented (line 15) and is no longer equal to two. This bug indicates

```
1  public static class C {
2    static int Fc = 0;
3
4    public static void P() {
5      // static initializer of 'D'
6      Fc = 2;
7      // static initializer of 'D'
8      assert Fc == 2;
9      // static initializer of 'D'
10     if (D.Fd == 3)
11       Fc = E.Fe;
12   }
13
14   static class D {
15     public static int Fd = C.Fc++;
16   }
17
18   static class E {
19     public static int Fe = 11;
20   }
21 }
```

Figure 4.3: A C# example illustrating the non-determinism introduced by static initializers with before-field-init semantics. The assertion on line 8 fails if `D`'s static initializer is triggered on line 7.

that the unit under test is affected by the non-deterministic behavior of a static initializer with before-field-init semantics. Such errors are particularly difficult to detect with standard unit testing since they might not manifest themselves reproducibly.

Critical points

A static initializer with before-field-init semantics may be triggered at any point before the first access to a static field of its class. To reduce the non-determinism that needs to be explored during testing, we use a static analysis to determine the *critical points* in a unit under test, that is, those program points where triggering a static initializer might actually affect the execution of the unit under test. All other program points can be ignored during testing because no new behavior of the unit under test will be exercised.

A critical point is a pair consisting of a program point i and a class C. It indicates that there is an instance or static field f that is accessed both by the instruction at program point i and the static initializer of class C such that the instruction, or the static initializer, or both modify the field. In other words, a critical point indicates that the overall effect of executing

the static initializer of C and the instruction at i depends on the order in which the execution takes place. Moreover, a pair (i, C) is a critical point only if program point i is not dominated in the control-flow graph by an access to a static field of C, that is, if it is possible to reach program point i without first initializing C.

In the example of Fig. 4.3, there are five critical points: $(6, \texttt{C})$, $(6, \texttt{D})$, $(8, \texttt{D})$, $(10, \texttt{D})$, and $(11, \texttt{E})$, where we denote program points by line numbers. Note that even though the static initializer of class `E` could be triggered anywhere before line 11, there is only one critical point for `E` because the behavior of method `P` is the same for all these possibilities.

We determine the critical points in a method under test in two steps. First, we use a simple static analysis to compute, for each program point i, the set of classes with before-field-init initializers that might get triggered at point i. This set is denoted by *prospectiveClasses*(i). In principle, it includes all classes with before-field-init initializers in the entire program, except those that are definitely triggered earlier, that is, before i. Since it is not feasible to consider all of them during testing, we focus on those classes whose static fields are accessed by the method under test. This is not a restriction in practice: even though the Common Language Infrastructure *standard* [53] allows more initializers to be triggered, the Common Language Runtime *implementation*, version 4.0, triggers the initialization of exactly the classes whose static fields are accessed by the method. Therefore, in Fig. 4.3, *prospectiveClasses*(8) is the set $\{\texttt{D}, \texttt{E}\}$.

Second, we use a static analysis to determine for each program point i and class C in *prospectiveClasses*(i) whether (i, C) is a critical point. For this purpose, the static analysis approximates the read and write effects of the instruction at program point i and of the static initializers of all classes in *prospectiveClasses*(i). Recall from the previous chapter that the *read effect* of a statement is the set of fields read by the statement or by any method the statement calls directly or transitively. Analogously, the *write effect* of a statement is the set of fields written by the statement or by any method the statement calls directly or transitively. The pair (i, C) is a critical point if (1) i's read effect contains a (static or instance) field f that is included in the write effect of C's static initializer, or (2) i's write effect contains a (static or instance) field f that is included in the read or write effect of C's static initializer. For instance, for line 8 of our example, $(8, \texttt{D})$ is a critical point because the statement on line 8 reads field `Fc`, which is written by the static initializer of class `D`, and `D` is in *prospectiveClasses*(8). However, even though class `E` is in *prospectiveClasses*(8), $(8, \texttt{E})$ is not a critical point because the effects of the statement on line 8 and of `E`'s static initializer are disjoint.

As in the previous chapter, read and write effects are sets of fully-qualified field names, which allows us to approximate them without requiring alias information. Our static effect analysis is inter-procedural. It explores

the portion of the whole program it can access (in particular, the entire assembly of the method under test) to compute a call graph that includes information about dynamically-bound calls.

A critical point (i, C) indicates that the dynamic symbolic execution should trigger the initialization of class C right before program point i. However, C's static initializer might lead to more critical points, because its effects may overlap with the effects of other static initializers and because it may trigger the initialization of additional classes, which, thus, must be added to *prospectiveClasses*. To handle this interaction, we iterate over all options for critical points and, for each choice, inline the static initializer and recursively invoke our static analysis.

Dynamic symbolic execution

We instrument the unit under test to include a marker for each critical point (i, C). We enhance the DSE function called from Alg. 4.1 to trigger the initialization of class C when the execution hits such a marker. If there are several markers for one class, the DSE function explores all paths of the unit under test for each possible option. Conceptually, one can think of adding an integer parameter n_C to the unit under test and interpreting the n-th marker for class C as a conditional statement `if` $(n_C == n)$ { $\texttt{init}_C$ }, where $\texttt{init}_C$ calls the static initializer of class C if it has not been called earlier during the execution of the unit test. Dynamic symbolic execution will then explore all options for the initialization of a class C by choosing different values for the input n_C.

Since $(8, \texttt{D})$ is a critical point in our example, DSE will trigger the initialization of class `D` right before line 8 during the symbolic execution of method `P`. As a result, the assertion violation is detected.

Discussion of correctness guarantees

Our technique for handling static initializers with before-field-init semantics may yield irrelevant critical points (for instance, when an instruction and a static initializer both have an instance field f in their effects, but at runtime, access f of different objects) and, thus, produce redundant unit tests.

Our technique may also miss critical points (for instance, when it fails to consider a method override in an assembly that is not accessible to the static analysis) and, thus, the generated unit tests might not explore all possible behaviors.

As a final remark, note that, even though errors detected with this technique are possible according to the Common Language Infrastructure standard, they may manifest themselves very rarely in practice. This might also

be the case for certain classes of concurrency errors, but does not degrade the value of techniques for their detection.

4.4 Experimental evaluation

We have evaluated the effectiveness of our technique on 30 open-source applications written in C#. These applications were arbitrarily selected from applications on Bitbucket, CodePlex, and GitHub. Our suite of applications contains a total of 423,166 methods, 47,515 (11%) of which directly access static fields. All classes of these applications define a total of 155,632 fields (instance and static), 28,470 (18%) of which are static fields; 14,705 of the static fields (that is, 9% of all fields) are static read-only fields. There is a total of 1,992 static initializers, 1,725 (87%) of which have precise semantics, and 267 (13%) of which have before-field-init semantics.

To determine which of the 47,515 methods that directly access static fields are most likely to have bugs, we implemented a lightweight scoring mechanism. This mechanism statically computes a score for each method and ranks all methods by their score. The score for each method is based on vulnerability and accessibility scores. The *vulnerability score* of a method indicates whether the method directly accesses static fields and how likely it is to fail at runtime because of a static field, for instance, due to failing assertions, or division-by-zero and arithmetic-overflow exceptions involving static fields. This score is computed based on nesting levels of expressions and how close a static field is to an operation that might throw an exception. The *accessibility score* of a method indicates how accessible the method and the accessed static fields are from potential clients of the application. In particular, this score indicates the level of accessibility from the public interface of the application, and suggests whether a potential bug in the method is likely to be reproducible by clients of the application. The final score for each method is the product of its vulnerability and accessibility scores.

To compare the number of errors detected with and without our technique, we ran Pex with and without our implementation on all methods with a non-zero score. There were 454 methods with a non-zero score in the 30 applications. Tab. 4.1 summarizes the results of our experiments on the applications in which bugs were detected. The first column of the table shows the name of each application. The second column shows the total number of methods with a non-zero score for each application. The two rightmost columns of the table show the number of errors that our technique detected in these methods. These errors do not include errors already detected by Pex without our technique; they are all caused by interactions of the methods under test with static state.

More specifically, column “init” shows the number of errors detected by

Application	**Number of methods**	**Number of errors**	
		init	init&inputs
Boggle[1]	60	-	24
Boogie[2]	21	-	6
Ncqrs[3]	38	1	1
NRefactory[4]	37	-	9
Scrabble[5]	64	-	2
Total	220	1	42

Table 4.1: Summary of our experiments. The first column shows the name of each application. The second column shows the total number of tested methods from each application. The two rightmost columns show the number of errors detected without and with treating static fields as inputs to the unit under test, respectively.

simply triggering static initializers at different points in the code. These errors are, thus, caused by calling static initializers (with both semantics) before or during the execution of the unit tests without treating static fields as inputs. Column "init&inputs" shows the number of errors detected by our technique, that is, by treating static fields symbolically and systematically controlling the execution of static initializers.

As shown in the last column of the table, our technique detected 42 bugs that are not found by Pex. Related work suggests that existing test case generators would not find these bugs either (see Sect. 4.5). A failed unit test does not necessarily mean that the application actually contains code that exhibits the detected bug; this uncertainty is inherent to unit testing since methods are tested in isolation rather than in the context of the entire application. However, all of the detected bugs may surface during maintenance or code reuse. In particular, for 25 of the 42 detected bugs, both the buggy method and the accessed static fields are public. Therefore, when the applications are used as libraries, client code can easily exhibit these bugs.

We have also manually inspected static initializers from all 30 applications and distilled their three most frequent usage patterns. Static initializers are typically used for:

1. Initializing static fields of the same class to constants or simple computations; these initializers are often inline initializers, that is, have

[1] http://boggle.codeplex.com, rev: 20226
[2] http://boogie.codeplex.com, rev: e80b2b9ac4aa
[3] http://github.com/ncqrs/ncqrs, rev: 0102a001c2112a74cab906a4bc924838d7a2a965
[4] http://github.com/icsharpcode/NRefactory, rev: ae42ed27e0343391f7f30c1ab250d729fda9f431
[5] http://wpfscrabble.codeplex.com, rev: 20226

before-field-init semantics. However, since they neither read static fields of other classes nor have side effects besides assigning to the static fields of their class, the non-determinism of the before-field-init semantics does not affect program execution.

2. Implementing the singleton pattern in a lazy way; these initializers typically have precise semantics.

3. Initializing public static fields that are mutable; these fields are often meant to satisfy invariants such as non-nullness. However, since the fields are public, these invariants can easily be violated by client code or during maintenance. This pattern is especially susceptible to static-field updates after the initialization, a scenario that we cover by treating static fields as inputs of the unit under test.

In none of these common usage patterns do initializers typically have side effects besides assigning to static fields of their class. This might explain why we did not find more bugs that are caused by static initialization alone (column "init" in Tab. 4.1); it is largely irrelevant when such initializers are triggered.

An interesting example of the third pattern was found in application Boggle, which uses the Caliburn.Micro library. This library includes a public static field `LogManager.GetLog`, which is initialized by `LogManager`'s static initializer to a non-null value. `GetLog` is read by several other static initializers, for instance, the static initializer of class `Coroutine`, which assigns the value of `GetLog` to a static field `Log`. If client code of the Caliburn.Micro library assigned null to the public `GetLog` field before the initialization of class `Coroutine` is triggered, the application might crash; `Coroutine` will then initialize `Log` with the null value, which causes a null-pointer exception when `Coroutine`'s `BeginExecute` method dereferences `Log`. Our technique reveals this issue when testing `BeginExecute`; it explores the possibility that `LogManager` is initialized before `BeginExecute` is called whereas `Coroutine` is not, and it treats `GetLog` as an input to `BeginExecute` such that the dynamic symbolic execution will choose null as a possible value. Note that this issue is indeed an initialization problem. Since `Coroutine.Log` is not public, a client could not cause this behavior by assigning null directly to `Log`.

4.5 Related work

Most existing automatic test case generation tools ignore the potential interactions of a unit under test with static state. These tools range from random testing (like JCrasher [42] for Java), over feedback-directed random testing (like Randoop [122] for Java), to symbolic execution (like Symbolic

Java PathFinder [125]) and dynamic symbolic execution (like Pex for .NET or jCUTE [127] for Java).

To the best of our knowledge, existing testing tools such as the above do not take into account the interference of static state with a unit under test, with the exception of JCrasher. JCrasher ensures that each test runs on a "clean slate"; it resets all static state initialized by any previous test runs either by using a different class loader to load each test, or by rewriting the program under test at load time to allow re-initialization of static state. Nevertheless, JCrasher does not address the four issues described at the beginning of this chapter.

Unit testing frameworks, like NUnit for .NET and JUnit for Java, require the tester to manage static state manually in set-up methods in order to ensure the clean execution of the unit tests. Therefore, the tester must be aware of all interactions of the unit under test with static state. As a result, these frameworks become significantly less automatic for unit tests that interact with static state.

Static analyzers for object-oriented languages, such as Clousot [59] for .NET and ESC/Java [62] for Java, do not reason about static initialization. An extension of Spec# [100] supports static verification in the presence of static initializers, but requires significant annotation overhead.

We are, therefore, not aware of any tool that automatically takes static state into account and detects the kinds of errors described in this chapter.

4.6 Summary and remarks

To automatically check the potential interactions of static state with a unit under test and thoroughly evaluate its oracles, we have proposed a novel technique in automatic test case generation based on static analysis and dynamic symbolic execution. Our technique treats static fields as input to the unit under test and systematically controls the execution of static initializers. We have implemented this technique as an extension to Pex and used it to detect errors in open-source applications. Our results shed light on which kinds of errors are currently missed by existing automatic test case generators, demonstrate the effectiveness of our technique, and bring us a step closer to our goal of automatically providing evidence on whether a program is correct.

As future work, one could prune redundant explorations more aggressively; this is promising since our evaluation suggests that many static initializers have very small read and write effects and, thus, very limited interactions with the unit under test.

Chapter 5

Delfy: Dynamic test generation for Dafny

In Ch. 2, we presented the benefits of effectively combining an unsound static analysis with systematic testing. Here, we discuss how beneficial it is to integrate a dynamic test generation tool into a sound and interactive verifier. More specifically, we present Delfy, a dynamic test generation tool for the Dafny programming language. In addition to handling advanced constructs of the language, Delfy is designed both to complement and be complemented by the Dafny verifier.

When complementing the verifier, Delfy can reduce the effort of debugging spurious verification errors. In particular, take an assertion that Dafny fails to verify. If the assertion is indeed failing and Delfy terminates in a sound search, then a concrete counterexample is generated. If, however, Delfy's exploration is sound and no counterexample is generated, then the verification error for that assertion is definitely spurious.

When being complemented by the verifier, Delfy can reduce the effort of trying to verify an incorrect program. Users of verifiers typically start by writing the most general specifications first, for instance, a method postcondition. They subsequently attempt to verify these general specifications by providing more information, such as loop invariants or assertions in the method body, which serve as auxiliary specifications for making a proof go through. In such common situations, Delfy can provide early feedback as to whether the original, general specifications hold in the first place. Moreover, in case Delfy terminates in a sound search without violating these specifications, they have been proven correct. Consequently, all auxiliary specifications are no longer necessary in order to prove, say, a method postcondition, thus reducing the annotation overhead for the user. Not only does this increase user productivity, but also speeds up the verification time by reducing the number of proof obligations for the verifier.

Outline. This chapter is organized as follows. In Sect. 5.1, we briefly present Dafny, and in Sect. 5.2, we give an overview of Delfy and explain how it handles interesting language and specification constructs of Dafny. Sects. 5.3 and 5.4 discuss the benefits of integrating a dynamic test genera-

tion tool into a verifier, and Sect. 5.5 describes the usage environment of this integration. We compare Delfy to a static verification debugger in Sect. 5.6, and we discuss related work in Sect. 5.7.

5.1 Dafny

Dafny [98] is a programming language with built-in specification constructs to support sound static verification. The Dafny program verifier is powered by the Boogie verifier [11] and the Z3 constraint solver [47], and targets functional correctness of programs.

Dafny is an imperative, sequential, class-based programming language that builds in specification constructs, including pre- and postconditions, frame specifications (that is, read and write sets), loop invariants, and termination metrics. To allow more expressive specifications, the language also offers updatable ghost variables, non-deterministic statements, uninterpreted functions, and types like sets and sequences. The Dafny compiler produces C# code but omits all specifications and ghost state, which are used only during verification.

The Dafny verifier is preferably run in an integrated development environment (IDE) [103, 104], which extends Microsoft Visual Studio. The verifier runs in the background of the IDE while the programmer is editing the program. Consequently, as soon as the verifier produces errors, the programmer may respond by changing the program accordingly to achieve a proof of functional correctness.

5.2 Dynamic test generation for Dafny

Delfy implements [56, 132] systematic dynamic test generation as shown in Alg. 1.1 of Sect. 1.1, with the only difference that the concrete and symbolic executions of a unit under test happen simultaneously. In particular, we have created a new compiler from Dafny directly to .NET bytecode, which is much more efficient than the existing Dafny-to-C# compiler. The new compiler inserts call-backs to Delfy in the compiled code. These call-backs pass to Delfy the Dafny code that should be executed symbolically, that is, the symbolic execution happens on the Dafny level. In other words, given a Dafny unit under test, Delfy compiles the code into .NET bytecode and runs the compiled unit under test. The symbolic execution in Delfy runs whenever a call-back occurs in the code, and all constraints are solved with Z3. Delfy implements various exploration strategies, namely, depth-first, breadth-first, and generational-search strategies.

As we mentioned in Sect. 5.1, the existing Dafny compiler omits all specifications, which are used only during verification. For this reason, we extended the support of our new compiler to translate Dafny specifications

into Code Contracts [58], including loop invariants, termination metrics, pre- and postconditions, assumptions, assertions, and old-expressions. (Old-expressions can be used in postconditions, and stand for expressions evaluated on entry to a method.) The Code Contracts library methods are called on the bytecode level.

Delfy checks frame specifications, that is, read or write sets, by determining whether every object, which is read or modified in a function[1] or method body and is not newly allocated, is in the specified read or write set. Fresh-expressions, which are allowed in method postconditions, say that all non-null objects specified in the expression must be allocated in the method body. Delfy checks fresh-expressions by determining whether the objects actually allocated in the method body include those specified in the fresh-expression of the postcondition.

Dafny has support for non-deterministic assignments, non-deterministic if-statements, and non-deterministic while-statements. A `*` in Dafny denotes the non-deterministic value, as in the assignment `var n := *`. For each non-deterministic value, the symbolic execution in Delfy introduces a fresh symbolic variable, that is, a new input to the unit under test. Consequently, the symbolic execution collects constraints on such variables, and the constraint solver generates inputs for them, such that execution is guided toward all those unexplored program paths that are guarded by the non-determinism.

To be more expressive, Dafny also supports uninterpreted functions and assign-such-that-statements. Delfy handles uninterpreted functions again by introducing a fresh symbolic variable for their return value, which is however constrained by an assumed condition (see Sect. 1.1), of the form $\textsc{Assume}(c)$, saying that the return value must satisfy the function specifications. Recall that, when an assumed condition is added to the path constraint, its condition c is never negated.

Assign-such-that-statements assign a value to a variable such that a condition holds. For example, the statement `var n :| 0 <= n <= 7` assigns a value to variable `n` such that `0 <= n <= 7`. Delfy executes this statement by introducing a fresh symbolic variable for the assigned variable, and an assumed condition saying that the condition of the assign-such-that statement must hold for its value. When the assumed condition is not satisfiable, execution is aborted, otherwise all paths guarded (directly or indirectly) by the assigned variable can be explored.

To express constraints on sets, Delfy uses the extended array theory in Z3 [48], which allows us to axiomatize set theory with boolean algebra. The only operation on sets that cannot be axiomatized with the array theory is set cardinality. We axiomatize sequences using a pair that contains the

[1] Dafny functions are mathematical functions, whose body, if any, must consist of exactly one expression.

sequence length and a function mapping indexes to elements. By using universal and existential quantification, we express all operations on sequences.

5.2.1 Handling input-dependent loops

For the systematic dynamic test generation to terminate in a reasonable amount of time, existing tools typically impose a bound on the number of iterations of input-dependent loops. We define an *input-dependent loop* as a loop whose number of iterations depends on an input of the unit under test. Such loops can cause an explosion in the number of constraints to be solved and in the number of program paths to be explored [74]. However, arbitrarily bounding their number of iterations might leave certain program paths unexplored and, thus, bugs may be missed.

As an example, consider the Dafny code in Fig. 5.1. Method `Mul` computes the product of its inputs `a`, `b`, and contains a loop whose number of iterations indirectly depends on `a`.

If we test this method with an initial input value $a = 7$ for `a`, where a is the symbolic variable associated with `a`, the loop condition alone triggers the generation of the following constraints $(0 < a) \wedge (0 < a - 1) \wedge ... \wedge \neg(0 < a - 7)$. Note that a, and not aa, appears in this (partial) path constraint because the symbolic execution in dynamic test generation tracks only direct data dependencies on the inputs of the unit under test. (This is because the solution to a path constraint should immediately correspond to an assignment to inputs of the unit under test that is likely to steer execution

```
method Mul(a: int, b: int)
  requires 0 <= a && 0 <= b;
{
  var aa := a;
  var sum := 0;
  while (0 < aa)
    invariant 0 <= aa <= a;
    invariant sum == (a - aa) * b;
  {
    sum := sum + b;
    aa := aa - 1;
  }
  assert sum == a * b;
}
```

Figure 5.1: A Dafny example illustrating how Delfy handles input-dependent loops. Method `Mul` computes the product of its inputs `a`, `b`, and contains a loop whose number of iterations indirectly depends on `a`.

along an unexplored program path.)

