Test Generation for High Coverage with Abstraction Refinement and Coarsening (ARC)

Doctoral Dissertation submitted to the

Faculty of Informatics of the *University of Lugano*

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

presented by

Mauro Baluda

under the supervision of

Prof. Mauro Pezzè

co-supervised by

Prof. Giovanni Denaro

April 2014

Dissertation Committee

Dissertation accepted on 30 April 2014

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Mauro Baluda
Lugano, 30 April 2014

Abstract

Testing is the main approach used in the software industry to expose failures. Producing thorough test suites is an expensive and error prone task that can greatly benefit from automation. Two challenging problems in test automation are generating test input and evaluating the adequacy of test suites: the first amounts to producing a set of test cases that accurately represent the software behavior, the second requires defining appropriate metrics to evaluate the thoroughness of the testing activities.

Structural testing addresses these problems by measuring the amount of code elements that are executed by a test suite. The code elements that are not covered by any execution are natural candidates for generating further test cases, and the measured coverage rate can be used to estimate the thoroughness of the test suite. Several empirical studies show that test suites achieving high coverage rates exhibit a high failure detection ability. However, producing highly covering test suites automatically is hard as certain code elements are executed only under complex conditions while other might be not reachable at all.

In this thesis we propose Abstraction Refinement and Coarsening (ARC), a goal-oriented technique that combines static and dynamic software analysis to automatically generate test suites with high code coverage. At the core of our approach there is an abstract program model that enables the synergistic application of the different analysis components. In ARC we integrate Dynamic Symbolic Execution (DSE) and abstraction refinement to precisely direct test generation towards the coverage goals and detect infeasible elements. ARC includes a novel coarsening algorithm for improved scalability.

We implemented ARC-B, a prototype tool that analyses C programs and produces test suites that achieve high branch coverage. Our experiments show that the approach effectively exploits the synergy between symbolic testing and reachability analysis outperforming state of the art test generation approaches. We evaluated ARC-B on industry relevant software, and exposed previously unknown failures in a safety-critical software component.

Contents

Figures

Tables

Chapter 1

Introduction

Testing is the premier technique for assessing the quality of software products in industry, it consists of executing a program with a sample of the possible inputs to observe the program behavior and detect failures. Testing activities are responsible for a large part of the overall cost of software production, and therefore a high degree of automation in testing is highly desirable, not only to reduce costs, but also to improve the predictability of the quality of software products [PY07; Ber07]. In this Thesis we present and evaluate Abstraction Refinement and Coarsening (ARC) a novel approach to Automated Test Data Generation (ATDG) that combines different software analysis techniques to produce test suites systematically.

Because of the enormous amount of behaviors that even small software systems can exhibit, exhaustive testing is not practical. For this reason test designers have to sample the input space on a finite set of test cases. Quality managers can evaluate the maturity of test suites using different criteria, one of the most successful being structural coverage. Structural coverage measures the thoroughness of testing based on the code elements exercised by executing the test suites. Given a specific type of code element, coverage is measured as the ratio between the elements of that type that the test exercises and the total number of those elements. Statement and branch coverage are the most popular criteria in the industrial setting so far.

On one hand, structural coverage is certainly useful in finding new test targets to direct the testing effort. In fact, coverage analysis can be used to identify the set of code elements that are not executed yet by any test case, and adding new test cases that cover these elements is an obvious way for improving the quality of a test suite. On the other hand it would be desirable to use structural coverage as a proxy measure for the quality assessment of a test suite. As coverage measures are numerical and unbiased, they would provide quality managers with a solid justification for the decision of continuing or terminating the testing effort.

A large body of research studied the problem of the correlation between the structural coverage and the failure detection power of a test suite. While some evidence for this correlation have been found, the debate is still active [HFGO94; FI98]. In the last few years, several empirical studies questioned the effectiveness of coverage as a good measure for testing effectiveness by analyzing larger and more realistic subjects [WMO12; GGZ+13; FSM+13; IH14].

A deeper look into the empirical studies on the effectiveness of structural testing shows that these apparently contrasting results arise from the analyses of test suites obtaining different levels of coverage. Most studies agree that test suites that achieve high structural coverage (over 90%) are highly effective: For example, Hutchins et al. show that test effectiveness is correlated with the level of coverage and high levels of coverage indicate highly effective test suites [HFGO94]. Frankl et al. obtained similar results analyzing larger test subjects and noticed a low structural testing effectiveness with low coverage levels [FI98].

This distinction between low and high levels of structural coverage is acknowledged by the most recent papers on the topic as well. The following quotation from the recent work by Inozemtseva et al. that contrasts the use of coverage criteria as a test quality assurance measure, highlights the point [IH14]:

> “Some of the previous studies found that a relationship between coverage and effectiveness did not appear until very high coverage levels were reached. Since the coverage of our generated suites rarely reached very high values, it is possible that we missed the existence of such a relationship. That said, it is not clear that such a relationship would be useful in practice. It is very difficult to reach extremely high levels of coverage, so a relationship that does not appear until 90% coverage is reached is functionally equivalent to no relationship at all for most developers.”

Our research is motivated by the observation that structural coverage close to 100% is rarely obtained either with manually or automatically generated test suites, and this factor is the main responsible for the reduced effectiveness of structural testing. The inability of generating test suites that reach high coverage levels stems from the difficulty of both generating test cases that satisfy complex branch conditions and detecting infeasible elements.

To approximate 100% structural coverage, test designers are expected to produce a set of inputs that drive the program execution towards all the coverage targets. Unfortunately, computing exactly such set of inputs is known to be an infeasible problem. In practice, generating a test input that covers a given code element can be very hard, it requires deep understanding of the program internals and the solution of a complex chain of interdependent constraints.

Moreover, undetected infeasible elements that cannot be executed under any program input, can divert the test generation effort and prevent the quality manager from obtaining a precise coverage measure. Infeasible elements can appear in code for several reasons ranging from bad programming practice to deliberate use of infeasible code elements for example in defensive programming [HH85]. Moreover in the case of complex structural testing criteria, several paths statically identified on the program control structure may be not executable even in the absence of dead code elements [Wey90].

1.1 Research Hypothesis and Contributions

In my PhD work I approached automatic structural testing as a twofold problem by employing existing and novel analysis techniques to 1) generate test cases that exercise code elements not yet covered, and 2) identify a large portion of the infeasible elements. The intuition that supports this idea is that testing can identify large portions of reachable elements, while detecting unreachable elements prevents from wasting effort in searching test inputs able to execute such elements. The simple sequential combination of techniques for generating test cases and detecting infeasible elements does not produce the expected results because of the risk of being trapped in the failing attempt to execute complex execution paths that may become a sink for the technique. We hypothesize that the continuous interplay between the two types of technique and the sharing of their partial analysis results can be effective in achieving high coverage and scaling to reasonable sized applications.

This dissertation aims to investigate the following research hypothesis: *Can an analysis technique that is based on scalable program abstractions and that exploits the synergies between symbolic test generation and reachability analysis generate test suites with high code coverage for industrially relevant software systems?*

We detailed the main research hypothesis in the following research questions:

Q1 Can the symbolic approach to test input generation be combined with reachability analysis techniques to produce test suites with high coverage?

Q2 Can such a hybrid framework scale to industrially relevant software by exploiting efficient program abstractions?

Q3 Is the ARC approach able to expose previously unknown failures in industrially relevant software?

This thesis contributes to the state of the research by:

- Proposing a framework for high-coverage structural testing leveraging the interplay of different program analysis techniques.
- Evaluating existing techniques to tackle the problem of testing code while at the same time identifying its infeasible portion.
- Designing a scalable abstraction management technique able to support both test generation and reachability analysis

- Evaluating empirically the effectiveness of the proposed approach.

1.2 Structure of the Dissertation

Automatic Structural Testing was studied extensively in the past decades as a means to automate the tedious and error prone activity of selecting a set of test inputs achieving full structural coverage. Chapter 2 describes the state of the art in ATDG and discusses the main open challenges.

ARC combines dynamic test generation techniques with static reachability analysis. Chapter 3 introduces the fundamental software analysis approaches that constitute the foundations of ARC: Weakest Precondition (WP), Symbolic Execution (SE), Constraint Solving and Abstraction Refinement.

ARC augments a test suite combining concolic execution with abstraction refinement, driven by a model of the coverage domain. A coarsening procedure makes the analysis practical by keeping the size of the coverage model minimal. Chapter 4 describes how the envisioned interplay can produce higher structural coverage rates by both augmenting the number of code elements that get executed and identifying a large fraction of the infeasible code.

The ARC approach was implemented in a prototype tool called ARC-B which is able to generate test cases for C programs. ARC-B targets branch coverage and deals with dynamic memory access and procedures. Chapter 5 discusses the design and implementation decisions involved in the construction of ARC-B.

The ARC approach was evaluated by applying ARC-B on a number of industrial programs. ARC-B obtained higher coverage rates respect to state of the art tools by covering more elements and detecting several infeasible ones. Chapter 6 reports about the experimental campaign we conducted and the evidence we could collect.

Chapter 7 summarizes the contribution of the dissertation and delineates the conclusions and the future directions.

Chapter 2

Related Work

Automated Test Data Generation is one of the main challenges for researchers working on software testing and has been studied extensively in the past decades. Test automation has the potential to reduce dramatically the cost of testing and improve ultimately the reliability of software systems. This chapter describes the state of the art in Automated Test Data Generation focusing particularly on structural testing, and discusses the main open challenges.

Automated Test Data Generation (ATDG) is one of the main challenges for researchers working on software testing towards the possibility of achieving 100% automated testing [Ber07]. By automating the tedious and error prone activity of selecting a set of test inputs to achieve some predefined test goal, ATDG has the potential to reduce dramatically the cost of software quality assurance that constitutes large part of the cost of software development. This in turn would ultimately lead to an increased reliability of software systems.

ATDG was studied extensively in the past decades because of its academic interest and strong industrial importance. Starting from the seminal work of the seventies [Bur67; BEL75; DN84], a rich variety of techniques and approaches have been proposed by the Software Engineering researchers community [McM04; PV09; FWA09b]. This chapter describes the state of the art in ATDG with a particular focus on structural testing criteria, it discusses the main challenges that still need to be addressed.

Section 2.1 describes a common classification schema used in several ATDG literature surveys. The schema offers three orthogonal dimensions each of them composed of three classes. The nine classes correspond to nine different properties of ATDG approaches:

1. *Design approach:* Structural, Functional, Non Functional

2. *Automation approach:* Random, Goal Oriented, Path Oriented
3. *Implementation technique:* Dynamic, Static, Hybrid

Section 2.2 presents an alternative classification schema that focuses on the background techniques employed by the different ATDG approaches. It appears in fact clear from the literature review that most of the proposed approaches build on top of few fundamental techniques:

1. Symbolic Execution
2. Model based Testing
3. Random testing and Adaptive Random Testing
4. Search-based Testing

Section 2.3 overviews the State of the Research in ATDG organizing the different approaches based on their background techniques which are defined in Section 2.2. Each of the discussed approaches are further classified according to the schema defined in Section 2.1. Despite not being exhaustive, the literature review aims to highlight the open challenges in ATDG thus motivating the approach proposed in this thesis.

Section 2.4 discusses the problem of identifying infeasible code elements, an active research area that is contiguous to test case generation. The complementarity between the two problems is a well known concept from the theory of software analysis but the existing approaches for the solution of either of them, seldom benefit from such insight. The section discusses the preeminent approaches to Automated Reachability Analysis and their relation with ATDG.

2.1 Classification of Test Data Generation Approaches

In this section I present a classification schema for ATDG techniques that is commonly found in literature and allows us to easily compare and link different approaches based on their properties. The schema identifies three orthogonal dimensions and three classes for each of them. As ATDG uses software analysis techniques to automate the task of generating test data, two of the three dimensions proposed can be tracked back to well known classifications of test design approaches and software analysis techniques.

2.1.1 Structural, Functional and Non Functional test Design

A first dimension for ATDG classification is derived directly from the test design approaches that should be automated. Such classification can be applied to any test generation technique, manual or automatic [PY07].

Structural testing defines test adequacy criteria based on the code of the software under test. Coverage rates are defined as the fraction of code elements of a certain type (for example statements) that are executed by a test suite. Structural coverage gives an objective measure of the test thoroughness and highlights the code that should be targeted by further test effort. The direct correlation between high structural coverage (i.e. between 90% and 100%) and the fault detection power of a test suite is acknowledged by practitioners and was measured in several empirical studies [FW93; HFGO94].

The largest part of the approaches presented in this chapter targets Structural Testing as it is certainly the most popular candidate to automation and the one targeted by the approach presented in this thesis.

Functional testing verifies that a software system behaves as it is intended. A specification is a formal or informal description of such behavior and can be used to derive test cases. Automation can be achieved to the extent to which a specification can be analyzed automatically.

Non-functional properties are qualities that describe how a system works, in opposition to functional qualities that concern what the system does. Non-functional testing is used to verify qualities of software like reliability, security, performance and usability.

2.1.2 Dynamic and Static Implementation Techniques

A second classification dimension for ATDG is inherited from program analysis terminology where *Dynamic* and *Static* approaches are distinguished based on the fact that they do require or do not require actual program execution, respectively.

A *Dynamic* ATDG approach observes the execution of the program under test and uses the gathered information to inform the reasoning that lies behind the Test Input Generation. On the contrary a *Static* ATDG approach analyses the program code in its textual form and deduces the expected program behavior from it.

As it is well know from program analysis theory, dynamic approaches are precise but incomplete. If dynamic analysis detects a failure then the failure is real but on the other hand an observed correct behavior gives little information about other possible behaviors. On the contrary static approaches are generally complete but imprecise. If static analysis detects a failure, it might be a false alarm due to the analysis inherent imprecision, on the other hand static analysis can certify the absence of certain classes of failure.

Hybrid ATDG approaches augment static analysis with dynamic information thus mitigating the imprecision of the results. Many researchers advocate hybrid approaches as a means to mitigate the complementary limitations of static and dynamic ones [YBS06]

2.1.3 Random, Path Oriented and Goal Oriented Automation

The third dimension is specific to ATDG and was proposed and formalized by Edvardsson in 1999 and widely adopted subsequently [Kor90; Edv99]. It divides ATDG methods in three classes based on the automation approach they employ:

Random approaches are conceptually simple and very flexible. As a matter of fact, every data type is ultimately a bit stream and it is therefore always possible to generate a random input for a program by generating a random bit stream. A large number of random test cases can be generated quickly and executed in parallel but as the program input space is normally huge, random testing is rarely exhaustive. In particular random testing does not perform well in testing for corner cases or detecting *semantically small* faults that are faults with a small probability of being executed. Random testing is often used as a benchmark to compare other ATDG techniques against and performs sometimes surprisingly well especially in terms of cost-effectiveness [DN84].

Path oriented approaches analyze the code of the program under test and select a specific execution path in the program Control Flow Graph (CFG) that should be covered by the test execution. If the testing goal is to achieve a certain level of structural coverage, path oriented approaches would need 1) to recognize all the path leading to a certain coverage target 2) select one path out of the many 3) generate the appropriate test input. The path selection strategy highly influence the overall effectiveness of the approach in particular as paths found by static analysis might be infeasible, that is, they may not correspond to any program computation.

The *Goal oriented* approach refers strictly to a specific dynamic ATDG technique proposed by Korel in 1992 and is a precursor of modern global search-based approaches [Kor92]. Korel proposes a technique that avoids the need for selecting a specific program path that should be executed but instead proposes to generate inputs based on an objective function that can be computed statically from a program CFG. By extension every ATDG technique that aims at a global goal instead of trying to cover specific program paths is classified in this category.

2.2 Background Techniques for Automated Test Data Generation (ATDG)

A different classification schema for ATDG techniques is based on the background employed technique. Such classification is particularly useful in describing the state of the research ATDG as the different classes also represent four adjacent but distinct sub communities of Software Testing researchers [ABC$^+$13]. This section briefly describes each of the four technique.

2.2.1 Symbolic Execution

Symbolic Execution generates symbolic constraints that characterize program path as functions of the program inputs. This is achieved by simulating a program execution along a certain path and assigning appropriate symbolic values to variable instead than concrete ones. Automated theorem provers can be used to generate program inputs satisfying such constraints, and therefore executing the selected path.

Symbolic Execution (SE) has been studied since the Seventies [BEL75; Kin76; Cla76] but both fundamental and technological aspects limited its practical application: In particular, exhaustive SE requires the enumeration of the paths of a program, this is rarely possible for large software systems. Moreover, proving mathematical theorems is a creative activity that is particularly hard to automate. Nonetheless SE is still a productive area of research thanks to the advances in both directions obtained in the last decades [PV09].

As SE is one of the building blocks of the ATDG technique presented in this Thesis, a detailed description of it is given in Section 3.2 of Chapter 3.

2.2.2 Model Based Testing

Model Based Testing uses formalized program requirements (models) to generate test cases. Software engineers have proposed a large variety of models for software that can represent both the intended behavior of a system and the structural aspects of its implementation. Model Based ATDG techniques use models of software systems to define test adequacy criteria.

Model Based approaches proved to be particularly suitable for Functional Testing. In this case the model (usually a finite state model) encodes aspects of the specification of the system under test. Normally a certain degree of manual modeling effort is needed but recent advances in the automatic generation of behavioral models from program execution traces, open the way to fully automatic model based ATDG approaches.

Model Checking is a formal verification approach that can provide counterexamples witnessing the violation of desired properties. Counterexamples can be interpreted as test cases, and model checking can be used to generate inputs satisfying certain testing criteria defining appropriate temporal properties.

2.2.3 Random Testing and Adaptive Random Testing

Random Testing produces test suites sampling the input space of a program randomly. Randomized approaches are common in testing hardware devices and applied to software since the seventies. The prospect of a ATDG that is completely unbiased and permits certain statistical guarantees has to confront with the enormous size of program input state space, combined with the non linear nature of software. It is in fact very unlikely to be able to detect software failures that happen rarely using naive random testing. For this reason in recent research Random Testing is normally complemented with dynamic and static analysis techniques that can inform the test case generation and orient the random search in "promising" areas of the input space.

2.2.4 Search-based Testing

Search-based Testing employs advanced Optimization Algorithms to automate the search for test data that maximizes a certain goal therefore sharing many of the fundamental limitations of random testing. Evolutionary testing is one of the most studied search-based approaches, inspired by biological evolution, it generates new test cases mutating and combining existing ones, a higher chance to influence new test inputs is given to the fittest tests based on a pre-determined fitness function. Possible fitness functions include measures of structural, functional, as well as non functional properties of software.

2.3 State of the Research in Automated Test Data Generation (ATDG)

ATDG is vast and specialized research field and typically surveys focus on specific sub fields [McM04; ATF09; PV09; FWA09b]. This section overviews the ATDG literature according to the four different classes identified in Section 2.2 that correspond to four different background techniques: SE, Model Based, Random and Search-based Testing.

We identified a second classification schema referring to the discussion in Section 2.1. The classification schema identifies three dimensions and three classes for each dimension producing a total of 27 combinations. While we believe that a coarse classification like the one described in Section 2.2 is better suited for a literature survey, we find that the second schema can be useful in comparing competing approaches and highlighting trends and open research areas.

Table 2.1 shows the ATDG techniques discussed in the rest of the chapter where approaches with similar properties are aggregated independently from their technical background and according to the schema from Section 2.1. In the table, columns map different Design Approaches while rows correspond to the Automation Approaches. Each of the nine table cell is further divided vertically according to the used implementation technique.

We marked in gray the empty cells in the table that in our opinion cannot be populated because of intrinsic incompatibility of the features, it is for example the case for the combination of random and static technique. The remaining empty cells could be in principle populated but we did not find any approach in our literature

		Design Approach								
		Structural			Functional			Non Functional		
		Dyn	Stat	Hyb	Dyn	Stat	Hyb	Dyn	Stat	Hyb
Automation Aproach	Random	[GDGM01] [CLM05] [WJMJ08] [CLOM08] [FZ11]	▨	[GKS05] [MS07]	[WRF$^+$11] [NZK12]	▨		[GFX12]	▨	[HHZ12]
	Path Or.	▨	[CSE96] [Bal03] [Bal04] [BCLKR04] [BCH$^+$04] [VPK04] [TdH08] [CDE08] [PDEP08] [McM10] [SP10] [LGR11] [BUZC11] [JMNS12] [LSWL13]	[PMB$^+$08]	[SMA05] [CS05] [SPPV05] [YBS06] [MS07] [GLM08] [BS08] [XTdHS09] [EGL09] [BNR$^+$10] [SJP$^+$13]	[MS07]			[BJSS09] [BJS09] [ZED11]	[XGM08]
	Goal Or.	[Kor90] [Kor92] [FK96] [MMS01] [Ton04] [MHBT06] [GFZ12] [KPD$^+$13] [VMGF13]		[LHM08] [IX08] [BBDP11] [AH11] [HJL11] [BHH$^+$11]	[TCM98] [BPS03] [WB04] [BW08] [VLW$^+$13]			[TCM98] [KZHH05] [TSWW06] [DPCE$^+$07] [BC07]		

Table 2.1. Classification of the State of the Research according to the dimensions defined in section 2.1: Design approach, Automation approach, Implementation technique

survey. Our analysis suggests that such areas of the spectrum of technical solutions to ATDG could be worthwhile investigating in the future.

In the rest of the section, we present our survey on ATDG approaches based on their main background analysis technique. While the overview of the literature presented is broad, it cannot be considered complete, moreover some of the work analyzed escape the proposed classification by overlapping different categories. The discussion is focused on the approaches more closely related to the one presented in the thesis namely hybrid symbolic approaches.

2.3.1 Symbolic Execution

SE is a well known program analysis technique that executes programs using symbolic values for variables instead of concrete ones. SE was first proposed in the seventies as a way to generate test cases but its practical application to industrial programs was always limited by scalability issues. Section 3.2 introduces SE since it is one of the fundamental background technique used in the ATDG approach presented in this paper.

The challenge of making SE scalable, fostered a large amount of scientific work. In the past decades, researchers proposed a number of algorithmic improvements to SE being able to analyze increasingly complex software. At the same time the increased availability of cheap computing power and progresses in the field of decision procedures make people believe that automated SE will be practical soon. This renovated interest in SE is witnessed by the emergence os many SE tools for the most common programming languages like C and Java [VPK04; APV07; CDE08; BDP13].

Three main issues that limit SE effectiveness for test case generation are *Path explosion*, *Expressiveness* and *Infeasibility*:

Path explosion:
Path explosion occurs as SE analyzes each and every execution path in a program. As the number of paths grows exponentially with the number of conditions in the program it is generally infeasible to enumerate all of them. Moreover executing *Loops* symbolically may never terminate as the number of Symbolic states to be analyzed might be unbounded.

Expressiveness:
While executing a program, SE builds predicates in certain logical theory to maintain a symbolic state of the execution. The *Expressiveness* of the logical theory employed is limited by the availability of an efficient automatic prover for it. For this reason only elementary programming language constructs are completely supported by most SE tools.

Infeasibility:
ATDG approaches based on symbolic execution need to select feasible paths leading

to code elements to be covered to be able to generate the corresponding test inputs. Static analysis of the control flow graph is perhaps the oldest, and best known, approach to identify such paths but it can only provide over-approximated feasibility information. SE may need to analyze many infeasible paths, be able discard them, without making progresses towards the selected coverage criterion.

In this section we discuss the main approaches explored by researchers to improve scalability of ATDG based on SE:

Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE) attempts to mitigate scalability issues by using dynamic information from concrete executions to guide SE. DART and CUTE select a random input for the program and execute it, SE is first directed along that same path that is proved feasible and then explores systematically alternative program paths in depth-first order [GKS05; SMA05]. The systematic search is obtained by fuzzing the path constraints collected on the concrete execution path. The new path predicates characterize test cases that both reach not-yet-covered branches and discover new paths. DSE can also replace symbolic values with concrete ones every time the theorem prover cannot deal with them thus partially overcoming SE expressiveness limitations at the expense of some precision [TdH08]. Such approach is known as Concolic Testing and is described in more details in Section 3.2.4.

Concolic testing is not the only DSE incarnation: Păsăreanu et al. tackle the state explosion problem combining exhaustive SE on code at the unit level with the concrete execution of system level tests [PMB+08]. Majumdar and Sen propose hybrid concolic testing that alternates random and concolic testing [MS07]; Random testing guarantees an extensive exploration of the program state space while concolic testing performs an exhaustive local search. Chipounov et al. introduce relaxed execution consistency models that allow to alternate symbolic and concrete execution reducing the number of explored paths and operating directly on binaries. The framework allows to choose a performance/accuracy trade-off suitable for a given analysis [CKC11].

DSE allows to restrict the symbolic reasoning to the part of the code that is relevant for the properties that need to be verified. Xu et al. proposed Splat, opting for a mostly dynamic approach that searches for memory violations using symbolic variables only to represent buffer length thus obtaining a very lightweight analysis [XGM08].

One important advantages of DSE approaches is the ability to deal with dynamically generated code where static analysis is generally inapplicable. Emmi et al. tested programs that interact with databases [EMS07]. Thanks to a solver for string constraints they were able to generate tests for dynamically generated SQL queries. Artzi et al. proposed Apollo, a tool for testing dynamically-generated Web pages combining concrete and symbolic execution of PHP code [AKD+08].

Search Space Prioritization

Both static and dynamic SE techniques suffer from the path explosion problem as they may diverge trying to explore an infinite number of paths that traverse only a small subset of program elements. For this reason, several studies investigated SE effectiveness when coupled with different path exploration strategies. Such strategies use heuristics to prioritize the analysis towards paths that are more promising.

Burnim and Sen defined heuristics that guide SE to achieve higher structural coverage. The authors propose to measure the distance of an execution from uncovered branches based on the program control flow graph and negate the nearest condition to continue the exploration. A simpler approach obtains similar performance by selecting the condition to negate using a random sampling of visited part of the symbolic execution tree [BS08]. Xie et al. introduced an heuristic for path selection that tries to minimize the distance between the selected paths and the code elements that are not yet covered taking into account also the state of the execution [XTdHS09]. Su et al. propose a predictive path search strategy that drives the path exploration towards code parts more dense in coverage goals [SJP$^+$13].

Papadakis and Malevris proposed a path selection strategy to reduce the effect of infeasible paths while targeting branch coverage [PM10]. Their strategy builds on the empirical observation that the path feasibility likelihood is correlated to the size of the path constraints. The approach prioritizes short execution paths as a proxy for paths with simpler constraints.

Li et al. identify less explored paths measuring the *length-n subpath program spectra* to approximate full path information. Length-n spectra generalize branch coverage to by profiling the execution of loop-free program paths with length n. The corresponding SE strategy proceeds by selecting paths in the direction that, up to that moment, exercised the smallest number of length-n subpaths [LSWL13].

Other approaches prioritize the state space exploration with the goal of maximizing program failures. Csallner et al. proposed Check 'n' Crash, a tool that generates test cases by trying to verify possible faults detected by means of static analysis. DSD-Crasher adds a preliminary step that performs dynamic invariant detection to build preconditions that can be used to reduce the number of executions analyzed [CS05; CSX08]. Cadar et al. proposed EXE a tool that drives symbolic execution towards possible failures and detects if the current path constraints allow any value that causes a failure, and generate the revealing test case automatically [CGP$^+$06].

Related approaches use SE to check symbolic runtime error conditions along concretely executed paths. Predictive Testing and ZESTI implement this idea and can in therefore detect failures that might occur during the execution of any test following the same program path as the observed one [JSS07; PDM12].

Marinescu et al. target the analysis of software patches that are often source of failures in evolving software [PDM13]. They developed KATCH, a SE tool that implements path selection heuristics that quickly drive the symbolic execution to the new code by using a combination of static and dynamic technique. KATCH prioritizes paths that better satisfy the software patches preconditions using as heuristic branch distance, weakest precondition and dataflow analysis.

Parallelization

As multi-core processors and distributed systems are getting more and more common, researchers are investigating parallelizing strategies that could accelerate the exploration of the symbolic space. The path explosion problem is caused by SE exploring each execution path independently, on the other side this independence makes SE algorithms parallelizable.

Staats and Păsăreanu apply a *Static Partitioning* technique to symbolic execution trees inspired by model checking parallelization approaches. Static Partitioning computes statically the preconditions that characterize distinct subtrees of the SE tree, such subtrees can be analyzed independently and in parallel [SP10]. Authors report time speedups up to 90x using 128 workers.

Starting from the observation that static balancing cannot predict precisely the actual workload of the worker nodes, Bucur et al. proposed Dynamic Distributed Exploration (DDE) [BUZC11]. DDE is implemented in the tool *Cloud9* and consists of a set of worker nodes performing SE independently. The worker nodes are coordinated by a load balancer whose goal is to dynamically partition the execution tree when the processing resources are underutilized.

Specialized Static Analyses

A different line of research explores the possibility of reducing the cost of SE by introducing ad hoc static analysis steps.

Anand et al. propose type-dependence analysis, a technique that identifies statically the program variables that can contain symbolic values. Type-dependence reduces the instrumentation needed to perform SE and supports users in identifying program parts that cannot be treated(e.g., third-party libraries) [AOH07].

Babic et al. execute binary programs symbolically and drive executions towards potential vulnerabilities. After executing a first set of test cases and observing dynamically the jump instructions in the code, a static analysis phase detects potentially vulnerable paths by augmenting the observed control flow with statically computed jumps [BMMS11]. The vulnerabilities are verified by producing the appropriate test case.

Program abstraction

Program abstraction is a family of program analysis techniques that can also be effective in mitigating some of the SE issues. The abstraction approaches that showed to be useful in symbolic execution are very diverse and targeted to different problems.

Lazy initialization was proposed by Visser et al. for symbolically executing methods that take complex data structure as inputs [VPK04]. When a method for a complex data structure is executed for the first time, SE creates the corresponding symbolic structure but leaves it uninitialized. Subsequently, when an uninitialized field is accessed, the executor creates the corresponding symbolic values considering all the possible alias conditions systematically. The executor considers nondeterministically the three cases where: 1) the field is initialized to *null*, 2) the field references a fresh object with uninitialized fields, 3) the field references a previously created object. The approach was shown to drastically improve the performance of SE in several application scenarios.

State matching is a technique that allows to check if a state that is visited during SE is subsumed by another symbolic state. This information is useful as it can prevent SE from reanalyzing the same states over and over again. As the number of symbolic states may be infinite, state matching quickly becomes impractical. Anand et al. proposed Abstract Subsumption, a technique that enables state matching by exploiting specific program abstractions in particular for lists and arrays [APV06].

Jaffar et al. propose to use interpolating theorem provers to mitigate the path-explosion problem [JMN13]. Automatic interpolation can provide abstract preconditions for the execution of program paths that may reduce the cost of state matching in SE. The authors show empirically that using interpolation they can achieve higher path coverage using less resources, they note however that the approach might be not cost-effective for branch coverage.

Compositionality

Another recent approach to improve scalability of SE is to reason about program modules separately, deriving function *summaries* that can be then reused compositionally. Such approach is mutuated from interprocedural analysis design techniques.

