UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

POLITÉCNICA

Generación de Casos de Prueba en Programación Orientada a Objetos

Tesis Doctoral

José Miguel Rojas

Diciembre, 2013

UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

POLITÉCNICA

Test Case Generation in Object-Oriented Programming

PhD Thesis

José Miguel Rojas

December, 2013

Departamento de Lenguajes,
Sistemas Informáticos e Ingeniería del Software
FACULTAD DE INFORMÁTICA

Generación de Casos de Prueba en Programación Orientada a Objetos

(Test Case Generation in Object-Oriented Programming)

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Software and Systems

Author: **José Miguel Rojas**
Computer Engineer
Universidad Autónoma Gabriel René Moreno, Bolivia

Advisors: **Elvira Albert**
Doctor of Philosophy in Computer Science

Miguel Gómez-Zamalloa
Doctor of Philosophy in Computer Science

Tutor: **Germán Puebla**

December, 2013

Abstract

Testing is nowadays the most used technique to validate software and assess its quality. It is integrated into all practical software development methodologies and plays a crucial role towards the success of any software project. From the smallest units of code to the most complex components and their integration into a software system and later deployment; all pieces of a software product must be tested thoroughly before a software product can be released. The main limitation of software testing is that it remains a mostly manual task, representing a large fraction of the total development cost. In this scenario, test automation is paramount to alleviate such high costs.

Test case generation (TCG) is the process of automatically generating test inputs that achieve high coverage of the system under test. Among a wide variety of approaches to TCG, this thesis focuses on structural (white-box) TCG, where one of the most successful enabling techniques is symbolic execution. In symbolic execution, the program under test is executed with its input arguments being symbolic expressions rather than concrete values. This thesis relies on a previously developed constraint-based TCG framework for imperative object-oriented programs (e.g., Java), in which the imperative program under test is first translated into an equivalent constraint logic program, and then such translated program is symbolically executed by relying on standard evaluation mechanisms of Constraint Logic Programming (CLP), extended with special treatment for dynamically allocated data structures.

Improving the scalability and efficiency of symbolic execution constitutes a major challenge. It is well known that symbolic execution quickly becomes impractical due to the large number of paths that must be explored and the size of the constraints that must be handled. Moreover, symbolic execution-based TCG tends to produce an unnecessarily large number of test cases when applied to medium or large programs. The contributions of this dissertation can be summarized as follows.

(1) A compositional approach to CLP-based TCG is developed which overcomes the

inter-procedural path explosion by separately analyzing each component (method) in a program under test, stowing the results as method summaries and incrementally reusing them to obtain whole-program results. A similar compositional strategy that relies on program specialization is also developed for the state-of-the-art symbolic execution tool Symbolic PathFinder (SPF).

(2) Resource-driven TCG is proposed as a methodology to use resource consumption information to drive symbolic execution towards those parts of the program under test that comply with a user-provided resource policy, avoiding the exploration of those parts of the program that violate such policy.

(3) A generic methodology to guide symbolic execution towards the most interesting parts of a program is proposed, which uses abstractions as oracles to steer symbolic execution through those parts of the program under test that interest the programmer/tester most.

(4) A new heap-constraint solver is proposed, which efficiently handles heap-related constraints and aliasing of references during symbolic execution and greatly outperforms the state-of-the-art standard technique known as lazy initialization.

(5) All techniques above have been implemented in the PET system (and some of them in the SPF tool). Experimental evaluation has confirmed that they considerably help towards a more scalable and efficient symbolic execution and TCG.

Resumen*

Las pruebas de software (Testing) son en la actualidad la técnica más utilizada para la validación y la evaluación de la calidad de un programa. El testing está integrado en todas las metodologías prácticas de desarrollo de software y juega un papel crucial en el éxito de cualquier proyecto de software. Desde las unidades de código más pequeñas a los componentes más complejos, su integración en un sistema de software y su despliegue a producción, todas las piezas de un producto de software deben ser probadas a fondo antes de que el producto de software pueda ser liberado a un entorno de producción. La mayor limitación del testing de software es que continúa siendo un conjunto de tareas manuales, representando una buena parte del coste total de desarrollo. En este escenario, la automatización resulta fundamental para aliviar estos altos costes.