Now consider that each of the above constraints is later negated to generate a new test case, and that Dafny supports mathematical (unbounded) integers. As a result, an infinite number of test cases could be generated to exercise the loop in method `Mul`. Moreover, in case an input-tainted branch condition occurs after the loop, all loop iterations must be explored for each of the two possible values of the condition.

Summarizing input-dependent loops

These issues can be significantly alleviated when the programmer provides a loop invariant. In particular, an invariant of an input-dependent loop may serve as a summary for the loop, in addition to being checked as part of the test oracle. Note that, in this chapter, we abuse the term "summary" to express that reasoning about many loop iterations happens in one shot, although we do not refer to a logic formula of loop pre- and postconditions, as is typically the case in compositional symbolic execution [66, 2].

Our approach consists in the following steps. We check the loop invariant as part of the test oracle in all test cases for which the number of loop iterations does not exceed a user-specified bound. In case it is feasible to generate a test case for which the number of loop iterations exceeds the bound, we no longer check the loop invariant as part of the test oracle. Instead, we assume it accurately summarizes the loop body, as is typically done in modular verification [63, 81].

For example, assume that the user-specified bound on the number of loop iterations is set to two. To test method `Mul` of Fig. 5.1, Delfy generates inputs, that is, values for parameters `a` and `b` of `Mul`, such that the following cases are exercised: (1) the loop is not entered (say, for `a := 0`, `b := 3`), (2) the loop is entered and the loop bound is not exceeded (for `a := 1`, `b := 3`), and (3) the loop is entered but the loop bound is exceeded (for `a := 3`, `b := 3`). In all of these cases, the loop invariant is *checked* (with Code Contracts assertions) both before executing the while-statement, and at the end of the loop body (for the first and second iterations).

When the bound on the number of loop iterations is exceeded, Delfy stops the execution of the current test case and warns about unsound coverage of the unit under test. This means that any code occurring after the loop remains completely unexercised when the loop iterates more than the specified number of times. When this is the case, like for method `Mul`, we address the issue by running the test case that exceeded the loop bound again (for `Mul`, `a := 3`, `b := 3`), but this time the loop invariant is treated as a summary and is *assumed* to hold.

More specifically, the loop is executed only concretely, and not symbolically, until it terminates. At that point, all loop targets (that is, all variables that are potentially written to in the loop body) are havocked: we

associate a fresh symbolic variable with each loop target, which, in case of an input-tainted loop target, is equivalent to forgetting all symbolic constraints collected for this loop target before the loop. In our example, variable `aa` is a loop target, since it is modified in the loop. As a result, after the loop terminates, we havoc `aa` and effectively forget that its value is non-negative, which follows from the precondition and the first assignment in method `Mul`. Note that `sum` is also a loop target, and that loop targets are computed using a lightweight static analysis, as we explain below.

It is necessary to havoc *all* loop targets, not only the input-tainted ones. We explain the reason through an example. Imagine that a local variable, initialized to zero, counts the number of iterations of an input-dependent loop. This variable is, of course, a loop target, but it is not input-tainted. Assume that, after the loop, there is a conditional statement on a value of this variable, both branches of which are feasible. If we have not assigned a fresh symbolic variable to the counter after the loop, we will not exercise the branch of the conditional statement that is not satisfied by the concrete value of the counter for this particular execution. In fact, we will not even consider the unexercised branch as a feasible program path for subsequent explorations.

After havocking all loop targets, we add two assumed conditions to the path constraint. First, we assume the negation of the loop condition. In our example, we assume $\neg(0 < aa')$, where aa' is the fresh symbolic variable associated with variable `aa`. All explorations that contain this negated loop condition in their path constraint exercise code deeper in the execution tree (that is, after the loop).

Second, we assume the loop invariants as if they were a summary constraining the possible values of the loop targets. In our example, we assume $0 \leq aa' \leq a \wedge sum' = (a - aa') * b$, where a, b are the symbolic variables originally associated with parameters `a`, `b`, and sum' is the fresh symbolic variable associated with `sum`. Note that, after the loop, all symbolic constraints on input-tainted loop targets refer to them by their fresh symbolic variables.

Consequently, the symbolic execution now also tracks indirect data dependencies on the inputs of the unit under test, through the fresh symbolic variables. These can be considered as additional inputs to the unit under test that are, however, not taken into account when generating new test cases, that is, any values assigned to the fresh symbolic variables by the solver do not correspond to actual inputs of the unit under test. Concretely, constraints on these variables are used to determine which program paths are infeasible. For example, after having assumed the negated loop condition and the loop invariants in method `Mul`, Delfy determines that a program path violating the assertion is infeasible.

Note that, when the loop targets include object references, we havoc all variables (in the concrete and symbolic stores) of the same type as these ref-

erences, to handle aliasing conservatively (just like in verification [97]). This is done to avoid falsely considering an alias of a loop target as unmodified by the loop, and potentially missing bugs along paths guarded by this alias. Dafny does not support subtyping, which makes determining these additional variables an easy task. However, as any source of over-approximation, this can result in the generation of *false positives*.

Our approach for summarizing input-dependent loops enables exercising code of the unit under test occurring after such loops regardless of their number of iterations. This comes at the cost of potentially generating test cases with *diverging* or *insane* path constraints, as we explain next.

If the user-provided loop invariant is too weak, that is, it does not precisely constrain all loop targets, the symbolic execution will try to explore execution paths that are infeasible in practice. For example, if we drop the second loop invariant in method `Mul`, the symbolic execution will try to explore the execution path that makes the assertion fail, which it would otherwise know to be infeasible due to the assumed conditions. Since sum' is unconstrained, the solver will generate values that satisfy the negation of the asserted condition, that is, $\neg(sum' = a * b)$. For this new test case however, the execution will not follow the expected program path that leads to an assertion violation (since the assertion holds). This situation in which a new test case does not follow the program path it was generated to exercise is called a *divergence*.

Now imagine that the loop invariants in method `Mul` are too strong, for instance, the first loop invariant becomes `0 < aa <= a`. In this case, when assuming the negated loop condition $\neg(0 < aa')$ and the first loop invariant $0 < aa' \leq a$ after the loop terminates, the path constraint for the current execution evaluates to false. In other words, even though the concrete execution is actually exercising a program path, the path constraint indicates that this path is infeasible. We call such path constraints *insane*.

Another consequence of this approach is that the body of an input-dependent loop might not be thoroughly exercised when the number of loop iterations exceeds the user-specified bound. In particular, since beyond the bound we execute the loop only concretely, and not symbolically, program paths and bugs might be missed.

Despite these limitations, our approach prevents the exploration from getting stuck in a loop body by targeting code occurring after the loop instead.

Checking loop invariants

Delfy addresses the issue of unsound coverage of input-dependent loop bodies (when the number of loop iterations exceeds the pre-defined bound), by implementing a mode for thoroughly checking loop invariants.

To thoroughly check whether an invariant is maintained by an input-

dependent loop, Delfy again uses a verification technique [63, 81], similar to the one described above. Concretely, on entry to an input-dependent loop, Delfy havocs all loop targets (and their potential aliases). Delfy then assumes the loop condition and the loop invariant, using assumed conditions as before. We determine whether the invariant is indeed maintained by the loop by checking whether its assertion at the end of the loop body holds along all execution paths within the loop. In other words, the invariant is maintained if and only if Delfy exercises all execution paths within the loop body in a sound search, and the invariant is never violated.

Note that this technique is also susceptible to false positives, divergence, and insane path constraints, depending on the over-approximation of the loop targets and the strength of the given loop invariant. However, when Delfy proves that a loop invariant is maintained by a loop, the overhead of auxiliary specifications that the programmer writes within a loop body in order to prove the invariant (with the verifier) can be reduced. Moreover, Delfy can reduce the effort of trying to verify an incorrect invariant, by generating tests that violate the invariant.

Also note that an invariant on loop entry is correct if and only if Delfy exercises all execution paths leading to the loop in a sound search, and the invariant is never violated.

5.3 Complementing verification with testing

The Dafny verifier is sound. Delfy can, therefore, complement the verifier by targeting only those properties in a unit under test that have not been verified. In other words, all verified properties have been soundly proven correct and need not be tested. The remaining properties have not been verified, either because there is a bug in the unit under test, or the programmer has not provided all necessary specifications, or the verifier "runs out of steam", that is, the specifications are too advanced for the verifier to reach a proof within a certain time limit.

As an example, consider the alternative implementation of method `Mul`, shown in Fig. 5.2. In this implementation, we first check whether parameters `a` and `b` are equal. If this is the case, we call method `Square` with `a` as an argument. Otherwise, we use the previous implementation of `Mul`, described in Sect. 5.2.1. Without providing a postcondition to method `Square` (as shown in a comment in the code), or calling the mathematical function `Square'` instead of `Square` in method `Mul`, the verifier cannot prove the first assertion in the code; the programmer has not provided all necessary specifications. Note that postconditions in functions, like in `Square'`, are typically not needed, since the function bodies give their full definition.

To avoid testing correct code and specifications, Delfy exercises only those paths of the unit under test that reach an unverified property. (Un-

```
method Mul(a: int, b: int)
  requires 0 <= a && 0 <= b;
{
  var sum := 0;
  if (a == b)
  {
    sum := Square(a);
    assert sum == a * b;
  }
  else
  {
    var aa := a;
    while (0 < aa)
      invariant 0 <= aa <= a;
      invariant sum == (a - aa) * b;
    {
      sum := sum + b;
      aa := aa - 1;
    }
    assert sum == a * b;
  }
}

method Square(n: int) returns (sq: int)
  // ensures sq == n * n;
{
  sq := n * n;
}

function method Square'(n: int): int
{
  n * n
}
```

Figure 5.2: An alternative implementation of method `Mul` from Fig. 5.1 illustrating how Delfy complements the verifier.

verified properties are marked by Dafny with the annotation language we presented in Ch. 2.) All other paths must be correct. Consequently, Delfy in combination with the verifier generates a smaller number of tests in comparison to Delfy alone, when the number of unverified program paths (that is, paths leading to at least one unverified property) is strictly less than the total number of paths in the unit under test, as we also showed in the experimental evaluation of Ch. 2. For example, for the implementation of `Mul` of Fig. 5.2, Delfy need only test the path along which parameters `a` and `b` are equal, since this is the only path that reaches an unverified assertion. All other paths are proven correct by the verifier and need not be tested.

As in Ch. 2, along an unverified path, all verified properties are added to the path constraint as assumptions, to avoid exploring their negation (see Alg. 1.1 of Sect. 1.1). Given that these properties are correct, the solver would most likely determine that the path-constraint prefix leading to an input-tainted, verified property, in conjunction with the negation of the property, corresponds to an infeasible program path. However, by treating all verified properties along an unverified path as assumptions, we prevent such redundant calls to the solver.

Verified paths, that is, paths that reach no unverified properties, need not be tested. For this reason, we apply *static* symbolic execution to generate all path-constraint prefixes leading to an unverified property. We then pass this set of prefixes to Delfy, which explores only those program paths whose path constraints start with a prefix from the set.

To prevent the static symbolic execution from exploring the entire unit under test in search for paths that reach unverified properties, we apply the following optimization. We first build the control-flow graph of the unit under test in a top-down manner. As soon as we have reached all the unverified (by the verifier) properties, we stop. As a result, we now have a possibly partial control-flow graph, which contains at least all unverified program paths and potentially a number of additional verified paths. For the example of Fig. 5.2, we build a control-flow graph containing only the if-branch and the first assertion in the code.

As a second step, we traverse the control-flow graph in a bottom-up manner, starting from the nodes that correspond to the unverified properties, for instance, the first assertion in `Mul`. While traversing the graph, we precisely determine for which program paths we need the static symbolic execution to generate a path-constraint prefix. For the example of Fig. 5.2, we instruct the static symbolic execution to generate a prefix only for the path along which the condition of the if-statement holds. As a result, Delfy generates a test case exercising the successful branch of the first assertion in method `Mul`. The dynamic symbolic execution in Delfy then determines that the failing branch of the assertion is infeasible. We, therefore, fully exercise method `Mul` by generating only one test case and avoiding the input-dependent loop in the else-branch.

We intend for the static symbolic execution to be a very fast intermediate step, which runs after the verifier and before the dynamic test generation tool. We, consequently, avoid making the analysis very precise at the cost of performance. In particular, when there is an input-dependent loop along an unverified path, we do not generate a path-constraint prefix for each number of loop iterations after which an unverified property is reached. We, instead, generate a prefix only until the entry of the input-dependent loop. Delfy, subsequently, explores the loop as described in Sect. 5.2.1, and dynamically filters out any test cases that do not exercise the unverified property.

In comparison to the code instrumentation of Ch. 2 for pruning paths

that have been statically verified, this approach, based on static symbolic execution, is more effective for (input-dependent) loop-free code. It reduces the search space for dynamic symbolic execution even more, as it is path precise and does not operate on an abstraction of the unit under test, which over-approximates the set of possible executions of the unit under test. For code with input-dependent loops, the instrumetation of Ch. 2 could be combined with this approach for even more precise results.

When a verification attempt does not go through the Dafny verifier, the Dafny IDE indicates with a red dot the return path along which the error is reported. By clicking on a red dot, the Dafny IDE displays more information about the corresponding error. We have integrated Delfy with the Dafny IDE in the following way. If a programmer selects a red dot in a unit under test and runs Delfy, the static symbolic execution generates only those path-constraint prefixes leading to the corresponding unverified property. Delfy then generates test cases exercising only this property, regardless of whether there are other unverified properties in the unit under test.

Delfy can also use the verification counterexample that corresponds to an error emitted by the verifier, that is, to a red dot, in order to generate a failing test case. In particular, Delfy is able to generate concrete inputs to a unit under test from the counterexample that the verifier computes. However, it is possible that these concrete inputs do not lead to an assertion failure at runtime. This can happen when the verifier havocs variables that affect the values of the inputs to the unit under test before the failing assertion is reached in the control flow. As a result, the initial values of the inputs (that is, at the beginning of the unit under test) from the verification counterexample might actually not provide any indication on how to cause the assertion failure. In such situations, when the test case that is generated based on the verification counterexample does not fail, Delfy falls back to the approach described above, which is based on static symbolic execution.

5.3.1 Assigning confidence to verification errors

As described above, Delfy generates only one successful test case for the alternative implementation of method `Mul` (Fig. 5.2). However, a programmer cannot be certain whether Delfy achieves sound coverage of the program paths reaching the unverified property. It is, therefore, unclear when the programmer should be confident about the correctness of the code. In practice, the programmer would still need to verify the code with Dafny, which is inefficient and does not reduce the verification effort.

To address this issue, we propose an assignment of confidence to each unverified property, based on the number of program paths that reach the particular property. More specifically, the programmer may be confident that a property, which has not been verified by Dafny, is correct only when Delfy exercises all program paths reaching the property, without generat-

ing any failing test cases or reporting any sources of unsoundness in the exploration. In other words, the confidence of correctness for an unverified property is computed over the number of all program paths that exercise the property, denoted as $|paths|$. Moreover, a generated test case exercising one of these program paths increases the confidence by $\frac{1}{|paths|}$, if and only if the property does not fail during execution of the test, and no unsoundness of Delfy is reported at any point during execution of the test.

Note that if an unsoundness is reported for a test case *after* the unverified property has been executed, the confidence of correctness for the property is *not* increased. This is because the unverified property could occur in the body of an input-dependent loop, or a recursive function or method. When this is the case, the test might not explore all executions of the property. We, therefore, cannot claim that the property holds along this program path.

Assigning confidence to errors emitted by the verifier can significantly reduce the time a programmer needs to spend on determining whether the code is indeed correct, when it does not go through the verifier. In addition, the verification experience is improved, since the programmer can focus on proving more interesting or complex properties, and leave the remaining mundane ones, which are assigned a high confidence score, to testing.

5.4 Complementing testing with verification

When Delfy assigns full confidence to a property, the property has been soundly proven correct by dynamic test generation. Consequently, such a property may now be assumed by the verifier (using the annotations of Ch. 2), which simplifies the verification task.

Therefore, complementing Delfy with the verifier can also be beneficial, for one, because the verification time of Dafny is likely to be reduced. Moreover, the overhead for the programmer of providing auxiliary specifications to prove a particular property might be alleviated—systematic testing could achieve sound coverage of the property instead. We also expect this combination to decrease the number of programmer attempts to verify an incorrect piece of code, by providing early feedback through failing tests.

From our experience with Delfy, we have observed that it can achieve additional verification results in comparison to the verifier alone in the following cases: (1) when Dafny's modular reasoning prevents it from reaching a proof (as in Fig. 5.2), (2) when path constraints for a unit under test are easier for the solver to reason about than a large verification condition, and (3) when an input-dependent loop can be exhaustively tested within the user-specified bound on the number of loop iterations. There might also be cases when the encoding of Delfy for sets and sequences, which is different than the corresponding axiomatization in Dafny, is simpler for the solver, but we have not yet come across such a situation.

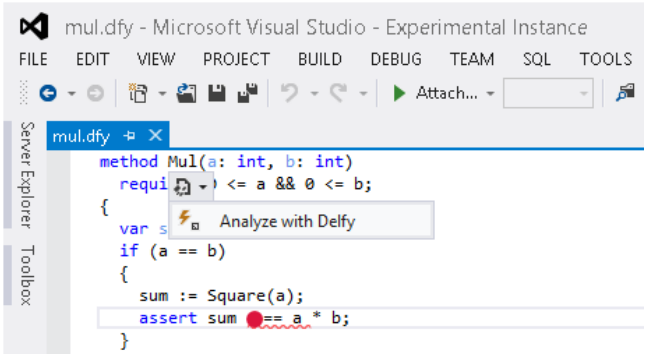

Figure 5.3: A smart tag allowing the user to invoke Delfy on a method under test, and a verification error emitted by the verifier (denoted by the dot in the assertion).

5.5 Incorporating Delfy into the Dafny IDE

We now present how we have incorporated Delfy into the Dafny IDE. The IDE runs both the verifier and Delfy in the background and in alternation. Given a program, the IDE starts by running Dafny or Delfy, in an attempt to prove as many properties as possible. Fig. 5.3 shows the error emitted by the verifier (denoted by the dot) for the first assertion in method `Mul` from Fig. 5.2. The subsequent tool, in this case Delfy, can then use the previous verification results to prove even more properties. Delfy is run automatically by the IDE in alternation with Dafny, or through the smart tag, shown in Fig. 5.3, by the user. Fig. 5.4 shows how the test cases generated by Delfy are displayed. When the automatic alternation of tools is no longer effective in achieving better results, any remaining, unverified properties are prioritized to the user for their verification.

The main characteristics of this IDE integration are the following:

– *Color coding of assertions.* To give users a sense of where they should focus their manual efforts, the IDE uses colors for assertions. A green color for a particular assertion shows that the assertion has been automatically proven correct, either by Dafny or Delfy. In Fig. 5.4, the first assertion in method `Mul`, which is proven by Delfy and not by Dafny, is highlighted with a green color. It is also marked with a verified-attribute; similarly to the notation of Ch. 2, this attribute denotes that the assertion has been verified under the premise true, that is, under no unsound assumptions.

  On the other hand, a red color for the assertion denotes that it does not hold, in other words, Dafny has emitted a verification error, and Delfy has generated a failing test case for this assertion. An orange color indicates that the assertion requires the attention of the user for

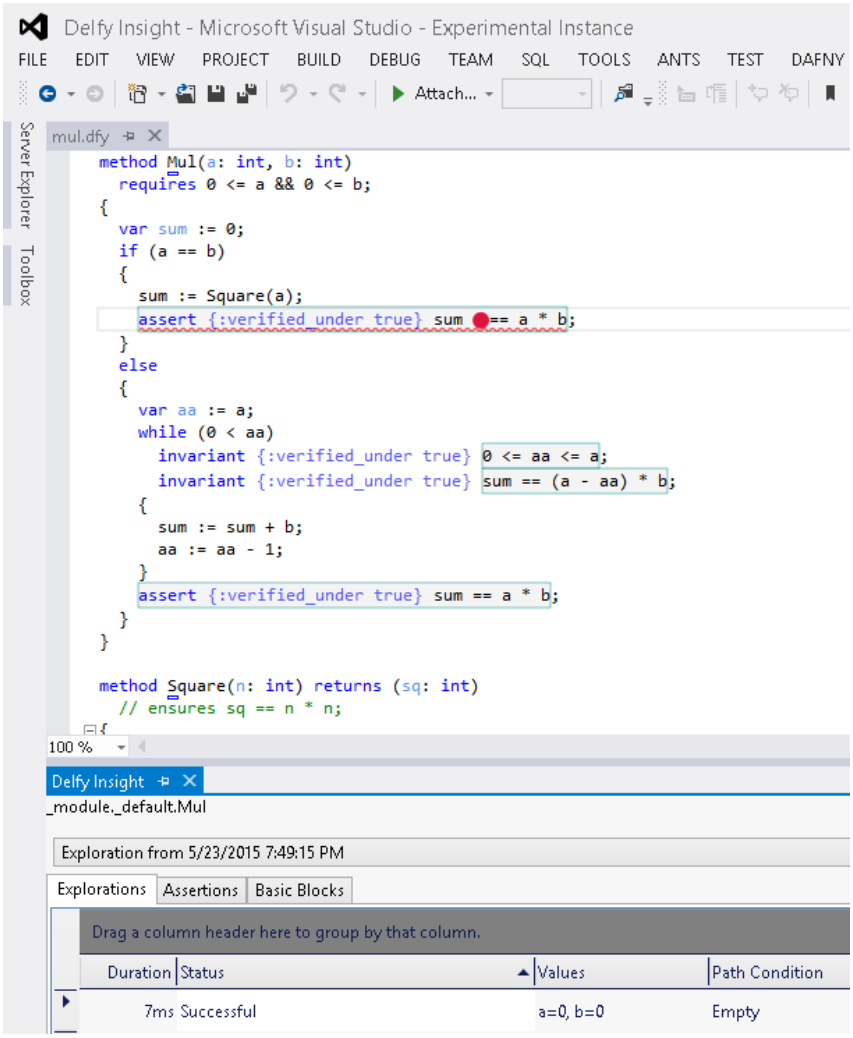

Figure 5.4: Delfy displays the generated tests and highlights the proven assertions in green.

its verification; Dafny has emitted a verification error, and Delfy has not generated any failing test, but its search was unsound.