Godefroid et al. proposes to use summaries in the context of concolic testing [God07]. Summaries consist of predicates that act as pre and post conditions of a function and predicate on input and output variables, respectively. If the precondition of a function encountered in symbolic execution is verified by the entry symbolic state, the function effect on the state can be summarized by the postcondition. As functions might be called over and over again in a program, reusable summaries can greatly reduce the number of analyzed paths. SMASH builds summaries on demand and distinguishes between *may* and *must* summaries that encode respectively over-approximated and under-approximated abstractions [GNRT10]

Solver Optimization

Invoking a decision procedure to solve constraints generated with SE is usually the most time consuming step of symbolic test case generation. Despite the continuous improvements in the performance of Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers, automated theorem proving is still a largely unsolved problem. In the last years though, it appeared clear that for some specific application scenarios automated theorem proving can be of practical use.

Researchers in ATDG normally use theorem provers as black box tools and take advantage of the improvements in the field by simply replacing old provers for new ones in their tools. In the context of SE it was noticed that it is usually impossible to select a prover that works best in all situations. Being largely based on heuristics and implementing different solving approaches, different tools perform better then others for specific types of constraints. Results from the annual competition for SMT solvers SMT-COMP show that none of the participants is competitive in all categories [BDM+13].

For this reason the most advanced symbolic test generation tools run several theorem provers in parallel and make use of the results that are computed faster. Saswat et al. introduced multi-solver support in the SE extension of Java Path Finder and Palikareva and Cadar implemented multi-solver support in KLEE [APV07; PC13]. In their papers they observe that different solvers react differently to changes in the timeout settings and the constraint caching policies implemented in the SE engine. The idea of running solvers competitively in parallel is justified by the fact that their relative performance proved to be unpredictable and by the growing availability of multicore systems that can be used for this purpose.

Erete and Orso propose that symbolic execution and constraint solving techniques should be better integrated, they suggest that the performance of constraint solvers during symbolic execution can be optimized using domain and contextual information [EO11]. In their paper they evaluate a novel optimization strategy called *DomainReduce* that eliminates potentially irrelevant constraints based on dynamic execution information on program dependencies. Their extensive empirical evaluation considers DomainReduce and other two classical strategies, namely *Incremental Solving* and *Constraint Subsumption*, providing evidence of the effectiveness of constraint optimization strategies.

2.3.2 Model Based Testing

Model Based Testing (MBT) uses models of software to derive test cases. The area is vast and diverse and is rooted in the tradition of formal methods for verification. While traditional formal methods aim to prove that a program adheres to its specification expressed as a formal model, MBT gives up on completeness and uses models to select test cases, especially focusing on functional testing [ABC+13].

The most popular software models for MBT are graph based, in particular Finite State Machines (FSM) and Labelled Transition Systems (LTS). In such cases test selection works by defining an adequacy criteria on the paths represented in the model, it does not directly generate test inputs but abstract tests that need to be instantiated. Nguyen et al. enrich MBT models with input domain specifications to enable test input generation [NMT12]. Combinatorial algorithms are used to reduce the number of generated test cases while maintaining effectiveness high.

Other MBT approaches make use of Software Model Checking, a technique that attempts to prove properties of a program by exhaustively checking its state space [JM09]. Structural coverage criteria can be expressed as temporal logic formulas and model checking algorithms can produce counterexamples that compose the desired test suite [CSE96; BCH+04; FWA09b]. Scaling model checking to large software is an open challenge and a major limitation for its direct application to testing [FWA09a]. Model checking in fact must deal with state space explosion, and may diverge while trying to cover infeasible elements.

2.3.3 Random Testing and Adaptive Random Testing

Random testing generates test data by sampling randomly the input space of a program. Random testing is conceptually simple and very flexible: As a matter of fact, every data type is ultimately a bit stream and it is therefore always possible to generate a random input for a program by generating a random bit stream. For its wide applicability, random testing is one of the most popular testing technique, alone and in combination with other approaches, proving itself effective also in areas like functional programming that are rarely considered by other approaches [CH00]. Moreover random testing is one of the few technique that allow a theoretical analysis of fault detection by controlling the statistical properties of the selected sampling technique.

Random testing can easily produce large and diverse test suites, but it is rarely able to obtain high structural coverage. This depends on the fact that the *semantic size* of a code element, that is the probability of executing it, can be very small [OP97]. Similarly, when the relative size of the input sub-domain for which a failure is observed is small (small semantic size) random testing shows low failure detection power.

Empirical studies showed however that failure-causing inputs tend to form contiguous regions in the input space [CCMY96]. This suggested the idea that if the already executed test cases do not show any fault, new test cases should be searched far from the previous ones. To exploit this fact, Chen et al. propose an enhancement to random testing called Adaptive Random Testing (ART) that maximizes the distance between test inputs based on a certain metric of the state space [CLM05].

Following this line of research Wu et al. proposed Antirandom Testing that select test inputs whose total distance from all previous tests is maximum [WJMJ08]. Different distance metrics can be designed and each of them defines a different ART algorithm. Of particular interest is ARTOO, an approach proposed by Ciupa et al. that defines a metric for the input space of Object Oriented software taking into account object values (be it primitive or reference), their dynamic types and attribute values [CLOM08].

Despite recent criticism to its practical effectiveness, it is generally recognized that ART outperforms random testing when considering their F-measure: that is the expected number of test cases required to detect the first failure [AB11].

2.3.4 Search-based Software Testing

Search Based Software Testing (SBST) uses search-based optimization algorithms to automatically generate test data. Considering the input domain of the program under test as the search space, SBST can find input vectors that satisfy a given test goal. Test goals must be formulated as a numerical "fitness function" that is used to guide the search towards promising areas of the search space. Being largely a stochastic approach, SBST shares in principle the same limitations with random testing in exercising semantically small faults. SBST is superior to random testing when the search objective can be approximated and the optimal solution is not necessarily needed.

SBST is a wide research area that is only partially related with the work presented in this thesis, it is rather a complementary approach to ATDG [McM04]. For this reason in this section we will overview the functioning of the most popular approaches to SBST and present only a selection of the recent work on the topic.

The first requirement to cast the problem of test generation as a search one is to define a suitable encoding of test cases. Test input vectors are normally used for this purpose, the search landscape in the space of program inputs however is discontinuous and highly non linear while most search techniques work better with smooth landscapes. A different encodings is usually used for the case of unit testing of Object Oriented software, in this case chromosomes are represented by linear sequences of method calls [Ton04].

As we already anticipated, defining an appropriate fitness function for the desired test goal is a fundamental aspect of every SBST technique. Due to the generality of the approach, researchers demonstrated different types of test goals using SBST. Moreover SBST allows to balance different test goals thanks to the possibility to define multi-objective fitness functions: the size of the test suite, the readability of the generated tests and their generality are examples of secondary objectives [FZ11].

A substantial part of the SBST research effort has targeted Structural Coverage. Wegener et al. targeted branch coverage considering every branch as a separate goal. For every program branch and test input, the fitness function describes how close the input was to executing that specific branch and is composed of two parts: the *approach level* and the *branch distance* [WBS01]. The *approach level* considers the path of the executed test and measures how many of the control dependencies of the target branch were not executed. The *branch distance* is the minimum distance to the target branch that the test execution path reached in its execution, measured on the program CFG. The two terms are added together after normalization to produce the final fitness function:

$$approach(branch, input) + distance(branch, input) \tag{2.1}$$

Search-based techniques can be directed to alternative test goals by defining appropriate fitness functions: researchers have proposed ah hoc solutions for mutation testing [FZ12], data-flow coverage [VMGF13], functional testing of self-parking systems [WB04], non functional testing [ATF09], Object Oriented testing [Ton04], regression testing [WSKR06], integration testing [CAaVP11], system testing of Graphical User Interfaces (GUIs) [GFZ12], and web testing [AH11].

The second main element that characterizes a SBST approach is the choice of the search algorithm. SBST has been approached using different *metaheuristic* optimization algorithms: general stochastic strategies that are not problem specific and are often nature-inspired. The variety of approaches includes local algorithms that improve a single individual solution and global algorithms that evolve populations of solutions. Hill Climbing, Simulated Annealing, Genetic Algorithms, Ant Colony Optimization and Particle Swarm Optimization are just some of the many metaheuristic optimization algorithms, in the remaining of this section we will discuss the approaches that had larger application in ATDG.

Local Search:
Hill Climbing is a simple local search algorithm that tries to improve one initial solution by visiting the search space neighborhood. Korel proposed to improve a test input by considering the input variables one by one and raising or lowering their numerical value until the maximum fitness is reached [Kor90]. Such approach suffers from the presence of local maxima, solutions that despite being better then the neighboring ones are not globally optimal.

Global Search:
Simulated annealing is a global search algorithm that takes the name from a strengthening metallurgic treatment. Compared to hill climbing, simulated annealing avoids being trapped in local optima defining a more relaxed neighborhood relation. At every iteration the algorithm has a certain probability to switch from the old solution to a new one that depends on the fitness of the two solutions and on a global “temperature” value that decreases over time. The probability of discarding the current solution is higher when the new solution has a better fitness, still the ability of traversing areas of suboptimal solutions make simulated annealing able to escape local optima. Tracey et al. used simulated annealing to generate test cases for both functional and non-functional properties, as well as for testing exception conditions showing that metaheuristic testing can be a cost effective approach for ATDG problems [TCM98].

Genetic Algorithms:
Genetic algorithms are loosely inspired by evolutionary biology and obtained a lot of

attention in the last decade. “Evolutionary” testing is a popular approach to test case generation in which genetic algorithms are used to evolve a population of test cases into a test suite maximizing given goals. Genetic algorithms are designed so that every individual in a population explores a part of the search space independently of the others, the best performing individuals have higher chances to influence the evolution of the population as a whole.

The initial population is typically created via RANDOM SEEDING and after that five steps are executed iteratively: FITNESS EVALUATION, SELECTION, CROSSOVER, MUTATION and REINSERTION. Optionally other stages can be introduced inspired by higher level mechanisms that can be observed in nature such as *speciation*, *competition* and *migration*.

RANDOM SEEDING:
The genetic constitution of the initial population can strongly influence evolutionary testing efficiency. Typical sources of genetic material include random values, constants extracted from source code (e.g., numbers and string) and hand crafted test inputs. MacMinn et al. use Web searches to improve coverage of Java classes that require string inputs [MSS12]. Alshahwan and Harman use dynamic analysis to seed string inputs for dynamic web applications [AH11]. Fraser and Arcuri evaluated different seeding strategies empirically for testing Java classes using SBST approaches and concluded that the more domain specific information can be included in the initial population, the better the results [FA12].

FITNESS EVALUATION:
At every iteration of the the evolutionary algorithms, each individual in the population is evaluated and is assigned a fitness value. The fitness value measures on a continuous scale the distance between the individual and the search solution. Fitness evaluation efficiency is crucial to scale evolutionary algorithms to large populations.

SELECTION:
Normally selection favors “fitter” individuals that exhibit a higher fitness value but weaker individuals are also preserved to avoid lost in feature diversity. A common selection strategy assigns a selection probability that is proportional to the individual fitness value. Kifetew et al. propose to favor the diversity in the population by adding to the population individuals that express orthogonal features [KPD+13].

CROSSOVER:
The next stage of a genetic algorithm is crossover: individuals are removed paired randomly and their genes (e.g. input variables values) are recombined into “offsprings”. Offsprings have equal probability to receive their genes from either parent.

MUTATION:
The mutation phase operates then a mutation to the offspring genes with a low probability. This step is needed to introduce features in the population that were not present in previous generations. A common mutation operator is the flip of a bit in the input sequence.

REINSERTION:
A reinsertion strategy decides which part of the old population should be replaced by the new one. An elitist strategy might replace 10% of the old population with the offspring that rank best.

SPECIATION:
Speciation and migration influence the crossover stage by limiting the freedom in which couples can be formed. In this case only individuals with a certain degree of similarity can be part of a crossover. MacMinn et al. propose to use different species to investigate the different paths towards a test coverage target [MHBT06].

COMPETITION:
Competitive approaches contemplate the presence of subpopulations that compete to supply a larger fraction of individuals to the global population. In this case the probability of influencing the population evolution is not dependent exclusively on the fitness of the single individuals but also on their group of belonging.

Evolutionary algorithms perform these steps iteratively until all the given time budget is consumed.

2.3.5 Hybrid Symbolic and Search-based Testing

Symbolic and Search-base testing are certainly the two most popular fully-automatic test generation approaches. The two techniques are largely complementary and their performance differs greatly for different types of software and testing goals. The most relevant complementaries between the two approaches derives from the fact that SBST is black box while DSE is white box.

Black box approaches do not need program code for their analysis and are therefore robust in presence of instructions for which symbolic reasoning is challenging. The definition and the evaluation of a fitness functions for instance do not need special care when dealing with non linear expressions or with native code. White box approaches on the contrary can generate more precise and focused tests as they have access to the program code and their precision is only limited by their background constraint solvers. It is therefore expected that the two families of techniques could be combined together fruitfully, producing better results then the ones obtained when used separately.

The hybrid approaches proposed in literature vary in the degree of integration they provide. A first line of research uses DSE in alternation with SBST to support specific programming languages aspects. Inkumsah et al. use SBST to produce method sequences for testing Object Oriented (OO) software and DSE for generating their parameter values [IX08]. Lakhotia et al. use DSE in a search-based approach to reason about pointers and data structures [LHM08].

Other approaches use DSE as a component in some of the steps that constitute a meta heuristic search algorithms. Baars et al. propose to use DSE in the fitness evaluation step of a SBST approach targeting branch coverage [BHH$^+$11]. Malburg and Fraser propose to use DSE as an extra mutation operator for SBST [MF11].

Galeotti et al. address the problem of coordinating the alternation between SBST and DSE by proposing an heuristic approach [GFA13]. Their evolutionary testing algorithms switches temporarily to SE every time the mutation operator applied to primitive values in the test sequence influences the fitness value. Their experiments show an increase in code coverage of up to 63% (11% on average).

Harman et al. proposed a technique to generate test suites that target strong mutation testing. Their approach generates test inputs reaching mutated instructions using DSE and then trying to reveal the mutation using SBST [HJL11]. The intuition behind this approach is that while reaching a mutation is in essence a coverage problem, revealing a failure can be addressed as an optimization problem. Their schema uses SBST to generate a test case that maximizes the distance between the execution path in the mutated code and in the original one.

A radically different line of research uses meta heuristic search algorithms to find in the space of the execution paths the ones that are most promising for DSE exploration. Xie et al. use a state-dependent fitness function to guide the symbolic path exploration towards paths that are less distant from the test targets [XTdHS09].

2.4 State of the Research in Reachability Analysis

The approaches presented in this chapter so far focus on ATDG without considering the dual problem of detecting infeasible code elements. With Abstraction Refinement and Coarsening (ARC) we advocate the need to tackle the two problems together as the inability to recognize that a relevant number of uncovered code elements are in reality infeasible undermines the effectiveness of structural coverage testing criteria. Moreover from the efficiency point of view it can be observed that, in presence of infeasible code, the testing effort might be wasted in the impossible task of finding a test suite achieving 100% coverage, without obtaining any increase in confidence about the quality of the software under test.

Saturation based criteria, that focus on the speed at which coverage increases during test execution, can be used to predict the effort needed to discover new coverage elements and terminate the testing effort when cost-effectiveness becomes too low [SDE09]. While saturation rates do not relate to the quality of the produced test suite, detecting infeasible elements of the coverage domain would influence directly the computed coverage rates. The rest of the chapter discusses some software analysis techniques developed in the context of property checking and dead code elimination that can be applied to exclude elements from structural testing requirements.

Many interesting properties of programs can be expressed as reachability problems, that is the problem of deciding if a code element can be executed under any program input or not. Safety properties, facts that should never be true in any execution of the program, belong to this category. Even if it is known that the problem is not decidable in general, a large body of research on reachability analysis and detection of infeasible paths has been carried out in the past with applications to code optimization, safety assurance, and testing.

Proving the infeasibility of a code element is the dual problem of finding a test case (counterexample) that executes it. The positive solution of one of the two problems implies the impossibility of solving the other. For this reason the ability to identify and remove infeasible code would greatly facilitate test case generation as it would prevent the hopeless search for covering test cases. In this section we review the main approaches for reachability analysis an in particular its applications to testing.

Optimizing compilers detect dead code using data flow analysis, a static technique that over approximates the set of possible values for variables at all program locations [Hec77]. State of the art dead code elimination algorithms exploit Static Single Assignment code representation for higher efficiency [CFR$^+$91]. Data flow analysis however is not path sensitive and can sometimes propagate data flow facts across infeasible paths producing imprecise results.

Symbolic Execution can overcome data flow limitations as it is able to determine paths feasibility. The increased precision is however paid in terms of efficiency: as already discussed, symbolic analysis suffers from the path explosion problem and often does not terminate. Combinations of the two approaches can be used to obtain simultaneous symbolic analysis of sets of program paths for example by generalizing results obtained on one path to a possibly infinite number of paths equivalent from the data flow point of view [DBG10].

Software model checking can be naturally applied to feasibility analysis. Traditional explicit state and symbolic model checking can scale to millions of states, but for general purpose software, with an unbounded state space, this is still not enough [BMDH90].

Counterexample Guided Abstraction Refinement (CEGAR) approaches to model checking contrast the state explosion by building a finite abstraction of the program behaviors on which they check the desired properties. Due to the abstraction however, a violation detected in the model might be spurious in the context of the real program. The identification of spurious counterexamples (property violations) triggers a refinement step that adds information to the model thus excluding some infeasible behaviors. This process is performed in a loop that might diverge but is guaranteed to progress [HJMS02]. Incremental software model checking approaches based on satisfiability checking tools proved to be very effective and parallelizable [Bra11; CG12].

Gulvani et al. propose to use symbolic test case generation to provide counterexamples for the abstraction refinement steps [GHK$^+$06; BNR$^+$10]. When SE detects an infeasible path, abstraction refinement is performed in the form of predicate abstraction: a predicate on the state variable of the program is introduced in the model to exclude the infeasible path from the abstraction. Beyer et al. propose to refine not one infeasibility at a time, but thanks to the identification of paths invariants, i.e. conditions valid on a set of infeasible paths, to remove a large number (possibly infinite) of spurious paths [BHMR07].

While the approaches described so far aim for maximum precision, imprecise approaches try to detect a large portion of infeasible code using shallow syntactic analysis. Such approaches might lead to false positives meaning that some feasible paths can be wrongly classified as infeasible. Ngo and Tan use pattern matching to detects infeasible paths [NT07; NT08]. They identified a small number of code patterns that are responsible for the infeasibility of large part of the paths including identical/complement-decision and mutually-exclusive-decision. The approach consists of detecting such code patterns using textual pattern matching. The authors report experiments on a small set of medium sized Java programs where they obtain a precision of 96% and a recall of 100%.

Chapter 3

Foundations

Abstraction Refinement and Coarsening (ARC) is an approach for Automated Test Data Generation that combines several static and dynamic software analysis techniques. The foundations of ARC lay in a small number of well known software analysis approaches. This chapter provides a short introduction to Weakest Precondition calculus, Symbolic Execution, Abstraction Refinement and Constraint Solving, with the purpose of supporting the understanding of the remaining of the Thesis.

This thesis proposes an approach to Automated Test Data Generation (ATDG) called Abstraction Refinement and Coarsening (ARC) that combines state of research and novel static and dynamic software analysis techniques. Before discussing the components of our approach and describing their interaction (Chapter 4), we introduce the fundamental software analysis approaches they are built upon.

Section 3.1 describes deductive verification and in particular Dijkstra's Weakest Precondition (WP) Calculus. Section 3.2 describes Symbolic Execution (SE) and reports about its applications to the ATDG problem. Section 3.3 describes Abstraction Refinement, an approach that makes Model Checking techniques scale to Software Verification problems. Section 3.4 describes constraint solving, a technique that automatically generates values that satisfy given logical constraints. Constraint solving enables the automation of a large class of software analysis techniques.

3.1 Weakest Precondition Calculus

This section introduces *Deductive Verification*, the axiomatic approach to software verification by E. W. Dijkstra. We complement the original WP formalization with recent improvements and extensions that allow its application to modern programming languages and increase its performances.

3.1.1 Deductive Verification

Deductive Verification attempts to verify formal properties of programs, it requires the program to be specified in predicative logic form, and aims to prove deductively that the program implementation implies the specification. Deductive verification achieves the goal by employing a logical calculus that defines the semantics of a programming language and provides rules to reason about the behavior of programs. Application of Deductive Verification are static proof of the absence of runtime errors, program comprehension, software optimization, code generation and debugging.

The origin of Deductive Verification can be traced back to the late sixties and in particular to the work of Robert W. Floyd and Tony Hoare [Flo67; Hoa69]. Floyd and Hoare started a line of research that aims to use mathematical logic as a foundation for computer programming with the goal of providing a sound technique to design algorithms and prove the correctness of their implementation.

As Hoare himself noted in a recent retrospective, the impact of Deductive Verification was hard to foresee [Hoa09]. In particular while Deductive Verification was intended as an alternative to Software Testing, Testing become the preeminent technique for assessing software reliability. Hoare observes that the recent progresses of automated verification tools and the growing interest in software security might be the key to wider industrial application of Deductive Verification. As the world economy relies more and more on Software Systems, the higher economic impact of software bugs may justify the application of more expensive verification approaches in some application domains.

3.1.2 Reasoning about Programs

To reason about the correctness of an imperative program it is useful to consider it together with its specification as a triple:

$$\{P\}\ c\ \{Q\} \tag{3.1}$$

Expression 3.1 is called a “Hoare Triple” and includes a proposition P called precondition, a proposition Q called postcondition and a command c. Deductive Verification provides rules to build Hoare triples that are correct, that means that assuming the precondition P the command c produces a program state where the postcondition Q is valid.

Deductive Verification promotes the view of a program as a predicate transformer. Predicates can be used to characterize program states and imperative program are functions that map program states to program states. Deductive Verification provides semantics for imperative programming languages by specifying how each command of the language affects program states.

Two levels of correctness are generally considered for a Hoare triple: total and partial correctness. Total correctness prescribes that if command c is executed in a state in which proposition P holds, then it terminates in a state in which Q holds. Partial correctness states that if command c is executed in a state in which proposition P holds, then it terminates in a state in which proposition Q holds unless it aborts or does not terminate. In a nutshell total correctness implies partial correctness but requires the termination of command c.

Deductive Verification approaches might support forward or backward reasoning. A forward approach defines how the postcondition Q can be computed as a function of a command c and the precondition P. Conversely a backward approach allows to compute a precondition P that guarantees that Q is a postcondition of the execution of command c. Hoare Calculus is an example of forward Deductive Verification, in the following of this section we introduce Dijkstra's Weakest Precondition which is the main backward verification approach.

3.1.3 Dijkstra's Weakest Precondition

Edsger W. Dijkstra introduced the Weakest Precondition Calculus as a generalization of Hoare Calculus to reasons about total correctness of imperative programs [Dij75; Dij97]. The Weakest Precondition function $wp(c, Q)$ takes a command c and a postcondition Q and gives a precondition for c that guarantees the postcondition Q, thus satisfying the Hoare triple 3.2.

$$\{wp(c, Q)\}\ c\ \{Q\} \tag{3.2}$$

Moreover the precondition $wp(c, Q)$ is the weakest of all the prepositions P satisfying the triple $\{P\}\ c\ \{Q\}$. This property implies that to prove a preposition $\{P\}\ c\ \{Q\}$ we can instead compute the $wp(c, Q)$ and prove the implication $P \implies wp(c, Q)$.

WPs are defined recursively based on the abstract syntax of statements. Dijkstra developed the language of *Guarded Commands*, a small language for writing abstract nondeterministic programs. The wp function for guarded commands is defined recursively and follows the syntactic structure of the language constructs: For each language construct c, Dijkstra gives an equation that defines the $wp(c, Q)$ function in terms of the postcondition parameter Q.

Dijkstra's WP constitutes an axiomatic semantics for the language of guarded commands, and gives a calculus for Hoare logic that allows to build valid deductions about program properties. Unlike other approaches, the goal of WP is to help programmers develop correct software by construction instead of merely define unambiguously a programming language. As the title of the book "A Discipline of Programming" suggests, Dijkstra aims to create an effective programming methodology by reducing the problem of verifying programs to the problem of proving theorems.

3.1.4 The WP Functions

In the following we introduce the *wp* function for an imperative language like C following Dijkstra definitions. In this work we focus on deterministic programs and therefore we do not consider the nondeterministic features of the Guarded Commands language.

Empty Statement:

$$wp(\texttt{skip}, Q) = Q \tag{3.3}$$

The empty statement has no effect on the program state. Q itself is therefore the precondition that guarantees the execution to terminate in a state where the postcondition Q is valid.

Assignment:

$$wp(x\texttt{:=}\ t, Q) = Q[t/x] \tag{3.4}$$

The assignment statement modifies the state of the program replacing the variable value on the left hand side of the statement with the value on the right hand side. Similarly the WP function operates on the postcondition Q by replacing all the occurrences of variable t with the value x.
For a concrete example consider the postcondition $Q = a > 10$ and the command that increments variable a by 1. To assure that after the increment variable a is bigger then 10 is enough to guarantee that before the execution, variable a is bigger then 9:

$$wp(a\texttt{:=}\ a+1, a > 10) = a+1 > 10 = a > 9$$

Sequence of Statements:

$$wp(s1; s2, Q) = wp(s1, wp(s2, Q)) \tag{3.5}$$

The second instruction in a sequence operates on the state as it was modified by the first instruction. Similarly the WP function of a sequence of commands is the composition of the WP functions of the commands in the sequence.
Consider the postcondition $Q : a > 10$ and the sequence of two commands $a = a \times a; a = a + 1$. To assure that after the increment variable a is bigger then 10 is enough to guarantee that the before the execution variable a is bigger then 9 which can be guaranteed by the precondition that $a \times a > 9$.

$$wp(a\texttt{:=}\ a \times a; a\texttt{:=}\ a+1, a > 10) =$$

$$= wp(a\texttt{:=}\ a \times a, wp(a\texttt{:=}\ a+1, a > 10)) =$$

$$= wp(a\texttt{:=}\ a \times a, a > 9) =$$

$$= a^2 > 9$$

Conditional:

$$wp(\texttt{if } b \texttt{ then } c1 \texttt{ else } c2, Q) = (b \land wp(c1, Q)) \lor (\neg b \land wp(c2, Q)) \tag{3.6}$$

In executing a conditional statement, command $c1$ in the *then* part of the instruction is executed only if the condition b evaluates to true, if b is false command $c2$ is executed instead. Similarly the WP function for a conditional is reduced to the WP of $c1$ if b is valid and to the WP of $c2$ otherwise. Consider the postcondition $Q : a > 10$ and the code to compute the absolute value of variable a: `if` $a < 0$ `then` a`:=`$-a$. The WP function distinguishes the two cases in which the condition $a < 0$ is valid or not valid and reduces to the disjunction of the WPs of the two commands a`:=`$-a$ and `skip` respectively.

$$wp(\texttt{if } a < 0 \texttt{ then } a\texttt{:=}-a \texttt{ else skip}, a > 10) =$$
$$= (a < 0 \land wp(a\texttt{:=}-a, a > 10) \lor (a \geq 0 \land wp(\texttt{skip}, a > 10)) =$$
$$= (a < 0 \land -a > 10) \lor (a \geq 0 \land a > 10)$$

Loop:

$$wp(\texttt{while } b \texttt{ do } c, Q) = \exists k : k \geq 0 \land P(k) \tag{3.7}$$

Where $P(k)$ is defined inductively:

$$\begin{cases} P(0) = \neg b \land Q \\ P(k+1) = b \land wp(c, P(k)) \end{cases} \tag{3.8}$$

As Dijkstra's calculus aims for total correctness, termination must be enforced. Formula 3.7 states that there must exists a finite number of iterations of the loop that ensure termination in a state satisfying postcondition Q. In Formula 3.8 it is possible to recognize the intuitive semantics of a loop: at every iteration the body c of the loop is executed on the state resulting from the execution of the previous iteration. After the last iteration the loop condition b is not valid anymore and the postcondition Q must be satisfied.
A classic example that illustrates the WP computation for a loop is to prove that the sum of the first n natural odd numbers is equal to n^2. Consider the precondition $n \geq 0$ postcondition $Q : s = n^2$ and the code

$$i\texttt{:=}0; s\texttt{:=}0; \tag{3.9}$$
$$\texttt{while } i \neq n \texttt{ do} \tag{3.10}$$
$$\quad i\texttt{:=}i + 1; \tag{3.11}$$
$$\quad s\texttt{:=}s + 2i - 1; \tag{3.12}$$

In this particular case, a closed form for $P(k)$ can be found using formula 3.8 for a small number of iteration and then generalizing:

$$P(0) : (i = n) \wedge (s = n^2)$$

$$\begin{gathered} P(1) : (i \neq n) \wedge (i + 1 = n) \wedge (s + 2(i + 1) - 1 = n^2) \\ (i \neq n) \wedge (i = n - 1) \wedge (s = n^2 - 2n + 1) \\ (i = n - 1) \wedge s = (n - 1)^2 \\ P(2) : (i \neq n) \wedge (i + 1 = n - 1) \wedge (s + 2(i + 1) - 1 = (n - 1)^2) \\ (i = n - 2) \wedge (s = (n - 2)^2) \\ \dots \end{gathered}$$

$$P(k) : (i = n - k) \wedge (s = (n - k)^2) \tag{3.13}$$

The precondition of the loop is therefore obtained from formula 3.7:

$$\exists k : k \geq 0 \wedge i = n - k \wedge s = (n - k)^2 \tag{3.14}$$

Continuing backward in the code, we can now compute the WP of the instructions in Line 3.9 given the postcondition 3.14 and finally prove that:

$$n \geq 0 \implies \exists k : k \geq 0 \wedge 0 = n - k \wedge 0 = (n - k)^2 \tag{3.15}$$

Choosing $k = n$ the implication is verified and therefore the property for the whole program.

The existential WP function for loops in 3.7, although intuitive, is seldom useful for the effective derivation of proofs of programs. Closed forms for the $P(k)$ are not always possible and therefore alternative proof strategies are needed. The *Fundamental Invariant Theorem for Loops* [Dij97] allows the proof to abstract from the exact number of times that a loop body is executed:

$$I \wedge wp(\texttt{while } b \texttt{ do } c, True) \implies wp(\texttt{while } b \texttt{ do } c, I \wedge \neg b) \tag{3.16}$$

where I is called the Invariant of the loop. I holds at the beginning of the loop and is preserved by the loop body c and is therefore valid at every iteration of the loop: $I \implies wp(c, I)$

In theorem 3.16 the final postcondition $I \wedge \neg b$ is the one that holds after the last iteration of the loop, when the condition b is not valid anymore. The antecedent of the implication states that I holds initially and that the loop terminates, in fact termination is required by the conjunct $wp(\texttt{while } b \texttt{ do } c, True)$. Theorem 3.16 states that an invariant I that is strong enough to verify the property $I \wedge \neg b \implies Q$

implies the $wp($`while` b `do` $c, Q)$ and can therefore be used in proofs of properties as a stronger replacement of the actual WP.