– *Caching of results.* The Dafny IDE already caches verification results along with computed dependencies of what is being verified [103, 104]. Before starting a new verification task, the system first consults the cache, to determine whether the targeted verification results exist in the cache and have not been invalidated by a change in the program.

  We have extended this functionality of the IDE to only re-launch an exploration of a unit under test with Delfy when the code has been modified since the last exploration. The timestamp of the last explo-

```
var aa := a;
while (0 < aa)
  invariant {:verified_under true} 0 <= aa <= a;
  invariant {:verified_under true} sum == (a - aa) * b;
{
  sum := sum + b;
  aa := aa - 1;
}
assert {:verified_under true} sum == a * b;
```

Figure 5.5: Delfy shows coverage information for a unit under test.

ration is shown above the generated test case in Fig. 5.4.

In the future, we intend to enhance this feature with information about which execution paths are affected by a program change, enabling Delfy to exercise only these affected paths and not re-generate test cases for the remaining paths. As a result, the interaction between the user and the IDE will become much more effective and efficient, when Delfy is run on the unit under test. This will allow for more productive and continuous processing of programs by the user.

– *Coverage information.* For large units under test, we find it difficult to determine which program path is exercised by a particular test case. This is why we highlight the covered code of the unit under test, when the user selects a generated test case on mouse click. The user can also choose to see the coverage that is achieved for the unit under test by all the generated tests, as shown in Fig. 5.5.

– *Generating concrete counterexamples.* As we described in Sect. 5.3, Delfy allows the user to select a property that has not been verified by

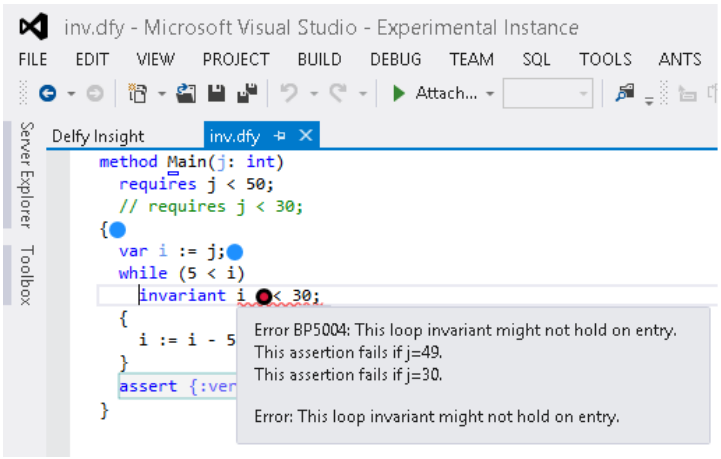

Figure 5.6: Verification counterexamples emitted by the verifier for a loop-invariant violation.

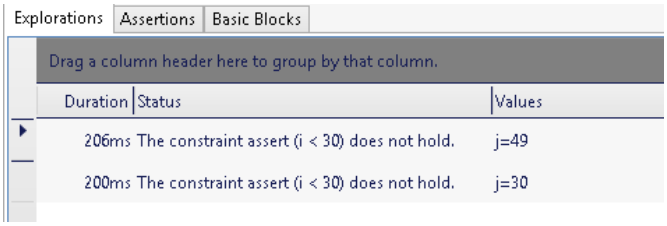

Figure 5.7: Delfy generates two failing tests from the verification counterexamples of Fig. 5.6.

Dafny, and generate test cases that cover only this selected property (optionally by using the verification counterexample).

Fig. 5.6 shows the verification counterexamples generated by Dafny for a very simple method with an overly weak precondition. As a consequence of this weak precondition, the loop invariant cannot be verified. Fig. 5.7 shows the failing tests that Delfy generates from these verification counterexamples. Note that the user can select to inspect all generated tests, or categorize them based on their outcome.

– *Debugging of test cases.* Delfy also makes it possible to debug the generated test cases with the .NET debugger, such that users can step through their execution and watch the values of variables of their choice. Fig. 5.8 shows a smart tag that allows the user to debug a verification error by running the failing test cases, generated for the particular error, in the .NET debugger. The options in the smart tag of Fig. 5.8 correspond to the failing tests shown in Fig. 5.7. Fig. 5.9 shows a debugging session for the second failing test of Fig. 5.7, during the progress of which we are watching the value of variable `i`.

5.6 Comparing Delfy to BVD

In this section, we compare Delfy to BVD, which refers to the Boogie Verification Debugger [92] and is already deeply integrated into the Dafny IDE.

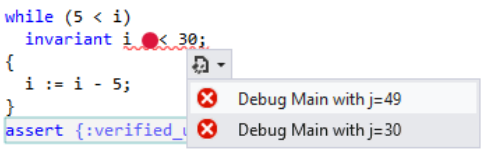

Figure 5.8: Delfy enables debugging of a verification error by running the failing tests, generated for the particular error, in the .NET debugger.

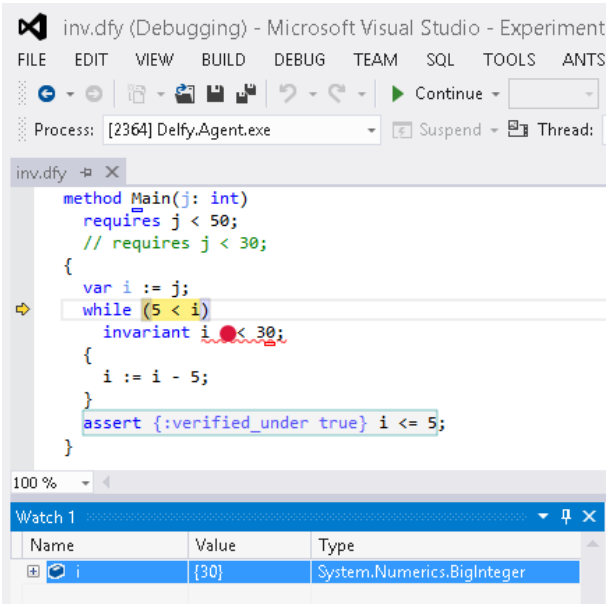

Figure 5.9: A debugging session of a failing test, during the progress of which a variable is being watched.

Our goal is to determine in which cases a verification error is easier to understand using a dynamic test generation tool over a static verification debugger, and vice versa.

To compare Delfy to BVD, we used both tools in an effort to understand all errors emitted by the Dafny verifier for all programs of the Dafny test suite, with and without seeding bugs. Here are our observations:

– *Heap data structures*: When a unit under test manipulates heap data structures, we found that verification errors are easier to understand with BVD. The reason is that Delfy provides only the input to the unit under test for which an error occurs, while BVD shows all step-by-step modifications to the data structures. However, Delfy makes it possible to debug the generated test cases with the .NET debugger and watch the value of any variable (as in Figs. 5.8 and 5.9), in which case Delfy becomes more usable.

– *Function or method calls*: When a unit under test contains function or method calls, we noticed that errors are easier to understand using Delfy. Since BVD is a modular tool, it is difficult to determine which particular call to a function or method in the unit under test causes an error to occur in the callee. With Delfy, this is much easier as the path constraint and coverage information of each generated test case indicates the executed program path.

– *Loops*: When the unit under test contains input-dependent loops whose number of iterations does not exceed the user-specified bound, we found Delfy to be more useful than BVD in determining the cause of an error. This is because Delfy provides the input to the unit under test for which the error occurs. For errors occurring after the loop, BVD shows the state only on loop entry, which in some cases does not even correspond to the state that will cause the error after the loop (due to havocking).

– *Sets and sequences*: We noticed that, when Delfy is able to generate concrete sets or sequences, so is BVD. The reason is that both tools use similar axiomatizations for sets and sequences and the same underlying constraint solver.

In general, we found Delfy and BVD to be complementary. Delfy provides concrete inputs to the unit under test and shows the followed program path for each test case; however, its exploration of the unit under test might not be sound. BVD is a static, modular tool that shows the state of all relevant parts of the heap at almost all program points.

5.7 Related work

We have discussed various integrations of verification with systematic testing in Sect. 2.4 of Ch. 2.

Complementing verification with systematic testing is not a new idea. In contrast to Check 'n' Crash [43], DSD-Crasher [44], and DyTa [65], which integrate unsound static checkers with dynamic test generation, our tool integration relies on a sound verifier. Consequently, these techniques might miss bugs by pruning execution paths that are unsoundly considered verified.

Our work is more closely related to YOGI [121] and SANTE [26]. More specifically, given a property, YOGI searches both for a test that violates the property and an abstraction that proves the property correct. Similarly to CEGAR [34], YOGI uses error traces from the abstraction to guide the test generation, but also unsatisfiable path constraints from the test generation to refine the abstraction. In comparison to YOGI, our combination of Dafny and Delfy is less tight, as the verification methodology of Dafny is not adjusted based on constraints from failed test generation attempts of Delfy. In fact, Dafny can only take advantage of prior verification results achieved by Delfy. Exactly like SANTE, our test generation also prunes execution paths that do not lead to unverified assertions.

Complementing systematic testing with verification, however, is a new idea. With the exception of the second part of this dissertation, we are not aware of any test generation approach to have been pushed toward reaching verification results about functional correctness properties, like the

ones presented here. Since testing tools could not claim such verification results until now, it did not seem sensible to complement them by verifiers.

Loop summarization in dynamic test generation has been studied before [74]. SAGE [73] uses simple loop-guard pattern-matching rules to, dynamically and on-the-fly, infer partial loop invariants for a certain class of input-dependent loops. Summaries for these loops are then derived from the inferred invariants, without any static analysis, theorem proving, or user-provided specifications. In comparison, Delfy can summarize any input-dependent loop as long as the user has written an invariant. However, the quality of the summary depends on the strength of the invariant, whereas dynamic loop summarization has been proven sound and complete when certain restrictions are met.

5.8 Summary and remarks

We have presented the integration of a sound verifier with a dynamic test generation tool and discussed its benefits. These include shorter testing and verification times, reduced annotation overhead, facilitated debugging of spurious verification errors, and increased programmer productivity. To make these benefits easily accessible to the users, we have designed an integrated development environment around this tool integration.

In the future, we plan on making Delfy available, through this IDE, to undergraduate students of formal methods courses. We also plan on making the tool available to the contributors of the Ironclad project [79], which is currently the largest project using Dafny, developed by the systems and security groups at Microsoft Research. Our goal is to carefully and empirically assess the usability of the Delfy tool and the aforementioned benefits of its combination with Dafny.

Part II

Pushing systematic testing toward verification

Chapter 6

Toward proving memory safety of the ANI Windows image parser using compositional exhaustive testing

In dynamic test generation, a path constraint is a symbolic generalization of a set of concrete executions, which represents an equivalence class of input vectors that drive the program execution along a particular path. In other words, the search space of program inputs is partitioned into equivalence classes, each of which exercises a different program path and potentially exhibits a new program behavior. Dynamic test generation amounts to program verification when the following three conditions hold: (1) there are finitely many program paths and, thus, equivalence classes, (2) all feasible paths are exercised, and (3) the constraint solver is sound and complete, such that path feasibility is decided correctly and an input vector is generated for each equivalence class.

However, in the presence of loops whose number of iterations depends on a program input, there can be an explosion of paths to be exercised. In practice, dynamic test generation tools bound the number of iterations of input-dependent loops, consequently ignoring some equivalence classes of inputs and potentially missing bugs.

Although bugs may be missed, dynamic test generation has proven particularly successful in detecting security vulnerabilities in hundreds of programs processing structured files, such as image processors, media players, file decoders, and document parsers [16]. Despite this success, systematic testing has never been pushed toward verification of such a program. In this chapter, we assess to what extent the idea of reaching verification with systematic testing is realistic. Specifically, we report how we used and enhanced systematic dynamic test generation to get closer to proving memory safety of the ANI Windows image parser.

The ANI parser is responsible for reading files in a structured graphics file format and processing their contents in order to display "ANImated" cursors and icons. Such animated icons are ubiquitous in practice (like the spinning

ring or hourglass on Windows), and their domain of use ranges from web pages and blogs, instant messaging and e-mails, to presentations and videos. In addition, there are many applications for creating, editing, and converting these icons to and from different file formats, such as GIF or CUR. The ANI parser consists of thousands of lines of low-level C code spread across hundreds of functions (referring to C functions, not mathematical ones).

We chose the ANI parser for this work as it is one of the smallest image parsers embedded in Windows. The implementation of the parser is within the scope of dynamic symbolic execution since it is neither concurrent nor subject to real-time constraints. Despite this, there are still significant challenges in proving memory safety of the ANI parser, including reasoning about memory dereferences and exception-handling code. Our choice was also motivated by the fact that in 2007 a critical out-of-band security patch was released for code in this parser (MS07-017) costing Microsoft and its users millions of dollars. This vulnerability was similar to an earlier one reported in 2005 (MS05-002) meaning that many details of the ANI parser have already been made public over the years [131, 83]. This parser is included in all distributions of Windows, i.e., it is used on more than a billion PCs, and has been tested for years. Given the ubiquity of animated icons, our goal was to determine whether the ANI parser is now free of security-critical buffer overflows.

We show in this chapter how systematic dynamic test generation can be applied and extended toward program verification in the context of the ANI parser. To achieve this, we alleviate the two main limitations of dynamic test generation, namely, imperfect symbolic execution and path explosion. For the former, we extended the tool SAGE [73] to improve its symbolic execution engine so that it could handle all the x86 instructions along all the explored code paths of that specific ANI parser. To deal with path explosion, we used a combination of function inlining, restricting the bounds of input-dependent loops, and function summarization. We also used a new tool, named MicroX [67], for executing code fragments in isolation using a custom virtual machine designed for testing purposes. We emphasize that the focus of our work is restricted to proving the absence of attacker-controllable memory-safety violations (as precisely defined in Sect. 6.2).

At a high-level, the main contributions of this chapter are:

- We report on the first application of dynamic test generation to verify as many executions as possible of a real, complex, security-critical, entire program. Our work sheds light on the shrinking gap between systematic testing and verification in a model-checking style.
- To our knowledge, this is the first attempt to prove that an operating-system (Windows or other) image parser is free of security-critical buffer overflows, modulo the soundness of our tools and several additional assumptions.

– We are also not aware of any past attempts at program verification without using any static program analysis. All the techniques and tools used in this work are exclusively dynamic, thus seeking verification of the *execution* of the parser, including complicated x86 code patterns for structured exception handling and stack-guard protection, which most static analysis tools cannot handle (see Sect. 6.7 for more details).

Outline. This chapter is organized as follows. In Sect. 6.1, we recall basic principles of compositional symbolic execution. In Sect. 6.2, we precisely define memory safety, show how to verify it compositionally, and discuss how we used and extended SAGE and MicroX for verification purposes. Sect. 6.3 presents an overview of the ANI Windows image parser. In Sect. 6.4, we present our approach and verification results in detail. During the course of this work, we discovered several memory-safety violations in the ANI parser code and came across a number of unexpected challenges, which are discussed in Sects. 6.5 and 6.6, respectively. We review related work in Sect. 6.7.

6.1 Compositional symbolic execution

Systematically testing and symbolically executing all feasible program paths does not scale to large programs. Indeed, the number of feasible paths can be exponential in the program size, or even infinite in the presence of loops with an unbounded number of iterations. This *path explosion* can be alleviated by performing symbolic execution *compositionally* [66, 2].

In compositional symbolic execution, a summary ϕ_f for a function (or any program sub-computation) f is defined as a logic formula over constraints expressed in theory T. Summary ϕ_f can be generated by symbolically executing each path of function f, then generating an input precondition and output postcondition for each path, and bundling together all path summaries in a disjunction. More precisely, ϕ_f is defined as a disjunction of formulas ϕ_{w_f} of the form

$$\phi_{w_f} = pre_{w_f} \wedge post_{w_f}$$

where w_f denotes an intra-procedural path in f, pre_{w_f} is a conjunction of constraints on the inputs of f, and $post_{w_f}$ a conjunction of constraints on the outputs of f. An *input* to a function f is any value that can be read by f, while an *output* of f is any value written by f. Therefore, ϕ_{w_f} can be computed automatically when symbolically executing the intra-procedural path w_f: pre_{w_f} is the path constraint along path w_f but expressed in terms of the function inputs, while $post_{w_f}$ is a conjunction of constraints, each of the form $v' = S(v)$, where v' is a fresh symbolic variable created for each program variable v modified during the execution of w_f (including the return

value), and where $S(v)$ denotes the symbolic value associated with v in the program state reached at the end of w_f. At the end of the execution of w_f, the symbolic store is updated so that each such value $S(v)$ is replaced by v'. When symbolic execution continues after the function returns, such symbolic values v' are treated as inputs to the calling context. Summaries can be re-used across different calling contexts.

For instance, given the function `is_positive` below,

```
int is_positive(int x) {
  if (0 < x)
    return 1;
  return 0;
}
```

a summary ϕ_f for this function can be

$$\phi_f = (0 < x \wedge ret = 1) \vee (x \leq 0 \wedge ret = 0)$$

where *ret* denotes the value returned by the function.

Symbolic variables are associated with function inputs (like x in the example) and function outputs (like *ret* in the example) in addition to whole-program inputs. In order to generate a new test to cover a new branch b in some function, all the previously known summaries can be used to generate a formula ϕ_P symbolically representing all the paths discovered so far during the search. By construction [66], symbolic variables corresponding to function inputs and outputs are all bound in ϕ_P, and the remaining free variables correspond exclusively to whole-program inputs (since only those can be controlled for test generation).

For instance, for the program `P` below,

```
#define N 100

void P(int s[N]) { // N inputs
  int i, cnt = 0;
  for (i = 0; i < N; i++)
    cnt = cnt + is_positive(s[i]);
  if (cnt == 3)     // (*)
    abort;
}
```

a formula ϕ_P to generate a test covering the then-branch `(*)`, given the above summary ϕ_f for function `is_positive`, can be

$$(ret_0 + ret_1 + \ldots + ret_{N-1} = 3) \wedge$$

$$\bigwedge_{0 \leq i < N} ((0 < s[i] \wedge ret_i = 1) \vee (s[i] \leq 0 \wedge ret_i = 0))$$

where ret_i denotes the return value of the i-th call to function `is_positive`. Even though program `P` has 2^N feasible whole-program paths, compositional

test generation can cover symbolically all those paths with at most four test inputs: two tests to cover both branches in function `is_positive` plus two tests to cover both branches of the if-statement `(*)`. In this example, compositionality avoids an exponential number of tests and calls to the constraint solver at the cost of using more complex formulas with more disjunctions.

Note, however, that the number of execution paths in a function f could be infinite, when f contains loops whose number of iterations depends on an unbounded input. In practice, we limit the size of the summary of f by enforcing a bound on the number of execution paths that are explored in f. As shown in Ch. 5, loop invariants could also be used for generating more general and compact function summaries, in comparison to those generated by bundling together all path summaries in a disjunction.

In general, when, where, and how compositionality is worth using in practice is still an open question (e.g., [66, 2, 15, 89]), which we discuss later in this chapter.

6.2 Proving memory safety

In this chapter, we define memory safety, describe how to verify it compositionally, and explain how we used and extended SAGE and MicroX to get closer to verification.

6.2.1 Defining memory safety

To prove memory safety during systematic dynamic test generation, all memory accesses need to be checked for possible violations. Whenever a memory address a stored in a program variable v (i.e., $a = M(v)$) is accessed during execution, the concrete value a of the address is first checked "passively" to make sure it points to a valid memory region mr_a, that is, it does not point to an unallocated memory region (as done in standard tools like Purify [78], Valgrind [118], and AppVerifier [113]); then, if this address a was obtained by computing an expression e that depends on an input (i.e., $e = S(v)$), the symbolic expression e is also checked "actively" by injecting a new *bounds-checking* constraint of the form

$$0 \leq (e - mr_a.base) < mr_a.size$$

in the path constraint to make sure other input values cannot trigger a buffer under- or overflow at this point of the program execution [21, 72]. How to keep track of the base address $mr_a.base$ and size $mr_a.size$ of each valid memory region mr_a during the program execution is discussed in work on precise symbolic pointer reasoning [54].

As an example, consider the following function:

```
void buggy(int x) {
  char buf[10];
  buf[x] = 1;
}
```

If this function is run with `x = 1` as input, the concrete execution is memory safe as the memory access `buf[1]` is in bounds. In order to force systematic dynamic test generation to discover that this program is not memory safe, it is mandatory to inject the constraint $0 \leq x < 10$ in the current path constraint when the statement `buf[x] = 1` is executed. This constraint is later negated and solved leading to other input values for `x`, such as -1 or 10, with which the function will be re-tested and caught violating memory safety.

A program execution w is called *attacker memory safe* [69] if every memory access during w in program P, which is extended with bound checks for all memory accesses, is either within bounds, i.e., memory safe, or input independent, i.e., its address has no input-dependent symbolic value, and, hence, is not controllable by an attacker through the untrusted input interface. A program is called attacker memory safe if all its executions are attacker memory safe.

Thus, the notion of attacker memory safety is weaker than traditional memory safety: a memory-safe program execution is always attacker memory safe, while the converse does not necessarily hold. For instance, an attacker-memory-safe program might perform a flawless and sound validation of all its untrusted inputs, but might still crash. As an example, consider the following function:

```
void buggy() {
  char* buf = malloc(10);
  buf[0] = 1;
}
```

This function is attacker memory safe since it has no (untrusted) inputs, but is memory unsafe since the trusted system call to `malloc` might fail, resulting in accessing the null address.

Security testing for memory safety is primarily aimed at checking attacker memory safety since buffer overflows that cannot be controlled by the attacker are not security critical. In the rest of this chapter, we focus on attacker memory safety, but we will often refer to it simply as memory safety, for convenience.

6.2.2 Proving attacker memory safety compositionally

In order to prove memory safety compositionally, bounds-checking constraints need to be recorded inside summaries and evaluated for each calling context.

Consider the following function `bar`:

```
void bar(char* buf, int x) {
  if ((0 <= x) && (x < 10))
    buf[x] = 1;
}
```

If we analyze `bar` in isolation without knowing the size of the input buffer `buf`, we cannot determine whether the buffer access `buf[x]` is memory safe. When we summarize function `bar`, we include in the precondition of the function that `bar` accesses the address `buf + x` when the following condition holds: $(0 \leq x) \wedge (x < 10)$. A summary for this function executed with, say, `x = 3` and a non-null buffer can then be:

$$(0 \leq x) \wedge (x < 10) \wedge \texttt{buf} \neq \texttt{NULL} \wedge (0 \leq x < mr_{\texttt{buf}}.size) \wedge (\texttt{buf}[\texttt{x}] = 1)$$

Later, when analyzing higher-level functions calling `bar`, these bounds-checking constraints can be checked because the buffer bounds will then be known. For instance, consider the following function `foo` that calls `bar`:

```
void foo(int x) {
  char* buf = malloc(5);
  bar(buf, x);
}
```

If, during execution, `foo` calls `bar` with its parameter `x = 3`, the precondition of the above path summary for `bar` is satisfied. The bounds-checking constraint can be simplified with $mr_{\texttt{buf}}.size = 5$ in this calling context, and negated to obtain the new path constraint,

$$(0 \leq x) \wedge (x < 10) \wedge \neg(0 \leq x < 5)$$

which after simplification is:

$$(0 \leq x) \wedge (x < 10) \wedge ((x < 0) \vee (5 \leq x))$$

(Note that the constraint $\texttt{buf} \neq \texttt{NULL}$ in the original summary for `bar` is now trivially true and is, therefore, omitted here.) The above path constraint is satisfiable with, say, $x = 7$, and running `foo` and `bar` with this new input value will then detect a memory-safety violation in `bar`.

To sum up, the procedure we use for proving memory safety compositionally is as follows. We record bounds-checking constraints in the preconditions of intra-procedural path-constraint summaries. Whenever a path summary is used in a specific calling context, we check whether its precondition contains any bounds-checking constraint. If so, we check whether the size of the memory region appearing in the bounds-checking constraint is known. If this is the case, we generate a new alternate path constraint defined as the conjunction of the current path constraint and the negation of the bounds-checking constraint, where the size of the memory region is replaced by the current size. We then attempt to solve this alternate path

constraint with the constraint solver, which generates a new test if the constraint is satisfiable.

For real C functions, the logic representations of their pre- and post-conditions can quickly become very complex and large. We show later in this chapter that, by using summarization sparingly and at well-behaved function interfaces, these representations remain tractable.

We have implemented in SAGE the compositional procedure for proving memory safety described in this section.

6.2.3 Verification with SAGE and MicroX

This work was carried out using extensions of two existing tools: SAGE [73] and MicroX [67].

Recall from Sect. 1.1 that SAGE implements systematic dynamic test generation and performs dynamic symbolic execution at the x86 binary level, for detecting security vulnerabilities. In order to use SAGE to verify as many executions as possible, we *turned on maximum precision* for symbolic execution: all runtime checkers (for buffer over- and underflows, division by zero, etc.) were turned on as well as precise symbolic pointer reasoning [54], any x86 instruction unhandled by symbolic execution was reported, every path constraint was checked to be satisfiable before negating constraints, we checked that our constraint solver, the Z3 automated theorem prover [47], never timed out on any constraint, and we also checked the absence of any *divergence*, which occurs whenever a new test generated by SAGE does not follow the expected program path. When all these options are turned on and all the above checks are satisfied, symbolic execution of an individual path has *perfect precision*: path constraint generation and solving is *sound and complete* (see Sect. 1.1).

Moreover, we *turned off* all the *unsound state-space pruning* techniques and heuristics implemented in SAGE to restrict path explosion, such as bounding the number of constraints generated for each program branch, and constraint subsumption, which eliminates constraints logically implied by other constraints injected at the same branch (most likely due to successive iterations of an input-dependent loop) using a cheap syntactic check [73]. How we dealt with path explosion is discussed in Sects. 6.4.2 and 6.4.3.

MicroX is a newer tool for executing code fragments in isolation, without user-provided test drivers or input data, using a custom virtual machine (VM) designed for testing purposes. Given any user-specified code location in an x86 binary, the MicroX VM starts executing the code at that location, intercepts all memory operations before they occur, allocates memory on-the-fly in order to perform these read- and write-memory operations, and provides input values according to a customizable *memory policy*, which defines what read-memory accesses should be treated as inputs. By default, an input is defined as any value read from an uninitialized function argument,

or through a dereference of a previous input (recursively) that is used as an address. This memory policy is typically adequate for testing C functions. (Note that, under the default memory policy, values read from uninitialized global variables are not considered inputs.) No test driver or harness is required: MicroX discovers automatically and dynamically the input/output signature of the code being run. Input values are provided as needed along the execution and can be generated in various ways, e.g., randomly or using some other test generation tool, like SAGE. When used with SAGE, the very first test inputs are generated randomly; then, SAGE symbolically executes the code path taken by the given execution, generates a path constraint for that (concrete) execution, and solves new alternate path constraints that, when satisfiable, generate new input values guiding future executions along new program paths.

As we describe in Sect. 6.4, we used MicroX in conjunction with SAGE with the purpose of proving memory safety of individual ANI functions in isolation. Memory safety of a function is *proven for any calling context* (soundly and completely) by MicroX and SAGE if all possible function input values are considered, symbolic execution of every function path is sound and complete, all function paths can be enumerated and tested in a finite (and small enough) amount of time, and all the checks defined above are satisfied for all executions. Instead of manually writing a test driver that explicitly identifies all input parameters (and their types) for each function, MicroX provided this functionality automatically [67].

During this work, many functions were not verified at first for various reasons: we discovered and fixed several x86 instructions unhandled by SAGE's symbolic execution engine, we also fixed several root causes of divergence (by providing custom summaries for non-deterministic-looking functions, like `malloc` and `memcpy`, whose execution paths depend on memory alignment), and we fixed a few imprecision bugs in SAGE's code. These SAGE limitations were much more easily identified when verifying small functions in isolation with MicroX, rather than during whole-application testing. After lifting these limitations, we were able to verify that many individual ANI functions are memory safe (see Sect. 6.4.1). The remaining functions could not be verified mostly because of path explosion due to input-dependent loops (see Sect. 6.4.2) or due to too many paths in functions lower in the call graph (see Sect. 6.4.3).

6.3 The ANI Windows parser

The general format of an ANI file is shown in Fig. 6.1. It is based on the generic Resource Interchange File Format (RIFF) for storing various types of data in tagged chunks, such as video (AVI) or digital audio (WAV). RIFF has a hierarchical structure in which each chunk might contain data or a list

```
RIFF ACON
  [ LIST INFO
      IART <artist>
      ICOP <copyright>
      INAM <name>
  ]
  anih <anihdr>
  [ seq <seqinfo> ]
  [ rate <rateinfo> ]
  LIST fram icon <iconfile> ...
```

Figure 6.1: The ANI file format (partial description).

of other chunks. Animated icons contain the following information:

- a `RIFF` chunk, whose header has the identifier `ACON`, specifies the type of the file,
- an optional `LIST` chunk, whose header has the identifier `INFO`, contains information about the file, such as the name of the artist,
- an `anih` header chunk contains information about the animation, including the number of frames, i.e., Windows icons, and the number of steps, i.e., the total number of times the frames are displayed,
- an optional `seq` chunk defines the order in which the frames are displayed,
- an optional `rate` chunk determines the display rate for each frame in the sequence, and
- a `LIST` chunk, whose header has the identifier `fram`, contains a list of icons.

This file format already provides an indication of the size and complexity of the ANI parser.

The high-level call graph of the parser code is shown in Fig. 6.2. The main component of the architecture, *Reading and validating file*, reads and

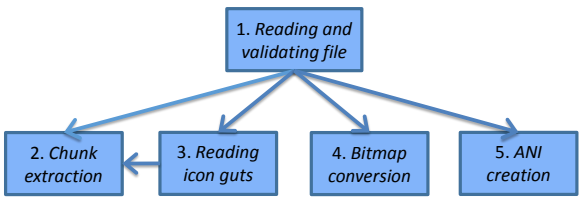

Figure 6.2: The high-level call graph of the ANI parser.

validates each chunk of an ANI file. If an extracted chunk is a `LIST fram`, the *Reading icon guts* component is invoked to read and validate the first icon in the list. In case the icon is valid, it is converted to a physical bitmap object by the *Bitmap conversion* component. Once this process has been repeated for all the icons in the list, the animated icon is created from their combination (*ANI creation* component).

The ANI parser is written mostly in C, while the remaining code is written in x86 assembly. The implementation involves at least 350 functions defined in five Windows DLLs. The parsing (that is, testing in branch statements) of input bytes from an ANI file takes place in at least 110 functions defined in two DLLs, namely, in `user32.dll`, which is responsible for 80% of the parsing code, and in `gdi32.dll`, which is responsible for the remaining 20%[1]. `user32.dll` creates and manages the user interface, such as windows, mouse events, and menus. Many functions defined in `user32.dll` call into `gdi32.dll`, which is the graphics device interface associated with drawing and handling two-dimensional objects as well as managing fonts. There are 47 functions in `user32.dll` that implement functionality of the parser. These functions alone compile to approximately 3,050 x86 instructions.

6.4 Approach and verification results

To try to prove memory safety of the ANI Windows image parser, we targeted the 47 functions that are defined in `user32.dll` and are responsible for 80% of the parsing code (see Sect. 6.3). The remaining 20% refers to at least 63 `gdi32.dll` functions that are called (directly or indirectly) by the 47 `user32.dll` functions. In addition to these `user32.dll` and `gdi32.dll` functions, the parser also exercises code in at least 240 other functions (for a total of at least 350 functions). However, all these other functions do not (directly or indirectly) parse any input bytes from an ANI file, and are by definition attacker memory safe. For the purpose of this work, the `gdi32.dll` and all these other functions can be viewed as *inlined* to the `user32.dll` functions, which are the top-level functions of the parser. Verifying the 47 `user32.dll` functions while inlining all remaining sub-functions is, thus, equivalent to proving attacker memory safety of the entire ANI parser. The call graph of the 47 `user32.dll` functions is shown in Fig. 6.3. The functions are grouped depending on the architectural component of the parser (see Fig. 6.2) to which they belong. Note that there is no recursion in this call graph.

In this section, we describe how we attempted to prove memory safety of the ANI parser using compositional exhaustive testing. Our results were obtained with 32-bit Windows 7 and are presented in three stages.

[1]These percentages were obtained by comparing the number of constraints on symbolic values that were generated by SAGE for each of the two DLLs.

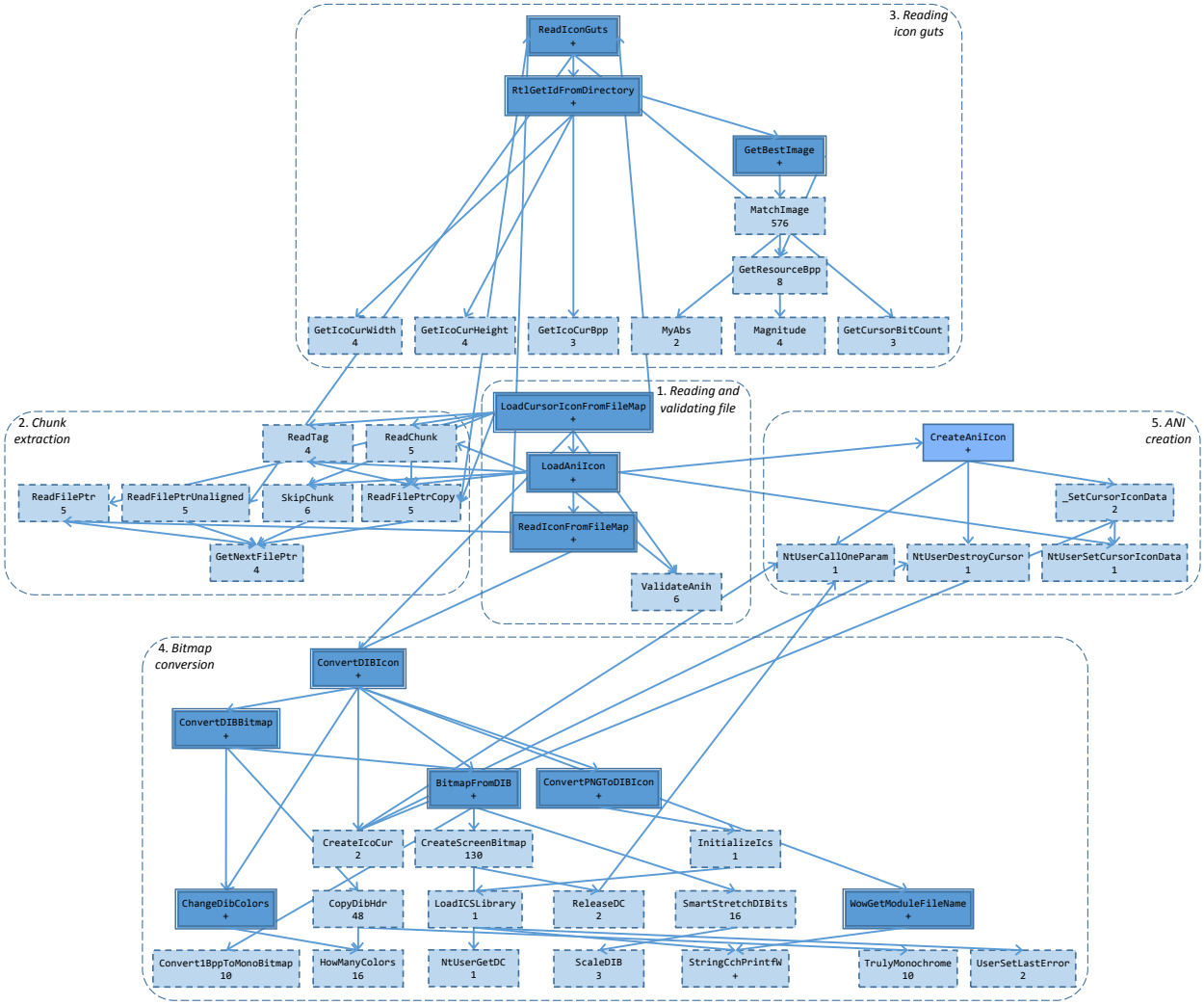

Figure 6.3: The call graph of the 47 `user32.dll` functions implementing the ANI parser core. Functions are grouped based on the architectural component of the parser to which they belong. The different shades and lines of the boxes denote the strategy we used to prove memory safety of each function. The lighter shade and dotted lines indicate functions verified with the bottom-up strategy (Stage 1), the medium shade and single solid lines, functions unsoundly verified by restricting the bounds of input-dependent loops (Stage 2), and the darker shade and double solid lines, functions (soundly or unsoundly) verified with the top-down strategy (Stage 3). Functions are annotated with the number of their execution paths. A + indicates that a function contains too many execution paths to be exhaustively enumerated within twelve hours without using any techniques for controlling path explosion.

6.4.1 Stage 1: Bottom-up strategy

For attempting to verify the ANI parser, we started with a bottom-up strategy with respect to the call graph of Fig. 6.3. We wanted to know how many functions of a real code base can be proven memory safe for any calling context by simply using exhaustive path enumeration. Our setup for this verification strategy consisted in trying to verify each `user32.dll` function (one at a time) starting from the bottom of the call graph. If all execution paths of the function were explored in a reasonable amount of time, i.e., less than twelve hours, and no bugs or other unsoundness-check violations were ever detected (see Sect. 6.2.3), we marked the function as memory safe. To our surprise, 34 of the 47 functions shown in Fig. 6.3 could already be proven memory safe this way, and are shown with the lighter shade and dotted lines in the figure.

Inlining

Function `StringCchPrintfW` of the *Bitmap conversion* component writes formatted data to a specified string, which is stored in a destination buffer. It takes as input arguments the destination buffer that receives the formatted string, the size of the destination buffer, the format string, and the arguments that are inserted in the format string. Exploring all execution paths of function `StringCchPrintfW` that may be passed a destination buffer of any length and a format string with any number of format specifiers does not complete in twelve hours, and is actually very complex.

To deal with this function, we just *inlined* it to each of its callers. Inlining a function means replacing the call sites of the function with the function body. In our context, inlining a function means that the function being inlined is no longer treated as an isolated unit that we attempt to verify for any (all) calling contexts. Instead, it is being included in the unit defined by its caller function(s) and verified only for the specific calling context(s) defined in the caller function(s). For instance, function `LoadICSLibrary`, which takes no input arguments, calls function `StringCchPrintfW`. By inlining `StringCchPrintfW` to `LoadICSLibrary`, we can exercise the single execution path in `LoadICSLibrary` and prove attacker memory safety of both functions.

Verification results

With the simple bottom-up strategy of this section, we were already able to prove attacker memory safety of 34 `user32.dll` functions out of 47, or 72% of the top-level functions of the ANI Windows parser. So far, we had to inline only one function, namely, `StringCchPrintfW` to `LoadICSLibrary` of the *Bitmap conversion* component. The `gdi32.dll` functions (not shown in Fig. 6.3), which are called by the 47 `user32.dll` functions of Fig. 6.3,

were also inlined (recursively) in those `user32.dll` functions. The boxes with the lighter shade and dotted lines of Fig. 6.3 represent the 34 functions that were verified with the bottom-up strategy. All these functions, except for those that were inlined, were verified in isolation for any calling context. This implies that all bounds for all loops (if any) in all these functions either do not depend on function inputs, or are small enough to be exhaustively explored within twelve hours. Recall that accesses to function input buffers are not yet proven memory safe at this stage of the verification process since input buffer sizes are still unknown (see Sect. 6.2.2).

6.4.2 Stage 2: Input-dependent loops

For the remaining 13 `user32.dll` functions of the ANI parser, path explosion is too severe, and exhaustive path enumeration does not terminate in twelve hours. Therefore, during the second stage of the process, we decided to identify and restrict the bounds of *input-dependent loops* that might have been preventing us from verifying functions higher in the call graph of the parser in Stage 1. This is where we gave up soundness. In our context, an input-dependent loop is a loop whose number of iterations depends on bytes read from an ANI file, i.e., whole-program inputs. In contrast, when the number of iterations of a loop inside a function depends on function inputs that are not whole-program inputs, path explosion due to that loop can be eliminated by inlining the function to its caller(s).

Restricting input-dependent loop bounds

In order to control path explosion due to input-dependent loops, we manually *fixed the bounds*, i.e., the number of iterations, of those loops by assigning a concrete value to the program variable(s) containing the input bound(s). We extended MicroX for the user to easily fix the value of arbitrary x86 registers or memory addresses. Of course, fixing an input value to a specific concrete value is like specifying an input precondition, and the verification of memory safety becomes restricted to calling contexts satisfying that precondition.

As an example, consider function `CreateAniIcon` of the *ANI creation* component. Function `CreateAniIcon` calls functions `NtUserCallOneParam` and `NtUserDestroyCursor`, which have one execution path each, as well as function `_SetCursorIconData`, which has two execution paths, as shown in Fig. 6.3. Despite the very small number of paths in its callees, function `CreateAniIcon` contains too many paths to be explored in twelve hours, which is indicated by the + in Fig. 6.3.

This path explosion is due to two input-dependent loops inside that function, shown in Fig. 6.4. The loop bounds `frames`, which refers to the number of frames in an animated cursor, and `steps`, which refers to the

```
for (i = 0; i < frames; i++)
  frameArrT[i] = frameArr[i];
for (i = 0; i < steps; i++) {
  if (rateArr == NULL)
    rateArrT[i] = rate;
  else
    rateArrT[i] = rateArr[i];
  if (stepArr == NULL)
    stepArrT[i] = i;
  else
    stepArrT[i] = stepArr[i];
}
```

Figure 6.4: The input-dependent loops in function `CreateAniIcon` (code fragment). Variables `frames`, `steps`, `rateArr`, and `stepArr` are inputs to `CreateAniIcon`. All arrays are allocated such that their length is greater than the corresponding loop bound, and therefore, there are no buffer overflows in this code.

number of steps, are both inputs to `CreateAniIcon`, and so are the values of variables `rateArr` and `stepArr`. Since `frames` and `steps` are of type `int` (four bytes), each loop may iterate up to 2^{32} times, which leads to the exploration of 2^{32} possible execution paths and is intractable in practice. Consequently, to control path explosion and unsoundly verify this function, we fixed the values of `frames` and `steps`. For any fixed value of `frames`, the first loop of Fig. 6.4 has only one execution path, while for any fixed value of `steps`, the second loop has always four execution paths due to the tests on the other inputs `rateArr` and `stepArr`. Thus, by fixing these loop bounds to any value from one to 2^{32}, the number of execution paths in the loops of Fig. 6.4 is always four. Tab. 6.1 summarizes how the number of paths in the loops of `CreateAniIcon` changes when fixing `frames` and `steps` to different values. As Tab. 6.1 shows, we can soundly prove memory safety of function `CreateAniIcon` for any fixed number of frames and steps in an animated cursor.