Loop invariants cannot in general be identified algorithmically and constitute the main challenge in automating the WP calculus. Software developers are required to manually annotate loops with appropriate invariants to support machine checked program verification. Researchers proposed heuristic solutions to the problem of inferring loop invariants automatically from the target postconditions [SGF09; FM10].

3.1.5 Weakest Precondition for Industrial Programming Languages

The guarded commands language proposed by Edsger W. Dijkstra is an abstract programming language that focuses on simplicity. Industrial programming languages support higher level program constructs like procedures and classes, and need to be interpreted efficiently by digital computers. Building a WP Calculus for such languages poses major challenges, both technical and conceptual. In this section we will discuss extensions to the Dijkstra's WP that are commonly employed to analyze industrial programs in tools like Frama-C [CKK+12], ESC/Java2 [FLL+02] and KeY [BHS07].

Procedures:
The Guarded Commands language does not support procedures. Dijkstra justified his design decision pointing out the complexity arising from supporting recursion: "the semantics of a repetitive construct can be defined in terms of a recurrence relation between predicates, whereas the semantic definition of a general recursion requires a recurrence relation between predicate transformers. This shows quite clearly why I regard general recursion as an order of magnitude more complicated than just repetition" [Dij97].

Refinement Calculus addresses the problem by introducing *Specification Statements*. The simple specification statement $[P, Q]$ comprises two predicates over the program variables and means "assuming an initial state satisfying P, establish a final state satisfying Q" [Mor88]. Specification statements treat procedures as black boxes in the WP and enable compositional reasoning.

Refinement Calculus advocates a top-down approach to program verification. In fact every specification statement can be used as an assumption in proofs even before being verified with respect to its own implementation. This strategy is known as *Stepwise Verification*, an iterative approach where abstract formal specifications are gradually refined to concrete executable code.

Machine Arithmetics:
Dijkstra's WP calculus is independent from the theory in which the predicates involved in the verification are defined. It can be shown that WP is a *relatively complete* calculus, meaning that its completeness depends on the completeness of the underlying theory [Coo78]. Under such assumptions every valid specification

can be derived applying the *wp* function.

A practical implementation of WP calculus however needs to find a compromise between accurate modeling of machine arithmetics and completeness. Moreover automating the verification process requires the availability of suitable and efficient decision procedures. Several models of arithmetics have been proposed for machine integral and floating point arithmetics the most common being the *Natural*, the *Real* and the *Bitvector* models.

The *Natural* model uses the infinite mathematical integers, which are well supported by state of the art automated provers. Such solution does not take into account bit-level machine operations as well as integer overflow. While overflow can be modeled using the modulo operator on natural integers, such strategy reduces significantly the efficiency of automated solvers.

The *Real* model can be used to approximate floating-point operations. The distance between the semantics of mathematical real numbers and IEEE floating point numbers is however considerable. Higher precision can be obtained by considering the different rounding modes prescribed by the standard but at the cost of reduced performance [DM10].

The *Bitvector* model mirrors directly the binary representation of data as used in physical machines. This solution represents the most precise modeling of machine arithmetics, especially because automated solvers for bitvector arithmetics support all the machine operators, including non linear ones. Bitvector models are preferred for the verification of code where non linear arithmetics plays an important role as it is the case for example in cryptographic libraries.

Heap Memory:
Another common feature of industrial programming languages is the ability to allocate memory dynamically on the *memory heap* and access it directly using pointers. The memory heap might be accessed through dynamically computed addresses and pointer variables might alias each other, that is refer to the same memory location using different names. Reasoning about the heap requires the definition of an abstract *memory model* and a suitable logic for it.

The simplest model commonly used in axiomatic reasoning approaches considers the whole memory heap as a single array of contiguous memory locations. The heap is therefore modeled as a function *mem* that takes one argument, the memory address a, and returns the value v stored at that memory location.

$$v = mem(a) \tag{3.17}$$

The logical theory of *equality* with *uninterpreted functions* matches such model and is supported by many automated provers (see Section 3.4.2).

However, such model, which is characterized by the congruence axiom (see Equation 3.20), can only represent immutable memory states. Imperative programming languages instead have the ability to change the program state. Two (interpreted) functions, *select* and *store*, can be used to model the familiar imperative memory access semantics:

select : takes an array *mem* and a memory address a and returns a value v.

store : takes an array *mem*, a memory address a and a value v and returns a new array mem' that is equal to *mem* at every location except for location a where the stored value is v.

John McCarty formalized the axiomatic semantics for arrays introducing the following axiom that defines the functions *select* and *store* [McC93]:

$$\mathit{select}(\mathit{store}(\mathit{mem}, a, v), a') \Leftrightarrow \text{if } a = a' \text{ then } v \text{ else } \mathit{select}(\mathit{mem}, a') \tag{3.18}$$

Informally one could say that the function *store* "masks" the original value in the array with the new value, modeling in this way the modified array.

Equality between arrays can be expressed via the *extensionality axiom* which is borrowed from the Zermelo-Fraenkel set theory:

$$\forall a(\mathit{select}(\mathit{mem}, a) = \mathit{select}(\mathit{mem}', a)) \Rightarrow \mathit{mem} = \mathit{mem}' \tag{3.19}$$

Informally it states that if two arrays store the same value at every location, then they are equal.

Combinatory Array Logic (CAL) is an extension to the classic array theory developed by Moura et al. that tries to balance expressivity and efficiency [dMB09]. CAL adds *constant-value arrays* and *maps* to McCarty's theory and supports a restricted form of extensionality. An efficient decision procedure for CAL is implemented in the Z3 constraint solver by reduction to the theory of uninterpreted functions.

The Global array memory model makes no assumptions on the structure of data stored in the heap. This can lead to reduced performance in program verification as any symbolic memory address can be alias of any other. Static may-alias analysis can be used to overapproximate the set of alias conditions that hold at every program point. Beckman et al. observe that may-alias analysis are rarely precise enough to produce satisfactory results and use instead dynamic alias information to compute path-oriented weakest precondition [BNR$^+$10].

The *Disjoint Regions* memory model represents memory as a collection of arrays corresponding to the memory region allocated in the program. Regions have a distinct base address, a fixed size and do not overlap. Pointers are defined as offsets with respect to the region they point to. This approach can rule out infeasible aliases between different memory regions resulting in improved performance [CMTS09].

A different approach that enables reasoning about dynamically allocated arrays and pointer arithmetic is Separation Logic (SL). Instead of using traditional first order logic to reason about theories suitable for the description of memory, SL extends Hoare Logic to include specific commands and logic operators for allocating, accessing and modifying memory [Rey02]. The goal is to have a logic that fosters local reasoning by writing predicates that refer only to the portion of memory accessed by the program and not to the entire system state.

The language used for SL predicates correspond better to the way programmers reason about data structures. It has been observed in fact that despite the unquestionable expressivity of first order logic, graph based representation of data structures is more intuitive, concise and effective [BRC$^+$12]. SL predicates about heap memory are easily expressible in graphical form and they proved to be effective especially in proving inductive properties on linked data structures.

Object Orientation:
Object Oriented (OO) languages allow programmers to reason at a higher abstraction level then procedural ones, promoting modular software development. While some of the fundamental features of OO languages like *encapsulation* and *inheritance* can be naturally casted to a rely/guarantee framework compatible with WP based approaches, dynamic features like *dynamic dispatching* and *reflection* pose a major challenge to deductive verification. For this reason OO verification is usually limited to *abstract data types* with inheritance, an abstract construction that only models the static features of OO languages for example by restricting inheritance in presence of pointers and references.

The specification for an abstract data type can be given in terms of the functional specification of the operations (or methods) that the type supports. The specification of a method is sometimes called its *contract*, as it both describes the method behavior and the conditions that should hold in the caller when the method is executed. Different types of conditions have been proposed in the context of the *Design by Contract (DbC)* design methodology including method *preconditions*, *postconditions*, *writes clauses* and *class invariants* [Mey92].

A condition on the caller of a method is called a method *precondition*, it is a predicate that should hold on the method entry. A condition on the method itself is called a *postcondition*, a predicate that should hold just before the execution control exits the method. Method contracts can also include *writes clauses* that specify which parts of the program state might be modified by the method execution. Moreover it is often useful to specify type *invariants*, predicates valid in any of the visible states of the abstract data type, thus whose validity is preserved by the execution of all methods.

The *DbC* methodology uses a specification language that includes all of the elements listed in the previous section and poses three semantic restrictions on how contracts can interact with inheritance:

- Preconditions cannot be strengthened in a subtype.

- Postconditions cannot be weakened in a subtype.
- Invariants of the supertype must be preserved in a subtype.

These restrictions are equivalent to the design principle known as the Liskov's substitution principle that defines the same behavioral requirements for subtyping in a concise and abstract form:

> *"Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T." [LW94]*

Although DbC is a design methodology, the availability of contracts can be exploited for quality assurance tasks including testing and deductive verification. Execution environments for languages that support DbC can use contracts to test the validity of the specified conditions at runtime, and in this way detect deviation of the system behavior from the specified one. The most well known deductive verification systems for OO languages like Frama-C, ESC/Java2 and KeY, use specification languages inspired from DbC and a form of Refinement Calculus to reason about inheritance [CKK$^+$12; FLL$^+$02; BHS07].

Reasoning about inheritance remains an open challenge for automated approaches to OO deductive verification. In particular most techniques pose strong limitations to the possibility of a subtype to restrict the behavior of its base type. This allows the reuse of a large part of properties that have been proved for supertypes in the verification of their subtypes. Allowing a derived type to restrict the behaviour of its supertype requires in fact that every inherited method is verified from scratch. In the context of inheritance-rich code that is typical of the application domains that mostly benefit from OO design, both the enforcement of strong subtyping restrictions and the repeated verification of common properties is infeasible.

Parkinson and Bierman proposed Separation Logic (SL) as a means to support the verification of OO inheritance with minimal reverification [PB08]. This approach requires the developers to make an extra specification effort: static specifications are needed to verify method implementations and direct method calls, and dynamic specifications for calls that are dynamically dispatched. The relationship between static and dynamic specification also needs to be stated explicitly. The dynamic specification is however the only part involved in the dynamic subtyping verification thus reducing the overhead of supporting all forms of inheritance.

jStar is a tool for automatic verification of OO Java programs that combines SL with symbolic execution. The specification requirements are reduced thanks to abstract interpretation techniques able to annotate automatically loops with non-trivial invariants. jStar was used to verify the implementation of four popular OO design patterns (subject/observer, visitor, factory, and pooling) that escape traditional verification approaches [DP08].

Concurrency:
While Dijkstra WP calculus can be used to verify sequential algorithms, its extension to concurrency poses major challenges. In fact despite WP based semantics for concurrent language constructs have been proposed, automated deductive verification of concurrent programs remains impractical [SZ92].

The predominant approach to deal with concurrency is based on *temporal logics* but it is limited to finite state verification. Temporal logics introduce time modalities into the specification language so that time constraints can be expressed. Efficient model checking algorithms have been discovered for certain classes of temporal logic.

Concurrent SL aims to bridge the gap between sequential and concurrent verification, its goal is to enable the verification of global properties of concurrent programs by reasoning locally. In particular Concurrent SL can reason about the dynamic transfer of ownership of shared mutable data among execution threads [Bro07].

3.2 Symbolic Execution

Symbolic Execution (SE) is static program analysis technique that finds application in software testing, in particular for test input generation [PY07]. SE explores the program paths and computes the conditions for their execution. The test case executing a certain path can be obtained by finding concrete values satisfying the corresponding conditions.

Similarly to Hoare calculus presented in Section 3.1.2, SE uses the predicate transformers semantics of programs and applies deductive forward reasoning to analyze their behavior. Unlike the verification approaches presented in the first part of this chapter, SE is path-oriented as it analyzes program paths individually. As programs paths are typically infinite, SE is an incomplete analysis and only considers a subset of the possible program behaviors. This explains why SE finds applications mainly in testing while Hoare Calculus and WP are better suited for proving program properties.

The main feature that distinguish SE over Hoare calculus is the fact that SE is a fully automated analysis. The requirement of full automation prevents many of the techniques discussed for WP to be fruitfully applied in the SE context, in particular it is the case for loop invariants and SL, whose automation is still an open research problem.

The idea of generating test cases using SE dates back to late 70's with the seminal works of Boyer et al., King and Clarke [BEL75; Kin76; Cla76]. An account for the recent progress in SE for test case generation is given in Section 2.3.1. This section introduces the SE analysis technique and its dynamic extension named *Concolic Execution* with emphasis on their application to testing.

3.2.1 The Symbolic Execution Analysis

SE executes programs using symbolic values as inputs. The values of the variables, and more in general the program state, are represented using symbolic expressions on those inputs symbolic values. At every moment during the execution, the (symbolic) state of the program being executed includes the program counter pointing to the next instruction, the symbolic values of the variables and the Path Condition (PC). The PC is a predicate on the symbolic input values, that represents the condition that the input has to satisfy for the execution to follow the associated path.

When SE encounters a conditional instruction, the branch condition is evaluated in the symbolic state and conjuncted to the PC. If the updated PC is satisfiable, the obtained input values constitute a test input driving the execution through that program path. If the PC is not satisfiable, the corresponding program path is infeasible and its exploration is therefore stopped.

The space of all the possible symbolic executions can be represented in a tree: the *symbolic execution tree*. In such tree nodes represent symbolic program states and arcs represent program statements. SE analysis can explore the symbolic execution tree according to different strategies, depth-first and breadth-first being common choices. The detection of an infeasible path makes the exploration strategy backtrack and continue along a new direction.

```
    int x, y;
1:  if (x > y) {
2:    x = x + y;
3:    y = x - y;
4:    x = x - y;
5:    if (x - y > 0)
6:      assert(false);
    }
```

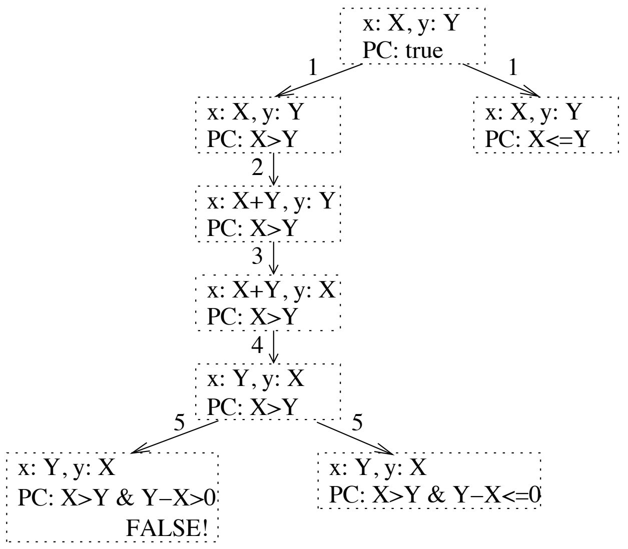


Figure 3.1. Code that swaps two integers when the first is greater then the second, and the corresponding symbolic execution tree, where transitions are labeled with program control points. Image from Kurshid et al. [KPV03].

In the following we illustrate the SE analysis technique on a simple example by

Kurshid et al. [KPV03]. Figure 3.1 shows the code that swaps the values of two variables x and y when the initial value of x is greater then the initial value of y. The right side of the figure reports the corresponding symbolic execution tree produced by the analysis.

The analysis starts with an empty symbolic state. Variables `x` and `y` are the inputs of the program and therefore they are assigned free symbolic variables: X and Y respectively. The PC is initialized to true as every execution path traverses the program entry point. The program counter is initialized to 1, the first instruction that needs to be executed.

SE encounters first the conditional instruction at line 1. The execution follows nondeterministically each of the two branches of the code, creating two different paths characterized by disjoint PCs. When following the `else` branch, SE evaluates the condition `!(x > y)` in the current symbolic state obtaining $X <= Y$, it updates the PC accordingly and backtracks as there is no other instruction to be executed on that path. The execution proceeds with the `then` branch of the conditional instruction. SE updates the PC with the symbolic evaluation of the condition `x > y` that is $X > Y$. The program counter is changed to 2, the next instruction along the current path.

Instructions 2, 3 and 4 are assignments that modify the symbolic execution state. The right-hand side of the assignment is first evaluated to a symbolic expression in the current symbolic state, the symbolic value of the variable on the left-hand side is then replaced with the computed expression. This behavior matches the semantics of assignments as defined using the WP calculus and expressed by the Equation 3.4.

The execution of statement 5 determines a second branching in the symbolic execution tree. Along the `then` branch, the condition `x - y > 0` is evaluated to `Y-X>0` and conjuncted to the current PC producing the predicate `X>Y & Y-X>0`. As this path condition is contradictory, the analysis can conclude that the path 1→2→3→4→5→6 is infeasible and backtrack to analyze the `else` branch. Along the `else` branch, the condition `!(x - y > 0)` is evaluated to `Y-X<=0` and conjuncted to the current PC producing the predicate `X>Y & Y-X<=0`. SE has now reached the end of the program and all branches have been explored, the analysis is therefore concluded.

The SE analysis we performed, explored three distinct paths in the program, each of them characterized by a different PC. Generating a test case exercising each of these paths amounts to solve the corresponding PCs if possible. The PC of path (1) is `X<=Y` which is satisfied for example by the values $X = 1, Y = 2$. The PC of path (1,2,3,4,5,6) is contradictory and is therefore infeasible. The PC of path (1,2,3,4,5) is `X>Y & Y-X<=0`, and is satisfied by $X = 2, Y = 1$.

3.2.2 The Path Explosion Problem

The program in Figure 3.1 can only be executed along three different paths. Moreover each of the execution can be analyzed symbolically without any approximation that would sacrifice completeness. However in general, enumerating all the paths in a program is unrealistic as the number of paths grows exponentially with the number of conditional instructions. Moreover in the presence of loops or recursion, the length of a single path may become unbounded and therefore its symbolic analysis might never terminate. These problems are known as the *path explosion* problem and the *path divergence* problem, respectively, and represent the main limitation to the industrial applicability of SE, together with the decidability of the underlaying language semantic model (see Section 3.1.5).

At this point it is important to consider that when the goal of the analysis is testing, the theoretical incompleteness of the approach might not be a problem. Given a test acceptance criterion, for instance a structural coverage criterion, it might be possible to perform a SE analysis powerful enough to satisfy the criterion. In this perspective researchers developed several strategies for exploring the SE tree, with the goal of optimizing specific testing goals.

Search heuristics aim to guide the exploration of SE trees prioritizing paths that are likely to achieve the search objectives fast. Such heuristics are often based on the Control Flow Graph (CFG) of a program and take advantage from the higher level of abstraction of such models to approximate the likelihood of a path to cover a target test element via graph algorithms like shortest path or path spectra [BS08; PM10; LSWL13]. Other heuristics are based on the dynamic analysis of test executions and determine a state-based fitness function for a candidate symbolic path [XTdHS09]. Heuristics are hard to evaluate and compare, and it is always possible to find specific cases for which they do not perform well. The work presented in this PhD thesis can be seen as a search strategy for symbolic executions trees, in particular ARC proposes a goal-oriented bidirectional search strategy that aims to overcome the problem of existence of cases for which a particular heuristic does not perform well.

Beside incompleteness, another aspect that clearly sets apart SE for testing from deductive verification techniques is the focus on automation. While WP sacrifices complete automation for completeness, for example by requiring manual annotations to the code and integrating interactively with theorem provers, SE techniques aims to be fully automatic and are therefore based on fully automated theorem provers like the Satisfiability Modulo Theories (SMT) solvers (see Section 3.4). The focus on automation has implications especially in the treatment of loops: as automatic loop invariants are seldom available SE analysis unrolls loops to sequences of conditionals and apply heuristic techniques to avoid non termination.

In a recent empirical study, Xiao et al. observed that the most common strategies to deal with the path explosion and path divergence caused by loops is to bound the number of loop iterations and use heuristics to guide the search to uncovered branches and away from infinite paths [XLXT13]. They considered the problem of generating test cases for branch coverage and concluded the following: (1) Loops that are executed a fixed number of times and loops that are input dependent but do not compromise the coverage of subsequent branches can be easily handled by bounded iteration and search guiding heuristics. (2) These two techniques can address loop problems caused by about 65% of the remaining loops (input dependent and affecting the coverage of subsequent branches). The study identifies directions to improve the treatment of loops in SE including novel heuristics and loop summarization.

3.2.3 Complex Constraints

The program in Figure 3.1 only uses integers and linear arithmetic. As we already observed in Section 3.1.5, industrial software makes use of programming language features that are not easily modeled in ways that allows for efficient deductive reasoning. SE might as well produce path conditions with complex constraints that are not manageable by automatic constraint solvers.

It may not always be possible to solve path constraints as the problem of solving the general class of constraints is undecidable. Moreover, it is possible that the computed path constraints become too complex (e.g., constraints involving non-linear operations such as multiplication and division and mathematical functions such as sin and log), and thus, cannot be solved using available constraint solvers. The inability to solve path constraints reduces the number of distinct feasible paths that a symbolic execution system can discover

In SE, the analysis of complex constraints can be simplified by dynamic software analysis techniques. As a matter of fact, unlike the other deductive verification approaches, SE is path-based and can therefore be coupled naturally with concrete executions. This idea is exploited in the class of approaches called Dynamic Symbolic Execution (DSE). The most popular DSE technique is *concolic execution*, it synthesizes in the name its main components: concrete and symbolic execution. Section 3.2.4 introduces concolic testing and shows how it helps overcome some of the limitations of static symbolic analysis.

3.2.4 Concolic Testing

Concolic Testing has been proposed as a fully automated testing technique based on dynamic SE [GKS05; SMA05]. It uses dynamic information extracted from test executions to assist the symbolic analysis of code, thus overcoming some of the imprecision inherent to static analysis. Concolic Execution is fully automatic and does not requires manual intervention for identifying test interfaces or designing test drivers.

Concolic Testing generates concrete test inputs randomly and executes the corresponding tests both concretely and symbolically. Concolic execution analyzes the path conditions collected by SE and produces new program inputs that direct systematically the execution along alternative program path. The process is repeated until all the execution paths are covered or the allotted time budget is consumed.

Concolic testing can use concrete values of program variables to simplify the symbolic constraints that are too complex for the underlying SE engine. In fact, automated solvers cannot deal effectively with non linear conditions or floating point numbers and symbolic analysis cannot make sense of binary code. Using concrete values for variables that cannot be evaluated symbolically introduces imprecision in the analysis but in practice the approximation is often good enough to guarantee the progress of SE. This capability enables the analysis of programs that are out of reach for traditional SE techniques.

The following steps are a streamlined, high-level description of the concolic testing algorithm. The algorithm is executed iteratively until all execution paths are covered or a timeout is reached:

1. execute a test case Concolically, i.e., both concretely and symbolically, collecting the symbolic path constraints at every branching condition.

2. negate one constraint in the path condition formula to obtain a new path condition. The new condition characterizes all the executions following the same path up to the selected branch and diverging at that point.

3. generate a new test input by solving the new path condition using an automated theorem prover

4. execute the new test case concolically checking that the execution indeed follows the alternative path at the desired branch

5. Repeat the process until all execution paths are covered

Concolic execution is an incomplete but sound analysis. A first source of incompleteness is substituting symbolic values with concrete ones: this simplification is unsound and the generated path conditions might therefore not guarantee the coverage of the desired path. Interestingly such event can be detected while executing the newly generated test and reported to the users. The second and more obvious reason why concolic testing might not terminate is the fact that programs might have a very large number of execution paths and executing all of them might be impractical.

Concolic testing might succeed in revealing a failure in cases where static analysis or symbolic execution based on the same theorem prover might fail. We use the code in Figure 3.2 to illustrate the behavior of concolic testing in when dealing with nonlinear constraints in comparison with static analysis and traditional SE.

```
1 nonlin(int x, int y){
2   if (x*x*x > 0){
3     if (x>0 && y==10)
4       abort();
5   } else {
6     if (x>0 && y==20)
7       abort();
8   }
9 }
```

Figure 3.2. A function that uses non linear conditions (from [GKS05])

The execution of the `abort()` instruction at line 4 is indeed reachable and would make the execution fail, on the contrary the one at line 7 is infeasible. The failing instructions however are dominated by the non linear condition at line 2.

Given an automated theorem prover that cannot reason about non-linear arithmetic, a static analysis tool, for instance an abstraction refinement tool, would not be able to detect the infeasibility of the condition at line 6 and would therefore conservatively report a false alarm about the reachability of the failing instruction at line 7. A static SE testing tool instead would try to generate an input value that satisfies the condition at line 2, but, since the condition is non linear, the solver would not be able to produce a correct assignment for the input variables, and would therefore terminate without producing a test case executing the failing line 4.

Concolic execution starts producing a random test input. Let us assume that the generated input satisfies the condition $(x > 0 \wedge y \neq 10)$, which has almost 50% probability, for example the test input $x = 1, y = 0$ that leads to the execution of lines 1, 2, 3 and 9. The concolic execution computes the path condition $(x * x * x > 0 \wedge x > 0 \wedge y \neq 10)$ up to the branching condition at line 3. It then computes a new path condition $(x * x * x > 0 \wedge x > 0 \wedge y = 10)$ by negating the last constraint. Since the condition contains a non linear expression, the concolic executor replaces the symbolic value of variable x with its concrete value 1, and passes the remaining symbolic constraint $(y = 10)$ to the theorem prover. The prover produces the test input $x = 1, y = 10$ that executes the `abort()` instruction at line 4 revealing the fault.

Let us consider what would happen if the initial random test case assigns a value less or equal 0 to x. The values $x = 0, y = 0$ would drive the execution through lines 1, 2, 6 and 9. The linearized path condition would be $(y \neq 20)$, the theorem prover would solve the negated condition $(y = 20)$ producing the test input $x = 0, y = 20$ where the value for x is obtained dynamically. The execution of the test would again follow the same path through lines 1, 2, 6 and 9, thus failing producing a test case for the `abort()` instruction on line 7, this behavior is compatible with the expectations, since concolic execution is a sound analysis technique that only executes feasible paths.

This example in the previous section shows that concolic execution might be incomplete when path conditions are simplified using dynamic data. A second type of incompleteness stems from the fact that the number of paths in a programs might be enormous and an exhaustive search might therefore result impractical. An active are of research in concolic testing tries to address this limitation with strategies that prioritize execution paths of particular interest. We have surveyed the main prioritization strategies that can be used in concolic testing in Section 2.3.1.

The development of DSE analysis tools follows two alternative implementation strategies, either instrumentation or interpretation. Instrumentation is the standard technique to perform dynamic analysis on compiled languages and works as follows: A program is first statically analyzed to discover the code locations of interest. At these locations the instrumentation process inserts calls to the instrumentation functions that encode the desired dynamic analysis and are provided as an external library. Afterwards the program is compiled normally and linked to the instrumentation library. Performing the dynamic software analysis task amounts simply to executing the instrumented program.

Concolic execution requires a major instrumentation as every part of the execution needs to be replicated by its symbolic counterpart. The instrumented program is therefore significantly bigger and slower then the original. Nonetheless as the instrumented program is compiled to a binary and executed natively, the instrumented SE can benefit from state of the art compiler optimizations.

The alternative strategy is more suitable for interpreted higher level programming languages. An ad hoc implementation of the interpreter performs the concolic execution on the unmodified code taking advantage of the infrastructure provided by the original interpreter. This strategy has been exploited to build concolic interpreters that run at the level of virtual machine monitors (hypervisors). This solution enables the analysis of binary code.

3.3 Abstraction Refinement

Abstraction promises to scale the application of model checking techniques to software systems. Model checking is an exhaustive property verification technique for finite-state systems that had enormous success in the context of verification of hardware properties. The main idea behind abstraction is to simplify an infinite state system in to a finite abstraction so that model checking can be applied. An abstraction is sound if every provable property is also valid in the corresponding infinite system, it is complete if every property of the system is provable on the abstraction. A large body of research in software model cheeking is devoted to the identification of methods to build sound abstractions automatically.

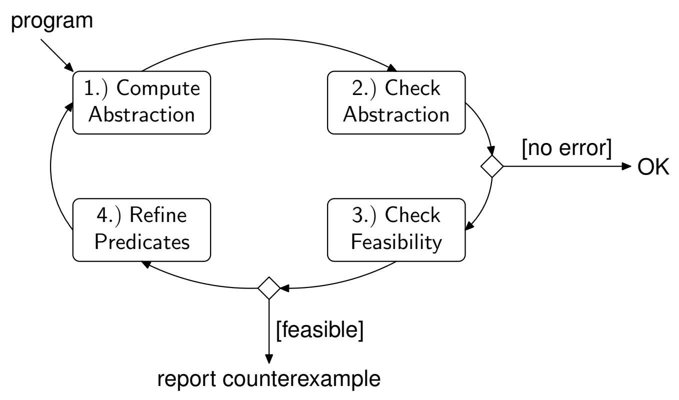


Figure 3.3. The CEGAR iterative verification loop.

The most well known approach to software model checking is Counterexample Guided Abstraction Refinement (CEGAR) [CGJ$^+$03]. CEGAR is an iterative process that starting from a coarse abstraction of a software system is able to automatically improve its precision using refinement steps. Spurious counterexamples are property violations found in the abstract model that are not present in the original system. The analysis of counterexamples is used to generate a refinement of the abstraction that is more detailed then the previous one and in particular excludes the known counterexample. This process is executed iteratively until no spurious counterexamples are found in the abstraction proving the property valid for the original system.

Figure 3.3 represents the CEGAR verification loop which is composed by four main steps:

1) A sound abstraction is computed starting from the original program.

2) The desired property is checked on the abstraction, if no errors are detected the property holds for the abstraction and for the original program.

3) If instead a property violation is found, the algorithm checks its feasibility in the original system. If the error is real then the counterexample is reported to the user.

4) If otherwise the violation is a false positive, the abstraction is refined in a way that preserves safety and excludes the spurious behavior.

In the case of sound abstractions, CEGAR is a sound analysis as it never detects a false error and never produces a false property proof. CEGAR is complete for finite state programs as the refinement guarantees the analysis progress by increasing monotonically the abstraction precision. The analysis however might not terminate in case of infinite systems as the abstraction might produce an infinite number of spurious counterexamples.

ARC uses a CEGAR style property checking approach to detect infeasible elements of the coverage domain and exclude them from the coverage rate computation. Our infeasibility detection algorithm is based on Synergy [GHK$^+$06], a CEGAR algorithm for checking safety properties. Synergy builds a conservative abstract transition model of the program under analysis and checks every abstract path that reaches the error statements using SE. A property violation is confirmed when SE generates a test case that executes the faulty statement and discarded if SE shows that the path is infeasible. In this last case Synergy refines the current abstract model using a pattern oriented approach and a WP-based refinement predicate generation technique.

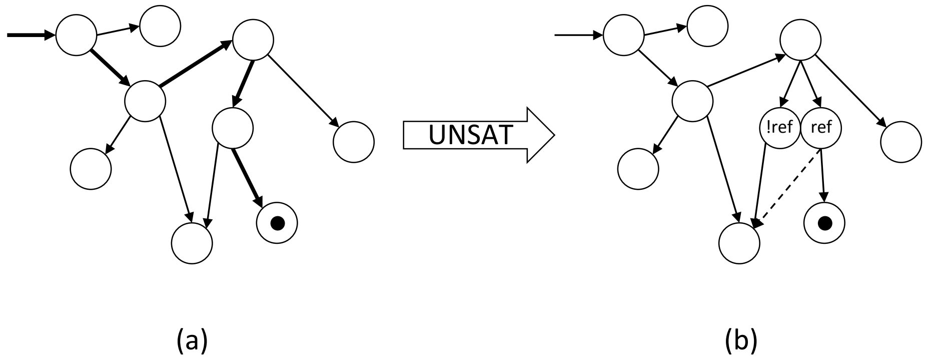


Figure 3.4. The pattern oriented refinment approach in Synergy.

Figure 3.4 shows the pattern based transformation of an abstract model into a refined one. Error states are marked with a dot and the abstract path under analysis is composed by thick transitions. The model refinements are triggered by the impossibility to generate a concrete test case that follows a specific path.

In our example, the SE-based feasibility checking has confirmed that the path to the faulty state highlighted in Figure 3.4 (a) is infeasible (UNSAT). Synergy updates the model by splitting in two the abstract state that comes before the last transition in the path. One of the two substates is annotated with a predicate `!ref` that characterizes the part of the state that is guaranteed not to lead to the error state. The transition to the error state can therefore be safely removed. The second substate is annotated with the complementary predicate `!ref`.

To guarantee that the resulting refined model (b) is sound, Synergy builds the refinement predicate `!ref` by negating the WP predicate of the error state along the last transition in the path. SE was not able to direct the execution to the error state, and the computed WP characterizes all the states that could lead to the error state. The substate annotated with the predicate `!ref` is therefore necessarily disjunct from the part of the abstract state that was explored by SE.

The refinement step reduces the reachability of the error state to the reachability of the substate annotated with the predicate `ref`. Such substate represents by construction only concrete states that were not included in the SE exploration. Subsequent iterations will push refinements back to the entry node of the model where a node split would disconnect the error state from the rest of the model proving it unreachable.

Synergy is an abstract intra procedural analysis and does not consider programming language feature that allocate memory dynamically, like pointers and memory heap. Computing precise refinement predicates in the general case would require an alias analysis that is path sensitive, thus strongly limiting the scalability of the approach. The pattern based refinement approach can nonetheless be successfully extended to cases where computing an exact WP predicate is not practical.

Beckman et al. developed the Dash algorithm to deal with pointers using dynamic alias conditions in the refinement predicates in a way that preserves the soundness of the produced abstraction [BNR+10]. Dash generates refinement predicates that are weaker then the exact WP but are consistent with the observed dynamic behavior. This is obtained by restricting the WP predicate to the observed alias conditions.

The dashed transition in Figure 3.4 (b) accounts for this last case. As the refinement predicate `ref` is weaker then the exact WP predicate, it cannot be guaranteed that a concrete execution satisfying `ref` will hit the error state. On the other side **ref** is strong enough to guarantee that an execution satisfying the complementary predicate `!ref` cannot hit the error state.

3.4 Constraint Solving

Constraint solving problems require to find a state for a set of variables that satisfy given constraints. In its general form constraint solving includes undecidable problems but researchers have successfully identified several decidable instances of the problem. Constraint Satisfaction Problems (CSP)s for example are NP-complete but only consider variables over finite domain.

In the last few decades the boolean satisfiability problem (SAT) and the Satisfiability Modulo Theories problem (SMT), two specific forms of CSPs, have received a lot of attention. Despite the fact that NP-complete problems are generally considered intractable, researchers have developed efficient approaches that can solve large SAT and SMT instances arising from concrete industrial problems. Software engineers could take advantage of the impressive progress in the field and proposed solutions based on SAT and SMT solvers that tackle problems in the fields of software design, analysis and verification as well as security, bioinformatics and more [DMB11].

3.4.1 Satisfiability

Propositional boolean satisfiability (SAT) is the problem of deciding if there exists an assignment that satisfies a given boolean formula. Satisfiability Modulo Theories (SMT) problems consider logical formulas expressed in classical first-order logic with equality.

SMT generalizes SAT by replacing boolean variables with predicates from a variety of underlying theories. This can be obtained by combining SAT with theory-specific solvers (T-solvers) that only need to handle conjunctions of predicates. The Nelson-Oppen method is the preeminent approach to obtain SMT solvers that combine different T-Solvers [NO79].

As SAT is a decidable albeit NP-complete problem, the decidability of a SMT problem only depends on its underlying theories. Researchers have carefully identified several decidable theories and constantly improved the corresponding T-solvers. Non decidable theories have been studies as well producing solvers that sacrifice completeness but can nonetheless be useful, in particular for the treatment of quantifiers.

3.4.2 SMT Theories

In this section we describe the SMT theories that gained top popularity in software engineering, thanks to their ability to express industrially relevant problems in a natural form. Assuming the point of view of a user of an SMT tool, we exemplify for each theory the class of problems that it can express and the most common domains of application.

Equality with Uninterpreted functions:
This is the simplest theory supported by SMT solvers, as the name suggests, it abstracts from the semantics of the modeled functions assuming that the validity of a formula is independent of the actual function behavior. The *congruence axiom* is generally enforced to guarantee functional consistency:

$$a = a' \Rightarrow f(a) = f(a') \tag{3.20}$$

It states that instances of the same function return the same value if given equal arguments.

Despite the extreme simplicity of this theory many properties can indeed be proved independently of the behavior of the specific functions involved. Predicates like the following for example, are easily recognized as unsatisfiable.

$$a \circ (f(b) + f(c)) = d \quad \wedge \quad b \circ (f(a) + f(c)) \neq d \quad \wedge \quad a = b \tag{3.21}$$

As we discussed in Section 3.1.5, uninterpreted functions lay at the base of the axiomatic formalization of arrays and dynamic memory access. In the hardware domain they can be used to model blocks that transform or evaluate data and analogously, for software components for which an implementation is not available with the assumption of absence of side effects.

Finite bit vectors with arbitrary size:
This theory considers fixed sized vectors of bits which can in turn represent any fixed width data types (int, short, long, etc.). Unsigned integers and two's complement arithmetic can be also encoded using bit vectors. All the traditional arithmetic operations can be used, including non linear ones like division and bit-wise logical operations (and, or, shift, extract etc.). This enables in turn the encoding of floating point arithmetics.

The remarkable expressivity ofthe bit vector theory comes at the price of increased complexity. On one side the size of the data types commonly used in industrial software makes expressions grow very big. On the other side the high granularity of bitwise operations produces many conditional clauses that challenge the underlaying SAT solvers. For this reason the bit vectors theory is mainly used when the ability to reason precisely about non linear arithmetics is critical, like in the verification of device drivers or cryptographic functions.

Linear artihmetics:
Linear arithmetics solvers can reason about boolean combination of linear constraints of the following form:

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n \bowtie b, \text{ with} \bowtie \in \{<, >, \leq, \geq\} \tag{3.22}$$

where the a_is, the x_is and b are in the domain $\mathbb{Z}$ of integer numbers or $\mathbb{Q}$ of rational ones.

Linear solvers are the most popular in software verification and automated test generation. They constitute a sweet spot in the trade-off between precision and efficiency as they match the abstraction level typical of application programming. This type of software in fact often ignores the low level details of machine arithmetics like two's complement representation, overflow or rounding.

Specialized solvers exists for the sub-theory of difference logic that includes predicates of the following form:

$$x - y \bowtie k \tag{3.23}$$

Difference logic has industrial application for problems like the optimization of job schedules and timing verification in electronic circuits.

Strings:
Kiezun et al. have recently proposed Hampi, a solver for string constraints over fixed-size string variables [KGG$^+$09]. String constraints often appear in web applications

and database driven software, often expressed in the form of regular expressions. Hampi translates regular language definitions to bit vector constraints that are then solved using off-the-shelf SMT solvers. If all the constraints are satisfiable, the output of the SMT solver is finally reinterpreted as a string.

Inductive data types:
Inductive data types (IDT) are defined using constructors, selectors and testers. A classical example is the inductive definition of a list of integers [Bar00]:

- Constructors: $cons$: (int, list) $\to$ list, $null$: list
- Selectors: car : list $\to$ int, cdr : list $\to$ list
- Testers: is_cons, is_null

SMT solvers for the first order theory of an inductive data type associates function symbols with constructors and selectors, and predicate symbols with testers. The following is an example predicate for the list data type.

$$\forall x : list.(x = null \lor \exists y : int, z : list.x = cons(y, z)) \tag{3.24}$$

While in the general case the satisfiability problem of IDTs predicates is NP-complete, there exist polynomial T-solvers for IDTs with a single constructor [Opp80; PW13].

Chapter 4

Abstraction Refinement and Coarsening

This chapter presents Abstraction Refinement and Coarsening (ARC), the main contribution of this thesis. ARC is a program analysis technique for automatically generating test suites that approximate full structural coverage. ARC aims to produce test suites that execute all feasible code elements while detecting the infeasible parts of programs. The technique promotes the synergies between static and dynamic program analysis for test case generation and infeasibility detection. Such synergy is made possible by a novel representation of the program state space called Generalized Control Flow Graph (GCFG) and is made applicable to industrial size programs by Coarsening, an algorithm that controls the redundancy level in the GCFG.

This chapter presents Abstraction Refinement and Coarsening (ARC), a program analysis technique that automatically generates test suites that approximate full structural coverage. ARC combines static and dynamic analysis techniques for test case generation and feasibility analysis. By generating new test cases and at the same time detecting infeasible elements, ARC is able to compute a precise code coverage rate that excludes infeasible elements from the coverage count.

ARC is designed with the goal of combining program analysis techniques for test input generation and infeasibility detection in a novel framework that enables the exploitation of their complementarity. The core element of ARC is the Generalized Control Flow Graph (GCFG) model that enables the interaction of different analysis techniques by providing an unified representation of their intermediate results. The efficiency of ARC is largely affected by the complexity of the GCFG model. Coarsening is a novel goal-oriented algorithm for optimizing abstraction complexity with respect to the desired structural coverage objective.

Section 4.1 introduces the guiding principle in the design of ARC that is to exploit the possible synergies between complementary analysis techniques at different levels of abstraction. Section 4.2 describes the ARC iterative process that combines test generation and reachability analysis in a unitary framework. Section 4.3 describes the GCFG model that is at the core of ARC and enables the interaction between its different components. Section 4.4 introduces the important concept of GCFG frontier that guides the ARC analysis towards the achievement of the desired structural coverage objectives. Section 4.5 describes the test generation component of ARC. Section 4.6 describes the infeasibility detection component of ARC. This component enables the implementation of a bidirectional search strategy making ARC a goal-oriented test generation approach. Section 4.7 introduces coarsening, the algorithm that allow the scalability of ARC.

4.1 Obtaining High Structural Coverage

In the past decades several empirical studies have shown that test suites approximating full structural coverage can guarantee high levels of confidence in the quality of software systems (see Section 1). Obtaining full structural coverage however requires that every code element of the type specified by the selected criteria is analyzed and either covered with a test or proven infeasible. In the following we discuss on how the two alternative results, traditionally addressed with specialized techniques, can be synergetically merged using the ARC unitary test generation approach. The guiding principle in the development of ARC is in fact the identification and exploitation of increasingly tighter levels of integration between test generation techniques and infeasibility detection techniques, ranging from the problem definition level to the analysis design and implementation level.

4.1.1 Ordering Coverage Targets

A first level of synergy that can be exploited concerns the ordering of the analysis of different code elements in a coverage domain: as code elements are rarely independent, it is likely that the analysis targeting a certain code element impacts some other ones. For example, a test case selected to cover a specific branch will cover other branches in its execution trajectory (i.e., collateral coverage), thus making their further analysis unnecessary. Similarly the proof of infeasibility for a code element that dominates other elements can be trivially extended to them.

The effect of ordering coverage targets in test generation techniques can be dramatic. As the cost of Automated Test Data Generation (ATDG) targeting specific code elements is not predictable, it can happen that the testing effort is misspent on a small number of hard to reach targets despite the availability of many easily reachable ones. Fraser at al. proposed and compared different ordering strategies in the context of ATDG via model checking [FGW09]. The results highlight the relevance of the problem and show that heuristic solutions do not perform consistently better then random ordering approaches.

ARC is carefully designed so that no particular coverage ordering is enforced by construction. This goal is achieved by guaranteeing that the GCFG conservatively overapproximates all the possible program paths that might reach a coverage target. Section 4.4 describes how the identification of a *frontier* on the GCFG directs the analysis nondeterministically towards all the coverage targets.

4.1.2 Static and Dynamic Analysis

A second level of synergy in ARC is achieved by exploiting the complementarity of conservative static program analysis with incomplete but sound dynamic analysis. ARC solves the problem of achieving high code coverage rates, by combining static reachability analysis with dynamic test case generation. On one side, when a code element is recognized as infeasible, it is safe to conclude that test generation activities that target that same element will never produce a covering test case and should therefore be dropped. On the other side, when a code element is covered by a test case, its infeasibility can be excluded immediately.

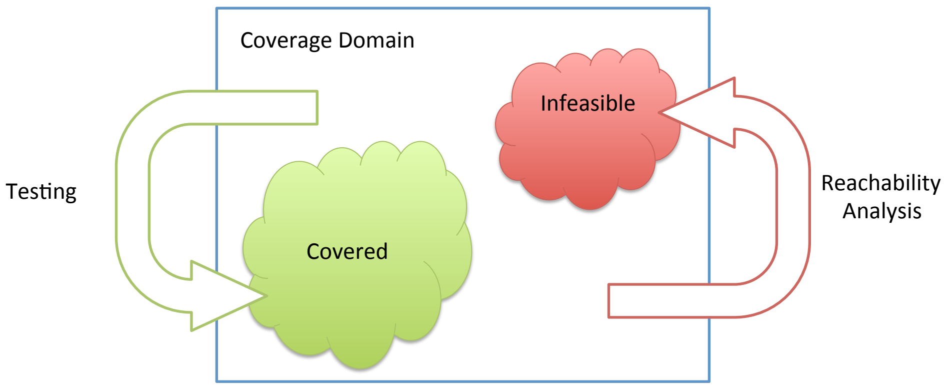


Figure 4.1. Testing and Reachability Analysis benefit each other when applied to the same program elements (the coverage domain)

Figure 4.1 represents testing as a process that takes elements from the unexplored part of the coverage domain and classify them as covered using test case generation techniques. Similarly reachability analysis detects infeasible elements. Each of the two analyses has the potential to reduce the unexplored part of the coverage domain and therefore benefit the complementary analysis as well.

4.1.3 Analysis-specific Synergies in ARC

The design of ARC draws on the intuition that even deeper synergies between complementary software analyses can be identified and exploited to improve their combined performance. In particular 1) test generation can be directed by the partial or inconclusive results about the reachability of code elements and 2) reachability analysis can assume the feasibility of the program (sub-)paths traversed by tests without further computation. Similarly to the situation depicted in Figure 4.1, this deep synergy can be exploited only if the combined techniques share and contribute to the same model of the coverage domain. The GCFG model is the ARC component that is responsible for the deep integration of static reachability analysis with dynamic test data generation (see Section 4.3).

ARC combines a test generation analysis technique based on Dynamic Symbolic Execution (DSE) with a reachability analysis technique based on Abstraction Refinement and Weakest Precondition (WP) calculus (see Chapter 3). The basic techniques that are combined into the ARC test generation approach have been selected based on two basic criteria: their expected effectiveness for the specific problem of obtaining very high code coverage and their potential for integration.

As discussed in Chapter 2, symbolic approaches to ATDG have better chances to cover corner cases and code elements that map to small regions of the program input space with respect to random or search-based testing approaches. This aspect is expected to become more and more important as the coverage rate grows towards approximating 100% and therefore leaving behind an increasingly small uncovered input space.

Abstraction Refinement and Deductive Verification are the premier technique for analyzing reachability properties of infinite state software systems. While abstract interpretation and data flow analyses can efficiently detect dead code, they usually favor scalability at the cost of precision by focusing on abstract domains with high granularity. Deductive verification approaches seems to be better suited for detecting the infeasibility of specific code elements thanks to their ability to reason at the level of program paths. ARC uses traditional dead code elimination as it can be obtained through a state of the art optimizing compiler as a preprocessing step.

Symbolic testing and abstraction refinement based on WP allow for a high degree of integration. In ARC the test cases generated symbolically serve as reachability counterexamples that can be readily used for refinement (see Section 3.3). Moreover dynamic information can be used to produce abstraction refinement predicates that are path-specific, this opportunity is exploited in ARC by following the schema proposed by Beckman et al. to compute alias aware WP predicates [BNR$^+$10].

On the other hand abstraction refinement produces reachability conditions for code elements as intermediate results of the analysis. Such conditions constraint the reachability of code elements of different granularity from program paths to sub-paths. In ARC symbolic testing takes advantage of reachability conditions, expressed as predicates on the program state space, to avoid directing the analysis effort towards infeasible directions.

4.2 The ARC Iterative Analysis

The ARC analysis for ATDG is an iterative process designed to maximize the interaction opportunities of its components. As already discussed, the first step to fruitfully combine dynamic symbolic testing with abstraction refinement-based feasibility checking is to have them share seamlessly the results of their analysis. In particular it is useful making even partial and incomplete results immediately available. To this purpose we based ARC on a novel state-transition model called GCFG (see Section 4.3).

The GCFG represents the structural coverage targets of the program and all the possible program executions that can exercise them. While the test generation progresses, the GCFG model is updated accordingly by removing covered elements from the set of targets and excluding infeasible sub-paths from further analysis. ARC updates the model using abstraction refinement to produce annotations that direct further test generation efforts.

The GCFG elements for which we have either dynamic or static analysis results constitute the coverage frontier (see Section 4.4). Focusing the test generation effort on the frontier elements increases chances to produce concrete executions exercising coverage criterion targets that have not been covered yet. The frontier is in fact composed of reachable elements that can potentially lead to the coverage targets.

To scale to large programs, ARC must control the size of the GCFG that could grow unbounded when the abstraction refinement component of the analysis become predominant. To this purpose we introduced *Coarsening*, a novel algorithm that operates on the abstract model of the program under test and optimizes it based on the chosen coverage criteria (see Section 4.7). Coarsening prunes information irrelevant for the analysis from the GCFG, while guaranteeing that at any given moment during the analysis, the GCFG contains the information relevant to the reachability of the remaining coverage targets.

The interaction scheme used in ARC guarantees progress to the test generation and prevents the reachability analysis to waste resources analyzing program sub-paths that are part of known test cases. ARC does not employ any explicit path exploration strategy for symbolic execution, the order in which paths get analyzed is goal oriented and implicitly derived from the selected coverage criteria. By avoiding premature decisions on the path exploration strategy to employ, ARC preserves as much nondeterminism as possible in the analysis, and such nondeterminism can be potentially exploited for parallelization. The extra memory requirements that derive from the need of storing the complete coverage frontier in the GCFG model is reduced with coarsening.

```
1: ARC(program, criterion=BRANCHES, tests=NULL, infeasible=NULL)
2:    model = generalizedCFG(program, criterion)
3:    do model.updateFrontier(tests, infeasible)
4:       tests += generateTests(model.frontier)
5:       infeasible += findInfeasible(model.frontier)
6:    while model.frontier != NULL
```

Figure 4.2. Pseudocode for the ARC execution loop.

Figure 4.2 shows the ARC iterative process in pseudo-code: Line 1 shows the signature of ARC: it takes as input a program to test and a coverage criterion (branch coverage by default), optionally ARC can be used to extend an existing test suite and can be initialized with a set of infeasible elements. The algorithm starts by creating the GCFG model for the program under test with respect to the desired coverage criterion (line 2). Lines 3 to 6 are executed iteratively: Line 3 identifies the coverage frontier on the model by dynamically tracing test executions and excluding infeasible paths from the count. Line 4 tries to augment the current test suite so that the coverage frontier would be expanded towards the target elements. Line 5 detects infeasible paths that traverse the coverage frontier. Line 6 checks if the entire coverage domain is already classified as either covered or unreachable and otherwise jumps back to line 3 to continue the process.

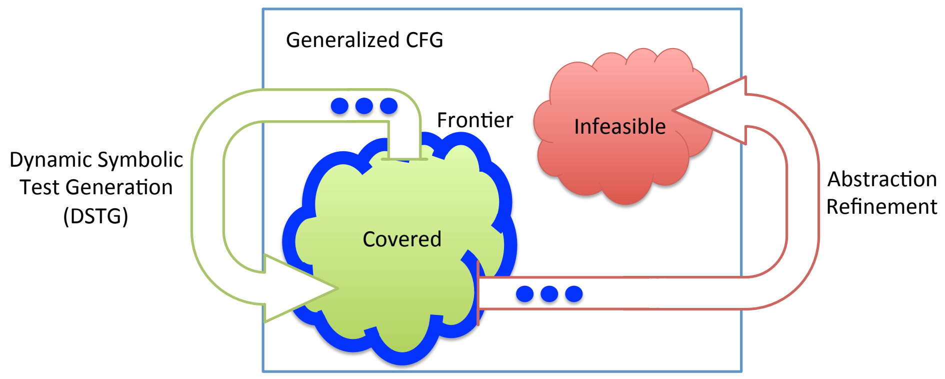


Figure 4.3. The iterative analysis performed by ARC and its main components.

Figure 4.3 illustrates the interaction between symbolic tests generation and abstraction refinement in ARC. The *Coverage Domain Frontier* (blue in Figure 4.3) is the region of the GCFG that contains the elements on which the interaction occurs. The coverage domain frontier is the interface between code elements that are already covered by existing tests and elements that are not yet covered and could lead program executions towards coverage targets. At every iteration, ARC selects elements from the Coverage Domain Frontier, analyses them and updates the model with the analysis results. Section 4.4 defines the coverage domain frontier and describes how it can be identified in an interprocedural GCFG.

ARC extends an existing test suite to cover new code element using a form of DSE directed by the GCFG. To maximize the chances to find new test cases, ARC analyses the elements on the coverage domain frontier. This guarantees that the newly generated test case makes the analysis progress towards increasing the desired coverage criteria. When symbolic testing finds new test cases, it augments the frontier with the newly discovered transitions that might be exploited to direct new executions to the target elements. Section 4.5 describes the process that generates new test inputs to exercise specific coverage targets.

ARC detects infeasible code elements using abstraction refinement, a static model analysis algorithm that refines the GCFG by propagating feasibility conditions of the coverage frontier elements toward the program entry point. Even if abstraction refinement might be unable to conclusively prove the infeasibility of a frontier element, the partial feasibility conditions are reported in the GCFG model. Such model annotations can be used by the Symbolic Execution (SE) component to steer new executions towards the coverage targets via different paths. Section 4.6 describes how ARC detects infeasible elements and produces model annotations.

The ARC iterative process is guaranteed to progress as both symbolic tests case generation and abstraction refinement reduce monotonically the state space to be analyzed. However as the state space of programs is typically unbounded, it is possible that the size of GCFGs grows indefinitely. Abstraction coarsening is a dynamic model checking algorithm that is responsible for minimizing the GCFG with respect to the existing test suite and the remaining coverage targets. Coarsening removes redundancy from the model, that is, every element that is not potentially leading to a coverage target prescribed by the selected coverage criterion. Coarsening is the key to scalability of the ARC approach and is discussed in Section 4.7.

4.3 The GCFG

The GCFG is an abstract state-transition model of program executions. It represents regions of the program state space (abstract states) as nodes and program execution paths as edges in the graph. It plays a central role in ARC as it enables the interactions between its static and dynamic components. ARC keeps track of new test case executions marking the covered abstract regions of the GCFG and annotating them with information dynamically extracted from test executions. During the progress of the ARC analysis, the GCFG model evolves as abstract program states can be either refined to smaller ones or merged together. Abstract states are characterized by predicates on the program state space.

The GCFG represents the state space of the program in an abstract and complete model. Each node in the graph represents an abstract state that is a subset of the program state space. Each edge in the graph represents all the possible program executions that can lead from the abstract state represented by the source node to the abstract state represented by the destination node.

The GCFG model is *unsound* as it can model infeasible program states (states that cannot be reached by any program executions) and spurious transitions (paths that cannot be followed by any program execution). The model is *complete* as all the execution traces of a program can be mapped to a path in the graph that connects the abstract states corresponding to the visited concrete program states. The model can be refined when new information about the program behavior is acquired using annotations.

The GCFG is refined by splitting nodes and transitions. Refined nodes are annotated with predicates over the program state space that characterize the abstract program states. A model can be refined to lower its abstraction level by excluding infeasible program states and infeasible execution paths from the representation.

We restrict the valid GCFG annotations to the ones that are conservative in preserving the completeness of the model. Moreover we require that GCFG refinement predicates can be evaluated in the concrete state of reaching test cases. This implies for instance that a valid refinement predicate can only refer to program variables that are in the scope of every test case that can reach the refined node.

As the above description suggests, it is possible to build an infinite number of GCFGs for a given program under test. Models can represent the program state space at different abstraction levels, but they all abstract the same concrete model.

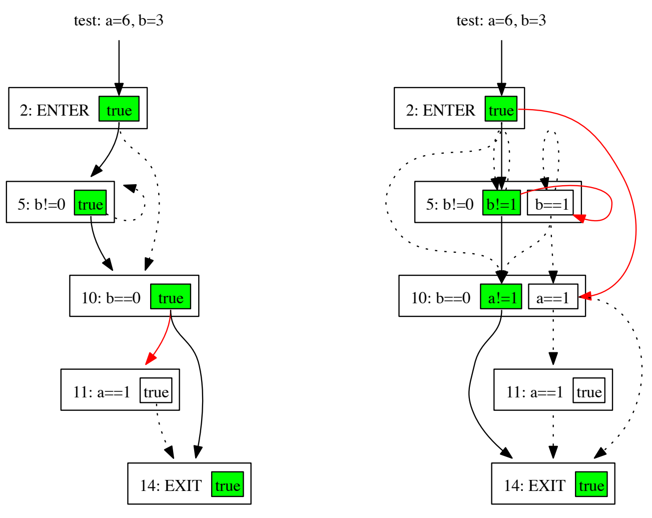
The traditional Control Flow Graph (CFG) is a simple GCFG in which each abstract state represents a branch of the program or, to define it in terms of the program execution state, each abstract state represents all the concrete state with the same program counter. Since our ATDG analysis targets branch coverage, we initialize the GCFG with the CFG. For this reason, the program counters are left implicit, and the abstract states of the initial model are annotated with the predicate `true`.

Figure 4.4 shows two of the GCFGs of the simple function `coprime`. The function computes the Greatest Common Divisor between the parameters a and b using Euclid's algorithm (lines 2-8) and returns `true` if they are coprime (i.e., their only common divisor is 1) or `false` otherwise. The models represent the program annotated with the execution trace of the test case with input $\{a = 6, b = 3\}$.

In the figure, green nodes represent abstract states that are covered by the execution of a test case, black solid arrows represent covered transitions, and dashed arrows represent transitions that are not covered by any of the test cases. Red transitions are the transitions that belong to the frontier. They are not covered by

```
 1: bool coprime(a,b){
 2:   temp=0;
 3:
 4:   while(b!=0) {
 5:     temp = b;
 6:     b = a%b;
 7:     a = temp;
 8:   }
 9:
10:   if(a==1){
11:     return true;
12:   }
13:   return false;
14: }
15: //test: a=6, b=3
```

(a)



(b) (c)

Figure 4.4. code for function `coprime` (a), the initial GCFG (b) and a refined GCFG model (c).

the test cases executed so far and lead to uncovered abstract states (see Section 4.4).

While the initial model (Figure 4.4 b) is isomorphic to the CFG of the function, the refined model (Figure 4.4 c) is enriched with extra states, transitions and state predicates. Both GCFGs overapproximate the behavior of function `coprime`. The completeness of the refined model (Figure 4.4 c) is guaranteed by the fact that every refined node is paired with a companion node that represents the states with the complementary predicate. As the complementary predicate is never used directly in the test generation algorithm, it is never explicitly computed or stored in the model.

ARC uses the GCFG to direct test generation towards uncovered code elements. While the test generation component of ARC tries to extend existing test cases beyond the coverage frontier, the static analysis component annotates the GCFG with the reachability conditions that need to be satisfied by test cases that aim at the remaining coverage targets. The GCFG model represented in Figure 4.4 c includes the following information: to be able to reach line 11, a test case needs to reach line 10 from line 2 in a state where the value of variable a is 1 or alternatively reach line 5 with b equal to 1 from within the loop. Such conditions are not met by any of the current test cases. The test generation can try to extend the test cases reaching line 2 using the condition $a == 1$ or the test cases reaching line 5 using the condition $b == 1$.

4.3.1 The Interprocedural GCFG Model

Classically there are two main approaches to generalize an intra-procedural program analysis to the interprocedural case [PS81]. The first strategy is called the *functional approach*, it works by combining context insensitive function summaries in a bottom-up fashion. Summaries are the result of intra-procedural analyses that can be combined compositionally thus contributing to the analysis of higher level functions. This strategy is popular for overapproximated static analysis techniques but suffers from the complexity of functional compositions, for example in case of recursive functions and does not allow to trade off precision for performance to reduce complexity.

The second strategy is called the *call string approach* and is context sensitive as it regards context information as part of the analysis domain. Call strings are used to represent possible calling contexts and the program analysis needs to deal with them as part of the program state. This second approach is the one we chose for ARC as it is more suitable for a sound but incomplete analysis like ours. Moreover in ARC, context information is easily accessible as both test generation and infeasibility detection analyses are path-oriented.

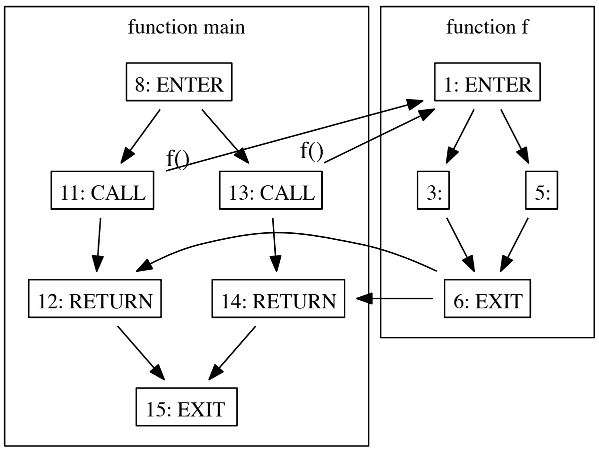
To properly deal with procedures, interprocedural GCFGs use special nodes to represent the program states at the entry and exit points of a procedure as well as call and return points. This representation is commonly used for interprocedural CFG and gives a complete flow representation, but at the cost of introducing a further level of imprecision [RHS95]. It is in fact possible in such model to follow paths where the correct pairing between procedure invocation and termination is not preserved.