Verification results

During this stage of the process, we unsoundly proved memory safety of only one additional `user32.dll` function of the ANI parser, namely, of `CreateAniIcon`. The box in Fig. 6.3 with the medium shade and single solid line represents `CreateAniIcon` that was unsoundly verified in Stage 2.

Tab. 6.2 presents a complete list of the input-dependent loop bounds that we fixed during this entire verification exercise on the parser. As described above, to (unsoundly) verify memory safety of function `CreateAniIcon` of

Input values		Number of paths
`frames`	`steps`	
0	0	1
1	0	1
any fixed	0	1
0	1	4
0	any fixed	4
any fixed	any fixed	4

Table 6.1: The number of paths in the loops of function `CreateAniIcon` (shown in Fig. 6.4) changes when fixing the input-dependent loop bounds `frames` and `steps` to different values.

the *ANI creation* component (component 5 of Fig. 6.3), we had to fix two input-dependent loops using two whole-program input parameters (namely, frames and steps). In the remainder of this work (see Sect. 6.4.3), we also had to fix two other whole-program input parameters to control a few other input-dependent loops. First, in the *Reading icon guts* component (component 3 of Fig. 6.3), there are three other input-dependent loops, located in functions `ReadIconGuts` and `GetBestImage`. The number of iterations of all these loops depends on the number of images contained in each icon, which corresponds to two bytes per frame of an ANI file. (A single icon may consist of multiple images of different sizes and color depths.) To limit path explosion due to these three loops, we had to fix the number of images per icon of the animated cursor to a maximum of one. Second, in the *Read-*

Type of loop bound	Component	Maximum loop bound
Frames (4 bytes)	5	2^{32}
Steps (4 bytes)	5	2^{32}
Images/frame (2 bytes/frame)	3	1
File size	1	110

Table 6.2: All the input-dependent loop bounds fixed during our attempt to verify the ANI parser. For each loop bound, the table shows the corresponding number of bytes in an ANI input file, the component of the parser containing loops with this bound (numbered as in Fig. 6.3), and the maximum value of the bound that we could soundly verify in twelve hours.

ing and validating file component (component 1 of Fig. 6.3), there are two input-dependent loops, located in functions `LoadCursorIconFromFileMap` and `LoadAniIcon`, whose number of iterations depends on the size of the input file, which we had to restrict to a maximum of 110 bytes.

In summary, it is perhaps surprising that the number of input-dependent loop bounds in the entire parser is limited to a handful of input parameters read from an ANI file, for a total of around ten bytes (plus the input file size) as shown in Tab. 6.2.

6.4.3 Stage 3: Top-down strategy

For the remaining twelve `user32.dll` functions still to be explored in the higher-level part of the call graph of Fig. 6.3, path explosion was still too severe even after using inlining and fixing input-dependent loop bounds. Therefore, after having enforced a bound on the number of execution paths in the parser in the previous stage, we adopted a different, top-down strategy using sub-function summaries to get closer to proving memory safety, in a compositional way (see Sects. 6.1 and 6.2). This strategy was now possible as we had bounded the size of the summaries to be generated by previously fixing the number of iterations of input-dependent loops. Consequently, verification of functions with this strategy is only sound if their number of paths has not been bounded by restricting the number of iterations of an input-dependent loop lower in the call graph, otherwise it is unsound.

Summarization

As we explained earlier, summarizing sub-functions can alleviate path explosion in these sub-functions at the expense of computing re-usable logic summaries that capture function pre- and postconditions, expressed in terms of function inputs and outputs, respectively. For this trade-off to be attractive, it is therefore best to summarize sub-functions (1) that contain many execution paths, and (2) whose input/output interfaces with respect to higher-level functions are not too complex, so that the logic encoding of their summaries remains tractable. Moreover, to prove memory safety of a sub-function with respect to its input buffers, all bounds-checking constraints inside that sub-function must be included in the precondition of its summary (see Sect. 6.2.2).

Verification results

To verify as many executions of the remaining `user32.dll` functions, we manually devised the following summarization strategy, based on the previous data about the numbers of paths in explored sub-functions (i.e., the numbers of paths in the boxes of Fig. 6.3), and by examining the input/output interfaces of the remaining functions. Specifically, we attempted to

verify one by one the top-level function of each remaining component of the parser, namely, `ReadIconGuts` of the *Reading icon guts* component, `ConvertDIBIcon` of *Bitmap conversion*, and `LoadCursorIconFromFileMap` of *Reading and validating file* (since the *Chunk extraction* and *ANI creation* components had already been explored during the previous stages).

Verification of `ReadIconGuts`. (*Reading icon guts* component) We fixed the bounds of the input-dependent loops of this component to a single loop iteration (see Tab. 6.2), as discussed in Sect. 6.4.2, and summarized function `MatchImage`. This function only returns an integer (a "score") that does not influence the control-flow execution of its caller `GetBestImage` for one loop iteration, so its visible postcondition $post_f$ is very simple. Moreover, `MatchImage` takes only one buffer as input, therefore, the precondition of its summary includes only bounds-checking constraints for that buffer. In its caller `GetBestImage`, the size of this buffer is always constant and equal to the size of a structure, so `MatchImage` is attacker memory safe. Overall, when restricting the bounds of the input-dependent loops in the *Reading icon guts* component, summarizing `MatchImage`, and inlining all the other functions below it in the call graph, `ReadIconGuts` contains 468 execution paths that are explored by our tools in 21m 53s.

Verification of `ConvertDIBIcon`. (*Bitmap conversion* component) We soundly verified this function after summarizing sub-function `CopyDibHdr`, whose summarization is also tractable in practice. After summarization, `ConvertDIBIcon` contains 28 execution paths exercised in 1m 58s. Note that, in the *Bitmap conversion* component, there are no input-dependent loops and, consequently, this verification is sound; although sub-function `ConvertPNGToDIBIcon` has loops whose numbers of iterations depend on this function's inputs and therefore could not be verified in isolation, inlining it to its caller `ConvertDIBIcon` eliminated this source of path explosion, and it was then proven to be attacker memory safe.

Verification of `LoadCursorIconFromFileMap`. (*Reading and validating file* component) This is the very top-level function of the parser and the final piece of the puzzle. Since this final step targets the verification of the entire parser, it clearly requires the use of summarization to alleviate path explosion.

Fortunately, and perhaps surprisingly, after closely examining the implementation of the ANI parser's components (see Fig. 6.2), we realized that it is common for their output to be a single "success" or "failure" value. In case "failure" is returned, the higher-level component typically terminates. In case "success" is returned, the parsing proceeds but without reading any other sub-component outputs and with reading other higher-level inputs

(such as other bytes that follow in the input file), i.e., completely independently of the specific path taken in the sub-component being summarized. Therefore, the visible postcondition of function summaries with such interfaces is very simple: a "success" or "failure" value. This is the case for the top-level functions of the lower-level components *Reading icon guts*, *Bitmap conversion*, and *ANI creation*. This was not the case for the *Chunk extraction* component, which mainly consists of auxiliary functions, but does not significantly contribute to path explosion and was not summarized.

More specifically, for the exploration of `LoadCursorIconFromFileMap`, we used three summaries for the following top-level functions of architectural components of the parser:

- `ReadIconGuts`, which returns a pointer to a structure that is checked for nullness in its callers. Then, caller `LoadCursorIconFromFileMap` returns null when this pointer is null. In caller `ReadIconFromFileMap`, in case the pointer is non-null, it is passed as argument to function `ConvertDIBIcon`, which has already been verified for any calling context, as described above.

- `ConvertDIBIcon`, which also returns a pointer to a structure that is checked for nullness in the callers of the function. This pointer is returned by function `ReadIconFromFileMap`, and in case it is non-null, `LoadAniIcon` stores the pointer in an array that is subsequently passed as argument to `CreateAniIcon`, which has already been verified for a restricted calling context (due to the input-dependent loops) during the second stage of the process.

- `CreateAniIcon`, which also returns a pointer to a structure. If this pointer is null, the parser fails and caller `LoadAniIcon` emits an error:

  ```
  if (frames != 0)
    ani = CreateAniIcon(...);
  if (ani == NULL)
    EMIT_ERROR("Invalid␣icon");
  ```

  Otherwise, the pointer is returned by `LoadAniIcon` and subsequently by the top-level function of the parser.

Function `LoadCursorIconFromFileMap` also has an input-dependent loop whose number of iterations depends on the size of the input file being read and containing the ANI file to be parsed. By summarizing the top-level function of each of the above three lower-level components and fixing the file size, we were able to unsoundly prove memory safety of the parser up to a file size of 110 bytes in less than twelve hours. Fig. 6.5 shows the number of execution paths in the parser as well as the time it takes to explore these paths when summarizing components *Reading icon guts*, *Bitmap conversion*, and *ANI creation* and controlling the file size.

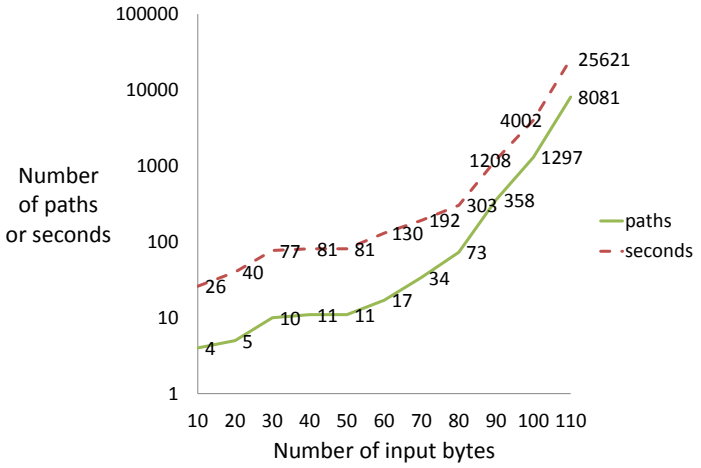

Figure 6.5: The number of execution paths in the top-level function `LoadCursorIconFromFileMap` of the ANI parser and the time (in seconds) it takes to exercise these paths versus the number of input bytes, when summarizing components *Reading icon guts*, *Bitmap conversion*, and *ANI creation*.

With this top-down strategy and the careful use of function summarization, we were able to (soundly or unsoundly) prove memory safety of the remaining twelve `user32.dll` functions of the parser. Note that by inlining functions that were previously verified in isolation, we also proved that accesses to input buffers of these functions are memory safe. The boxes in Fig. 6.3 with the darker shade and double solid lines represent the remaining `user32.dll` functions that were explored during this stage of the process.

6.5 Memory-safety bugs

In reality, the verification attempt of the ANI Windows parser was slightly more complicated than presented in the previous section because the ANI parser is actually *not memory safe*! Specifically, we found three types of memory-safety violations during the course of this work:

- real bugs (fixed in the latest version of Windows),
- harmless bugs (off-by-one non-exploitable buffer overflows),
- code parts not memory safe by design.

We discuss each of these memory-safety violations in this section. Details are omitted for proprietary reasons. The results presented in Sect. 6.4 were actually obtained *after fixing or ignoring* these bugs, as explained below.

Real bugs

We found several buffer overflows all related to the same root cause. Function `ReadIconGuts` of the *Reading icon guts* component allocates memory for storing a single icon extracted from the input file and returns a pointer to this memory. The allocated memory is then cast to a structure, whose fields are read for accessing sub-parts of the icon, such as its header. However, the size of an icon, and therefore the size of the allocated memory, depends on the (untrusted) declared size of the images that make up the icon. These sizes are declared in the ANI file and might not correspond to the actual image sizes. Consequently, if the declared size of the images is too small, then the size of the allocated memory is too small, and there are buffer overflows when accessing the fields of the structure located beyond the allocated memory for the icon. These buffer overflows have been fixed in the latest version of Windows, but are believed to be hard to exploit and, hence, not security critical.

Harmless bugs

We also found several harmless buffer overflows related to the bugs described above. For instance, function `ConvertPNGToDIBIcon` of the *Bitmap conversion* component converts an icon in PNG format to DIB (Device Independent Bitmap), and also takes as argument a pointer to the above structure for the icon. To determine whether an icon is in PNG format, `ConvertPNGToDIBIcon` checks whether the icon contains the eight-byte PNG signature. However, the allocated memory for the icon may be smaller than eight bytes, in which case there can be a buffer overflow. Still, on Windows, every memory allocation (call to `malloc`) always results in the allocation of a reserved memory block of at least eight bytes. So, technically, accessing any buffer `buf` of size less than eight up to `buf + 7` bytes is not a buffer overflow according to the Windows runtime environment—such buffer overflows are harmless to both reliability and security.

Code parts not memory safe by design

Finally, we found memory-safety violations that were expected and caught as runtime exceptions using try-except-statements. For instance, `CopyDibHdr` of the *Bitmap conversion* component copies and converts an icon header to a common header format. The size of the memory that is allocated in `CopyDibHdr` for copying the icon header depends on color information defined in the header itself. This color information is read from the input file, and is therefore untrusted. Specifically, it can make the parser allocate a huge amount of memory, which is often referred to as a *memory spike*. Later, the actual header content is copied into this memory. To check whether the declared size matches the actual size, `CopyDibHdr` uses a try-statement to

probe the icon header in chunks of 4K bytes, i.e., the minimum page size, and ensure that the memory is readable and properly initialized, as shown below:

```
try {
  DWORD offset;
  for (offset = 0; offset < alloc; offset += 0x1000)
    *(volatile BYTE*) ((LPBYTE)hdr + offset);
  *(volatile BYTE*) ((LPBYTE)hdr + alloc - 1);
} except (W32ExceptionHandler(FALSE, RIP_WARNING)) {
  return NULL;
}
```

In the code above, variable `alloc` is the untrusted declared size, while variable `hdr` is a pointer to a buffer whose size is the real header size. While probing the icon header inside the try-statement, the parser may access unallocated memory beyond the bounds of the header, which is a memory-safety violation. However, this violation is expected to be caught in the except-statement, which returns null and aborts parsing in higher-level functions.

False alarms

Since some of the above buffer overflows were detected when testing functions in isolation, for any calling context, we then determined whether these buffer overflows are real bugs, that is, whether they also manifest themselves in the context of the entire parser.

To achieve this, we identified the position of the (untrusted) input bytes, which were to blame for the bugs, in a well-formed input file. We then changed the values of these bytes in the well-formed input file to the "buggy" values we had previously found, and ran the entire ANI parser on the modified file. We could then witness that these buffer overflows were still triggered, hence proving that there was no input validation on the modified input bytes anywhere else in the ANI parser, and that these buffer overflows were reproducible and not false alarms. Note that we found no false positives during this work: all the buffer overflows we detected were indeed due to accesses to unallocated memory. In a similar way, we were able to demonstrate that code surrounding the try-except-statement shown above could be tricked in two different ways into allocating 1MB and 1.5GB of memory, respectively, in function `ReadIconGuts`; in both cases though, the memory is freed before the function returns, so memory spikes are not observable.

The above buffer overflows manifested themselves as cases of divergence from the program paths expected by SAGE and MicroX. In these cases, the cause of the divergence was either a bug in our tools (that we immediately fixed) or in the ANI parser. Moreover, exceptions thrown by the code of the parser were detected when the exception handling mechanism of the operating system was triggered. If a function under test throws an unhan-

dled exception, a potential bug has been detected depending on whether the exception is handled higher in the call graph of the parser. In such situations, we simply checked whether all calls to this function were wrapped in appropriate try-except-statements, which was always the case.

Validity of verification results

The results of Sect. 6.4 were obtained after fixing or ignoring the memory-safety bugs discussed in this section. Those results are therefore sound only with respect to these additional assumptions. For example, in the try-except-statement above, variable `alloc` is input dependent and the for-loop is an input-dependent loop (as defined in Sect. 6.4.2) causing severe path explosion. To avoid path explosion due to such memory-unsafe code patterns, we restricted the values that variables like `alloc` can take.

6.6 Challenges

To obtain the results described in Sect. 6.4, we came across a number of unexpected challenges.

For instance, exception handling can vary even between different versions of the same operating system, and our results for code that throws exceptions were interpreted differently on 32- and 64-bit Windows 7.

Another example involves cases of path divergence caused by the Fault Tolerant Heap (FTH) sub-system of Windows 7. FTH is responsible for monitoring application crashes, and may autonomously apply mitigations to prevent future crashes. This caused divergence when FTH unexpectedly interfered with certain test runs but not with others. We solved this problem by entirely disabling FTH on the system.

Win32 structured exception handling created further complications as it requires the execution of function prologue code that modifies the stack. These modifications can confuse MicroX when determining the inputs of a function that are passed through the stack, and special care had to be taken.

6.7 Related work

Traditional *interactive* program verification, using static program analysis, verification-condition generation, and theorem proving, provides a broader framework for proving more complex properties of a larger class of programs, but at the expense of more work from the user. For instance, the VCC [36] project verified the *functional correctness*, including memory safety and race freedom, of the Microsoft Hyper-V hypervisor [94], a piece of concurrent software (100K lines of C, 5K lines of assembly) that runs between x64 hardware and guest operating systems, and provides isolated execution environments,

called partitions. This verification effort required more than 13.5K lines of source-code annotations for specifying contracts, loop invariants, and ghost state in about 350 functions, by more than ten people and over several years.

As another impressive example, the seL4 project [88] designed and verified the C code of a microkernel, using the interactive theorem prover Isabelle/HOL [120], and requiring about 200K lines of Isabelle scripts and 20 years of research in developing and automating the proofs.

Moreover, Typed Assembly Language [112] (TAL) and the Boogie program verifier [11] were used to prove type and memory safety of the Verve operating system [141], which consists of a low-level "Nucleus", written in x86 assembly, and a higher-level kernel, written in safe C#. The exported functions of the Nucleus code (a total of 20 functions implemented in approximately 1.5K lines of x86 assembly) were verified and manually annotated with pre- and postconditions, loop invariants, and external function stubs for a total of 1,185 lines of annotations in about nine months of work.

In contrast, our verification attempt required only three months of work, no program annotations, no static program analysis, and no external function stubs, although our scope was more focused (attacker memory safety only), our application domain was different (sequential image parser versus concurrent or reactive operating-system code), and we required key manual steps, like user-guided program decomposition and summarization. Furthermore, contrary to the above verification projects, we gave up soundness by fixing a few input-dependent loop bounds, as discussed in Sect. 6.4. Note that our purely dynamic techniques and x86-based tools can handle complicated ANI x86 code patterns, such as stack-modifying, compiler-injected code for structured exception handling (SEH prologue and epilogue code for try-except-statements), and stack-guard protection, which most static analysis tools cannot handle. For example, the abstract interpreter ASTRÉE [41] does not support dynamic memory allocation.

Despite this limitation, statically proving memory safety of a program is possible. However, proving *attacker memory safety*, even more so *compositionally*, is novel: we prove that an attacker cannot trigger buffer overflows, but ignore other buffer overflows (for instance, due to the failure of trusted system calls). This requires a whole-program taint analysis to focus on what the attacker can control, performed using symbolic execution and the top-down strategy of Sect. 6.4.3. In contrast, other approaches, like verification-condition generation, bounded model checking, abstract interpretation, or traditional static analysis, lack this global taint view and treat all program statements alike, without prioritizing the analysis toward parts on the attack surface, which hampers scalability and relevance to security.

Moreover, static analysis involves reading the (source or binary) code of a program and analyzing it, e.g., by generating a logic formula representing the program or its transition relation, which is then unfolded. In contrast, dynamic symbolic execution does not know what the entire program is.

Static-analysis-based software model checkers, for instance, SLAM [8], BLAST [80], and YOGI [121], can *automatically* prove control-oriented API properties of specific classes of programs (specifically, device drivers). These tools rely on (predicate) abstraction in order to scale, and are not engineered to reason precisely about pointers, memory alignment, and aliasing. They were not designed and cannot be used as-is for proving (attacker) memory safety of an application as large and complex as the ANI Windows parser. In addition, some model checkers of this category, such as SLAM, are unsound [90].

SAT- and SMT-based bounded model checkers, like CBMC [35] and Corral [91], are another class of static analysis tools for automatic program verification. The program's logic representation generated by such model checkers is similar to verification-condition generation, and captures both data and control dependencies on all program variables, which is comparable to *eagerly* summarizing (as in Sect. 6.1) *every* program block and function. Even excluding all loops, such a monolithic whole-program logic encoding would quickly become too large and complex, and consequently not scale to the entire ANI parser. Corral, however, would perform better than other tools in this category by using stratified inlining, that is, by over-approximating the summaries it computes and then refining them based on counterexample-guided abstraction refinement (CEGAR) [34]. Similarly to our approach, bounded model checking is typically unsound.

As shown in Sect. 6.4, systematic dynamic test generation also does not scale to the entire ANI parser without the selective use of function summarization and compromising soundness by fixing a few input-dependent loop bounds. These crucial steps were performed manually in our work. Algorithms and heuristics for *automatic* program summarization have been proposed before [66, 2, 89] as well as other closely related techniques [15, 107] and heuristics [73], which can be viewed as approximations of sub-program summarization. However, none of this prior work on automatic summarization has ever been applied toward *verifying* an application as large and complex as the parser considered here.

We do not know which parts of the ANI code are in the subset of C for which tools like CCured [117] or Prefix [18] are sound, or how many memory-safety checks could be removed in those parts with such a sound static analysis. However, we do know that Prefix was run on this code for years, yet bugs remained, which is precisely why dynamic symbolic execution is performed later [16].

In a different context, dynamic test generation has been applied to check and potentially prove equivalence of floating-point and Single Instruction Multiple Data (SIMD) code [37], up to a certain input size and with the use of phi node folding to statically merge paths. A similar approach was later designed to crosscheck OpenCL and C/C++ programs, up to a certain input size and number of threads [38].

6.8 Summary and remarks

For the first time, we attempted to prove attacker memory safety of an entire operating-system image parser, in only three months of work, using compositional exhaustive testing, i.e., no static analysis whatsoever. These results required a high-level of automation in our tools and verification process, although key steps were performed manually, like fixing input-dependent loop bounds, guiding the summarization strategy, and avoiding memory-safety violations. Also, the scope of our work was only to prove attacker memory safety, not general memory safety or functional correctness, and the ANI parser is a purely sequential program. Finally, the verification guarantees provided by our work are valid only with respect to some important assumptions we had to make, mostly regarding input-dependent loop bounds. Overall, after this work, we are now confident that the presence of any remaining security-critical (i.e., attacker-controllable) buffer overflows in the ANI Windows parser is unlikely, but this conclusion is subject to the assumptions we made.

Here are some interesting findings that we did not expect at the beginning of this project:

- Many ANI functions are loop free and were easy to verify (Sect. 6.4.1).
- All the input-dependent loops in the entire ANI parser are controlled by the values of about ten bytes only in any ANI file plus the file size (Sect. 6.4.2).
- The remaining path explosion can be controlled by using only five function summaries with very simple interfaces (Sect. 6.4.3).

We expect that these findings are also representative of other image parsers, given that the ANI parser is organized in very common architectural components for this application domain, such as the *Chunk extraction* and *Bitmap conversion* components. Moreover, since the general format of an ANI file is based on RIFF, used for storing various types of data such as video or digital audio, we anticipate that our findings also generalize to certain media players.

In hindsight, there are several things we would now do differently. Mostly, the verification results obtained for the lower-level functions with the bottom-up strategy were often stronger than necessary for verifying the higher-level functions. Some of that work could have been avoided, although this stage was useful to familiarize ourselves with the code base, input-dependent loops, etc., and provided early, encouraging verification results.

Our work suggests future directions for automating the steps that were done manually, in particular, decomposing the program at cost-effective interfaces, and dealing with few, but critical, input-dependent loops. In the

next chapter, we investigate how to automatically perform program decomposition and summarization at simple interfaces, without any user input.

The path explosion caused by input-dependent loops could be addressed by providing loop invariants, which can be used to generate summaries for such loops. These invariants may be provided manually, like in Ch. 5, or automatically, for instance, with an abstract interpretation tool. In fact, an abstract interpreter could be applied to prove any remaining properties that have not been soundly verified by the dynamic test generation, such as those that have been verified under the assumption of a fixed number of iterations of an input-dependent loop. This is the opposite direction to combining verification and systematic testing than we explored in Ch. 2. In Ch. 8, we outline what we have learned from each of the two parts in this dissertation, and discuss what we now expect to happen in practice.

Chapter 7

IC-Cut: A compositional search strategy for dynamic test generation

In Ch. 6, we use SAGE [73] to evaluate to what extent systematic dynamic test generation can be pushed toward program verification of the ANI Windows image parser [29]. In this previous chapter, we limit path explosion in the parser with user-guided program decomposition and summarization [66, 2]. In particular, we *manually* identify functions for summarization whose input/output interfaces with respect to higher-level functions in the call graph are not too complex, so that the logic encoding of their summaries remains simple. Indeed, we find that it is common for functions to return a single "success" or "failure" value. If "failure" is returned, the higher-level function typically terminates. If "success" is returned, parsing proceeds with new chunks of the input, that is, completely independently of the specific path taken in the function being summarized. We, therefore, decompose the program at very few interfaces, of functions that parse independent chunks of the input and return a single "success" or "failure" value.

Based on these previous insights, we now define a new compositional search strategy for *automatically* and dynamically discovering simple function interfaces, where large programs can be effectively decomposed. IC-Cut, short for "Interface-Complexity-based Cut", tests the decomposed program units independently, records their test results as low-complexity function summaries (that is, summaries with simple logic encoding), and reuses these summaries when testing higher-level functions in the call graph, thus limiting overall path explosion. IC-Cut runs on-the-fly during the search to incrementally refine interface cuts as the search advances and increase its precision. In short, IC-Cut is inspired by compositional reasoning, but is only a search strategy, based on heuristics, for decomposing the program into independent units that process different chunks of the input. We, therefore, do not perform compositional verification in this work, except when certain particular restrictions are met (see Sects. 7.1.4 and 7.2).

The main contributions of this chapter are:

- We present a principled alternative to ad-hoc state-of-the-art search heuristics for alleviating path explosion.
- As our experiments show, IC-Cut preserves or even increases code coverage and bug finding in significantly less time, compared to the current generational-search strategy of SAGE.
- IC-Cut can identify which decomposed program units are exhaustively tested and, thus, *dynamically verified*.

Outline. This chapter is organized as follows. Sect. 7.1 explains the IC-Cut search strategy in detail. In Sect. 7.2, we present our experimental results obtained when testing the ANI Windows image parser. We review related work in Sect. 7.3.

7.1 The IC-Cut search strategy

In this section, we present the IC-Cut search algorithm, precisely define the low-complexity function summaries of IC-Cut, and discuss its correctness guarantees and limitations.

7.1.1 Algorithm

Alg. 7.1 presents the IC-Cut search strategy. IC-Cut consists of three phases, which are overlapping: learning, decomposition, and matching.

Learning

The learning phase of IC-Cut runs the program under test on a set of seed inputs. The goal is to discover as much of the call graph of the program. As a result, the larger the set of seed inputs, the more detailed is the global view that IC-Cut has of the program, and the fewer new functions are discovered in the next phase. Note that by the term "learning", we simply refer to discovering the call graph of the program, that is, we do not extrapolate based on a given training set as in classical machine-learning techniques.

On line 2 of Alg. 7.1, function CreateCallgraph returns the call graph of the program that is learned, dynamically and incrementally, by running the program on the seed inputs. Each node in the call graph represents a function of the program, and contains the function name and one seed input that steers execution of the program through this function. Each edge (f, g) in the call graph denotes that function f calls function g. Note that we assume no recursion.

Handling recursion is conceptually possible [66]. In practice, it is not required for the application domain of binary image parsers. Recursion in such parsers is very rare due to obvious performance, scalability, and reliability reasons, which is why we do not address it in this work.

Algorithm 7.1: The IC-Cut search algorithm.

```
1  function IC-Cut(p, seeds)
2    cg ← CreateCallgraph(p, seeds)
3    summaries ← {}
4    Explore(cg, p, summaries)
5
6  function Explore(cg, p, summaries)
7    workQueue ← GetLeaves(cg)
8    while IsNotEmpty(workQueue) do
9      f ← Peek(workQueue)
10     cg′, summaries ← Process(f, p, summaries)
11     if cg′ == cg then
12       workQueue ← Dequeue(workQueue)
13       predecessors ← GetPredecessors(f, cg)
14       workQueue ← Enqueue(predecessors, workQueue)
15     else
16       newFunctions ← GetNewFunctions(cg, cg′)
17       workQueue ← AddFirst(newFunctions, workQueue)
18       cg ← cg′
19
20 function Process(f, p, summaries)
21   seed ← GetSeed(f)
22   interface, cg′ ← DSE(f, p, seed, summaries)
23   if IsSummarizable(interface) then
24     summary ← GenerateSummary(interface)
25     summaries ← PutSummary(f, summary, summaries)
26   return cg′, summaries
```

Decomposition

During the decomposition phase, IC-Cut tests one function at a time, that is, it explores each function using dynamic symbolic execution. IC-Cut starts testing functions at the bottom of the learned call graph, and potentially records the function test results as a low-complexity summary (that is, a summary with a simple logic encoding, as defined in Sect. 7.1.2). This is done in function Explore of Alg. 7.1, which is called on line 4 and takes as arguments the call graph *cg*, the program under test *p*, and an empty map from call-graph nodes to function summaries *summaries*.

In particular, IC-Cut selects a function from the bottom of the call graph that has not been previously tested. This is shown on line 7 of Alg. 7.1, in function Explore, where we create a *workQueue* of the call graph leaf-nodes, and on line 9, where a function f is selected from the front of the

workQueue. The selected function is then tested independently (in function PROCESS) to determine whether its interface is simple enough to be cost-effective for summarization. To test the selected function, IC-Cut chooses an appropriate seed input, which in the previous phase has been found to steer execution of the program through this function (line 21 of Alg. 7.1). Subsequently, on line 22, IC-Cut tests the program starting with this seed input, using dynamic symbolic execution.

However, while testing the program, not all symbolic constraints that IC-Cut collects may be negated; we call the constraints that may be negated *open*, and all others *closed*. (In other words, an open constraint is added to the path constraint as a branching condition BRANCH(c), whereas a closed constraint as an assumed condition ASSUME(c), as these conditions are defined in Sect. 1.1.) Specifically, the constraints that are collected until execution encounters the *first* call to the selected function are closed. Once the function is called, the constraints that are collected until the function returns are open. As soon as the function returns, symbolic execution terminates. This means that IC-Cut tests only the selected function and for a single calling context of the program. Note that the function is tested using a *generational search*.

While testing the selected function, IC-Cut dynamically determines the complexity of its interface, as defined in Sect. 7.1.2. If the function interface is simple enough to be cost-effective for summarization (line 23 of Alg. 7.1), the test results of the function are recorded as a summary. On line 24, we generate the function summary, and on line 25, we add it to the *summaries* map. Note that function PROCESS describes our algorithm in a simplified way. If a function interface is found to be suitable for summarization, IC-Cut actually records the summary *while* testing the function. If this is not the case, IC-Cut aborts testing of this function. How summaries are generated is precisely documented in Sect. 7.1.2.

It is possible that new functions are discovered during testing of the selected function, i.e., functions that do not appear in the call graph of the learning phase. When this happens, IC-Cut updates the call graph. Of course, these new functions are placed lower in the call graph than the currently-tested function, which is their (direct or indirect) caller. IC-Cut then selects a function to test from the bottom of the updated call graph.

This is shown on lines 11–18 of Alg. 7.1. If no new functions are discovered during testing of the selected function (line 11), we remove this function from the *workQueue*, and add its predecessors in the call graph at the end of the *workQueue* (lines 12–14). When IC-Cut explores these predecessors, their callees will have already been tested. If, however, new functions are discovered (lines 15–16), we add these functions at the front of the *workQueue* (line 17), and update the call graph (line 18). Note that when new functions are discovered, IC-Cut aborts exploration of the currently-tested function; this is why this function is not removed from the *workQueue* on line 17.

The above process highlights the importance of the set of seed inputs in the learning phase: the better the set of seed inputs is in call-graph coverage, the less time is spent on switches between the decomposition and learning phases of IC-Cut.

Matching

In general, summaries can be reused by callers to skip symbolic execution of a summarized callee and, hence, alleviate path explosion caused by inlining the callee, i.e., by re-exploring all callee paths.

The matching phase decides whether a recorded summary may be reused when testing higher-level functions in the call graph. This is why function DSE of Alg. 7.1 (line 22) takes the *summaries* map as argument. On the whole, DSE explores (using dynamic symbolic execution) one function at a time, records its interface, and reuses previously-computed summaries.

In our context, while testing a higher-level function in the decomposition phase, the exploration might come across a call to a function for which a summary has already been computed. Note, however, that this summary has been computed for a particular calling context. Therefore, the matching phase determines whether the encountered calling context of the function matches (precisely defined in Sect. 7.1.2) the old calling context for which the summary has been computed. If this is the case, it is guaranteed that all execution paths of the function for the encountered calling context are described by the recorded summary. Consequently, the summary may be reused, since no execution paths of the function will be missed. If, on the other hard, the calling contexts do not match, the called function is tested as part of the higher-level function (that is, it is inlined to the higher-level function) as if no summary had been recorded, to avoid missing execution paths or generating false alarms. In other words, IC-Cut allows, arbitrarily and for simplicity, that a function is summarized only for a single calling context, and thus, summary reuse must be calling-context specific.

7.1.2 Function summaries

Before describing which constraints on interface complexity a function must satisfy to be summarized, we first precisely define function inputs and outputs. (These definitions differ from the ones presented in Ch. 6.)

Function inputs and outputs

– An *input* i_f of function f is any value that is read and tested by f. In other words, the value of i_f is not only read in f, but also affects which execution path of the function is taken at runtime.

- An input i_f of f is *symbolic* if it is a function of any whole-program inputs; otherwise, i_f is *concrete*.
- A *candidate output* co_f of function f is any value that is written by f.
- An *output* o_f of function f is any candidate output of f that is tested later in the program.

Consider program `P` below, which expects two non-negative inputs `a` and `b`:

```
int is_less(int x, int y) {
  if (x < y)
    return 1;
  return 0;
}

void P(int a, int b) {
  if (is_less(a, 0) || is_less(b, 0))
    abort;
  ...
}
```

For both calling contexts of function `is_less` in program `P`, `is_less` has one symbolic input (that is, `a` or `b`), one concrete input (that is, `0`), and one output (which is `0` or `1` and tested by the if-statement in `P`).

Generating summaries

Recall from the previous chapter that, in compositional symbolic execution [66, 2], a summary ϕ_f for a function f may be computed by symbolically executing all paths of function f, generating an input precondition and output postcondition for each path, and gathering all of these path summaries in a disjunction.

Precisely, ϕ_f is defined as a disjunction of formulas ϕ_{w_f} of the form

$$\phi_{w_f} = pre_{w_f} \wedge post_{w_f}$$

where w_f denotes an intra-procedural path in f, pre_{w_f} is a conjunction of constraints on values read by f, and $post_{w_f}$ a conjunction of constraints on values written by f. For instance, a summary ϕ_f for function `is_less` is

$$\phi_f = (x < y \wedge ret = 1) \vee (x \geq y \wedge ret = 0)$$

where *ret* denotes the value returned by the function. This summary may be reused across different calling contexts of `is_less`. In practice, however, these disjunctions of conjunctions of constraints can become very large and complex, thus making summaries expensive to compute. For this reason,

IC-Cut generates only low-complexity function summaries for specific calling contexts.

For a given calling context, a function f is summarized by IC-Cut only if the following two conditions are satisfied:

- All symbolic inputs of f are unconstrained, that is, they are completely independent of the execution path taken in the program until function f is called. In particular, the symbolic inputs of f do not appear in any of the closed constraints collected before the call to f. Therefore, the input precondition of f must be true.
- Function f has at most one output o_f.

Note that the latter condition is based on our insights from Ch. 6, also described at the beginning of this chapter. In principle, any number of outputs of f could be allowed for summarization. However, as we discuss in Sect. 7.1.4, the summaries computed by IC-Cut can be unsound under certain circumstances, and the larger the number of outputs the more execution paths might be missed by IC-Cut (see Thm. 2). If the above conditions are not satisfied, function f is inlined to its calling contexts (that is, not summarized). As an example, consider again program `P`. For the first calling context of function `is_less` in `P` (that is, `is_less(a, 0)`), the symbolic input of `is_less` is unconstrained, and the function has exactly one output. As a result, `is_less` is summarized by IC-Cut for this *first* calling context, as described in Sect. 7.1.1.

As a consequence of these conditions, the summaries considered in this work have a single precondition on all symbolic inputs, which is true, and a single precondition on all concrete inputs, which is of the form

$$\bigwedge_{0 \leq j < N} i_j = c_j$$

where i_j is a concrete input, c_j a constant representing its concrete value, and N the number of concrete inputs. Moreover, the summaries in this work have no output postconditions, as explained later in this section. As a result, when IC-Cut generates a summary for a function f, it actually records a precondition of the above form on all concrete inputs of f; this precondition also represents the current calling context of f. In this chapter, we abuse terminology and call such preconditions "summaries", although we do not record any disjunctions or postconditions. For example, in the program `P` above, IC-Cut generates the following summary for the first calling context of function `is_less`

$$y = 0$$

which denotes that all inputs of `is_less` except for y are symbolic and unconstrained, and that y is a concrete input whose value is zero in the

particular calling context. This summary indicates that function `is_less` has been tested for a calling context in which x may take any value, while y must have the value zero.

Reusing summaries

While testing a higher-level function in the decomposition phase of IC-Cut, the exploration might come across a call to a function for which a summary has already been generated. Then, the matching phase determines if this summary may be reused by checking whether the new calling context of the function matches, i.e., is equally or more specific than, the old calling context for which the summary has been recorded (see Sect. 7.1.1).

- The new calling context is *as specific* as the old calling context only if (1) the function inputs that are symbolic and unconstrained in the old calling context are also symbolic and unconstrained in the new calling context, and (2) all other function inputs are concrete and have the same values across both calling contexts, except in the case of non-null *pointers* whose concrete values may differ since dynamic memory allocation is non-deterministic (see Sect. 7.1.4 for more details).

- The new calling context is *more specific* than the old calling context only if (1) the function inputs that are concrete in the old calling context are also concrete in the new calling context and have the same values (except in the case of non-null pointers), and (2) one or more function inputs that are symbolic and unconstrained in the old calling context are either symbolic and constrained in the new calling context or they are concrete.

Recall that, in our previous example about program `P`, IC-Cut records a summary for the first calling context of function `is_less` in `P`. This summary is also reused in the second calling context of `is_less` in `P` (that is, `is_less(b, 0)`), which is as specific as the first.

After having described *when* a recorded summary may be reused, we now explain *how* this is done. When the matching phase of IC-Cut determines that a function summary matches a calling context of the function, the following two steps are performed:

1. The function is executed only concretely, and not symbolically, until it returns.

2. The function candidate outputs are associated with fresh symbolic variables.

Step (1) is performed because all execution paths of the function have already been explored when testing this function independently for an equally

or more general calling context. Step (2) is used to determine whether the function has at most one output, as follows.

When testing a function f for a given calling context, we can determine all values that are written by f, which we call *candidate outputs*. Yet, we do not know whether these candidate outputs are tested later in the program, which would make them *outputs* of f. Therefore, when reusing a summary of f, we associate fresh symbolic variables with all of its candidate outputs. We expect that at most one of these candidate outputs is ever tested later in the program. If this condition is not satisfied, the summary of f is invalidated. In this case, the higher-level function that reused the summary of f is tested again, but this time, f is inlined to its calling contexts instead of summarized.

When reusing the summary of function `is_less` in program `P`, we associate a symbolic variable with the function's only candidate output, its return value. This symbolic variable is tested by function `P`, in the condition of the if-statement, thus characterizing the return value of `is_less` as a function output. Given that the summary of `is_less` is computed to be very simple and, when reusing it, the function is executed only concretely, IC-Cut misses some of those execution paths in `P` that are guarded by the value of the function output (that is, either the then- or the else-branch of the if-statement in `P`). Overall, IC-Cut explores both paths in `is_less` when computing its summary, and a single path in `P` (instead of four) when reusing the summary. This is because function `is_less` is executed only concretely for both calling contexts.

Note that when the outputs of the summarized functions do not affect which path is taken later in the program, IC-Cut does not miss any execution paths. For instance, consider the following program `Q`:

```
int* Q(int a, int b) {
  int buf[2];
  buf[0] = is_less(a, 0);
  buf[1] = is_less(b, 0);
  return buf;
}
```

In this case, IC-Cut again explores two paths in `is_less` when computing its summary, and a single path in `Q` (instead of four) when reusing the summary, but no paths or bugs are missed.

7.1.3 Input-dependent loops

We use constraint subsumption [73] to automatically detect and control input-dependent loops. Subsumption keeps track of the constraints generated from a given branch instruction. When a new constraint c is generated, SAGE uses a fast syntactic check to determine whether c implies or is implied by a previous constraint, generated from the same instruction during execution, most likely due to successive iterations of an input-dependent

loop. If this is the case, the weaker (implied) constraint is removed from the path constraint.

In combination with subsumption, which eliminates the weaker constraints generated from the same branch, we can also use *constraint skipping*, which never negates the remaining stronger constraints injected at this branch. When constraint subsumption and skipping are both turned on, an input-dependent loop is concretized, that is, it is explored only for a fixed number of iterations.

7.1.4 Correctness

We now discuss the correctness guarantees of the IC-Cut search strategy. The following theorems hold assuming symbolic execution has perfect precision, i.e., that constraint generation and solving are sound and complete for all program instructions.

We define an *abort-statement* in a program as any statement that triggers a program error.

Theorem 1. *(Completeness) Consider a program P. If IC-Cut reaches an abort, then there is some input to P that leads to an abort.*

Proof sketch. The proof is immediate by the completeness of dynamic symbolic execution [70, 66]. In particular, it is required that the summaries of IC-Cut are not over-approximated, but since these summaries are computed using dynamic symbolic execution, this is guaranteed. □

Theorem 2. *(Soundness) Consider a program P. If IC-Cut terminates without reaching an abort, no constraints are subsumed or skipped, and the functions whose summaries are reused have no outputs and no concrete, non-null pointers as inputs, then there is no input to P that leads to an abort.*

Proof sketch. The proof rests on the assumption that any potential source of unsoundness in the IC-Cut summarization strategy is conservatively detected. There are exactly two sources of unsoundness: (1) constraint subsumption and skipping for automatically detecting and controlling input-dependent loops, and (2) reusing summaries of functions that have a single output and concrete, non-null pointers as inputs.

Constraint subsumption and skipping remove or ignore non-redundant constraints from the path constraint to detect and control successive iterations of input-dependent loops. By removing or ignoring such constraints, these techniques omit paths of the program and are, thus, unsound.

When reusing the summary of a function with a single output, certain execution paths of the program might become infeasible due to the value of its output. As a result, IC-Cut might explore fewer execution paths than are feasible in practice. On the other hand, summaries of functions with

no outputs are completely independent of the execution paths taken in the program. Therefore, when such summaries are reused, no paths are missed. Note that by restricting the function outputs to at most one, we set an upper bound to the number of execution paths that can be missed, that is, in comparison to reusing summaries of functions with more than one output.

When reusing the summary of a function that has concrete, non-null pointers as inputs, execution paths that are guarded by tests on the values of these pointers might be missed, for instance, when two such pointers are compared for aliasing. This is because we ignore whether the values of such inputs actually match the calling context where the summary is reused, to deal with the non-determinism of dynamic memory allocation.