Consider for example the program sketched in Figure 4.5. The path traversing lines $8 \rightarrow 11 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 14 \rightarrow 15$, which can be traversed in the model, does not match any concrete program execution. In every realizable program execution in fact, each time the execution reaches function `f()` from line 11 it will continue to line 12 and every time `f()` is called from line 13 it will return to line 14. Paths in the GCFG that respect the call$\rightarrow$return sequence of the modeled program are called *Realizable Paths.*

Non realizable paths can be excluded from the model using suitable model refinements. For this purpose refinement predicates can include calling context constraints. Only executions that match the prescribed calling context constraints are mapped to the corresponding abstract state.

```
 1:  f(){
 2:    if(...)
 3:      ...
 4:    else
 5:      ...
 6:  }
 7:
 8:  main(){
 9:    ...
10:    if(...)
11:      f();
12:    else
13:      f();
14:    ...
15:  }
```

(a)



(b)

Figure 4.5. Interprocedural Generalized Control Flow Graph model: an example.

4.4 The Coverage Domain Frontier

The coverage domain frontier is the subset of the transitions on a GCFG that connect covered to uncovered abstract states and lead to coverage targets that are not covered yet. The frontier contains abstract transitions that are reached but not traversed by test cases. Extending a test suite coverage over the frontier results in augmenting its structural coverage.

ARC builds on the observation that the coverage domain frontier is the location of a GCFG where the interplay between DSE and abstraction refinement can be exploited more successfully. In fact, on the frontier transitions, both dynamic information from test case executions and static reachability conditions with respect to the coverage targets are available. This information allows extending the path conditions of test cases reaching the frontier with the predicates characterizing the target nodes past the frontier to generate targeted test cases (see Section 4.5). When the attempt to generate a test case traversing the frontier fails because the new path condition is unsatisfiable, ARC can improve the precision of the reachability conditions of coverage targets using abstraction refinement and taking advantage of the available dynamic information (see Section 4.6).

The function `updateFrontier(tests, infeasible)` in Figure 4.2 is responsible for tracking the execution traces on GCFG models and removing transitions that are proven infeasible. To this goal each transition that is touched by the model update is analyzed and added to the coverage frontier if it satisfies the following three criteria:

- The source node of the transition models a covered abstract state.
- The destination node models a non covered abstract state.
- A coverage target is reachable starting from the transition destination node.

4.4.1 Realizable Coverage Frontier

When considering interprocedural GCFGs the definition of coverage frontier needs to be adapted taking into account the overapproximation of the interprocedural models. In fact DSE and abstraction refinement can only analyze transitions that correspond to exactly one branch in an execution trace. Interprocedural GCFGs instead include spurious transitions and abstract transitions that map to larger code fragments. For instance, transitions between call and return nodes are purely abstract compound transitions as they model the whole body of a function and therefore cannot be part of the coverage frontier. Similarly, transitions between source nodes covered only in some specific calling context and incompatible targets cannot be included in the coverage frontier.

More precisely the coverage frontier of an interprocedural GCFG is composed only of transitions that belong to *realizable* execution paths. A realizable path is a path that exhibits a matching sequence of call and return nodes in its call string [RHS95]. Let us consider again the path $8 \rightarrow 11 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 14 \rightarrow 15$ in Figure 4.5, this path is non realizable as the call string $\{11, 14\}$ does not constitute a valid sequence of call and return nodes. Instead, the path $8 \rightarrow 13 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 14 \rightarrow 15$ is realizable as nodes 13 and 14 are a valid call-return sequence.

Deciding if a transition is part of the realizable coverage frontier amounts to checking if the path followed by the test case reaching the transition is compatible with the abstract state after the transition. The compatibility can be checked comparing the call string of the test execution up to the frontier with the call string of the abstract state in the frontier destination.

Consider Figure 4.6 that represents a refinement of the GCFG graph in Figure 4.5 where nodes 14 and 15 have been refined with predicates $p14$ and $p15$ respectively. Solid black arrows represent the transitions traversed by a test case following the path $8 \rightarrow 13 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 14 \rightarrow 15$. The five red transitions represent the coverage frontier computed using the intraprocedural definition while the three red solid transitions are the realizable subset.

It is easy to see that the red dashed transition 13-14 is not part of the realizable coverage frontier as it crosses an abstract call-return edge. The analysis of transition 6-12 is more complex and requires that we take into account the call string of the test case reaching node 6. We denote a call string with the list of call nodes traversed by

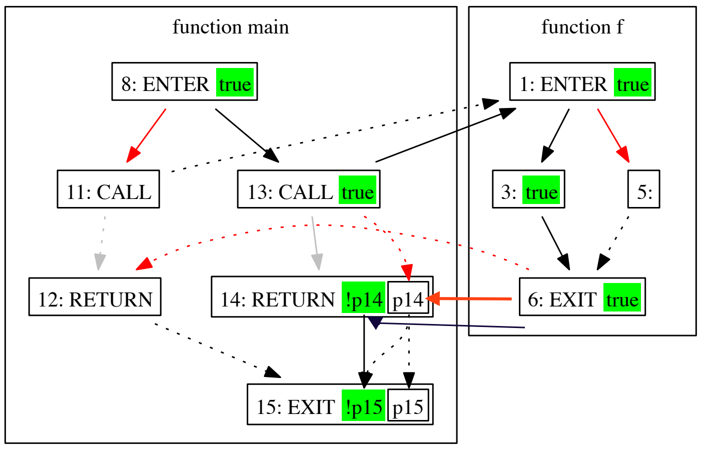


Figure 4.6. Interprocedural Coverage Frontier.

the test execution that are not balanced by a corresponding return node. The test represented in the figure reaches node 6 with calling context {13} and is therefore non compatible with the return node 12. Transition 6-12 is therefore not part of the realizable coverage frontier.

As we discussed in Section 4.3.1, refined abstract states can be annotated with predicates that restrict the set of valid calling contexts. In ARC, refinement predicates are computed as WP of the code along the path between a refinement and its target coverage goal (see Section 4.6), this guarantees that the call strings used to annotate abstract states are always realizable by construction. As both the test case calling context and the refinement context are well formed, their compatibility can be checked syntactically: the refinement context needs to be a *postfix* of the test calling context.

Two call strings do not need to be equal to be compatible because while the call string for a test case always starts from the outermost calling function `main`, the target abstract state might refer to a coverage target of an inner function. This case is exemplified in Figure 4.6 by the frontier transition 1-5 where the call string for the test case reaching node 1 is {13} and the call string of the abstract state in node 5 is empty. Transition 1-5 belongs to the realizable coverage frontier as it satisfies the postfix compatibility criterion.

4.5 Test Case Generation

ARC uses DSE to generate new test cases with the goal of exercising the targets of the selected coverage criterion. Unlike other DSE approaches that explore the program symbolic state space in a predetermined order (for example depth first), ARC is driven by the coverage frontier as it contains the symbolic transitions that can be traversed to diverge from existing test suites toward uncovered targets. In this section we describe the test generation algorithm used in ARC.

At every iteration, ARC selects one of the transitions in the coverage frontier. As the GCFG is annotated with the execution traces of the current test suites, ARC can identify the inputs of the test cases that reach the selected frontier transition. ARC uses the refinement predicate that is stored in the abstract state reached by the frontier transition to direct the test executions toward the coverage targets.

ARC generates a new test case that extends existing test cases and satisfies the frontier refinement predicate in three steps:

1. Execute the tests concolically up to the frontier, extract the path condition and the symbolic state.

2. Evaluate the refinement predicate in the symbolic states reached by the test executions obtaining a predicate expressed in terms of the program inputs.

3. Append the symbolic refinement predicate to the test path conditions obtaining an input condition for a new test case.

The new input condition produced in this way characterizes all the program inputs whose execution follows the same path as the original test up to the frontier, and then traverse the coverage frontier transition. The analysis continues as in a traditional DSE approach by using a constraint solver to check if the modified path condition is satisfiable. If the condition is satisfied the program is executed with the new input and its trace is recorded in the GCFG. The new test execution is expected to cover the frontier transition, but, as we already discussed, the concretization step performed during DSE may cause some imprecision (see Section 3.2.4). The execution of a new test case produces a new coverage frontier in the GCFG and possibly increases the overall coverage score. If on the contrary the path condition is not satisfiable, ARC triggers the infeasibility analysis step based on Abstraction Refinement that will eventually detect infeasible coverage elements (see Section 4.6).

In traditional DSE approaches, the execution of test cases is directed towards new paths producing a symbolic execution tree that gets progressively closer to the coverage targets. In the case of structural coverage criteria or test suite augmentation, the number of coverage targets is small with respect to the number of feasible paths. In this case DSE might spends most of the computing resources exploring paths that are not relevant to the testing goals. ARC test generation instead is informed by the refinement predicates about promising exploration directions, and is therefore able to look deeper in the symbolic execution tree towards the coverage targets.

Let us consider as an example the `coprime` function and the corresponding GCFG in Figure 4.4 c. If ARC selects transition $5 \rightarrow 5$ from the coverage frontier, it finds out that the only test case reaching the frontier is the test case with input {a=6,b=3}. Executing the input up to the frontier, ARC produces the path condition $B! = 0$ and the symbolic state $\{temp = B, b = A \mod B, a = B\}$, where A and B are the symbolic values for variable `a` and `b`, respectively. ARC evaluates the refinement predicate for node 5 (`b!=0 && b==1`) obtaining the symbolic value $A \mod B == 1$ and the complete path condition for the new test ($\{B \neq 0 \wedge A \mod B = 1\}$) that is satisfiable for example by the input {a=1,b=2} that reaches 100% branch coverage.

The modulo operator is non linear but is nonetheless supported by state of the art constraint solvers. Let us assume for a moment that our constraint solver does not support modulo arithmetic. In this case, the concolic execution would concretize the modulo operations and produce the symbolic state $\{temp = B, b = 0, a = B\}$. The complete path condition for the new test would be $\{B \neq 0 \wedge 0 = 1\}$ that is not satisfiable, and this would activate the abstraction refinement component of ARC.

4.5.1 Coverage Frontier Selection.

The example in the previous section shows that the selection of the element of the coverage frontier can be critical. For instance, selecting transition $2 \rightarrow 10$ instead of $5 \rightarrow 5$ easily produces the test case {a=1,b=0} using either a linear or a non linear constraint solver, reaching 100% branch coverage. However, there is no evidence that a particular frontier or target selection order performs significantly better in practice than random selection [FGW09].

ARC selects a new element of the frontier with a goal oriented approach that can reduce the number of paths that need to be explored symbolically. In fact, the coverage frontier contains only those transitions that may increase the coverage rate of the test suite. By maintaining the complete list of such transitions and selecting randomly, ARC avoids the pitfalls of search strategies, like depth first and heuristic search strategies (see Section 6.1). The goal oriented search strategy implemented in ARC can be framed in the schema of bidirectional search algorithms.

Traversing frontier transitions that correspond to loop back edges produces new test cases that generate longer traces by looping a higher number of times. Experimental evidence shows that shorter paths are less likely to be infeasible [PM10], for this reason a common search strategy in symbolic execution is to select only paths up to a fixed length, and therefore up to a small number of loop iterations. ARC does not avoid longer paths completely but selects forward transitions with higher probability then back edges, this results in giving higher priority to the generation of short test cases.

Another type of frontier transitions that are selected with a lower priority are the ones that lead to abstract states with non linear refinement predicates or, more in general, refinement predicates that are not accepted unmodified by the underlying constraint solver.

4.5.2 Tracing Executions on the GCFG Model.

ARC test cases are reexecuted every time their symbolic state is required for a new test generation step. In principle, it would be possible to store in the GCFG the symbolic state and the path condition of each of the executed test cases to avoid multiple executions. However to be useful, these data should be available for each branching decisions in the execution path, leading to an enormous number of stored symbolic state which is roughly the product of the number of test cases and their average path condition length. Clearly this approach is not feasible in practice as the model size would quickly become intractable.

Nonetheless, ARC stores some information about the test execution states every time a test trace traverses a node in the GCFG. The first piece of information is the number of code blocks, corresponding to the number of branching conditions that have been traversed by the test up to that execution point. This information is used to stop the reexecution of a test when it encounters the frontier transition. For example, when the frontier transition is in a loop, the transition source node may be traversed several times, many of which would not reach the frontier. The number of traversed code blocks is used to assure that the desired frontier transition is effectively reached. The second dynamic information stored in the model is the test execution calling context. This information is used for checking if a certain transition belongs to the coverage frontier of an interprocedural GCFG as described in Section 4.4.1.

4.6 Abstraction Refinement

ARC includes an abstraction refinement mechanism that aims to both detect infeasible parts of the coverage domain and direct test case generation towards coverage targets. The algorithm is based on pattern based refinement, a Counterexample Guided Abstraction Refinement (CEGAR) approach that uses WP predicates [GHK$^+$06]. It is triggered every time a test generation attempt fails as the theorem prover detects that the targeted program path is infeasible.

Abstraction refinement can progressively exclude infeasible paths from the GCFG model so that the next test generation attempts is driven towards new program paths. Paths can be removed from the GCFG model by splitting abstract states in two parts and annotating them with suitable predicates (see Section 3.3). This process is conservative as it only removes infeasible paths from the model. It guarantees to progress to the global ATDG analysis by reducing monotonically the amount of paths represented in the GCFG.

Consider again as an example the program `coprime` and the GCFG models in Figure 4.4. As we observed while simulating the test generation steps on model (c) the refinement predicates `a==1` in node 10 and `b==1` in node 5 are exactly the predicates that direct SE to the uncovered branch 11. Abstraction refinement can generate automatically these refinements predicate computing the WP for the state *true* along the program subpath $5 \to 10 \to 11$.

The code along the considered subpath includes three assignments and two conditions, the first (`b==0`) is the negation of the loop condition that must be valid for the execution to continue past the loop, the second (`a==1`) guards the execution of branch 11. The WP along the selected path is therefore computed as:

$$wp(\texttt{temp=b; b=a\%b; a=temp; if(b==0); if(a==1)}, \mathit{true}) \tag{4.1}$$

Rule 3.5 is first applied to the sequence of statements:

$$wp(\texttt{temp=b}, wp(\texttt{b=a\%b}, wp(\texttt{a=temp}, wp(\texttt{if(b==0)}, wp(\texttt{if(a==1)}, \mathit{true}))))) \tag{4.2}$$

Rule 3.6 is then applied twice to the conditional instructions:

$$wp(\texttt{temp=b}, wp(\texttt{b=a\%b}, wp(\texttt{a=temp}, wp(\texttt{if(b==0)}, a = 1)))) \tag{4.3}$$

$$wp(\texttt{temp=b}, wp(\texttt{b=a\%b}, wp(\texttt{a=temp}, a = 1 \wedge b = 0))) \tag{4.4}$$

Rule 3.4 computes the WP for the three assignment instructions:

$$wp(\texttt{temp=b}, wp(\texttt{b=a\%b}, \mathrm{temp} = 1 \wedge b = 0)) \tag{4.5}$$

$$wp(\texttt{temp=b}, \mathrm{temp} = 1 \wedge a \bmod b = 0) \tag{4.6}$$

$$b = 1 \wedge a \bmod b = 0 \tag{4.7}$$

Finally a well known property of the modulo operations is applied. In fact when $b = 1$ any variable a satisfies the predicate $a \bmod b = 0$, we can therefore omit this part of the constraint.

$$b = 1 \tag{4.8}$$

This concluses the refinement predicate computation that produces the GCFG in Figure 4.4 c.

In Section 4.5 we considered the case in which the constraint solver at our disposal is not able to handle nonlinearities like the modulo operator. In that case as we have shown, the attempt to cross the frontier 5→5 in program `coprime` c would lead to a further refinement on the model. This time the refinement predicate is computed by the WP function

$$wp(\texttt{temp=b; b=a\%b; a=temp; if(b!=0)}, b = 1) \tag{4.9}$$

The refinement predicate is computed analogously to the previous case and produces the following constraint:

$$a \bmod b = 1 \tag{4.10}$$

As the constraint contains the same non linear expression that caused the need for a further refinement, the frontier transitions leading to the corresponding abstract state will be selected with a low probability. Full branch coverage is finally obtained when ARC selects the frontier transition 2→10 for analysis (see Section 4.5.1).

In an Interprocedural GCFG, abstraction refinement produces both a refinement predicate as well as a *refinements context* that excludes from the model refinement frontier transitions along unrealizable paths. In Figure 4.6 for example, the refinement of the transitions 6→14 would produce a refinement context for node 6 denoted by the call string {13}. This refinement context specifies that only test cases reaching node 6 with a calling context compatible with the call string {13} might be extended towards the return node 14. The compatibility is checked syntactically by verifying that the refinement context call string is a postfix of the test context.

4.6.1 Detecting Infeasible Elements

In the following we illustrate with two examples how abstraction refinement achieves the two goals of detecting infeasible elements and directing test generation toward coverage targets. In both examples ARC is able to achieve 100% branch coverage. Section 6.1 shows how traditional SE approaches as well as heuristic-based concolic approaches cannot achieve full branch coverage on such programs.

As a first example we apply ARC to the analysis of the function `scan` in Figure 4.7 that contains an infeasible branch. The function `scan` calls the function `do_something` on each element of its input array `array`, starting from the element at position `start`. The function `do_something` checks that the position of the item is valid, being included in the interval $[0, size-1]$, and then prints the value of the array element at position `item`.

This example mimics a common schema employed for defensive programming. A function precondition is checked dynamically at the function entry point to verify the caller compliance. In our case the function `scan` always passes a valid index as parameter to the function `do_something` and therefore the validity check inside `do_something` can never fail. The branch at line 9 is therefore infeasible.

We show in Figure 4.8 the GCFG of the function `scan` at two stages of the ARC analysis. The models are simplified for clarity and compactness, in particular return

```
 1 void scan(int* array, int size, int start){
 2   while(start >= 0 && start < size){
 3     do_something(array, size, start);
 4     start = start + 1;
 5   }
 6 }
 7 void do_something(int *array, int size, int item){
 8   if(item < 0 || item >= size){
 9     printf("infeasible");
10     exit(-1);
11   }
12   printf("Item: %d", array[item]);
13 }
```

Figure 4.7. code for the function scan.

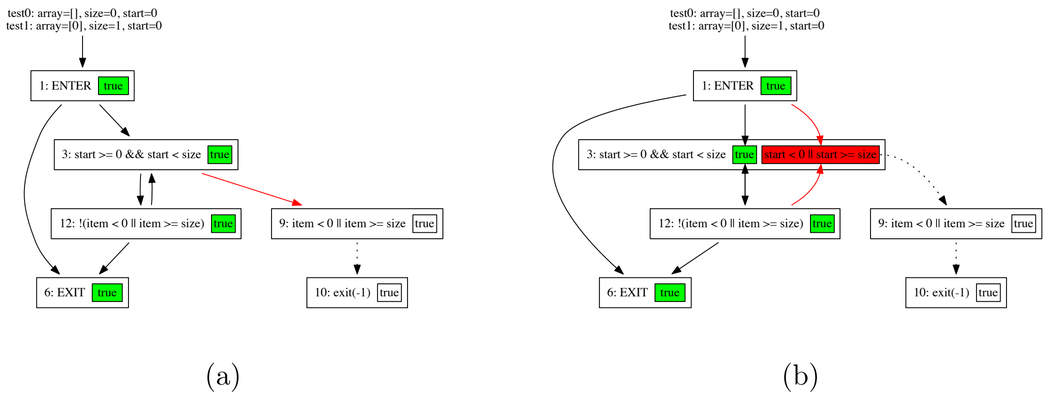

test0: array=[], size=0, start=0
test1: array=[0], size=1, start=0
1: ENTER
true
3: start >= 0 && start < size
true
start < 0 || start >= size
12: !(item < 0 || item >= size)
9: item < 0 || item >= size
6: EXIT
10: exit(-1)
(a)
(b)

Figure 4.8. Execution of Abstraction Refinement and Coarsening (ARC) on scan, the initial GCFG (a) and a refined GCFG model (b)

points are not unified and function calls are not modeled as they do not impact the analysis in this case.

Figure 4.8 a models the initial GCFG after the execution of two test cases: *test0*={array=[], size=0, start=0} and *test1*={array=[0], size=1, start=0}. The execution of the test *test0* follows the path 1→6, the execution of *test0* follows the path 1→3→12→6. The two inputs can be easily found with traditional SE by considering the two branches produced by the condition at line 2.

Analyzing the model in Figure 4.8 a, ARC identifies the coverage frontier that is constituted by the transition 3→9 that is reached by the test case *test1*. As the condition for the path 1→3→9 is infeasible, the test generation component of

ARC triggers a model refinement that produces the GCFG in Figure 4.8 b. The refinement predicate `start < 0 || start >= size` is computed as the WP along the transition 3→9 of the condition at line 9, replacing the formal parameter `item` with the actual parameter `start`.

As the predicate refining branch 3 contradicts the branch condition, ARC can immediately discard the corresponding node as infeasible (red in Figure 4.8 b). Removing the infeasible node makes branch 9 unreachable from the program entry point. ARC now concludes that no input can drive the program execution through branch 9 and therefore branch coverage cannot be further improved.

The precision of the refinement predicate computed using the WP calculus allows ARC to detect infeasible code reasoning locally. In our example ARC did not need to enumerate all the abstract execution paths reaching the infeasible elements, which would be impossible in the presence of loops. Traditional SE approaches as well as heuristic based concolic execution never terminate on this example, their analysis diverges trying to reach branch 9 through paths that include an increasing number of loop executions (see Section 6.1).

4.6.2 Directing Test Generation Precisely

```
 1 #define VALVE_NOT_WORKING(v) v == 0
 2 #define TOLERANCE 3
 3 int valves(int valves[], unsigned int size) {
 4   int count = 0, index = size;
 5   while(index >= 0){
 6     if(VALVE_NOT_WORKING(valves[index]))
 7       count++;
 8     index--;
 9   }
10   if(count > TOLERANCE)
11     printf("alarm\n");
12   return count;
13 }
```

Figure 4.9. Code for the function `valves`

In our experiments, we noticed that classic and heuristic SE search strategies can hardly cover branches that are executed only under a specific combination of several decisions along a path. Let us consider for instance the program `valves` that scans an array of integer values, tracks the count of all values equals to zero, and signals an alarm if the count is above a given threshold (Figure 4.9). The branch at line 11 is never covered using either depth-first or heuristic path selection strategies for concolic execution (see Section 6.1). This is because line 11 is executed only when the condition at line 6 holds true for several iterations of the loop. Executing the paths on which the condition at line 6 holds true for several but not enough iterations fails in covering the target branch and gives no indication on how to effectively approach the target.

As we observed experimentally ARC quickly generates refinement predicates that require progressively higher values of the variable `count` and propagates them inside the loop. The resulting coverage frontiers direct SE towards paths that satisfy the branch condition at line 6 for many iterations of the loop. In fact ARC is a target oriented approach, the combination of symbolic test case generation with abstraction refinement implements a *bidirectional search* in the symbolic execution tree of the program.

Bidirectional search is a search technique whose theoretical time complexity compares favorably to depth-first and breadth-first search. The idea of bidirectional search is to search forward from the initial state and backward from the goal state simultaneously. The goal state is reached when the two explorations meet.

For search trees with branching factor b in both directions, bidirectional search will find a solution at depth d in $O(b^{d/2})$ steps. Breadth-first search will require $O(b^d)$ steps while depth-first approaches might never terminate in case of infinite trees. These theoretical considerations suggest that bidirectional search could improve SE efficiency, but an efficient implementation is not always possible.

A first problem in implementing a bidirectional search algorithm is to define precisely what it means to search backwards from the goal in terms of a specific search space. In general, searching backwards means successively generating predecessors starting with the goal node, but, depending on the problem, calculating a node predecessor can be difficult in particular if the goal states are defined implicitly. In ARC we exploited the insight that SE and WP calculus compute effectively the same predicate transformer functions but operate in opposite directions on the code (see Section 3.1.2). Applying ARC to structural coverage targets allows us to define a bidirectional search strategy on the symbolic execution tree of a program having as targets an explicit list of symbolic program states.

A second problem is to identify an efficient way to check if each newly visited node appears in the search tree in the other search direction. In ARC this problem is approached with symbolic test case generation: if the path condition composed by combining the constraints collected in the forward and in the backward search is satisfiable then the symbolic states reached from the two opposite directions have an intersection that characterizes the solution. The refinement frontier makes the check more efficient as it greatly reduces the amount of satisfiability queries that need to be solved. In fact node pair that do not belong to the coverage frontier do not need any further analysis.

Bidirectional search requires that the state space explored by one of the two searches is kept in memory. This is crucial for detecting when the two searches meet and recognizing the finding of a solution. In ARC the refined GCFG maintains the global state of the backward search, the refined abstract states encode the extent to which the reachability of coverage targets has been explored. The symbolic states of the test during their execution, is instead recreated at every iteration by replaying their SE.

The space complexity for bidirectional search is $O(b^{d/2})$ which lays in between the space complexity of depth-first search (linear) and the complexity of breadth-first search ($O(b^d)$). The use of bidirectional search provides a good balance in the tradeoff between space and time efficiency. In the next section we discuss how ARC can scale to industrial size software using *coarsening*, a novel algorithm that minimizes the memory footprint of GCFG models.

4.7 The GCFG Coarsening

ARC uses abstraction refinement on GCFG models to reduce progressively the portion of the program state space that may be explored to augment the structural coverage. In traditional refinement algorithms, the size of the produced refinements grows monotonically until the desired property can be eventually proved. In ARC the reachability of all the coverage targets is analyzed simultaneously, this may produce large GCFGs and complex refinement predicates spanning distinct coverage targets, ultimately limiting the scalability of the approach to larger software systems.

During the ARC test generation process, coverage targets are progressively classified as either covered or infeasible. We observed that such partial results render parts of the refined GCFG redundant as it contains predicates that are irrelevant for the goal of increasing structural coverage. To minimize redundancy in the GCFGs and therefore reduce the space complexity of ARC, we developed *coarsening*, a novel algorithm that guarantees that each refined node in a GCFG is relevant for the analysis of the remaining coverage targets.

Coarsening is applied at every iteration of the ARC loop and undoes redundant model refinements. It can reduce both the number of nodes and transitions in the GCFGs model and the complexity of the refinement predicates that need to be checked by the theorem prover. Coarsening is one of the key contributions of our approach to ATDG and allows the application of ARC to industrial size programs.

Figure 4.10 shows the size of the GCFG during the analysis of the program `tcas`, a component of an aircraft traffic control and collision avoidance system available from the Software-artifact Infrastructure Repository (SIR) Repository [DER05]. ARC obtains 100% branch coverage for `tcas`, producing a test suite composed of 23 test cases in 735 iterations. The generated test suite covers 87 branches, the remaining 5 branches are correctly detected as infeasible (see Section 6.2).

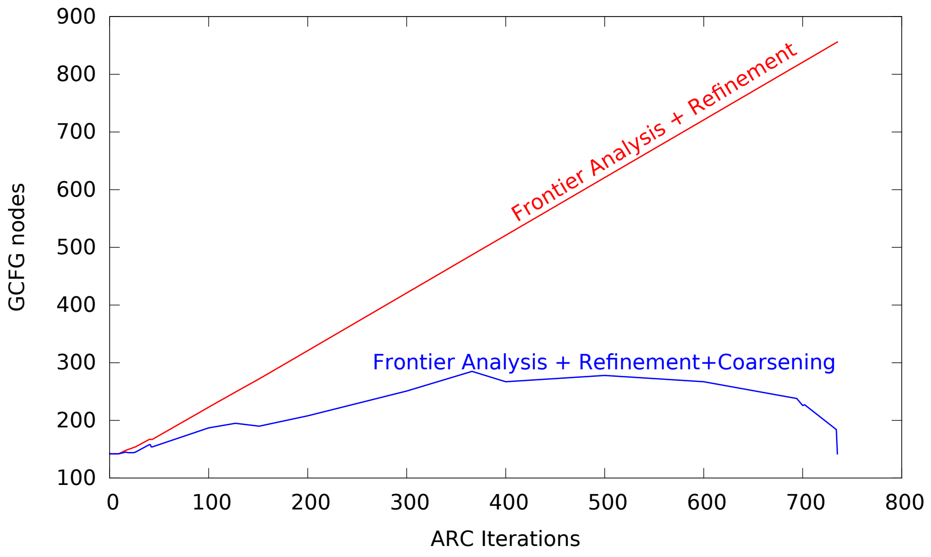


Figure 4.10. Comparison of the number of nodes in the GCFG during the analysis of the program `tcas` when coarsening is enabeld (blue), and disabled (red).

The red line in the plot shows the size of the GCFG using abstraction refinement without coarsening. Pattern based abstraction refinement creates a new node in the model at every iteration that does not produce a new test case, resulting in a monotonic a linear growth of the GCFG size. The blue line shows the size of the GCFG when coarsening is activated. Only the nodes that are relevant to the remaining coverage targets are represented in the GCFG model, it can be noticed for instance that when no more targets are left to be covered (iteration 735) the model size goes back to that of the original unrefined GCFG.

To enable coarsening, we need to slightly modify the abstraction refinement algorithm implemented in ARC. Every refinement node that gets added to the GCFG is mapped via a relation `split_for` to the target node for which the refinement was created. Considering the model in Figure 4.4 b for example, the `split_for` would include the pairs of nodes {`5:b==1` → `10:a==1`} and {`10:a==1` → `11:true`}.

When the execution of a new test case covers a node in the GCFG, the coarsening algorithm undoes all the refinements generated towards the analysis of that specific node. Coarsening then proceeds recursively along the structure defined by the `split_for` map to undo the complete chain of refinements leading to the newly covered node. In the model in Figure 4.4 b for example, the execution of the test case with input $\{a{=}1;\ b{=}0\}$ would cover the node `11:true`, provoking the coarsening of both the refinement node `10:a==1` and `5:b==1`.

Similarly, if an abstract state is detected as infeasible, coarsening ascends recursively the chain of refinements in the `split_for` map and undoes all the redundant ones. In this case however, all the GCFG nodes that are reachable from the detected infeasible state are checked as well. A coarsening procedure is started for each of the infeasible nodes.

Chapter 5

Prototype Implementation

The Abstraction Refinement and Coarsening (ARC) algorithm has been implemented in a prototype tool called ARC-B (ARC for Branch coverage testing). ARC-B generates test cases for C programs with the goal of evaluating the effectiveness of the ARC approach in obtaining full branch coverage for non-trivial software. This chapter discusses the design and implementation decisions involved in the construction of ARC-B.

This chapter introduces ARC-B, a prototype tool that has been employed to validate empirically the effectiveness of the Abstraction Refinement and Coarsening (ARC) approach (see Chapter 6). ARC-B is a robust and efficient Automated Test Data Generation (ATDG) tool for C programs that targets full branch coverage; this result was achieved also thanks to the extensive reuses of several third party, open source components. ARC-B is built on top of CREST[1], an automatic test generation tool based on concolic execution. CREST in turn relies on CIL[2] for the instrumentation and the static analysis of C code, and on the YICES[3] SMT solver. ARC-B extends the CREST functionalities by including four extra static and dynamic analysis components that concur to the implementation of the ARC algorithm: the *GCFG model*, the *ARC frontier analysis*, the *GDB* based *coverage tracer* and *ARC refinement*.