The program units for which the exploration of IC-Cut is sound and does not lead to an abort are dynamically verified. □

7.1.5 Limitation: Search redundancies

It is worth emphasizing that IC-Cut may perform redundant sub-searches in two cases: (1) partial call graph, and (2) late summary mismatch, as detailed below. However, as our evaluation shows (Sect. 7.2), these limitations seem outweighed by the benefits of IC-Cut in practice.

Partial call graph

This refers to discovering functions during the decomposition phase of IC-Cut that do not appear in the call graph built in the learning phase. Whenever new functions are discovered, testing is aborted in order to update the call graph, and all test results of the function being tested are lost.

Late summary mismatch

Consider a scenario in which function `foo` calls function `bar`. At time t, `bar` is summarized because it is call-stack deeper than `foo` and the interface constraint on `bar`'s inputs is satisfied. At time $t + i$, `foo` is explored while reusing the summary for `bar`, and `bar`'s candidate outputs are associated with symbolic variables. At time $t + i + j$, while still exploring `foo`, the interface constraint on `bar`'s outputs is violated, and thus, the summary of `bar` is invalidated. Consequently, testing of `foo` is aborted and restarted, this time by inlining `bar` to its calling context in `foo`.

7.2 Experimental evaluation

In this section, we present detailed experimental results obtained when testing the ANI Windows image parser, which is available on every version of Windows.

As described in the previous chapter, this parser processes structured graphics files to display "ANImated" cursors and icons, like the spinning ring or hourglass on Windows. The ANI parser is written mostly in C, while the remaining code is written in x86 assembly. It is a large benchmark consisting of thousands of lines of code spread across hundreds of functions. The implementation involves at least 350 functions defined in five Windows DLLs. The parsing of input bytes from an ANI file takes place in at least 110 functions defined in two DLLs, namely, in `user32.dll`, which is responsible for 80% of the parsing code, and in `gdi32.dll`, which is responsible for the remaining 20% [29] (see Ch. 6).

Our results show that IC-Cut alleviates path explosion in this parser while preserving or even increasing code coverage and bug finding, compared to the current generational-search strategy used in SAGE. Note that by "generational-search strategy used in SAGE", we mean a monolithic search in the state space of the entire program.

For our experiments, we used five different configurations of IC-Cut, which we compared to the generational-search strategy that is implemented in SAGE. All configurations are shown in Tab. 7.1. For each configuration, the first column of the table shows its identifier and whether it uses IC-Cut. Note that configurations A–E use IC-Cut, while F uses the generational-search strategy of SAGE. The second column shows the maximum runtime for each configuration: configurations A–E allow for a maximum of three hours to explore each function of the parser (since the exploration is per function), while F allows for a total of 48 hours to explore the entire parser (since the exploration is whole program). The four rightmost columns of the table indicate whether the following options are turned on:

- *Summarization at maximum runtime*: Records a summary for the currently-tested function when the maximum runtime is exceeded if no conditions on the function's interface complexity have been violated;
- *Constraint subsumption*: Eliminates weaker constraints implied by stronger constraints generated from the same branch instruction (using a fast syntactic check), most likely due to successive iterations of an input-dependent loop (see Sect. 7.1.3);
- *Constraint skipping*: Does not negate stronger constraints that imply weaker constraints generated from the same branch instruction (see Sect. 7.1.3);
- *Flip count limit*: Establishes the maximum number of times that a constraint generated from a particular program instruction may be negated [73].

Note that F is the configuration of SAGE that is currently used in production.

Configuration ID	IC-Cut	Maximum runtime	Summarization at maximum runtime	Constraint subsumption	Constraint skipping	Flip count limit
A	✓	3h/function		✓	✓	
B	✓	3h/function		✓	✓	✓
C	✓	3h/function	✓	✓	✓	
D	✓	3h/function		✓		
E	✓	3h/function	✓	✓		
F		48h		✓		✓

Table 7.1: All configurations used in our experiments; we used five different configurations of IC-Cut (A–E), which we compared to the generational-search strategy of SAGE (F).

Fig. 7.1 shows the instructions of the ANI parser that are covered by each configuration. We partition the covered instructions in those that are found in `user32.dll` and `gdi32.dll` (projected coverage), and those that are found in the other three DLLs (remaining coverage). Note that the instructions in `user32.dll` and `gdi32.dll` are responsible for parsing *untrusted* bytes and are, therefore, critical for bug finding. As shown in Fig. 7.1, configuration E, for which options "summarization at maximum runtime" and "constraint subsumption" are turned on, achieves the highest projected coverage. Configuration D, for which only "constraint subsumption" is turned on, achieves a slightly lower coverage. This suggests that summarizing when the maximum runtime is exceeded helps in guiding the search toward new program instructions; in particular, it avoids repeatedly

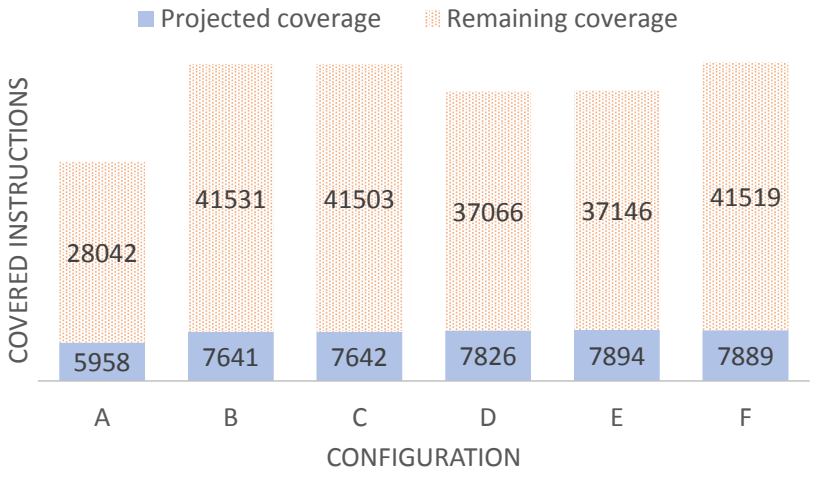

Figure 7.1: The instructions of the ANI parser that are covered by each configuration. The projected instruction coverage is critical for bug finding.

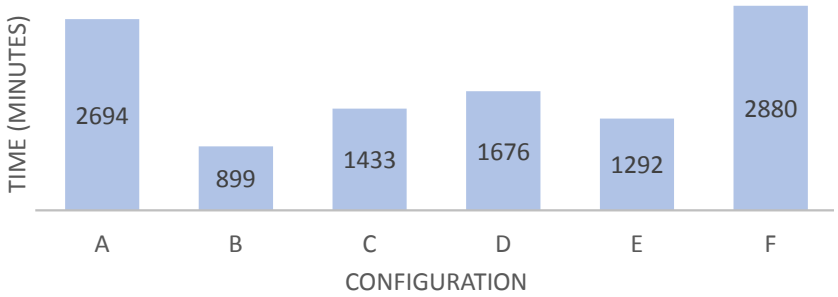

Figure 7.2: The time it takes for each configuration to stop exploring the ANI parser.

exploring the code of the summarized functions. In contrast, configurations A–C, for which "constraint skipping" is turned on, achieve the lowest projected coverage. This indicates that testing input-dependent loops for more than just a single number of iterations is critical in achieving higher code coverage.

Fig. 7.2 shows the time (in minutes) it takes for each configuration to stop exploring the ANI parser. Note that configuration B stops in the smallest amount of time (approximately 15 hours); this is because too many constraints are pruned due to options "constraint subsumption", "constraint skipping", and "flip count limit", which are turned on. Configuration D achieves almost the same projected coverage as F (Fig. 7.1) in much less time, indicating that ad-hoc heuristics such as flip count limits are no longer necessary with IC-Cut. Configuration E, which achieves the highest projected coverage, stops exploring the parser in the second smallest amount of time, that is, in approximately 21.5 hours—roughly 55% faster than the generational-search strategy used in production (configuration F).

In this amount of time, configuration E also detects the largest number of unique first-chance exceptions in the ANI parser. This is shown in Fig. 7.3,

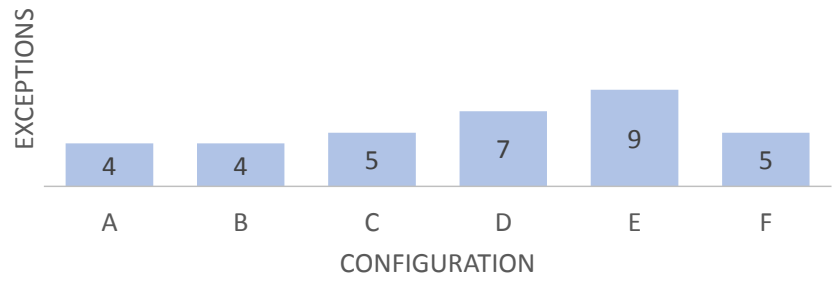

Figure 7.3: The number of unique exceptions that are detected by each configuration.

Maximum runtime		Coverage		Total time	First-chance exceptions	
		projected	remaining	(in minutes)	unique	duplicate
1	minute	5,421	36,250	23	0	0
90	minutes	7,896	37,183	683	8	7
3	hours	7,894	37,146	1292	9	10

Table 7.2: Performance of the winner-configuration E when the maximum runtime per function of the parser is one minute, 90 minutes, and three hours, respectively. Performance is measured in terms of covered instructions, total exploration time of the parser, and detected first-chance exceptions.

which presents how many unique exceptions are detected by each configuration. A *first-chance exception* is an exception (similar to an assertion violation) thrown at runtime (by the operating system) during program execution, but caught by the program using a C/C++ try/catch-mechanism (see [29]). Note that the nine exceptions found by configuration E are a superset of all other exceptions detected by the remaining configurations.

In summary, configuration E detects more unique exceptions than all other configurations combined. Compared to configuration F (generational search), E finds more exceptions (Fig. 7.3) and achieves the same projected instruction coverage (Fig. 7.1) in less than half the time (Fig. 7.2). E is the most effective configuration against path explosion.

Tab. 7.2 shows how the winner-configuration E performs when the maximum runtime per function of the parser is one minute, 90 minutes, and three hours, respectively. Performance is measured in terms of covered instructions, total exploration time of the parser, and detected first-chance exceptions. As shown in the table, IC-Cut performs better than configuration F even for a maximum runtime of 90 minutes per function: there is a noticeable improvement in projected code coverage and bug finding, which is achieved in approximately eleven hours (roughly 76% faster than configuration F). This is a strong indication of how much the summarization strategy of IC-Cut can alleviate path explosion.

Fig. 7.4 shows the number of functions that are explored by the winner-configuration E when the maximum runtime per function of the parser is one minute, 90 minutes, and three hours, respectively. This figure shows only functions for which SAGE generated symbolic constraints. The functions are grouped as follows: exhaustively tested and summarized, summarized despite constraint subsumption or an exceeded runtime, not summarized because of multiple outputs or constrained symbolic inputs. The functions in the first group constitute verified program components (according to Thm. 2), highlighting a *key originality of IC-Cut*, namely, that it can dynamically verify sub-parts of a program during testing. As expected, the larger the maximum runtime, the more functions are discovered, the fewer

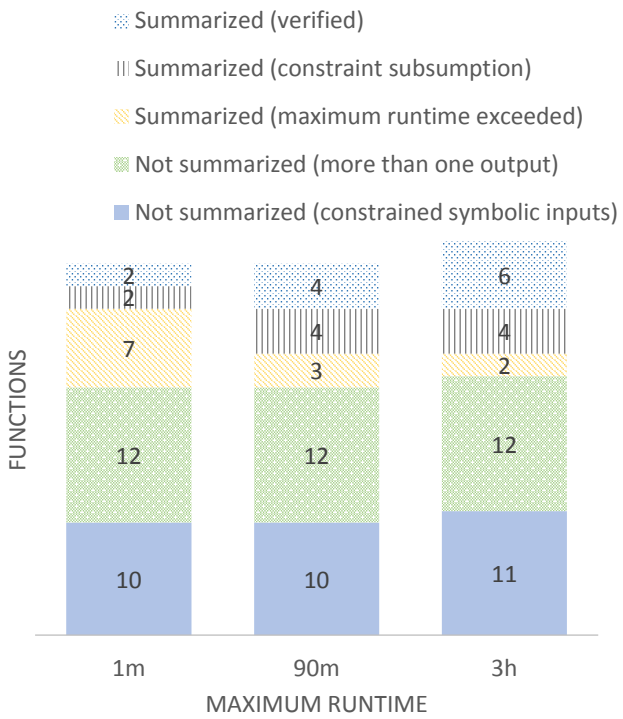

Figure 7.4: How many functions are explored by the winner-configuration E when the maximum runtime per function of the parser is one minute, 90 minutes, and three hours, respectively. Only functions for which SAGE generated symbolic constraints are shown.

functions are summarized at maximum runtime, and the more functions are verified. Interestingly, the functions that are not summarizable because of multiple outputs or constrained symbolic inputs are identified immediately, even for a maximum runtime of one minute per function.

We also used IC-Cut to test other image parsers, namely, GIF and JPEG. Unfortunately, our prototype implementation could not handle the size of these larger parsers. However, preliminary experiments showed that our restrictions for summarization on function interfaces apply to both GIF and JPEG. For instance, when running on GIF with a timeout of three hours per function, 16 out of 140 functions (with symbolic constraints) were summarized. When running on JPEG with the same timeout, 27 out of 204 functions (with symbolic constraints) were summarized.

7.3 Related work

Automatic program decomposition for effective systematic dynamic test generation [25] is not a new idea. Moreover, compositional symbolic execu-

tion [66, 2] has already been shown to alleviate path explosion. However, when, where, and how compositionality is most effective in practice is still an open problem.

Algorithms for automatic program summarization have been proposed before [66, 2, 75]. SMART [66] tests all program functions in isolation, encodes their test results as summaries expressed using input preconditions and output postconditions, and then reuses these summaries when testing higher-level functions. Demand-driven compositional symbolic execution [2] generates partial summaries that describe only a subset of all paths in a function and can be expanded lazily. SMASH [75] computes both may and must information compositionally using both may and must summaries. IC-Cut is inspired by this compositional reasoning and summarization although it does not generate full-fledged function summaries. Instead, IC-Cut records a single precondition on all concrete function inputs without disjunctions or postconditions. In contrast to SMART, IC-Cut generates summaries only for functions with low interface complexity. Similarly to demand-driven compositional symbolic execution, our summaries are partial in that they describe a single calling context. Furthermore, when testing a function in isolation, the closed symbolic constraints that IC-Cut collects before the first call to the function are similar to the lazily-expanded dangling nodes in the demand-driven approach.

Other closely related techniques [89, 4, 15, 107] can be considered as approximations of sub-program summarization. Dynamic state merging and veritesting [89, 4] merge sub-program searches, and RWset [15] prunes searches by dynamically computing variable liveness. Information partitions [107] are used to identify "non-interfering" input chunks such that symbolically solving for each chunk while keeping all other chunks fixed to concrete values finds the same bugs as symbolically solving for the entire input. Similarly to these techniques, our work also approximates sub-program summarization. Moreover, IC-Cut is closely related to reducing test inputs using information partitions. Both techniques exploit independence between different parts of the program input. However, IC-Cut does not require that the input is initially partitioned, and avoids the overhead of dynamically computing data and control dependencies between input chunks.

Overall, our algorithm does not require any static analysis and uses very simple summaries, which are nevertheless sufficient to significantly alleviate path explosion. As a result, it is easy to implement on top of existing dynamic test generation tools. Our purely dynamic technique can also handle complicated ANI code patterns (see Ch. 6), such as stack-modifying, compiler-injected code for structured exception handling, and stack-guard protection, which most static analyses cannot handle. Furthermore, a static over-approximation of the call graph might result in testing more functions than necessary and for more calling contexts. With an over-approximation of function interfaces, we would summarize fewer functions, given the restric-

tions we impose on function inputs and outputs, thus fighting path explosion less effectively.

In addition to our low-complexity function summaries, SAGE implements other specialized forms of summaries, which deal with floating-point computations [69], handle input-dependent loops [74], and can be statically validated against code changes [71].

7.4 Summary and remarks

We have presented a new search strategy inspired by compositional reasoning at simple function interfaces. However, we do not perform compositional verification in this work, except when certain particular restrictions are met, as detailed in Thm. 2 (see also Sect. 7.2).

IC-Cut uses heuristics about interface complexity to discover, dynamically and incrementally, independent program units that process different chunks of the input vector. Our search strategy is complete for bug finding, while limiting path explosion in a more principled and effective manner than in the current implementation of SAGE, with its simple, yet clever, search heuristics. Indeed, compared to the generational-search strategy of SAGE, our experiments show that IC-Cut preserves code coverage and increases bug finding in significantly less exploration time.

IC-Cut generates low-complexity summaries for a single calling context of functions with unconstrained symbolic inputs and at most one output. The previous chapter on proving memory safety of the ANI Windows image parser [29] shows that such simple interfaces exist in real, complex parsers, which is why we chose the above definition. However, our definition could be relaxed to allow for more than one calling context or function output, although our experiments show that this definition is already sufficient for large improvements. We leave this for future work. We also leave for future work determining how suitable such a definition is for application domains other than that of binary image parsers.

Chapter 8

Conclusion and future work

In this dissertation, we have explored different ways of narrowing the gap between static verification and systematic testing. In particular, we have presented how to complement static analyzers or verifiers with systematic testing, and how far we can push systematic testing toward reaching verification.

Summary

To minimize the test effort while maximizing code quality, we explore how to reduce redundancies of systematic testing with prior verification. We achieve this by exploiting the partial verification results both of an unsound static analyzer and a sound verifier, to guide dynamic test generation toward program properties and executions that have not already been checked statically. This approach yields several benefits, including smaller and more targeted test suites, higher overall code coverage, shorter testing time, and higher programmer productivity.

By complementing verification with systematic testing, we identify program properties that are both difficult to check statically and lie beyond the capabilities of existing test generation tools. These properties involve not sufficiently checking object invariants as part of the oracle, although invariants are used to filter valid input data, and not taking into account the potential interference of static state with a program. We have addressed these issues by developing two novel techniques in dynamic test generation that check such properties. We, therefore, enhance systematic testing with better means of checking program correctness and, thus, of supporting verification tools.

To assess how far we can push systematic testing toward verification without using any static analysis whatsoever, we try to address the main limitation of dynamic test generation, namely, path explosion. In the context of proving memory safety of a complex binary image parser, we discover that many functions are easy to verify, using inlining, manual program decomposition, and summarization at very few, yet simple, function interfaces. To

control the remaining path explosion, however, we sacrifice soundness of our approach by bounding the number of iterations of input-dependent loops.

Nonetheless, the insights from this verification exercise have proven significant in defining a new compositional search strategy in dynamic test generation, for automatically and dynamically decomposing large programs to fight path explosion. Indeed, this strategy outperforms the state-of-the-art generational search and its heuristics.

Significance of results

In the first part of this dissertation, the starting point is a static analysis. We identify and encode any verification gaps in this analysis (due to its deliberate unsoundness), generate runtime checks for these gaps, and apply automatic test generation to compensate for the gaps. In contrast, in the second part of the dissertation, we push systematic testing toward verification, but without any static analysis. Specifically, we identify the verification gaps of dynamic test generation in the application domain of binary image parsers. Therefore, this dissertation brings forward the strengths and weaknesses of both static analysis and dynamic test generation in achieving sound verification. For instance, the static analyzer Clousot does not reason soundly about arithmetic overflow, object invariants, or static initialization, which can be compensated for by dynamic test generation, as shown in Chs. 2, 3, and 4. On the other hand, dynamic test generation struggles with path explosion caused by input-dependent loops. This limitation could be addressed by providing loop invariants, either manually as in Ch. 5 through the Dafny verifier, or automatically as in Ch. 2 through the abstract interpreter Clousot.

Based on our techniques and findings, we anticipate a tighter integration of verification and systematic testing in practice. In particular, we expect the development of wizards that identify on which parts of a program a static analyzer can yield better or worse verification results in comparison to a test generation tool, apply the tools accordingly, and precisely combine their correctness guarantees. As a result, tool designers and users will know exactly which program properties remain to be validated and, therefore, the direction toward which they should focus their manual efforts. This will facilitate progress in tool development, such as the rise of novel verification techniques or search strategies for test generation, to address advanced tool limitations and smoothen the integration of existing tools, for instance, in terms of efficiency or precision.

We also expect software engineers and project managers to develop programming guidelines on how much effort should be spent on verification, depending on software quality, criticality, and budget constraints. Such guidelines will facilitate a more effective integration and use of static analyzers in industrial projects.

Our approach for complementing verification with systematic testing generalizes to a large class of assumptions made by static analyzers [30, 32]. It can, therefore, guide users of unsound analyzers in using them fruitfully, for instance, in deciding how to combine them with other analyzers or dynamic test generation tools. Moreover, it can assist designers of such analyzers in finding good trade-offs that are motivated empirically. In other words, we expect designers to perform experiments that measure the impact of their design compromises, in terms of whether these compromises can be compensated for by systematic testing. Such experiments could also derive language designs that mitigate the unsoundness of a static analyzer.

By pushing dynamic symbolic execution toward proving the absence of a certain class of errors in a particular application domain, we make the first step in concretely identifying and assessing the current limitations of systematic testing in reaching verification. We anticipate the development of static techniques that will address these limitations and automate further the verification process. We also expect dynamic test generation to be applied in even more challenging application domains and for proving the absence of more classes of errors. This will most likely reveal an increase in the gap between verification and systematic testing as presented in the second part of this dissertation, and unravel many more research problems.