Section 5.1 describes the ARC-B tool from a user viewpoint describing in particular how the source code of a program under test can be prepared for the analysis. Section 5.2 overviews the high level workflow that governs the ARC-B analysis. The subsequent sections focus on the four analysis components that are specific to the ARC-B approach, describing in detail the most relevant aspects of their design and implementation. Section 5.3 describes the implementation of the *Generalized*

[1]https://github.com/jburnim/crest/
[2]https://github.com/kerneis/cil/
[3]http://yices.csl.sri.com/

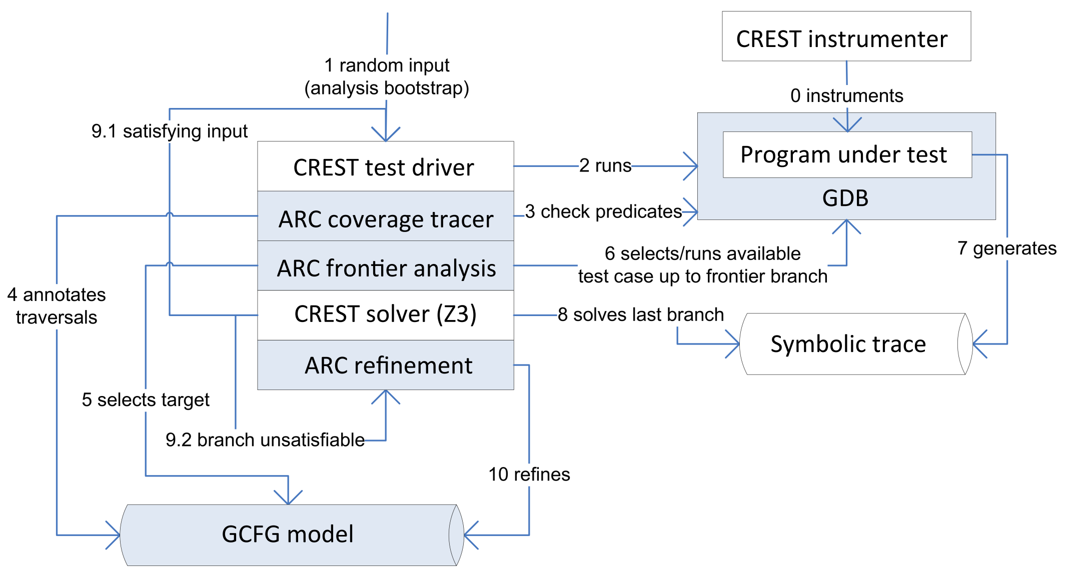


Figure 5.1. The logical modules and workflow of ARC-B.

Control Flow Graph (GCFG) model and the data structures that it provides to the other components. Section 5.4 describes how the *ARC frontier analysis* implements efficient data structures to identify and analyze the frontier transitions in the GCFG model. Section 5.5 describes how ARC-B uses a *GDB* based *coverage tracer* to dynamically analyze test case executions. Section 5.6 describes the implementation of the Weakest Precondition (WP) based *ARC refinement* and the coarsening algorithms.

5.1 Using ARC-B

The ARC-B prototype tool operates on programs written in the C programming language. ARC-B requires as input the source code of the program under analysis in a single .c file. This requirement is inherited from the CREST tool that we used as a base for ARC-B. The CREST distribution package includes a tool that mitigates this limitation by merging all the code mentioned in a Makefile into a single .c file.

To test a program with ARC-B, users need to write a test driver in the form of a `main` function. The driver instructs ARC-B about which program variables should be considered as the input of the program under test. An unconstrained symbolic value will be assigned to these variables by the ARC-B Symbolic Execution (SE) engine. The test driver typically creates a number of symbolic values using the ARC-B library functions, and invokes the functions under test using the symbolic values as parameters.

```
   int main() {
1:   int x;
2:   ARC_B_int(x, "X");
3:   ARC_B_assume(x >= 0);
4:   return foo(x);
   }
```

Figure 5.2. A simple test driver for ARC-B.

Figure 5.2 shows a simple driver for testing the function `foo` that accepts an integer value as input. Line 1 declares an integer variable `x` and line 2 assigns a symbolic value X of the appropriate type to it. Line 4 finally, calls the function `foo` using `x` as parameter.

The macro `ARC_B_assume` is used to introduce preconditions in the test drivers. Preconditions can be used to reduce the symbolic state space analyzed by ARC-B and can therefore have a big impact on the analysis performance. Line 3 in figure 5.2 shows how to use `ARC_B_assume` to limit the input space of function `foo` to a smaller range. ARC-B will automatically discard in this case all negative values for `x`. `ARC_B_assume` accepts any valid C condition as parameter.

ARC-B is a command line tool. When invoked, it instruments and compiles the program under test, linking it with functions that perform side by side concrete and symbolic execution and enable program execution monitoring. ARC-B repeatedly runs the program, building new tests cases at every step and detecting unreachable branches. Finally, when the analysis is completed or a given time budget is reaches, ARC-B estimates and reports the branch coverage achieved by the generated test suite, excluding unreachable branches from the total target count.

5.2 ARC-B Workflow

Figure 5.1 shows the logical modules of ARC-B and how they modify the original workflow of CREST. White rectangles represent the CREST components that are reused in ARC-B, while gray rectangles represent the new modules of ARC-B. The arrows, numbered from 0 to 10, model the execution steps that form an iteration of the ARC analysis loop, as well as the data dependencies between the ARC-B components.

ARC-B execution starts by instrumenting the program under test, manually annotated with symbolic input identifiers, using the CREST instrumenter (step 0 in Figure 5.1). The instrumentation code, which is generated using a customized version of the CIL library, implements a stack-based symbolic interpreter that runs alongside the normal program execution. The symbolic interpreter updates the symbolic execution state of the program under test and computes the path conditions.

The CREST test driver executes the instrumented program using an initial test suite if available, or using a randomly generated input otherwise (steps 1 and 2). ARC-B runs the test cases through the GDB[4] debugger, which allows the inspection of the program dynamic execution state. The extracted execution information is used by the ARC coverage tracer to dynamically check the validity of the abstraction refinement predicates and annotate the GCFG with test execution traces (steps 3 and 4).

In step 5, the ARC-B search engine selects an abstract frontier transition that has the potential to drive the execution towards uncovered elements. To find a new test that traverses the frontier, in step 6 ARC-B identifies the test cases that reach the frontier but do not traverse it. The compatible test cases are replayed up to the frontier, to generate the required path conditions (step 7). The CREST solver generates an input file for the Z3 solver that combines the path conditions with the eventual frontier predicate, and checks its feasibility (step 8).

Finally, if the solver succeeds in finding a test input that drives the program execution across the coverage frontier towards the target elements (step 9.1), the iterative process is restarted from step 2. If traversing the coverage frontier is instead infeasible (step 9.2), the ARC refinement component updates the GCFG model using WP based abstraction refinement (step 10), and the ARC-B analysis continues from step 5.

The ARC-B analysis iterates until all the coverage targets are classified as covered or infeasible by gradually removing infeasible program paths from the GCFG model. However, due to approximations in generating the path conditions and in the refinement predicates, the analysis does not always terminate but is stopped when reaching a predetermined time budget.

The Object Oriented design of CREST supports the implementation of custom strategies to explore the symbolic execution trees of the programs under test. This is obtained by extending the class `Search` and implementing the method `Run` which drives the SE analysis. Appendix A contains a streamlined version of the methods `Run` and `traverseFrontier` implemented in ARC-B. The method `Run` implements the workflow we just described and calls `traverseFrontier` to attempt to traverse a frontier and, when impossible, obtain an appropriate refinement predicate.

5.3 The GCFG Model

The GCFG model is at the core of the ARC technique as it represents the interface among the different components of the analysis and enables their interaction. In ARC-B we took special care in selecting appropriate data structures that sup-

[4]http://www.sourceware.org/gdb/

port efficient GCFG access and update methods. This section describes the most important design choices.

The ARC-B implementation of the GCFG is based on the `boost::graph`[5] library. `boost::graph` uses generic programming to obtain the highest grade of flexibility and extensibility without scarifying efficiency, in the style of the Standard Template Library (STL) [Aus98]. To support bidirectional searches in the GCFG, ARC-B represents the graph as an adjacency list with bidirectional edge access (access to both out-edges and in-edges of a node).

Property maps are the `boost::graph` mechanism that allows linking the abstract mathematical nature of graphs to concrete domain problems that can be expressed in terms of graph theoretical concepts. They can be used to attach generic properties to the vertices and edges of a graph. The abstract states of a GCFG are represented in ARC-B as `boost::graph` vertices. Several property maps are required to encode their properties:

- `vertex2branch`: links graph vertices to the branches of the program under test.
- `vertex2inputs`: links graph vertices to the set of test cases that reach the corresponding program branch.
- `vertex2predicate`: links graph vertices to the refinement predicate that denotes the abstract state it represents.
- `split_for`: links graph vertices that represent GCFG refinements to the graph vertices that are the target of those refinements.
- `is_target`: contains the vertices that map to coverage targets that are not covered yet.
- `is_refinement`: contains the vertices introduced in the model by abstraction refinement.
- `is_call, is_return, is_enter, is_exit`: contain the vertices that do not model actual program branches, but map to procedure call, return, enter or exit points, respectively.

GCFG abstract transitions are represented in ARC-B as `boost::graph` edges and are characterized using the following property maps:

- `is_frontier`: links edges to coverage frontier transitions.
- `is_back_edge`: links edges to loop back edges.

[5]http://www.boost.org/doc/libs/release/libs/graph/

In ARC-B, we encode predicates denoting abstract program states as strings in the SMT-LIB 2.0 format [BST10]. The predicates extracted from the C code of the test subjects are converted to the Satisfiability Modulo Theories (SMT) format in a preprocessing phase that uses a custom parser based on flex[6] and bison[7].

ARC-B extends the original CREST solver to support the Z3 SMT engine. Z3 accepts the SMT-LIB input format as well, reducing to a minimum the need of format conversions. In fact thanks to the common representation between the GCFG and the SMT solver, predicates generated with abstraction refinement are used directly to extend the path conditions obtained using symbolic execution.

```
(define-fun c_div ((x Int) (y Int)) Int
       (ite (>= x 0)(div x y)(div (- x) (- y))))
(define-fun c_mod ((x Int) (y Int)) Int
       (ite (>= x 0)(mod x y)(- (mod x y))))
```

Figure 5.3. Semantics of the C arithmetic operators / and %.

There exist some minor discrepancies between the arithmetic operators semantics in C and in Z3. We cite as an example the C operators / and % that, when the numerator operand is negative, behave differently than as prescribed by their standard mathematical semantics. In such cases it is useful to define appropriate helper functions in Z3 and specify their desired semantics. Figure 5.3 shows how ARC-B expresses the C semantics of the operators / and % in the Z3 language.

5.4 ARC Frontier Analysis

Each time an abstract transition is added to the GCFG model by a refinement step, and each time a new test case is executed, ARC-B needs to update the set of transitions that constitute the coverage frontier.

Figure 5.4 shows the code that implements the function `is_frontier` that is used to decide if an abstract transition belongs to the coverage frontier and that updates the frontier set accordingly. The code in lines 1 to 3, simply navigates the abstract transition modeled through the edge `e` in both forward and backward directions to find the vertices `fst_v` and `snd_v` that insist on it. Lines 4 and 5 encode the criteria that define an interprocedural coverage frontier. Finally lines 6 and 7 check that the frontier is interprocedurally realizable (see Section 4.4):

- `Line 4`: checks that the abstract state before the frontier transition is covered by a test case.

[6]http://flex.sourceforge.net/
[7]https://www.gnu.org/software/bison/

```
 1 bool is_frontier(edge_descriptor e){
 2     auto fst_v=source(e,graph);
 3     auto snd_v=target(e,graph);

 4     if(!vertex2inputs[fst_v].empty() &&
 5        (is_target[snd_v] || is_refinement(snd_v)) &&
 6        !(is_call[fst_v] && is_return[snd_v]) &&
 7        context_matches(fst_v, snd_v)){
 8           frontiers.insert(e);
 9           return true;
10     }
11     frontiers.erase(e);
12     return false;
13 }
```

Figure 5.4. Function `is_frontier` checks if a GCFG transition `e` is part of the coverage frontier.

- `Line 5`: checks that the abstract state after the frontier transition maps to a target branch or to a refinement that leads to a target branch.
- `Line 6`: excludes transitions that map to the whole body of a function (call-return transitions).
- `Line 7`: excludes unrealizable transitions that are transitions where the context of test cases reaching node `fst_v` and the context of the refinement node `snd_v` do not satisfy the *postfix compatibility criterion.*

Each of these four conditions must hold for the transition `e` to be added to the coverage frontier set `frontiers` (line 8), otherwise the transition is removed from the coverage frontier set (line 11).

While the first three conditions can be checked locally on the model, and require only the retrieval of values from the graph property maps, checking the condition at line 7 efficiently is harder. To achieve this goal we developed an efficient way to retrieve the set of test cases reaching the vertex `fst_v` with a context that is compatible with the one in the refinement node `snd_v`.

The code in Figure 5.5 defines the map `vertex2inputs` (line 4), the data structure that allows us to retrieve both the test cases based on the branches they traverse and their calling context. The key feature of the data structure implementation is the use of reverse lexicographic ordering as comparison function (line 3). The next paragraphs explain with an example how the reverse lexicographic ordering is related to the problem of verifying the postfix compatibility criterion for call strings.

```
//the type definition for a test input
typedef std::vector<value_t> input_t;
//the type definition for a calling context
typedef vector<vertex_descriptor> context_t;
//the type definition for a map that stores test inputs,
//indexed by calling context, sorted in reverse lexicographic order
typedef multimap<context_t,
                 const pair<const input_t, int>,
                 reverse_lexicographical_compare> inputs_map;
//a map that stores test input traces, indexed by covered branch
unordered_map<vertex_descriptor, inputs_map> vertex2inputs;
```

Figure 5.5. The `vertex2inputs` data structure supports the efficient retrieval of test inputs for frontier analysis.

Consider the problem of finding in a container all the call strings that are compatible with the refinement context {2, 3, 4} according to the postfix compatibility criterion, and suppose that the container includes the following call strings: {5, 1, 2, 3, 4}, {3, 4, 3, 4}, {5, 1, 3, 4}, and {3, 2, 3, 4}. We can visualize the reverse lexicographic ordering by first reversing all the call strings and using the familiar lexicographic ordering:

1. {4, 3, 1, 5}
2. {4, 3, 2, 1, 5}
3. {4, 3, 2, 3}
4. {4, 3, 4, 3}

We can immediately notice that the call strings at position 2 and 3 that are compatible with the refinement context {2, 3, 4} appear in a continuous sequence. If we reverse our refinement context we obtain the string {4, 3, 2}. In the sorted container, the sequence of the compatible test inputs is delimited on one side by the refinement context itself, and on the other by the smallest context that is not compatible with the refinement context: {4, 3, 3}.

The relation that we just discovered between the refinement context and the test contexts when sorted in reverse lexicographic order is illustrated in the following representation:

1. {4, 3, 1, 5}
2. **{4, 3, 2}**

3. {4, 3, 2, 1, 5}
4. {4, 3, 2, 3}
5. **{4, 3, 3}**
6. {4, 3, 4, 3}

The call strings at position 2 and 5 encode the refinement context and the smallest non-compatible context, respectively. As expected, they limit the sequence of the compatible test contexts, the ones that satisfy the postfix compatibility criterion.

```
   //finds all the tests reaching the vertex fst_v
   //with context compatible with ctx
   //returns a range of test inputs as a pair of iterators
 5 pair<inputs_map::iterator, inputs_map::iterator>
           get_compatible_tests(const context_t& ctx,
                                const vertex_descriptor& fst_v){
       //find the first test case with context compatible with ctx
 6     auto itlow=vertex2inputs[fst_v].lower_bound(ctx);
       //construct the smallest incompatible context
 7     ctx[0]=ctx[0]++;
       //find first test case with incompatible context
 8     auto itup=vertex2inputs[fst_v].lower_bound(ctx);
 9     return make_pair(itlow, itup);
10 }
```

Figure 5.6. Function that searches for test cases that are compatible with a given coverage target.

The function `get_compatible_tests` in Figure 5.6 returns a pair of iterators. They delimit the sequence of test inputs in the data structure `vertex2inputs` that cover the abstract state `fst_v` and are compatible with the refinement context `ctx`. The function implements the algorithm based on the reverse lexicographic ordering that we just exemplified.

Thanks to the expressivity of the C++ STL library, the implementation of the algorithm is very compact: Line 6 uses the method `std::multimap::lower_bound` to retrieve the iterator to the first element in the container that is not smaller then `ctx` in reverse lexicographic order. Line 7 generates a new context that is the smallest among the ones that are not compatible with `ctx`, and it is obtained by incrementing by one the first element of the call string. Line 8 uses again the method `std::multimap::lower_bound` to retrieve the iterator to the first element in the container that is not smaller then the smallest non-compatible context. The two iterators delimit the sequence of the test inputs that are compatible with the context `ctx`; they are returned as a pair in line 9.

It is important to notice that the ordering of elements in a `std::multimap` needs to satisfy the conditions of a *strict weak ordering* relation: irreflexivity, antisymmetry, transitivity and transitivity of equivalence[8]. Lexicographic ordering satisfies all these conditions being a total ordering relation.

5.5 Dynamic Analysis in ARC: GDB

ARC-B implements the dynamic analysis components of ARC using the GDB debugger to monitor and modify the state of running programs. GDB can be controlled programmatically using the GDB/MI interface, a text based interface that encodes GDB outputs in a machine friendly format. ARC-B interacts with the GDB/MI shell using Pstreams[9], a multi platform C++ interface to POSIX pipes.

GDB is used in ARC-B to extract both symbolic and concrete data from a program execution. This is possible because the programs under test are instrumented to execute concretely and symbolically at the same time. The symbolic state is therefore part of the concrete execution state of the program and can be accessed analogously.

5.5.1 ARC Coverage Tracer

The main role of dynamic analysis in ARC-B is to trace test executions on GCFG models. This is achieved by evaluating the refinement predicates that are stored in the model, at every branch encountered during executions. A test case covers an abstract state of the GCFG model when the corresponding refinement predicate evaluates to `true` in the test execution state.

As we already mentioned, refinement predicates are stored in the GCFG models as strings in the SMT-LIB 2.0 format. SMT-LIB is a functional language that uses a prefix notation for arithmetic and logical formula. To evaluate refinement predicates in GDB, we implemented an interpreter that translates SMT-LIB predicates to C statements.

The SMT-LIB interpreter for GDB evaluates SMT predicates in three stages. The first stage is external to the program under test and converts the prefix notation of SMT-LIB to valid C expressions before they are passed to the GDB/MI shell. The uninterpreted functions `select` and `store`, that are used in SMT-LIB to represent memory access via memory location references, are converted to calls to similarly named C functions.

[8] https://www.sgi.com/tech/stl/StrictWeakOrdering.html

[9] http://pstreams.sourceforge.net/

Consider as an example the following SMT predicate

```
(> (select (store mem id 5) id) b)     (5.1)
```

It compares the value stored at memory location `mem[id]` with the value of variable `b` after storing the value `5` at that same location. This predicate would be translated to the following valid C expression.

```
(select(store(mem, id, 5), id) > b)     (5.2)
```

Supposing that GDB is able to evaluate correctly the functions `select` and `store`, one could think of evaluating directly the valid C string that was produced in the previous step. The variable names that appear in the predicate, however, might not be visible in the current state of the running test case. This is because the variables in the refinement predicate might originate in a different stack frame and have reached the current abstract state via a refinement chain that traversed some interprocedural transitions.

The second stage of the dynamic predicate evaluation might therefore need to retrieve the dynamic values of variables from deeper stack frames. This is obtained using the GDB/MI command `-stack-select-frame` that changes the active stack frame. ARC-B finds the correct variable values by testing the availability of their names in the current stack frame and switching to increasingly deeper frames in case they cannot be found. The postfix compatibility criterion guarantees that every variable that is mentioned in a refinement predicate is accessible in the stack of a compatible test execution state.

At this point on the predicate evaluation, every symbolic variable in the predicate has been replaced by its concrete value. Let us suppose that the base address of the vector `mem` is 135025216, the value of the offset `id` is 1 and the value of variable `b` is 0. The example predicate we are considering would be evaluated to the following expression:

```
(select(store(135025216, 1, 5), 1) > 0)     (5.3)
```

The third and last stage of the dynamic predicate evaluation deals with the dynamic evaluation of the memory access functions `select` and `store`. An implementation of the two functions is available in the program under test as part of the instrumentation. As the implementation matches the semantics that `select` and `store` have in the SMT language, the expression can be interpreted directly by GDB to `true` as 5 is greater than 0. The C code that implements the purely functional interpretation of `select` and `store` is available in Appendix A.2.

5.6 ARC Refinement

The implementation of the GCFG model refinement in ARC-B matches closely the description in section 4.6. The function `split_region(frontier, predicate)` is responsible for adding new nodes to the model and reconnecting their in- and out-edges so that only the path that is detected as infeasible gets removed from the model (see Figure 3.3). The function is also responsible for populating the `split_for` map that links the newly create abstract state with the coverage target that caused the model refinement, thus enabling the coarsening step (see Section 4.7). When a new test is generated or a branch is proved infeasible, the function `coarsening` recursively traverses the `split_for` map undoing all the redundant refinements, that is the ones that do not refer to undecided coverage targets.

In this section we describe how the refinement predicates used as parameters for the function `split_region` are generated. As we described in the previous chapters, the refinement predicates in ARC are computed using the WP calculus along the coverage frontier that is being analyzed. While the traditional WP predicates can be computed statically by analyzing the program code, the computation of alias-aware refinement predicates combines instead static and dynamic information [BNR$^+$10]. In particular the refinement step needs to access the program state of the test case that is executed up to the frontier, with the goal of checking aliasing.

ARC-B computes the alias-aware refinement predicate in two steps. The first step is performed statically during the initialization of the analysis and consists of collecting the list of all the assignments that are performed by the program between every pair of consecutive branches, i.e. inside every linear code sequence. In fact given a predicate P, computing its WP along a GCFG edge e amounts to replacing the variables in P with the value they are assigned to along the edge e.

```
foreach(auto ass=assignments(b1, b2)){
    foreach(auto pred_var=pred.vars){
        if(gdb.is_alias(pred_var, ass.left)){
            //replace if alias of the assignment lhs
            pred.replaceVar(pred_var, ass.right);
            //build the alias-aware refinement predicate
            Predicate alias_pred=Predicate::parseSMT(pred_var);
            alias_pred.equal(ass.left);
            alias_pred.negate();
            pred.disjunct(alias_pred);
        }
    }
}
```

Figure 5.7. Computation of alias-aware, WP-based refinement predicates.

The actual variable replacement however is delayed to the dynamic test execution phase. Only when a tests get re-executed in ARC-B to recreate the symbolic states that determines the path infeasibility, is in fact possible to compute the alias conditions between the assigned variables and the predicate variables that are needed for generating the refinement predicates.

Figure 5.7 shows a simplified version of the code fragment that computes the refinement predicates used in ARC-B. The variable `pred` contains the predicate to be refined and the function `assignments(b1, b2)` returns the list of assignments performed between the branches `b1` and `b2`. Lines 1 and 2 create a loop that allows us to consider one by one all the possible combinations of predicate variables and assignments. Therefore, lines 3-8 get executed for every variable `pred_var` in predicate `pred` and every assignment `ass` performed between the branches `b1` and `b2`.

To compute the traditional WP refinement predicate, we would replace in the predicate `pred` every occurrence of the assignment left-hand side variable with the assignment right-hand side predicate. In ARC-B we want to replace not only the occurrences of the assignment left-had side but also all its aliases. The function `gdb.is_alias` at line 3, operates in a given concrete state represented by the object `gdb` and checks dynamically if `pred_var` and `ass.left` are alias by checking if they are stored in the same memory location. Only in this case the variable `pred_var` is replaced in the predicate `pred` (Line 4). Lines 5-8 finally correct the refinement predicate to guarantee the refinement soundness: the modified predicate `pred` combined using a disjunction with the negation of the observed alias condition.

Consider as an example the assignment `a=b` and the predicate `(c>0)`. Let us suppose that, on the frontier state, variables `a` and `c` are alias. The alias-aware refinement predicate would be computed as `(b>0 || &a!=&b)`. This abstract state will be a future target of the ARC-B analysis and will be eventually covered by an execution that either produces the same alias condition that we already observed on the frontier but verifies the predicate `b>0`, or by an execution that generates a different alias condition altogether.

Chapter 6

Evaluation

We evaluated the Abstraction Refinement and Coarsening (ARC) approach by applying the ARC-B prototype tool on a number of synthetic and industrial programs. This chapter describes the experimental setting and discussed the evidence we collected. We first show that the ARC goal oriented search strategy can indeed detect infeasible code and cover more code elements then depth first and heuristic based symbolic execution. Then we compare the coverage scores of ARC-B with competing techniques when applied on industrial programs. Finally we report about an independent study that evaluated ARC on a software component of a safety-critical control system, detecting several previously unknown failures.

Our main research hypothesis is that "*An analysis technique that is based on scalable program abstractions and that exploits the synergies between symbolic test generation and reachability analysis can generate test suites with high code coverage for industrially relevant software systems*".

In Chapter 1 we motivated the goal of obtaining high code coverage in the light of several classical and recent empirical studies and meta analyses that agree in correlating coverage scores that approximate 100% with high quality assurance levels. Chapter 4 describes our approach, Abstraction Refinement and Coarsening (ARC), that combines a test case generation approach based on Symbolic Execution (SE) with abstraction refinement based infeasibility detection.

To validate our research hypothesis we applied the ARC-B prototype tool described in Chapter 5 on several test subjects to estimate its ability of both generating test suites that cover feasible branches and identifying infeasible ones. In this chapter we report empirical data from two sets of experiments designed to answer the following research questions:

Q1 Can the symbolic approach to test input generation be combined with reachability analysis techniques to produce test suites with high coverage?

Q2 Can such a hybrid framework scale to industrially relevant software by exploiting efficient program abstractions?

Q3 Is the ARC approach able to expose previously unknown failures in industrially relevant software?

To address question Q1, we discuss in section 6.1 the results obtained while experimenting with artificial programs, built as variants of the codes of Figures 4.7 and 4.9 that we have discussed in Chapter 4. The goal of these initial experiments is to evaluate the ability of the ARC approach to solve problems that cannot be addressed well by symbolic and concolic tools, and to evaluate the performance of ARC-B for increasingly complex instances of these problems.

Section 6.2 presents experiments with industrial programs that contains feasible and infeasible branches. Throughout the experiments, we also compare the coverage rates obtained using ARC-B with the ones obtained using a representative set of concolic, symbolic and random testing tools. Part of the presented evaluation was published in [BBDP11]. New and larger test subjects have been added to demonstrate the recently developed support for procedures and dynamic memory access, as well as the improved scalability performance.

Section 6.3 reports our experience of applying ARC-B to industrial software with the goal of exposing software failures and addressing the research question Q3. The ARC-B prototype tool was applied to four incremental versions of a real-time software component that have been provided as case study by industrial partners in the context of the European FP7 project PINCETTE. The test suite produced with ARC-B exposed six unknown and subtle failures in the system, indicating that the ARC approach can be useful in improving the quality of software in realistic industrial settings [BDR+14].

6.1 Effectiveness of the ARC Search Heuristics

Table 6.1 reports the results obtained analyzing increasingly complex instances of the functions that we used in Chapter 4 to illustrate the ability of the ARC approach to detect infeasible code elements and direct test generation precisely towards uncovered ones. The subject of our experiments are labeled `scan`$\langle i \rangle$ and `valves_rep`$\langle i \rangle$, they refer to generated programs that are composed of a sequence of i replicas of programs `scan` (Figure 4.7) and `valves` (Figures 4.9), respectively. The programs named `valves_nest`$\langle i \rangle$ are similar to `valves_rep`$\langle i \rangle$ with the difference that the code replicas are not appended one to the other but are instead nested within the

		ARC-B					CREST-dfs		CREST-cfg		KLEE	
subject	**br**	**it**	**tc**	**cbr**	**ibr**	**cov**	**cbr**	**cov**	**cbr**	**cov**	**cbr**	**cov**
scan1	8	11	3	6	2	100%	6	75%	6	75%	6	75%
scan2	16	23	5	12	4	100%	9	56%	12	75%	8	50%
scan5	40	59	11	30	10	100%	18	45%	30	75%	14	35%
scan10	80	119	21	60	20	100%	33	41%	60	75%	24	30%
scan20	160	239	41	120	40	100%	63	39%	120	75%	44	28%
scan50	400	599	101	300	100	100%	153	38%	300	75%	104	26%
scan100	800	1203	200	600	200	100%	303	37%	600	75%	204	26%
valves1	6	29	4	6	0	100%	5	83%	5	83%	5	83%
valves_rep2	12	65	7	12	0	100%	7	58%	10	83%	8	67%
valves_rep5	30	161	16	30	0	100%	13	43%	25	83%	17	57%
valves_rep10	60	316	31	60	0	100%	23	38%	50	83%	32	53%
valves_rep20	120	633	61	120	0	100%	43	36%	100	83%	62	52%
valves_rep50	300	1594	151	300	0	100%	103	34%	250	83%	152	51%
valves_rep100	600	10000	300	599	0	99%	203	34%	500	83%	302	50%
valves_nest2	12	59	7	12	0	100%	5	42%	5	42%	5	42%
valves_nest5	30	149	16	30	0	100%	5	17%	5	17%	5	17%
valves_nest10	60	299	31	60	0	100%	5	8%	5	8%	5	8%
valves_nest20	120	599	61	120	0	100%	5	4%	5	4%	5	4%
valves_nest50	300	1499	151	300	0	100%	5	2%	5	2%	5	2%
valves_nest100	600	10000	217	431	0	71%	5	1%	5	1%	5	1%

br:	number of branches computed statically
it:	number of iterations of ARC-B
tc:	number of generated test cases
cbr:	number of covered branches
ibr:	number of identified infeasible branches
cov:	coverage [as percentage], i.e., cbr / (br - ibr) , in the case of ARC-B measured using the GNU `gcov` utility, in the case of CREST-dfs, CREST-cfg and KLEE

Table 6.1. Results of ARC-B, CREST-dfs, CREST-cfg and KLEE on variants of the programs `scan1` and `valves1`

body of the last `if`, that is, within the hard-to-cover branch. Column *br* of Table 6.1 reports the number of branches in each subject program, computed using static analysis.

We generated test cases for the subject programs with ARC-B, KLEE and CREST. KLEE implements the traditional approach based on static, depth-first symbolic execution [CDE08], while CREST exploits concolic execution and can be further configured to use either depth-first search (CREST-dfs) or control-flow graph guided heuristic path exploration (CREST-cfg) [BS08]. We ran the tools to generate test cases for each subject program with the goal of maximizing branch coverage, starting from a single input test case.

For each tool run, we report the number of covered branches (column *cbr*) and the branch coverage score (column *cov*). For ARC-B we also report the number of iterations performed (column *it*), the number of test cases produced (column *tc*) and the number of branches identified as infeasible (column *ibr*). In the case of CREST and KLEE, the number of generated test cases is equal to the number of iterations.

Since all the tools call the SMT solver once per iteration, the number of iterations gives a clear indication of the complexity of the analysis. We fixed a budget of 10,000 iterations for all the runs. ARC-B generates test cases that cover all feasible branches in all the experiments except for `valves_rep100` and `valves_nest100` where it does not terminate but achieves a coverage of 99% and 71%, respectively. CREST and KLEE never terminate when applied to the subject programs as their depth-first search strategy diverges for programs that contain loops.

The table shows that ARC-B steadily achieves very high branch coverage while the other tools cannot achieve the same coverage, and their results vary largely among the set of considered programs. ARC-B identifies all infeasible branches of programs `scan`$\langle i \rangle$, and covers all the difficult branches of both `valves_rep`$\langle i \rangle$ and `valves_nest`$\langle i \rangle$ up to 50 code replicas. It covers all but one branch of `valves_rep100` and about 30% of the branches of `valves_nest100`.

The performance of CREST-dfs and KLEE reduces progressively for programs with an increasingly higher number of loops. CREST-cfg has stable performance on the programs `scan`$\langle i \rangle$, where it covers all feasible branches, and on programs `valves_rep`$\langle i \rangle$, where it only misses one feasible branch per code replica. It shows very poor performance for the programs `valves_nest`$\langle i \rangle$, where most code is embedded into the hard-to-cover branches. All the tools show their worst performance for the program `valves_nest100`, where ARC-B scores about 71% branch coverage, while both CREST-dfs, CREST-cfg and KLEE do not go beyond 1% coverage.

This experiment answers positively the research question Q1, supporting the case for combining test generation and iterative refinement of abstract program models to achieve higher coverage. The target oriented approach implemented in ARC-B maintains the focus on the uncovered branches and can precisely identify and isolate infeasible branches. The ARC approach outperforms CREST-dfs and KLEE that instead engage themselves in the depth-first exploration of infinitely many program paths, it also outperforms CREST-cfg whose search heuristic based on the overapproximated program CFG, is deceived by the presence of infeasible branches.

6.2 Approximating Full Coverage on Industrial Programs

Table 6.2 lists 16 subject programs that we used in our experiments. The first 12 programs have been selected for a preliminary evaluation of the ARC approach. In fact despite their non-trivial branching structures, they are of limited size and make limited use of procedures [BBDP10]. This allows us to verify the ability of ARC-B

			ARC-B						
subject	**loc**	**br**	**tc**	**cbr**	**ibr**	**cov$_1$**	**it$_1$**	**cov$_2$**	**it$_2$**
linsearch	23	8	3	8	0	100%	3	100%	3
binsearch	39	12	4	12	0	100%	6	100%	6
tcas	180	92	23	87	5	95%	151	100%	735
week0	154	58	15	53	3	91%	50	96%	50
week1	154	58	16	53	3	91%	65	96%	65
week2	154	56	17	52	4	93%	70	100%	121
week3	154	56	16	52	4	93%	50	100%	108
week4	154	58	16	43	13	74%	55	96%	125
week5	154	58	16	53	3	91%	85	96%	90
week6	154	56	18	52	4	93%	90	100%	128
week7	154	56	18	52	4	93%	110	100%	139
week_	154	84	27	81	3	96%	135	100%	135
cdaudio	2 171	534	259	417	113	79%	20 965	99%	76 800
diskperf	1 104	194	59	166	14	86%	600	92%	13 200
floppy	1 144	234	75	206	14	88%	317	94%	14 700
kbfiltr	618	68	32	57	6	84%	129	92%	981
TOTAL	6 665	1 696	612	1 456	195	86%	22 881	97%	107 386

loc:	number of lines of code
br:	number of branches computed statically (after unrolling decisions with multiple conditions in equivalent cascade of single condition decisions)
tc:	number of generated test cases
cbr:	number of covered branches
ibr:	number of identified infeasible branches
cov$_1$:	branch coverage measured using the GNU `gcov` utility [as percentage]
it$_1$:	number of iterations to achieve cbr
cov$_2$:	cbr / (br - ibr) [as percentage]
it$_2$:	number of iterations to achieve both cbr and ibr

Table 6.2. Results of ARC-B execution on 16 industrial subject programs.

to achieve high branch coverage rates and at the same time to inspect manually the identified infeasible branches. Being able to manually inspect the identified infeasible branches helped us mitigating one of the major treats to the validity of our evaluation campaign, the possible incorrectness of the tool implementation.

The programs `linsearch` and `binsearch` implement the linear and binary search of an integer value in an array, respectively (the code is reported in Appendix B.1). The Program `tcas` implements a component of an aircraft traffic control and collision avoidance system, as available from the Software-artifact Infrastructure Repository (SIR) [DER05]. The programs `week0..7` and `week_` use the function `calc_week` from the MySQL database management system in different specialized ways (the code is reported in Appendix B.2). Function `calc_week` takes two parameters, `l_time` (a date) and `week_behavior`, and returns the corresponding week of the year (an integer value between 0 and 53). The parameter `week_behavior` is interpreted as a sequence of bits that indicate the day when the week starts (either Sunday or Monday), the baseline to count weeks (either 0 or 1), and the reference standard for the date representation (ISO standard 8601:1988 or not). In the context of a specific application, `calc_week` is typically used by passing a fixed constant value of `week_behavior` to all calls. This may cause some of the branches of `week_behavior` to be infeasible within a specific application. The programs `week0..7` pass the values 0 to 7 as `week_behavior` to the function `calc_week`, while the program `week_` uses a symbolic value.

The last four subjects in the table correspond to four device drivers for the Windows NT kernel: `cdaudio`, `diskperf`, `floppy` and `kbfiltr`. These programs have a very complex control flow that is reflected in a very large number of paths. In a recent work evaluating an interpolant-based Automated Test Data Generation (ATDG) tool based on CREST, Jaffar et al. used this same programs as a benchmark [JMN13]. This allows us to compare the ARC-B results with the state-of-the-research in symbolic test case generation despite the lack of access to the tools.

For each of the subject programs, column *loc* reports the size expressed in lines of code as counted by the GNU utility `wc`, and column *br* reports the number of branches as counted by ARC-B. ARC-B counts the branches in a program statically, right after the CIL pre-compilation that unrolls decisions with multiple conditions as a cascade of single condition decisions, and eliminates dead code based on constant propagation. We can observe that this fact determines slightly different counts of static branches across the different specializations of the program `calc_week`.

We launched ARC-B on all the subject programs starting from an initially empty test suite. Table 6.2 reports the number of test cases that ARC-B generated for each program (column *tc*), the number of covered branches (column *cbr*), the number of branches that ARC-B identified as infeasible (column *ibr*). Column cov_1 shows the branch coverage rated computed using the GNU `gcov` utility that ignores the ARC ability of detecting infeasible branches. Column cov_2 rescales the coverage score by removing infeasible branches from the coverage domain using the formula $\mathrm{cbr}/(\mathrm{br} - \mathrm{ibr})$. Columns it_1 and it_2 report how many iterations were required to reach the cov_1 and cov_2 scores respectively.

In this experiments, ARC-B has generated a total of 612 test cases that cover 1.456 out of 1.696 branches, and identified 195 infeasible branches, failing only on 45 branches that ARC-B could neither cover nor identify as infeasible. The total branch coverage adjusted with respect to the branches identified as infeasible is over 97%. Each run completed within 60 minutes on a common laptop.

ARC-B produced test suites of manageable size that cover most feasible branches (100% in many cases and 92% in the worst cases). The table shows clearly that the value of the branch coverage that ARC-B computes after identifying and eliminating infeasible elements (column cov_2) is more accurate than the value that ARC-B computes referring to the statically identified branches (column cov_1), although computing the former value requires more iterations (column it_1) than computing the latter value (column it_2) in most cases. This can be observed for all programs that contain infeasible elements. The improvement reaches 22% for `week4` where the coverage grows from 74% to 96%. On average the improvement is around 10%.

To evaluate the effectiveness of ARC-B we compared the coverage results we obtained with the ones produced by CREST using three different strategies: plain random testing (Rand), depth-first concolic execution (CREST-dfs) and an heuristic strategy that aims to maximize branch coverage (CREST-cfg) [BS08].

	Branch coverage (%)					
subject	**Rand**	**CREST-dfs**	**CREST-cfg**	**KLEE**	**ARC-B (cov$_1$)**	**ARC-B (cov$_2$)**
linsearch	100%	37%	100%	62%	100%	100%
binsearch	100%	83%	100%	75%	100%	100%
tcas	4%	93%	93%	93%	**95%**	100%
week0	79%	79%	79%	91%	91%	96%
week1	79%	79%	79%	91%	91%	96%
week2	80%	82%	82%	93%	93%	100%
week3	80%	82%	82%	93%	93%	100%
week4	53%	53%	53%	74%	74%	96%
week5	76%	76%	76%	91%	91%	96%
week6	79%	77%	79%	93%	93%	100%
week7	80%	80%	80%	93%	93%	100%
week_	67%	67%	67%	96%	96%	100%
cdaudio	2%	78%	77%	79%	79%	99%
diskperf	3%	75%	85%	76%	86%	92%
floppy	2%	86%	86%	87%	88%	94%
kbfiltr	38%	84%	84%	85%	84%	92%

Table 6.3. Comparison of branch coverage scores of ARC-B and CREST

Table 6.3 compares the branch coverage obtained with Rand, CREST-dfs, CREST-cfg and KLEE on the 16 subject programs of Table 6.2 with the values obtained with ARC-B. To make the results comparable, we ran all tools on exactly the same code, after the logical operators in the program conditionals were removed via the code transformation done by CIL. We terminated the executions after 60 minutes for each program.

We can observe that in no case CREST or KLEE generate test suites that cover more branches than the ones from ARC-B. In three cases (tcas, diskperf and floppy) ARC-B covers more branches then the alternative approaches proving that the bidirectional search it implements can precisely direct symbolic execution towards uncovered targets (cov_1). Most importantly we can observe that ARC-B is the only approach that systematically reaches branch coverage scores exceeding 90% (cov_2). As we discussed in chapter 1, test suites with coverage values in this range have been shown to possess a high failure detection ability. These results allow us to answer positively to our research question Q2: the ARC approach can be useful in generating test suites that approximate full branch coverage for software of industrial relevance.

6.2.1 Threats to Validity

The main threat to the internal validity of our experiments is the potential incorrectness of the analysis computed by ARC-B on the subject programs. We contrasted this threat in several ways. As we already discussed the first strategy was to analyze programs of increasing size and complexity.

For the synthetically generated programs and the 12 smaller industrial ones, we manually produced the correct branch reachability data. This allowed us to systematically test (and fix) the ARC-B prototype obtaining a robust core implementation. The extension of the core system toward supporting more advanced program constructs like heap memory access and procedures was done using a test first approach.

The correctness of the analysis of larger software has been confirmed indirectly by checking the ARC-B results using third-party testing tools: On one side, we used the GNU `gcov` tool to re-run the generated test suites and confirm that all the branches marked as covered by ARC-B are indeed traversed by the corresponding test cases. On the other side we crosschecked that none of the branches covered by other test generation tools was identified as infeasible by ARC-B. Moreover we confirmed manually the infeasibility of a sample of the branches from the device driver test subjects.

One of the key contributions of KLEE is its ability to *close* a program by mocking the behavior of UNIX system calls. This allowed KLEE for example to generate test cases for `coreutils`, the set of GNU utility programs. ARC-B does not currently include such a support. Despite the ability of ARC-B to handle a certain level of imprecision both in the SE and Weakest Precondition (WP) analysis components, this might constitute a major limitation to the successful application of the ARC-B prototype tool to system software.

Considering now the external validity of our evaluation, we need to discuss two aspects: the industrial applicability of the ARC approach and the comparison of its effectiveness with the state of the art. We strongly believe that the ability of ARC to inexpensively obtain a very high branch coverage score for the four NT device drivers does prove its usefulness in industrial settings. The actual benefit of the approach however, can be evaluated only by performing studies that consider automated testing as a component of a more general development process, and evaluate its impact on the quality and the cost of the final product and its maintenance. Such goal is outside the scope of this dissertation.

Our evaluation focused on the comparison of the effectiveness of ARC with respect to other symbolic testing techniques. The publicly available versions of

CREST and KLEE however, do not include many of the algorithmic improvement proposed in the literature in the recent years. The decision to use as a case study the four NT device drivers mitigates the impact of this limitation in the experimentation. From the analysis of the data presented in the paper [JMN13], we can conclude that ARC-B would obtain a higher branch coverage on the same experimental subjects. The authors in fact report that they observed that their approach can achieves the same branch coverage as vanilla CREST despite the higher path coverage. This allows us to assert that the performance of ARC-B in terms of branch coverage exceeds the previous state-of-the-research.

6.3 Failure Detection in Safety-Critical Software

This section provides a description of the experiments we conducted with the goal of evaluating the failure detection capabilities of ARC-B and thus answering the research question Q3. To this goal we set up an experiment that compares ARC-B against a sample of industrial software systems, with characteristics that are challenging SE based ATDG tools, including nonlinear and floating point arithmetics. The test subjects have been provided as case study by industrial partners in the context of the European FP7 project PINCETTE and include four incremental versions of a real-time software component that controls a robot responsible for the maintenance of the ITER nuclear fusion plant.

6.3.1 The Experiment Setup

In the following we describe the experiment subject programs and provide the core domain knowledge needed to understand the results of our testing activity.

ITER is part of a series of experimental reactors that are meant to investigate the feasibility of using nuclear fusion as a practical source of energy. Due to very specialized requirements, the maintenance operations of the ITER reactor requires developing and testing several new technologies related to software, mechanics, electric and control engineering. Many of these technologies are under investigation at the Divertor Test Platform (DTP2) at VTT Technical Research Centre of Finland. DTP2 embeds a real-time and safety critical control system for remotely operated tools and manipulation devices to handle the reactor components for maintenance.

The control system is implemented using the C, LabVIEW and IEC 61131 programming languages. The software component chosen for this study is part of the motion trajectory control system of the manipulation devices and is implemented in C. It provides an interface between the operator and the manipulator. The operator inputs the target position of the manipulator, along with the maximum velocity, initial velocity, maximum acceleration and maximum deceleration, as physical constraints on the generated trajectory. As a result, the software plans the movement of the manipulator, interpolating a trajectory between two given points in n-dimensional space, where n is the number of physical joints in the manipulator.

The output results are in the form of smooth motions, so that the manipulator joints accelerate, move and decelerate within the physical bounds until the target position is reached. This avoids excessive mechanical stress on the structure of the manipulator, ensuring its integrity and safety. It also keeps the desired output forces of the joints actuators in check. The correctness of such software plays a key role in the reliability of the control system of the ITER maintenance equipment.

A further requirement posed on the produced trajectories is that all the joints should start and finish their motion at the same time. The software ensures that all joints finish their motion at the same time by slowing down acceleration and velocities for certain joints. The component is designed to be compiled as a Dynamic Link Library (DLL) to work with Matlab or LabVIEW.

Our experiment considers four incremental versions of the subject software composed of 6 functions each. The subjects code size ranges between 250 and 1,000 lines of code and between 36 and 74 branches.

I) Baseline version

The baseline version is the main working implementation of the software, and can be compiled to run in the LabVIEW real-time environment. This version was used to test the motion characteristics of a hydraulic manipulator.

II) Platform change version

The second version considered in the study is a platform change of the baseline version that provides the same functionality, but is designed to compile as a DLL to work in the Matlab Simulink environment. It was implemented to plan and simulate the motions in a virtual environment before executing them on the real manipulator, aiming to enhance the safety of operations.

III) Fixed version

The third version considered in the study is a bug fix of the second one. In fact, the second version contains a particular bug causing the manipulator to violate the maximum velocity and acceleration limits. This bug remained in the software for several years before being detected and fixed in this version.

IV) Rewrite version

The fourth version considered in the study is a recently proposed alternative implementation that has not been tested in a real environment yet. It was developed from scratch to rectify some unwanted behaviors in the previous implementations.

The experiment subject software takes as input only floating point variables. The ARC-B test generator however, does not allow floating point variables as program inputs and therefore could not be used out of the box. The research for workarounds that could overcome the ARC-B inability to handle floating point inputs natively lead to the decision to simulate floating point arithmetics using suitably interpreted integer values.

This was achieved by linking the subject programs with a library that provides a simulation of the floating point IEEE 754 semantics over integer-typed values. In our experiments we used the SoftFloat library[1]. This solutions comes at the cost of increasing substantially the complexity of the subject programs which is quantifiable in a factor-10 increase in the number of branches.

Another major obstacle in the experiments was the absence of assertions in the subject programs. Interviews with the developers confirmed that writing assertions is uncommon in their software process.

None of the generated test cases resulted in runtime exceptions or program crash even though deeper analysis of the test results revealed that runtime problems were actually happening, such as floating point underflows and divisions by zero. The standard semantics of floating point operations in fact, prescribes that these special cases should be treated by returning special values, such as, NaN (not a number) or Inf (infinity). Analyzing the test suite runs, we found out that special values were being silently propagated by the subject programs.

For this reason we had to rely on manual oracles for evaluating the ability of the generated test suite to expose failures. This was achieved with the help of domain experts that confirmed our evaluations of the outputs of the subject programs when executed against the generated test inputs.

Having to rely on manual oracles, the ability to generate test suites of manageable size is of crucial importance. ARC-B is a goal oriented approach and for this reason it produces test suites that are generally smaller than the ones from traditional path-oriented SE tools. We could further reduce the test suite size by instructing the test generator to consider only the test cases that increase branch coverage.

6.3.2 Results

We ran the test generator against the subject programs up to saturation that we defined as experiencing no coverage increase for an arbitrary budget of 10,000 test generation attempts, obtaining 35 distinct test cases. We computed the coverage indicators with the GNU `gcov` utility and collected the test execution outputs. We manually inspected the test outcomes by looking into the plotted trajectories of the manipulator joints generated by the subject programs with the support of VTT experts for the analysis of the plots.

This entailed searching the 35 test outputs for unexpected 0, NaN or Inf values, and visually inspecting the generated plots for unexpected or inconsistent shapes

[1]`http://www.jhauser.us/arithmetic/SoftFloat.html`

across the subject programs. All test suites and problem reports from our testing activity have been submitted to developers at VTT to collect their feedback on the relevance of the test cases generated and the correctness of our observations.

We tabulated the failure data and clustered it obtaining 7 distinct failures summarized in Table 6.4. The table includes the known failure F4 produced by the test subject *I) Platform change*, as well as six other significant and previously unknown problems.

FailureID	**Description**	**SW Version**
F1	Floating point imprecision with small input values: in the presence of very small input values the program computes bad output values, e.g., unexpected 0.0 or Inf values	I) Baseline
F2	No robustness with all zero accelerations: if both the values of the maximum acceleration and maximum deceleration are set to zero, the program computes bad output values, e.g., unexpected 0.0 or Inf values	I) Baseline
F3	No robustness with negative accelerations: if either the value of the maximum acceleration or maximum deceleration is a negative number, the program computes bad output values, e.g., unexpected 0.0 or Inf values	I) Baseline
F4	Wrong peak velocity in presence of quiet joints: if there are quiet joints (same origin and destination positions), the program will issue movements at up to double or triple the maximum velocity	II) Platform change
F5	Quiet joints that move: if there are quiet joints other then the first one, the program will cause them to move	II) Platform change
F6	Slowness due to single instant peak velocity: the program issues a smooth progressive increase in acceleration up to peak velocity and a smooth progressive deceleration from then on; This results in (unwanted) slower movements than when applying the maximum acceleration and deceleration at once	IV) Rewrite
F7	Unaccounted maximum deceleration: the program refers to the value of maximum acceleration to compute both acceleration and deceleration movements, possibly exceeding the physical limits of the device when the maximum deceleration is lower than the maximum acceleration	IV) Rewrite

Table 6.4. Failures detected by automatically generated test cases

The test suites generated for the test subject *I) Baseline*, that is, the reference

LabVIEW version of the component under test, exposed failures F1, F2 and F3. The program fails when handling very small input values (failure F1), and particular combinations of the input parameters including zeros (failure F2) and negative values (failure F3) as the maximum acceleration/deceleration of the joints. The three failures manifest themselves as unexpected 0 and Inf values in the output.

Debugging revealed that failure F1 is due to floating point underflows in a multiplication, while failures F2 and F3 are caused by divisions by zero. We learned from the VTT experts that currently the program is never used with negative inputs. Such problems call for strengthening the input preconditions.

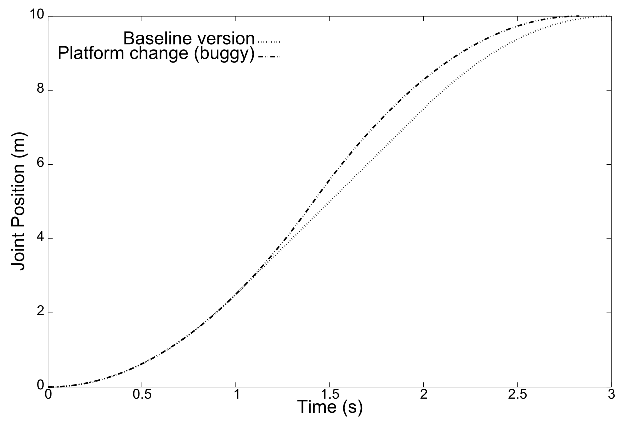


Figure 6.1. Movement of joint 2 when executing a test case

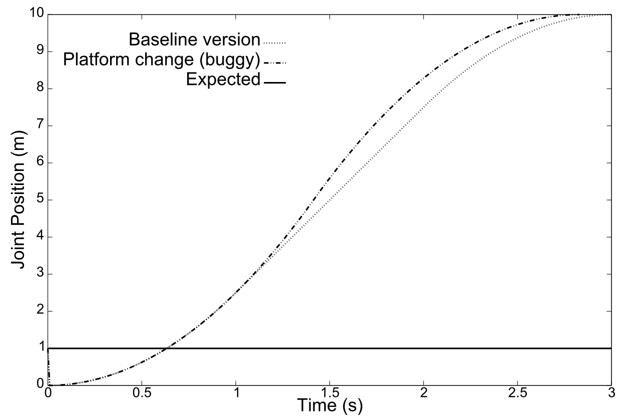


Figure 6.2. Movement of joint 3 when executing a test case

The test suites generated for the test subject *II) Platform change*, exposed the failures F4 and F5. Figure 6.1 plots data from a test case in which the input values for the origin and destination positions of joint 1 are equal. In this case, Joint 1 is expected to be a *quiet* joint, a joint that does not move. The plot shows the movements of joint 2 as issued by the subject *I) Baseline* and *II) Platform change* respectively. It can be checked visually how the test subject II issues a higher velocity than the Baseline due to failure F4.

At the code level the fault consists of a sequence of assignments that may double or triple the value of maximum velocity in the presence of quiet joints. The equality constraints that trigger the execution of these assignments are the typical case in which directed testing based on Dynamic Symbolic Execution (DSE) outperforms random testing; while the equality constraints are easy to solve, the probability of randomly generating equal values is infinitesimal.

The same test suite uncovered another unknown failure in the programs due to a division by zero that produces a NaN value in the trajectory data of quiet joints (failure F5). The NaN value interferes with the conditional control structures so that the program fails to update the position of the joint according to the prescribed trajectory. The observed outcome is that a supposedly quiet joint does instead move, replicating the movement of the joint that precedes it in the manipulator. Figure 6.2 illustrates this behavior with reference to a test case in which joint 3 is specified as a quiet joint. This bug has been confirmed and classified as very severe by VTT experts.

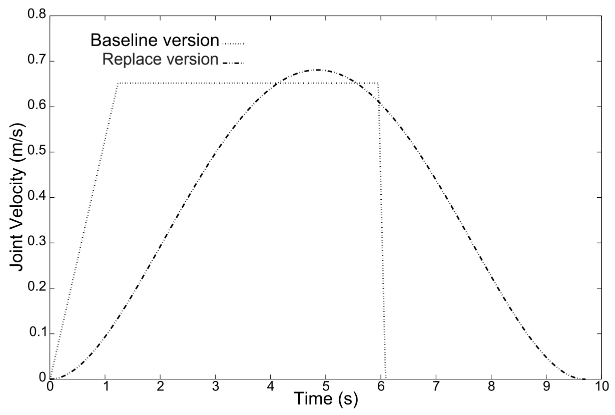


Figure 6.3. Velocity of joint 2 when executing a test case

Running the test suites generated for the test subject *IV) Rewrite*, that is, the recently proposed re-implementation of the functionality of the baseline version, we observed failures F6 and F7. While overall the test cases highlighted the expected change of behavior of version *IV) Rewrite* with respect to the baseline program, i.e. smoother accelerations of the movements, they also revealed two new problems. First, the subject *IV) Rewrite* computes accelerations that produce an instantaneous peak velocity followed by joint movements that are slower then necessary (failure F6). Second, the test subject does not take into account the maximum deceleration input value when it is different from the maximum acceleration value (failure

F7). This may entail important practical issues with the physical limitations of the manipulator. These problems can easily be spotted in Figure 6.3.

If we analyze the code coverage of the generated test suites we obtain the following results: ARC-B produced 20 test cases for the test subject *I) Baseline* that cover 86% of the branches. The test suite for subject *II) Platform change* augmented the test suite of the baseline with 12 additional test cases, resulting in a branch coverage of 88%. For the subject *III) Fixed*, the test generator did not produce any additional test case over the test suite of the previous version, the branch coverage reaches 83%. Finally ARC-B produced 3 additional test cases for the subject *IV) Rewrite* over the test suite of the baseline, obtaining a branch coverage of 86%.

The branch coverage rates we obtained on the four test subjects is consistently higher then 80%. While such a coverage rate is certainly high, it does not reach a level that strongly correlates with high failure detection abilities (see 1). We believe that the impossibility of reaching coverage levels approximating 100% is due to the difficulty of detecting infeasible elements in this type of software with the current version of ARC-B. In fact, the large amount of nonlinear arithmetic operations in the code limited the precision of the refinement predicates in the ARC-B Generalized Control Flow Graph (GCFG) models.

Ultimately we believe that the results we obtained by applying ARC-B to this set of test subjects, namely a test suite that exposed 7 relevant failures and covered of more then 80% of the branches, allow us to answer positively to the research question Q3: the ARC approach is able to expose previously unknown failures in industrially relevant software.

6.3.3 Threats to Validity

In this section we discuss the most important factors that may affect the internal and external validity of our study, and outline the strategies we used to mitigate their impact.

In the experiments, we selected test cases that produce increments of branch coverage up to saturation. Different selection or halting criteria might have induced different test cases and then different results. In this case the bias we introduced is pessimistic, we might in fact have halted the generation process too soon, or dropped test cases that would expose further failures without increasing branch coverage. We can, therefore, assume that this threat has low impact on the results.

Handling floating point calculations using the SoftFloat simulation library, leads to analyzing a transformation of the original subject programs, with an increase of the total number of branches up to a factor of 10. This may threaten the comparability of the results with respect to the coverage of the original code. We addressed this issue by using the transformed code only to generate the test suites, while we collected the failures and the coverage data on the original programs, thus producing precise results.

The features of the selected experimental subjects are representative of several other real-time control systems that are being developed at the VTT research centre, but we are aware that the results cannot be directly generalized. More specifically, it is debatable whether the results obtained generalize across software of different size, written in different programming languages and for different application domains.

Chapter 7

Conclusions

This thesis addresses the problem of automatically generating test cases that achieve very high structural coverage.

As several empirical studies have showed in the past, the fault detection power of test suites grows more and more quickly for higher coverage rates, becoming very high only for test suites approximating 100%. High coverage test suites are nonetheless very rare in practice because of the cost connected with their generation and maintenance. Moreover the presence of infeasible coverage elements might undermine the informativeness of coverage scores, in particular for fine grained coverage criteria.

To this goal we designed the Abstraction Refinement and Coarsening (ARC) analysis technique combining in a fully automatic approach Dynamic Symbolic Execution (DSE) based test data generation with Weakest Precondition (WP) based reachability analysis. ARC fosters the synergy between the different analysis components using a program abstraction called Generalized Control Flow Graph (GCFG) that allows sharing partial reachability results. The scalability of the ARC approach is greatly improved by the coarsening algorithm. Coarsening traces the coverage targets back to the model refinements that have been generated to analyze their reachability. This information allows to maintain the GCFG size under control by undoing the refinements that are not needed anymore, the ones that can be only traced back to covered or infeasible code elements.

We implemented the ARC analysis technique in the ARC-B prototype tool that targets branch coverage of C programs. The experiment we conducted showed that ARC-B can generate high-coverage test suites and scale to industrially relevant software. Furthermore the test suites generated with ARC-B have exposed several previously unknown failures in a safety-critical industrial software system.

7.1 Contributions

The main contribution of this thesis is to provide an approach for Automated Test Data Generation (ATDG) that can achieve high-coverage structural testing for industrially relevant software, combining dynamic and static analyses to direct test case generation towards code elements that are not covered yet and at the same time detect the infeasible ones. The analysis interplay is supported by a scalable program abstraction that represents partial reachability results. In particular the thesis makes the following specific contributions:

- **Proposing a framework for high-coverage structural testing leveraging the interplay of different program analysis techniques.** ARC combines a test case generation approach with reachability analysis, aiming to exploit the possible interplays between the two. The interaction between the different components of ARC boils down to the sharing of reachability analysis information. This is possible thanks to a scalable program abstraction that coordinates the analyses interplay.

- **Evaluating existing techniques to tackle the problem of testing code while at the same time identifying its infeasible portion.** ARC combines a specific set of program analysis techniques that are particularly suitable for combination. This is due to the fact that they can benefit each other from sharing even partial reachability information. In particular ARC combines DSE based test case generation with WP based abstraction refinement.

- **Designing a scalable abstraction management technique able to support both test generation and reachability analysis.** In ARC the different analysis components are coordinated via the GCFG model, a program abstraction that encode the progress of both the dynamic and the static program state space exploration. The GCFG size is controlled by coarsening, an algorithm that minimizes the model redundancy and improves significantly its scalability.

- **Evaluating empirically the effectiveness of the proposed approach.** The ARC approach was implemented in a prototype tool named ARC-B that targets branch coverage. ARC-B has been used to generate test suites that achieve high coverage on several test subjects. The experiments validate the effectiveness of the ARC approach in generating test suites approximating 100% coverage. Moreover ARC-B was able to expose several failures in a software component of an industrial safety-critical system.