Future work

As future work, we plan to improve the effectiveness of collaborative verification and systematic testing. In particular, we could precisely determine which compromises made by a static analyzer actually affect the verification of a certain assertion in the program. We currently assume that any assumption the analyzer makes before the assertion in the control flow, is necessary for its verification. However, by computing the smallest set of assumptions that indeed "pollute" the sound verification of the assertion, we also reduce the number of unverified executions, through the assertion, that need to be tested. As a result, test generation would become even more targeted—test suites would become smaller, testing times shorter, and redundancies with prior static analysis would be brought down to a minimum.

Moreover, we could infer compromises made by static analyzers using dynamic test generation. As an example, imagine that an assertion in the program is proven correct by an unsound static analyzer. When checking this assertion during testing, a failing test case is generated. Of course, the failing test demonstrates that the analyzer is unsound. At this point, the path constraint for the test case could contain, or at least indicate, an unsound assumption made by the analyzer. If, under this assumption, the assertion failure is no longer feasible, then all sources of unsoundness in the analyzer have been determined, with respect to the verification of this particular assertion.

We believe that collaborative verification and testing would also be beneficial beyond sequential programs, properties that can be expressed by contract languages, and the typical compromises made by abstract interpreters and deductive verifiers. For instance, we could exploit the datarace-free guarantee [1] for concurrent programs, which states that any execution of a datarace-free program under a relaxed memory model is equivalent to some sequentially consistent execution of that program. More specifically, we could, for instance, verify concurrent C programs using the VCC verifier [36], which assumes a sequentially consistent memory model. Datarace freedom could subsequently be checked with systematic testing, to show program correctness for weak memory models. In the same spirit, our approach could be applied to unsound type systems, as in the TypeScript [111] programming language.

Furthermore, the static and dynamic information computed during collaborative verification and testing could be used to determine a global level of correctness achieved for a given program, or to perform program repairs in code or specifications.

Having verification in mind, we also plan to address the current imperfections of dynamic symbolic execution, including the generation of complex input data and the handling of floating-point numbers. The more such limitations we alleviate, the further is dynamic test generation pushed toward more advanced verification.

References

[1] S. V. Adve and M. D. Hill. Weak ordering—A new definition. In *ISCA*, pages 2–14. ACM, 1990.

[2] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, volume 4963 of *LNCS*, pages 367–381. Springer, 2008.

[3] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *TSE*, 36:474–494, 2010.

[4] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *ICSE*, pages 1083–1094. ACM, 2014.

[5] T. Ball, B. Hackett, S. K. Lahiri, S. Qadeer, and J. Vanegue. Towards scalable modular checking of user-defined properties. In *VSTTE*, volume 6217 of *LNCS*, pages 1–24. Springer, 2010.

[6] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213. ACM, 2001.

[7] T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR–TR–2000–14, Microsoft Research, 2000.

[8] T. Ball and S. K. Rajamani. The SLAM toolkit. In *CAV*, volume 2102 of *LNCS*, pages 260–264. Springer, 2001.

[9] S. Balzer and T. R. Gross. Modular reasoning about invariants over shared state with interposed data members. In *PLPV*, pages 49–56. ACM, 2010.

[10] S. Balzer and T. R. Gross. Verifying multi-object invariants with relationships. In *ECOOP*, volume 6813 of *LNCS*, pages 359–383. Springer, 2011.

[11] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[12] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *CACM*, 54:81–91, 2011.

[13] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

[14] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional model checking: A technique to pass information between verifiers. In *FSE*, pages 57–67. ACM, 2012.

[15] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS*, volume 4963 of *LNCS*, pages 351–366. Springer, 2008.

[16] E. Bounimova, P. Godefroid, and D. A. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *ICSE*, pages 122–131. IEEE Computer Society/ACM, 2013.

[17] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, pages 123–133. ACM, 2002.

[18] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *SPE*, 30:775–802, 2000.

[19] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX, 2008.

[20] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, volume 3639 of *LNCS*, pages 2–23. Springer, 2005.

[21] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS*, pages 322–335. ACM, 2006.

[22] C. Cadar, V. Ganesh, R. Sasnauskas, and K. Sen. Symbolic execution and constraint solving (Dagstuhl seminar 14442). *Dagstuhl Reports*, 4:98–114, 2015.

[23] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *ICSE*, pages 1066–1071. ACM, 2011.

[24] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *CACM*, 56:82–90, 2013.

[25] A. Chakrabarti and P. Godefroid. Software partitioning for effective automated unit testing. In *EMSOFT*, pages 262–271. ACM, 2006.

[26] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. The SANTE tool: Value analysis, program slicing and test generation for C program debugging. In *TAP*, volume 6706 of *LNCS*, pages 78–83. Springer, 2011.

[27] M. Christakis, P. Emmisberger, and P. Müller. Dynamic test generation with static fields and initializers. In *RV*, volume 8734 of *LNCS*, pages 269–284. Springer, 2014.

[28] M. Christakis and P. Godefroid. IC-Cut: A compositional search strategy for dynamic test generation. In *SPIN*, volume 9232 of *LNCS*, pages 1–19. Springer, 2015.

[29] M. Christakis and P. Godefroid. Proving memory safety of the ANI Windows image parser using compositional exhaustive testing. In *VMCAI*, volume 8931 of *LNCS*, pages 370–389. Springer, 2015.

[30] M. Christakis, P. Müller, and V. Wüstholz. Collaborative verification and testing with explicit assumptions. In *FM*, volume 7436 of *LNCS*, pages 132–146. Springer, 2012.

[31] M. Christakis, P. Müller, and V. Wüstholz. Synthesizing parameterized unit tests to detect object invariant violations. In *SEFM*, volume 8702 of *LNCS*, pages 65–80. Springer, 2014.

[32] M. Christakis, P. Müller, and V. Wüstholz. An experimental evaluation of deliberate unsoundness in a static program analyzer. In *VMCAI*, volume 8931 of *LNCS*, pages 333–351. Springer, 2015.

[33] M. Christakis, P. Müller, and V. Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. Technical report, ETH Zurich, 2015.

[34] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[35] E. M. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, pages 368–371. ACM, 2003.

[36] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[37] P. Collingbourne, C. Cadar, and P. H. J. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *EuroSys*, pages 315–328. ACM, 2011.

[38] P. Collingbourne, C. Cadar, and P. H. J. Kelly. Symbolic testing of OpenCL code. In *HVC*, volume 7261 of *LNCS*, pages 203–218. Springer, 2011.

[39] L. Correnson, P. Cuoq, F. Kirchner, V. Prevosto, A. Puccetti, J. Signoles, and B. Yakobowski. *Frama-C User Manual*, 2011. `http://frama-c.com//support.html`.

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

[41] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.

[42] C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *SPE*, 34:1025–1050, 2004.

[43] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE*, pages 422–431. ACM, 2005.

[44] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *TOSEM*, 17:1–37, 2008.

[45] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang. Testing static analyzers with randomly generated programs. In *NFM*, volume 7226 of *LNCS*, pages 120–125. Springer, 2012.

[46] M. Czech, M.-C. Jakobs, and H. Wehrheim. Just test what you cannot verify! In *FASE*, volume 9033 of *LNCS*, pages 100–114. Springer, 2015.

[47] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[48] L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52. IEEE Computer Society, 2009.

[49] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18:453–457, 1975.

[50] M. Dimjašević and Z. Rakamarić. JPF-Doop: Combining concolic and random testing for Java. In *Java Pathfinder Workshop*, 2013. Extended abstract.

[51] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, volume 5142 of *LNCS*, pages 412–437. Springer, 2008.

[52] S. Du Toit. *Working Draft, Standard for Programming Language C++*, 2013. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3691.pdf`.

[53] ECMA. *ECMA-335: Common Language Infrastructure (CLI)*. ECMA, 2012.

[54] B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise pointer reasoning for dynamic test generation. In *ISSTA*, pages 129–140. ACM, 2009.

[55] P. Emmisberger. Bachelor's thesis "Dynamic Test Generation with Static Fields and Initializers", 2013. Department of Computer Science, ETH Zurich, Switzerland.

[56] P. Emmisberger. Research in computer science "Integrating Dynamic Test Generation with Sound Verification", 2015. Department of Computer Science, ETH Zurich, Switzerland.

[57] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69:35–45, 2007.

[58] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *SAC*, pages 2103–2110. ACM, 2010.

[59] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, volume 6528 of *LNCS*, pages 10–30. Springer, 2010.

[60] Federal Aviation Administration, Department of Transportation. Airworthiness directives; The Boeing company airplanes. *Federal Register*, 80:24789–24791, 2015.

[61] J. C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[62] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.

[63] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[64] P. Garg, F. Ivančić, G. Balakrishnan, N. Maeda, and A. Gupta. Feedback-directed unit test generation for C/C++ using concolic execution. In *ICSE*, pages 132–141. IEEE Computer Society/ACM, 2013.

[65] X. Ge, K. Taneja, T. Xie, and N. Tillmann. DyTa: Dynamic symbolic execution guided with static verification results. In *ICSE*, pages 992–994. ACM, 2011.

[66] P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54. ACM, 2007.

[67] P. Godefroid. Micro execution. In *ICSE*, pages 539–549. ACM, 2014.

[68] P. Godefroid, J. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25:30–37, 2008.

[69] P. Godefroid and J. Kinder. Proving memory safety of floating-point computations by combining static and dynamic program analysis. In *ISSTA*, pages 1–12. ACM, 2010.

[70] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.

[71] P. Godefroid, S. K. Lahiri, and C. Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*, volume 6887 of *LNCS*, pages 112–128. Springer, 2011.

[72] P. Godefroid, M. Y. Levin, and D. A. Molnar. Active property checking. In *EMSOFT*, pages 207–216. ACM, 2008.

[73] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, pages 151–166. The Internet Society, 2008.

[74] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *ISSTA*, pages 23–33. ACM, 2011.

[75] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, pages 43–56. ACM, 2010.

[76] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification*. Oracle, java SE 7 edition, 2012.

[77] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[78] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX*, pages 125–138. USENIX, 1992.

[79] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *OSDI*, pages 165–181. USENIX, 2014.

[80] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.

[81] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12:576–580, 1969.

[82] G. J. Holzmann. Mars code. *CACM*, 57:64–73, 2014.

[83] M. Howard. Lessons learned from the animated cursor security bug, 2007. `http://blogs.msdn.com/b/sdl/archive/2007/04/26/lessons-learned-from-the-animated-cursor-security-bug.aspx`.

[84] K. Inkumsah and T. Xie. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In *ASE*, pages 425–428. ACM, 2007.

[85] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26. ACM, 2001.

[86] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. Technical report, ETH Zurich, 2014.

[87] J. C. King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.

[88] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220. ACM, 2009.

[89] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *PLDI*, pages 193–204. ACM, 2012.

[90] A. Lal and S. Qadeer. Powering the static driver verifier using Corral. In *FSE*, pages 202–212. ACM, 2014.

[91] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *CAV*, volume 7358 of *LNCS*, pages 427–443. Springer, 2012.

[92] C. Le Goues, K. R. M. Leino, and M. Moskal. The Boogie verification debugger. In *SEFM*, volume 7041 of *LNCS*, pages 407–414. Springer, 2011.

[93] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. *JML Reference Manual*, 2011. `http://www.jmlspecs.org/`.

[94] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *FM*, volume 5850 of *LNCS*, pages 806–809. Springer, 2009.

[95] K. R. M. Leino. Efficient weakest preconditions. *IPL*, 93:281–288, 2005.

[96] K. R. M. Leino. Specification and verification of object-oriented software. Lecture notes of Marktoberdorf International Summer School, 2008.

[97] K. R. M. Leino. This is Boogie 2, 2008.

[98] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[99] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, volume 3086 of *LNCS*, pages 491–515. Springer, 2004.

[100] K. R. M. Leino and P. Müller. Modular verification of static class invariants. In *FM*, volume 3582 of *LNCS*, pages 26–42. Springer, 2005.

[101] K. R. M. Leino and P. Müller. Using the Spec# language, methodology, and tools to write bug-free programs. In *Advanced Lectures on*

Software Engineering, volume 6029 of *LNCS*, pages 91–139. Springer, 2010.

[102] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *FOSAD*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009.

[103] K. R. M. Leino and V. Wüstholz. The Dafny integrated development environment. In *Formal-IDE*, volume 149 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–15. Open Publishing Association, 2014.

[104] K. R. M. Leino and V. Wüstholz. Fine-grained caching of verification results. In *CAV*, volume 9206 of *LNCS*, pages 380–397. Springer, 2015.

[105] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. In *OOPSLA*, pages 19–32. ACM, 2013.

[106] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *CACM*, 58:44–46, 2015.

[107] R. Majumdar and R.-G. Xu. Reducing test inputs using information partitions. In *CAV*, volume 5643 of *LNCS*, pages 555–569. Springer, 2009.

[108] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[109] B. Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[110] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs that test themselves. *IEEE Computer*, 42(9):46–55, 2009.

[111] Microsoft Corporation. *TypeScript Language Specification, Version 1.4*, 2014.

[112] J. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *POPL*, pages 85–97. ACM, 1998.

[113] MSDN. Application verifier. `https://msdn.microsoft.com/en-us/library/windows/hardware/ff538115%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396`.

[114] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.

[115] P. Müller. Course on "Software Architecture and Engineering", spring semester, 2012. Department of Computer Science, ETH Zurich, Switzerland.

[116] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62:253–286, 2006.

[117] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL*, pages 128–139. ACM, 2002.

[118] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM, 2007.

[119] H. H. Nguyen, V. Kuncak, and W.-N. Chin. Runtime checking for separation logic. In *VMCAI*, volume 4905 of *LNCS*, pages 203–217. Springer, 2008.

[120] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[121] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The YOGI project: Software property checking via static analysis and testing. In *TACAS*, volume 5505 of *LNCS*, pages 178–181. Springer, 2009.

[122] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84. IEEE Computer Society, 2007.

[123] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, 1997.

[124] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *ISSTA*, pages 93–104. ACM, 2009.

[125] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, pages 15–26. ACM, 2008.

[126] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[127] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, volume 4144 of *LNCS*, pages 419–423. Springer, 2006.

[128] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC*, pages 263–272. ACM, 2005.

[129] K. Sen, H. Tanno, X. Zhang, and T. Hoshino. GuideSE: Annotations for guiding concolic testing. In *AST*, pages 23–27. IEEE Computer Society, 2015.

[130] D. X. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *ICISS*, volume 5352 of *LNCS*, pages 1–25. Springer, 2008.

[131] A. Sotirov. Windows animated cursor stack overflow vulnerability, 2007. `http://www.offensive-security.com/os101/ani.htm`.

[132] P. Spettel. Master's thesis "Delfy: Dynamic Test Generation for Dafny", 2013. Department of Computer Science, ETH Zurich, Switzerland.

[133] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *ESEC/SIGSOFT FSE*, pages 193–202. ACM, 2009.

[134] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *OOPSLA*, pages 189–206. ACM, 2011.

[135] N. Tillmann and J. de Halleux. Pex—White box test generation for .NET. In *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.

[136] N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC/SIGSOFT FSE*, pages 119–128. ACM, 2005.

[137] P. Tonella. Evolutionary testing of classes. In *ISSTA*, pages 119–128. ACM, 2004.

[138] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, pages 97–107. ACM, 2004.

[139] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Covana: Precise identification of problems in Pex. In *ICSE*, pages 1004–1006. ACM, 2011.

[140] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, volume 3440 of *LNCS*, pages 365–381. Springer, 2005.

[141] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *PLDI*, pages 99–110. ACM, 2010.

[142] S. Zhang, D. Jalali, J. Wuttke, K. Muslu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *ISSTA*, pages 385–396. ACM, 2014.

[143] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *ISSTA*, pages 353–363. ACM, 2011.

Appendix A

Methods under test

For the experiments of Ch. 2, we used 101 methods (written in C#) from solutions to 13 programming tasks on the Rosetta Code repository and from nine open-source projects. A complete list of the methods used in that evaluation can be found here:

- Rosetta Code repository
  (tasks downloaded at 10:00 on October 23, 2014)

  1. `http://rosettacode.org/wiki/Ackermann_function#C.23`
     - `Program.Ackermann`
  2. `http://rosettacode.org/wiki/Binary_search#C.23`
     (recursive version)
     - `Program.binarySearch`
  3. `http://rosettacode.org/wiki/Bulls_and_cows#C.23`
     - `Program.KnuthShuffle`
     - `Program.game`
  4. `http://rosettacode.org/wiki/Ethiopian_multiplication#C.23`
     - `EthiopianMultiplication_Task.EM_Linq`
     - `EthiopianMultiplication_Task.EM_Loop`
     - `EthiopianMultiplication_Task.EM_Recursion`
  5. `http://rosettacode.org/wiki/Fibonacci_n-step_number_sequences#C.23`
     - `Program.GetFibLikeSequence`
     - `Program.GetLucasNumbers`
     - `Program.GetNacciSeed`
     - `Program.GetNnacciNumbers`
  6. `http://rosettacode.org/wiki/Forest_fire#C.23`
     - `Program.InitializeForestFire`
     - `Program.IsNeighbor`
     - `Program.StepForestFire`
  7. `http://rosettacode.org/wiki/Greatest_common_divisor#C.23`
     - `Program.gcd`
  8. `http://rosettacode.org/wiki/N-queens_problem#C.23`
     - `Program.Allowed`
  9. `http://rosettacode.org/wiki/Number_reversal_game#C.23`
     (version for C# 1.0)

   - `NumberReversalGame.RandomPermutation`
   - `NumberReversalGame.check`

10. `http://rosettacode.org/wiki/Primality_by_trial_division#C.23`
    - `Program.isPrime`
11. `http://rosettacode.org/wiki/Pythagorean_triples#C.23`
    - `Program.Count_New_Triangle`
12. `http://rosettacode.org/wiki/Reduced_row_echelon_form#C.23`
    - `Program.rref`
13. `http://rosettacode.org/wiki/Rock-paper-scissors`
    - `RPSGame.GetWinner`

– Open-source projects

1. Autodiff
   (`http://autodiff.codeplex.com`, rev: `d8799882919c`)
   - `TermBuilder.Constant`
   - `TermBuilder.Exp`
   - `TermBuilder.Log`
   - `TermBuilder.Power`
2. Battleships
   (`http://github.com/guylr/Console-Battleships-CSHARP`, rev: `9986775f36`)
   - `Sea.ChangeDirection`
   - `Sea.PlaceShipInSea`
   - `Sea.SelectBlocks`
   - `Sea.UnSelectBlocks`
3. BBCode
   (`http://bbcode.codeplex.com`, rev: `80132`)
   - `BBCode.EscapeText`
   - `BBCode.ToHtml`
   - `BBCode.UnescapeText`
   - `BBCodeParser.ParseAttributeValue`
   - `BBCodeParser.ParseChar`
   - `BBCodeParser.ParseName`
   - `BBCodeParser.ParseWhitespace`
   - `SequenceNode.AcceptVisitor`
   - `SequenceNode.SetSubNodes`
   - `SyntaxTreeNodeCollection.InsertItem`
   - `SyntaxTreeVisitor.GetModifiedSubNodes`
   - `SyntaxTreeVisitor.Visit(SequenceNode)`
   - `SyntaxTreeVisitor.Visit(SyntaxTreeNode)`
   - `SyntaxTreeVisitor.Visit(TagNode)`
   - `TagNode.ReplaceAttribute`
4. BCrypt
   (`http://bcrypt.codeplex.com`, rev: `d05159e21ce0`)
   - `BCrypt.Char64`
   - `BCrypt.DecodeBase64`
   - `BCrypt.EncodeBase64`
   - `BCrypt.GenerateSalt`
   - `Bcrypt.StreamToWord`
5. Boggle
   (`http://boggle.codeplex.com`, rev: `20226`)

- BasicWordList.ContainsWord
- BasicWordList.ContainsWordStartingWith
- BasicWordList.IsWordWithinBounds
- BasicWordList.Load
- Extensions.AsString
- Extensions.LettersAt
- Extensions.Validate
- Game.ScoreLetterCount
- Tray.FilledWith
- Trie.Add
- Trie.Contains
- Trie.ContainsPrefix
- Trie.LastNodeOf

6. Bowling
(http://github.com/ardwalker/bowling-for-csharp, rev: eb706e4fc8)
   - BowlingGame.Roll
   - Strike.AddBonus
7. DSA
(http://dsa.codeplex.com, rev: 96133)
   - Numbers.Factorial
   - Numbers.Fibonacci
   - Numbers.GetHexSymbol
   - Numbers.GreatestCommonDenominator
   - Numbers.IsPrime
   - Numbers.MaxValue
   - Numbers.Power
   - Numbers.ToBinary
   - Numbers.ToHex
   - Numbers.ToOctal
   - Searching.ProbabilitySearch
   - Searching.SequentialSearch
   - Sets.Permutations
   - Sorting.BubbleSort
   - Sorting.Exchange
   - Sorting.MedianLeft
   - Sorting.MergeSort
   - Sorting.QuickSort
   - Sorting.ShellSort
   - Strings.Any
   - Strings.IsPalindrome
   - Strings.RepeatedWordCount
   - Strings.ReverseWords
   - Strings.Strip
   - Strings.WordCount
8. Scrabble
(http://wpfscrabble.codeplex.com, rev: 20226)
   - Board.TileAt
   - Board.TileExistsAt
   - BoardLocation.IsLocationWithinBounds
   - Extensions.AvailableLocationsIn
   - Extensions.Shuffle

9. Sudoku
   (http://github.com/sakowiczm/Sudoku-Solver-CSharp, rev: 3966be25f9)
   - `Cell.CompareTo`
   - `Extensions.CellsToString`
   - `SudokuSolver.CheckVertically`
   - `SudokuSolver.GetBlock`
   - `SudokuSolver.GetValues`
   - `SudokuSolver.IsPossible`