7.2 Future Directions

The work presented in this thesis opens several research problems. ARC constitutes a first step in the direction of generating high coverage test suites by combining test case generation with reachability analysis. Many alternative solutions can be imagined on the same line and should be investigated and validated. More specifically, we believe that the ARC algorithm could benefit from the following developments:

- **Parallelization.** At every single moment in the ARC analysis, a large subset of the transitions in the GCFG model is independently analyzable. They constitute the *coverage domain frontier*. We believe that this feature leads naturally to a parallel implementation of the ARC algorithm that could boost greatly the analysis performance. Key to the parallel version of ARC would be the parallel implementation of the GCFG model.

- **New frontier selection strategies.** In our definition and implementation of the frontier selection strategy that decides which of the frontier transitions should be analyzed next, we took special care in allowing the maximum level of nondeterminism. ARC implements a bidirectional search in the symbolic execution tree of the program under analysis but the specific forward and backward search strategies are not prescribed. Different selection strategies would produce different instances of the ARC analysis that might exhibit specific strengths.

- **New refinement predicate generation algorithms.** In ARC we use an alias-aware, WP-based, abstraction refinement algorithm. It would be interesting to study alternative options like the Craig interpolants or forms of dynamically informed WP calculus.

- **Further empirical validation.** The ARC-B tool was successfully used on several industrial case studies but nonetheless it still has the typical limitations of a research prototype. Further empirical studies would not only allow to better evaluate the ARC industrial applicability and scalability but would also allow to identify the critical areas that need improvement. In the analysis of our experiments with the VTT software for example we highlighted limitations with respect to nonlinear and floating point arithmetics that could be investigated further. The extension of this study to other industrial domains would allow to further generalize the current results.

Appendices

Appendix A

The ARC-B Prototype Code

In this Appendix we include some of the most significant code fragments that implement the ARC-B prototype tool.

A.1 The ARC-B Iterative Loop.

```
void ARC_B_Search::Run() {
    //Execution of the initial test suite
    LaunchProgram(tests);
    //Iterate until the targets are over
    for(; !target_branches.empty(); ++num_iters){
        //select a frontier transition (a pair of vertices)
        pair<int, int> frontier=selectFrontier();
        //extract the refinemnt context
        context_t ref_ctx=vertex2context[frontier.second];
        //find compatible test cases
        auto test=get_compatible_tests(ref_ctx, frontier.first);
        //try to traverse the frontier
        pair<bool, Predicate> success=traverseFrontier(test.input,
                                      frontier, test.steps);
        //refine if unsuccessful
        if(!success.first) {
            refineModel(frontier, success.second);
        }
    }
}
```

Figure A.1. Function Run that implements the main iterative loop of ARC-B.

```
//returns true and an empty predicate if the frontier is traversed
//returns false and an appropriate refinment predicate otherwise
pair<bool, Predicate> traverseFrontier(input_t input,
                           frontier_t frontier, size_t steps){
    //instantiate GDB and replay the program up to the frontier
    GDB gdb(program_, input);
    gdb.ReplayProgram(steps);
    //create SMT file and solve
    auto solution=Solve(makeSMT(frontier, gdb));
    // Run with new input or compute refinement predicate.
    if(solution.found){
        TrackTestOnModel(solution.input);
        return make_pair(true, Predicate());
    } else {
        ref_predicate=WP(frontier.first, frontier.second, gdb);
        return make_pair(false, solution.ref_predicate);
    }
}
```

Figure A.2. Function traverseFrontier that implements the coverage frontier analysis of ARC-B.

A.2 Functional Style Memory Access.

```
//Functions for accessing arrays in a functional style
struct _arrayS;
typedef __CREST_VALUE(*ArrayGet)(size_t type_size,
                                 struct _arrayS*, __CREST_ADDR);
struct _arrayS {
    __CREST_ADDR modifiedItem;
    __CREST_VALUE value;
    __CREST_ADDR array;
    struct _arrayS* aWrapper;
    ArrayGet getter;
};

#define get(type_size, _arrayS, i) (_arrayS->getter)(type_size,
                                                     _arrayS, i)
void _arrayS_free(struct _arrayS* a);
__CREST_VALUE _arrayS_get4Identity(struct _arrayS* a, __CREST_ADDR i);
__CREST_VALUE _arrayS_get4Store(struct _arrayS* a, __CREST_ADDR i);
```

```
struct _arrayS* _identity(__CREST_ADDR a);
#define identity() _identity((__CREST_ADDR)0)
struct _arrayS* _store(struct _arrayS* a, __CREST_ADDR i,
                       __CREST_VALUE val);
#define store(a, i, val) _store(a, (__CREST_ADDR)i,
                                (__CREST_VALUE)val)
__CREST_VALUE _select(size_t type_size, struct _arrayS* a,
                      __CREST_ADDR i);
#define select(type_size, a, i) _select(type_size, a,
                                        (__CREST_ADDR)i)

//implementation
void _arrayS_free(struct _arrayS* a){
    if(a->aWrapper) _arrayS_free(a->aWrapper);
    free(a);
}

__CREST_VALUE _arrayS_get4Identity(size_t type_size,
                                   struct _arrayS* a, __CREST_ADDR k){
    return getValFromAddr(a->array, k, type_size);
}

__CREST_VALUE _arrayS_get4Store(size_t type_size,
                                struct _arrayS* a, __CREST_ADDR k){
    return k == a->modifiedItem? a->value :
                                 get(type_size, a->aWrapper, k);
}

struct _arrayS* _identity(__CREST_ADDR a){
    struct _arrayS* arr=(struct _arrayS*)malloc(sizeof(struct _arrayS));
    arr->aWrapper=0;
    arr->array=a;
    arr->getter=_arrayS_get4Identity;
    return arr;
}

struct _arrayS* _store(struct _arrayS* a, __CREST_ADDR i,
                       __CREST_VALUE val){
    struct _arrayS* arr=(struct _arrayS*)malloc(sizeof(struct _arrayS));
    arr->modifiedItem=i;
    arr->value=val;
    arr->aWrapper=a;
    arr->getter=_arrayS_get4Store;
    return arr;
}

__CREST_VALUE _select(size_t type_size, struct _arrayS* a,
                      __CREST_ADDR i){
    __CREST_VALUE ret=get(type_size, a, i);
    _arrayS_free(a);
    return ret;
}
```

Appendix B

Experiments

B.1 The Linear and Binary Search Functions

```
int linsearch(void) {
    int n;
    ARC_B_Int(n);
    int v;
    ARC_B_Int(v);
    int a[n];
    for(int i=0; i<n; i++){
            ARC_B_Int(a[i]);
            int var=a[i];
            fprintf(stderr,"val %d\n",var);
    }

    //search
    if(v==0)
        return 0;
    for (int i=0; i<n; i++){
            int var=a[i];
            if (a[i] == v)
                return i;
    }
    return n;
}

int binsearch(void) {
    //INPUTS
    int key; ARC_B_Int(key);
    int dictSize; ARC_B_Int(dictSize);
    int dictKeys[dictSize];
    printf("key=%d\n",key);
    printf("dictSize=%d\n",dictSize);

    for(int i=0; i<dictSize; i++){
            ARC_B_Int(dictKeys[i]);
            printf("dictKeys[%d]=%d\n",i,dictKeys[i]);
            //enforce precondition (ordering)
            if(i!=0 && dictKeys[i]<dictKeys[i-1]){
                printf("non sorted\n");
                return 1;
            }
    }

    int low=0;
    int high = dictSize - 1;
    int mid;

    while (high >=low) {
            mid = (high + low) / 2;
            if (key > dictKeys[mid]) {
                // dictKeys[mid] too small; look higher
                low=mid+1;
            } else if (key < dictKeys[mid]) {
                // dictKeys[mid] too large; look lower
                high=mid-1;
            } else {//67
                printf("key %d found\n", dictKeys[mid]);
                    return dictKeys[mid];
            }
    }
    printf("%d not found\n", key);
    return 0;
}
```

B.2 Week Number Computation Function from MySQL

```
/* Flags for calc_week() function.  */
#define WEEK_MONDAY_FIRST    1
#define WEEK_YEAR            2
#define WEEK_FIRST_WEEKDAY   4

typedef struct
  TIME{uint year; uint month; uint day;} TIME;

/* Calc days since year 0 (from 1615) */
long calc_daynr(uint year, uint month, uint day);

/* Calc weekday from daynr: 0 for mon, 1 for tue... */
int calc_weekday(long daynr,
                 bool sunday_first_day_of_week);

/* Calc days in a year. works with 0 <= year <= 99 */
uint calc_days_in_year(uint year);

/* Meaning of the bits in week_behaviour:
   WEEK_MONDAY_FIRST (0): set ==> Mon, else Sun
   WEEK_YEAR (1): set ==> Week in range 1-53, else 0-53
   WEEK_FIRST_WEEKDAY (2): not set ==> ISO 8601:1988
*/
uint calc_week(TIME *l_time,
               uint week_behaviour, uint *year){
  uint days;
  ulong daynr =
    calc_daynr(l_time->year, l_time->month, l_time->day);
  ulong first_daynr = calc_daynr(l_time->year, 1, 1);
  bool monday_first =
    week_behaviour & WEEK_MONDAY_FIRST;
  bool week_year = week_behaviour & WEEK_YEAR;
  bool first_weekday =
    week_behaviour & WEEK_FIRST_WEEKDAY;

  uint weekday=calc_weekday(first_daynr, !monday_first);
  *year=l_time->year;

  if (l_time->month == 1 && l_time->day <= 7-weekday){
    if (!week_year && (first_weekday && weekday != 0 ||
                       !first_weekday && weekday >= 4))
        return 0;
    week_year= 1;
    (*year)--;
    first_daynr-= (days=calc_days_in_year(*year));
    weekday= (weekday + 53*7- days) % 7;
  }

  if ((first_weekday && weekday != 0) ||
      (!first_weekday && weekday >= 4))
    days= daynr - (first_daynr+ (7-weekday));
  else days= daynr - (first_daynr - weekday);

  if (week_year && days >= 52*7){
    weekday= (weekday + calc_days_in_year(*year)) % 7;
    if (!first_weekday && weekday < 4 ||
        first_weekday && weekday == 0){
      (*year)++;
      return 1;
    }
  }
  return days/7+1;
}
```

Bibliography

[AB11] Andrea Arcuri and Lionel Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 265–275, New York, NY, USA, 2011. ACM.

[ABC$^+$13] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, August 2013.

[AH11] Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 3–12, Washington, DC, USA, 2011. IEEE Computer Society.

[AKD$^+$08] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 261–272, New York, NY, USA, 2008. ACM.

[AOH07] Saswat Anand, Alessandro Orso, and Mary Jean Harrold. Type-dependence analysis and program transformation for symbolic execution. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'07, pages 117–133, Berlin, Heidelberg, 2007. Springer-Verlag.

[APV06] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. Symbolic execution with abstract subsumption checking. In *Proceedings of the 13th International Conference on Model Checking Software*, SPIN'06, pages 163–181, Berlin, Heidelberg, 2006. Springer-Verlag.

[APV07] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. Jpf-Se: A symbolic execution extension to Java pathfinder. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2007)*, pages 134–138, Braga, Portugal, March 2007.

[ATF09] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Journal of Information and Software Technology*, 51(6):957–976, June 2009.

[Aus98] Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[Bal03] Thomas Ball. Abstraction-guided test generation: A case study. Technical Report MSR-TR-2003-86, Microsoft Research, November 2003.

[Bal04] Thomas Ball. A theory of predicate-complete test coverage and generation. In *Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin / Heidelberg, November 2004.

[Bar00] Clark Barrett. Theory solvers in SMT. `http://www.stanford.edu/class/cs357/lectures/lec9_h.pdf`, 2000. Accessed: 03 March 2014.

[BBDP10] Mauro Baluda, Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Structural coverage of feasible code. In *Proceedings of the Fifth International Workshop on Automation of Software Test (AST 2010)*, AST '10, pages 59–66, New York, NY, USA, 2010. ACM.

[BBDP11] Mauro Baluda, Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Enhancing structural software coverage by incrementally computing branch executability. *Software Quality Journal*, 19(4):725–751, 2011.

[BC07] Renée C. Bryce and Charles J. Colbourn. One-test-at-a-time heuristic search for interaction test suites. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1082–1089, New York, NY, USA, 2007. ACM.

[BCH$^+$04] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 326–335. IEEE Computer Society, 2004.

[BCLKR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM 2004)*, pages 1–20. Springer, 2004.

[BDM+13] Clark Barrett, Morgan Deters, Leonardo Moura, Albert Oliveras, and Aaron Stump. 6 years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277, March 2013.

[BDP13] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 411–421, New York, NY, USA, 2013. ACM.

[BDR+14] Pietro Braione, Giovanni Denaro, Oliviero Riganelli, Mauro Baluda, and Muhammad Ali. Static/dynamic test case generation for ssoftware upgrades via ARC-B and DeltaTest. In Hana Chockler, Daniel Kroening, Leonardo Mariani, and Natasha Sharygina, editors, *Validation of Evolving Software. To Appear*, chapter 12, pages 163–210. Springer, 2014.

[BEL75] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT - a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10:234–245, April 1975.

[Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[BHH+11] Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. Symbolic search-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 53–62, Washington, DC, USA, 2011. IEEE Computer Society.

[BHMR07] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, PLDI '07, pages 300–309, New York, NY, USA, 2007. ACM.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[BJS09] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 463–473, Washington, DC, USA, 2009. IEEE Computer Society.

[BJSS09] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 161–169, Washington, DC, USA, 2009. IEEE Computer Society.

[BMDH90] Jerry R. Burch, Kenneth L. McMillan, David L. Dill, and LJ Hwang. Symbolic model checking: 10 20 states and beyond. *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439, 1990.

[BMMS11] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 12–22, New York, NY, USA, 2011. ACM.

[BNR$^+$10] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, Robert J. Simmons, Sai Deep Tetali, and Aditya V. Thakur. Proofs from tests. *IEEE Transactions on Software Engineering*, 36(4):495–508, July 2010.

[BPS03] André Baresel, Hartmut Pohlheim, and Sadegh Sadeghipour. Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In *Proceedings of the 2003 international conference on Genetic and evolutionary computation: PartII*, GECCO'03, pages 2428–2441, Berlin, Heidelberg, 2003. Springer-Verlag.

[Bra11] Aaron R. Bradley. Sat-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[BRC$^+$12] Jason Belt, Robby, Patrice Chalin, John Hatcliff, and Xianghua Deng. Efficient symbolic execution of value-based data structures for critical systems. In *Proceedings of the 4th International Conference on NASA Formal Methods*, NFM'12, pages 295–309, Berlin, Heidelberg, 2012. Springer-Verlag.

[Bro07] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, April 2007.

[BS08] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 443–446, 2008.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[Bur67] W.H. Burkhardt. Generating test programs from syntax. *Computing*, 2(1):53–73, 1967.

[BUZC11] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of EuroSys 2011*. ACM, 2011.

[BW08] Oliver Bühler and Joachim Wegener. Evolutionary functional testing. *Comput. Oper. Res.*, 35(10):3144–3160, October 2008.

[CAaVP11] Thelma Elita Colanzi, Wesley Klewerton Guez Assunção, Silvia Regina Vergilio, and Aurora Pozo. Integration test of classes and aspects with a multi-evolutionary and coupling-based approach. In *Proceedings of the Third International Conference on Search Based Software Engineering*, SSBSE'11, pages 188–203, Berlin, Heidelberg, 2011. Springer-Verlag.

[CCMY96] F.T. Chan, T.Y. Chen, I.K. Mak, and Y.T. Yu. Proportional sampling strategy: Guidelines for software testing practitioners. *Information and Software Technology*, 38(12):775 – 782, 1996.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, 2008.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, October 1991.

[CG12] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In P. Madhusudan and SanjitA. Seshia, editors, *Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293. Springer Berlin Heidelberg, 2012.

[CGJ$^+$03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGP$^+$06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*, pages 322–335, New York, NY, USA, 2006. ACM.

[CH00] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, pages 265–278. ACM, 2011.

[CKK$^+$12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[Cla76] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, May 1976.

[CLM05] Tsong Yueh Chen, Hing Leung, and Keith Mak. Adaptive random testing. In Michael Maher, editor, *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, volume 3321 of *Lecture Notes in Computer Science*, pages 3156–3157. Springer Berlin / Heidelberg, 2005.

[CLOM08] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: Adaptive random testing for object-oriented software. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 71–80, New York, NY, USA, 2008. ACM.

[CMTS09] Ernie Cohen, MichałMoskal, Stephan Tobies, and Wolfram Schulte. A precise yet efficient memory model for C. *Electronic Notes in Theoretical Computer Science*, 254:85–103, October 2009.

[Coo78] Stephan Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.

[CS05] Christoph Csallner and Yannis Smaragdakis. Check'N'Crash: Combining static checking and testing. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 422–431, 2005.

[CSE96] John Callahan, Francis Schneider, and Steve Easterbrook. Automated software testing using model-checking. In *Proceedings of the 1996 SPIN Workshop (SPIN 1996). Also WVU Technical Report NASA-IVV-96-022.*, 1996.

[CSX08] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–37, April 2008.

[DBG10] Mickaël Delahaye, Bernard Botella, and Arnaud Gotlieb. Explanation-based generalization of infeasible path. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 215–224, Washington, DC, USA, 2010. IEEE Computer Society.

[DER05] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[Dij97] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[DM10] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software*, 37(1):2:1–2:20, January 2010.

[dMB09] Leonardo de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 45–52. IEEE, 2009.

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[DN84] Joe W. Duran and S.C. Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, SE-10(4):438–444, 1984.

[DP08] Dino Distefano and Matthew J. Parkinson. Jstar: Towards practical verification for Java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 213–226, New York, NY, USA, 2008. ACM.

[DPCE$^+$07] Massimiliano Di Penta, Gerardo Canfora, Gianpiero Esposito, Valentina Mazza, and Marcello Bruno. Search-based testing of service level agreements. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1090–1097, New York, NY, USA, 2007. ACM.

[Edv99] Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering*, pages 21–28, 1999.

[EGL09] Bassem Elkarablieh, Patrice Godefroid, and Michael Y. Levin. Precise pointer reasoning for dynamic test generation. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 129–140, New York, NY, USA, 2009. ACM.

[EMS07] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 151–162, New York, NY, USA, 2007. ACM.

[EO11] Ikpeme Erete and Alessandro Orso. Optimizing constraint solving to better support symbolic execution. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 310–315, Washington, DC, USA, 2011. IEEE Computer Society.

[FA12] Gordon Fraser and Andrea Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 121–130, 2012.

[FGW09] Gordon Fraser, Angelo Gargantini, and Franz Wotawa. On the order of test goals in specification-based testing. *The Journal of Logic and Algebraic Programming*, 78(6):472 – 490, 2009.

[FI98] Phyllis G. Frankl and Oleg Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '98/FSE-6, pages 153–162, New York, NY, USA, 1998. ACM.

[FK96] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5:63–86, January 1996.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[Flo67] Robert W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19:19–32, 1967.

[FM10] Carlo A. Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, volume 6300 of *Lecture Notes in Computer Science*, pages 277–300. Springer, August 2010.

[FSM$^+$13] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 291–301, New York, NY, USA, 2013. ACM.

[FW93] Phyllis G. Frankl and Elaine J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, 1993.

[FWA09a] Gordon Fraser, Franz Wotawa, and Paul Ammann. Issues in using model checkers for test case generation. *J. Syst. Softw.*, 82(9):1403–1418, September 2009.

[FWA09b] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: A survey. *Software Testing, Verification and Reliability*, 19(3):215–261, September 2009.

[FZ11] Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Soft-*

ware Testing and Analysis, ISSTA '11, pages 364–374, New York, NY, USA, 2011. ACM.

[FZ12] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, March 2012.

[GDGM01] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, pages 5–, Washington, DC, USA, 2001. IEEE Computer Society.

[GFA13] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *IEEE International Symposium on Software Reliability Engineering*, 2013.

[GFX12] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 156–166, Piscataway, NJ, USA, 2012. IEEE Press.

[GFZ12] Florian Gross, Gordon Fraser, and Andreas Zeller. Search-based system testing: High coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 67–77, New York, NY, USA, 2012. ACM.

[GGZ$^+$13] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 302–313, New York, NY, USA, 2013. ACM.

[GHK$^+$06] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. Synergy: A new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 117–127, New York, NY, USA, 2006. ACM.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN*

2005 Conference on Programming Language Design and Implementation (PLDI 2005), PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2008)*, pages 151–166, 2008.

[GNRT10] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. *SIGPLAN Not.*, 45(1):43–56, 2010.

[God07] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier Science Inc., New York, NY, USA, 1977.

[HFGO94] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[HH85] David Hedley and Michael A. Hennell. The causes and effects of infeasible paths in computer programs. In *Proceedings of the 8th international conference on Software engineering*, ICSE '85, pages 259–266, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[HHZ12] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 38–38, Piscataway, NJ, USA, 2012. USENIX Association.

[HJL11] Mark Harman, Yue Jia, and William B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 212–222, New York, NY, USA, 2011. ACM.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-*

SIGACT symposium on Principles of programming languages, POPL '02, pages 58–70, New York, NY, USA, 2002. ACM.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of ACM*, 12(10):576–580, October 1969.

[Hoa09] C.A.R. Hoare. Viewpoint: Retrospective: An axiomatic basis for computer programming. *Commun. ACM*, 52(10):30–32, October 2009.

[IH14] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the International Conference on Software Engineering*, 2014.

[IX08] K. Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 297–306, Washington, DC, USA, 2008. IEEE Computer Society.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):21:1–21:54, 2009.

[JMN13] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. Boosting concolic testing via interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 48–58, New York, NY, USA, 2013. ACM.

[JMNS12] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. Tracer: A symbolic execution tool for verification. In *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12, pages 758–766, Berlin, Heidelberg, 2012. Springer-Verlag.

[JSS07] Pallavi Joshi, Koushik Sen, and Mark Shlimovich. Predictive testing: Amplifying the effectiveness of software testing. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 561–564, New York, NY, USA, 2007. ACM.

[KGG$^+$09] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, New York, NY, USA, 2009. ACM.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[Kor90] Bogdan Korel. Automated software test data generation. *Software Engineering, IEEE Transactions on*, 16(8):870–879, 1990.

[Kor92] Bodgan Korel. Dynamic method for software test data generation. *Software Testing, Verification and Reliability*, 2(4):203–213, December 1992.

[KPD$^+$13] Fitsum M. Kifetew, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Paolo Tonella. Orthogonal exploration of the search space in evolutionary test case generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 257–267, New York, NY, USA, 2013. ACM.

[KPV03] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 553–568, Berlin, Heidelberg, 2003. Springer-Verlag.

[KZHH05] H.G. Kayacik, A.N. Zincir-Heywood, and M. Heywood. Evolving successful stack overflow attacks for vulnerability testing. In *Computer Security Applications Conference, 21st Annual*, pages 8 pp.–234, 2005.

[LGR11] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. Klover: A symbolic execution and automatic test generation tool for c++ programs. In *CAV*, pages 609–615, 2011.

[LHM08] Kiran Lakhotia, Mark Harman, and Phil McMinn. Handling dynamic data structures in search based testing. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, pages 1759–1766, New York, NY, USA, 2008. ACM.

[LSWL13] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 19–32, New York, NY, USA, 2013. ACM.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, November 1994.

[McC93] John McCarthy. Towards a mathematical science of computation. In *Program Verification*, volume 14 of *Studies in Cognitive Systems*, pages 35–56. Springer Netherlands, 1993.

[McM04] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[McM10] Kenneth L. McMillan. Lazy annotation for program testing and verification. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 104–118. Springer Berlin / Heidelberg, 2010.

[Mey92] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[MF11] Jan Malburg and Gordon Fraser. Combining search-based and constraint-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 436–439, Washington, DC, USA, 2011. IEEE Computer Society.

[MHBT06] Phil McMinn, Mark Harman, David Binkley, and Paolo Tonella. The species per path approach to searchbased test data generation. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 13–24, New York, NY, USA, 2006. ACM.

[MMS01] Christoph Michael, Gary McGraw, and Michael Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, December 2001.

[Mor88] Carroll Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, July 1988.

[MS07] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[MSS12] P. McMinn, M. Shahbaz, and M. Stevenson. Search-based test input generation for string data types using the results of web queries. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 141–150, 2012.

[NMT12] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 100–110, New York, NY, USA, 2012. ACM.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, October 1979.

[NT07] Minh Ngoc Ngo and Hee Beng Kuan Tan. Detecting large number of infeasible paths through recognizing their patterns. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 215–224, New York, NY, USA, 2007. ACM.

[NT08] Minh Ngoc Ngo and Hee Beng Kuan Tan. Heuristics-based infeasible path detection for dynamic test data generation. *Information and Software Technology*, 50(7-8):641–655, 2008.

[NZK12] Razieh Nokhbeh Zaeem and Sarfraz Khurshid. Test input generation using dynamic programming. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 34:1–34:11, New York, NY, USA, 2012. ACM.

[OP97] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.

[Opp80] Derek C. Oppen. Reasoning about recursively defined data structures. *Journal of the ACM (JACM)*, 27(3):403–411, July 1980.

[PB08] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 75–86, New York, NY, USA, 2008. ACM.

[PC13] Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *Proceedings of the 25th international conference on Computer Aided Verification*, CAV'13, pages 53–68, Berlin, Heidelberg, 2013. Springer-Verlag.

[PDEP08] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th*

ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA, 2008. ACM.

[PDM12] Cristian Cadar Paul Dan Marinescu. Make test-zesti: A symbolic execution solution for improving regression testing. In *International Conference on Software Engineering (ICSE 2012)*, 6 2012.

[PDM13] Cristian Cadar Paul Dan Marinescu. Katch: High-coverage testing of software patches. In *European Software Engineering Conference / ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 235–245, 8 2013.

[PM10] M. Papadakis and N. Malevris. A symbolic execution tool based on the elimination of infeasible paths. In *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on*, pages 435–440, 2010.

[PMB⁺08] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[PS81] Amir Pnueli and Micha Sharir. Two approaches to interprocedural data flow analysis. *Program flow analysis: theory and applications*, pages 189–234, 1981.

[PV09] Corina S. Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 11(4):339–353, 2009.

[PW13] Tuan-Hung Pham and Michael W. Whalen. RADA: A tool for reasoning about algebraic data types with abstractions. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 611–614, New York, NY, USA, 2013. ACM.

[PY07] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques.* John Wiley and Sons, April 2007.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[SDE09] Elena Sherman, Matthew B. Dwyer, and Sebastian Elbaum. Saturation-based testing of concurrent programs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 53–62, New York, NY, USA, 2009. ACM.

[SGF09] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. VS3: Smt solvers for program verification. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 702–708, Berlin, Heidelberg, 2009. Springer-Verlag.

[SJP$^+$13] Ting Su, Siyuan Jiang, Geguang Pu, Bin Fang, Jifeng He, Jun Yan, and Jianjun Zhao. Automated coverage-driven test data generation using dynamic symbolic execution. Technical report, 2013.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13)*, pages 263–272, 2005.

[SP10] Matt Staats and Corina S. Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA 2010)*, pages 183–194. ACM, 2010.

[SPPV05] Corina S. Păsăreanu, Radek Pelánek, and Willem Visser. Concrete model checking with abstract matching and refinement. In *Proceeding of the 17th International Conference on Computer Aided Verification (CAV 2005)*, number 3576 in LNCS, pages 52–66. Springer, 2005.

[SZ92] D. Scholefield and H. S. M. Zedan. Weakest precondition semantics for time and concurrency. *Information Processing Letters*, 43(6):301–308, October 1992.

[TCM98] Nigel Tracey, John Clark, and Keith Mander. Automated program flaw finding using simulated annealing. In *Proceedings of the 1998 ACM*

SIGSOFT international symposium on Software testing and analysis, ISSTA '98, pages 73–81, New York, NY, USA, 1998. ACM.

[TdH08] Nikolai Tillmann and Jonathan de Halleux. Pex: White box test generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[Ton04] Paolo Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 119–128, New York, NY, USA, 2004. ACM.

[TSWW06] M. Tlili, H. Sthamer, S. Wappler, and J. Wegener. Improving evolutionary real-time testing by seeding structural test data. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 885–891, 2006.

[VLW$^+$13] Tanja E. Vos, Felix F. Lindlar, Benjamin Wilmes, Andreas Windisch, Arthur I. Baars, Peter M. Kruse, Hamilton Gross, and Joachim Wegener. Evolutionary functional black-box testing in an industrial setting. *Software Quality Control*, 21(2):259–288, June 2013.

[VMGF13] Mattia Vivanti, Andre Mis, Alessandra Gorla, and Gordon Fraser. Search-based data-flow test generation. In *ISSRE'13: Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering*. IEEE Press, November 2013.

[VPK04] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, ISSTA '04, pages 97–107, New York, NY, USA, 2004. ACM.

[WB04] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In Kalyanmoy Deb, editor, *Genetic and Evolutionary Computation – GECCO 2004*, volume 3103 of *Lecture Notes in Computer Science*, pages 1400–1412. Springer Berlin Heidelberg, 2004.

[WBS01] Joachim Wegener, Andre Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841 – 854, 2001.

[Wey90] Elaine J. Weyuker. The cost of data flow testing: An empirical study. *IEEE Trans. Softw. Eng.*, 16(2):121–128, February 1990.

[WJMJ08] Shen Hui Wu, Sridhar Jandhyala, Yashwant K. Malaiya, and Anura P. Jayasumana. Antirandom testing: A distance-based approach. *VLSI Design*, 2008(2):2:1–2:9, January 2008.

[WMO12] Yi Wei, Bertrand Meyer, and Manuel Oriol. Is branch coverage a good measure of testing effectiveness? In Bertrand Meyer and Martin Nordio, editors, *Empirical Software Engineering and Verification*, pages 194–212. Springer-Verlag, Berlin, Heidelberg, 2012.

[WRF$^+$11] Yi Wei, Hannes Roth, Carlo A. Furia, Yu Pei, Alexander Horton, Michael Steindorfer, Martin Nordio, and Bertrand Meyer. Stateful testing: Finding more errors in code and contracts. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 440–443, Washington, DC, USA, 2011. IEEE Computer Society.

[WSKR06] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Timeaware test suite prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 1–12, New York, NY, USA, 2006. ACM.

[XGM08] Ru-Gang Xu, Patrice Godefroid, and Rupak Majumdar. Testing for buffer overflows with length abstraction. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 27–38, New York, NY, USA, 2008. ACM.

[XLXT13] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *Proc. 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*, November 2013.

[XTdHS09] Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, pages 359–368, June-July 2009.

[YBS06] Greta Yorsh, Thomas Ball, and Mooly Sagiv. Testing, abstraction, theorem proving: Better together! In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 145–156, New York, NY, USA, 2006. ACM.

[ZED11] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. Automatic generation of load tests. In *Proceedings of the 2011 26th IEEE/ACM In-*

ternational Conference on Automated Software Engineering, ASE '11, pages 43–52, Washington, DC, USA, 2011. IEEE Computer Society.