La generación automática de casos de pruebas (TCG, del inglés test case generation) es el proceso de generar automáticamente casos de prueba que logren un alto recubrimiento del programa. Entre la gran variedad de enfoques hacia la TCG, esta tesis se centra en un enfoque estructural de caja blanca, y más concretamente en una de las técnicas más utilizadas actualmente, la ejecución simbólica. En ejecución simbólica, el programa bajo pruebas es ejecutado con expresiones simbólicas como argumentos de entrada en lugar de valores concretos. Esta tesis se basa en un marco general para la generación automática de casos de prueba dirigido a programas imperativos orientados a objetos (Java, por ejemplo) y basado en programación lógica con restricciones (CLP, del inglés constraint logic programming). En este marco general, el programa imperativo bajo pruebas es primeramente traducido a un programa CLP equivalente, y luego dicho programa

*The thesis is presented in English for its defense before an international committee. The inclusion of this summary in Spanish is prescribed by the PhD regulations of the Universidad Politécnica de Madrid.

*Este resumen de la Tesis Doctoral, presentada en lengua inglesa para su defensa ante un tribunal internacional, es preceptivo según la normativa de doctorado vigente en la Universidad Politécnica de Madrid.

CLP es ejecutado simbólicamente utilizando los mecanismos de evaluación estándar de CLP, extendidos con operaciones especiales para el tratamiento de estructuras de datos dinámicas.

Mejorar la escalabilidad y la eficiencia de la ejecución simbólica constituye un reto muy importante. Es bien sabido que la ejecución simbólica resulta impracticable debido al gran número de caminos de ejecución que deben ser explorados y a tamaño de las restricciones que se deben manipular. Además, la generación de casos de prueba mediante ejecución simbólica tiende a producir un número innecesariamente grande de casos de prueba cuando es aplicada a programas de tamaño medio o grande. Las contribuciones de esta tesis pueden ser resumidas como sigue.

(1) Se desarrolla un enfoque composicional basado en CLP para la generación de casos de prueba, el cual busca aliviar el problema de la explosión de caminos inter-procedimiento analizando de forma separada cada componente (p.ej. método) del programa bajo pruebas, almacenando los resultados y reutilizándolos incrementalmente hasta obtener resultados para el programa completo. También se ha desarrollado un enfoque composicional basado en especialización de programas (evaluación parcial) para la herramienta de ejecución simbólica Symbolic PathFinder (SPF).

(2) Se propone una metodología para usar información del consumo de recursos del programa bajo pruebas para guiar la ejecución simbólica hacia aquellas partes del programa que satisfacen una determinada política de recursos, evitando la exploración de aquellas partes del programa que violan dicha política.

(3) Se propone una metodología genérica para guiar la ejecución simbólica hacia las partes más interesantes del programa, la cual utiliza abstracciones como generadores de trazas para guiar la ejecución de acuerdo a criterios de selección estructurales.

(4) Se propone un nuevo resolutor de restricciones, el cual maneja eficientemente restricciones sobre el uso de la memoria dinámica global (heap) durante ejecución simbólica, el cual mejora considerablemente el rendimiento de la técnica estándar utilizada para este propósito, la "lazy initialization".

(5) Todas las técnicas propuestas han sido implementadas en el sistema PET (el enfoque composicional ha sido también implementado en la herramienta SPF). Mediante evaluación experimental se ha confirmado que todas ellas mejoran considerablemente la escalabilidad y eficiencia de la ejecución simbólica y la generación de casos de prueba.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software testing [AO08] is the most commonly used technique for validating the quality of software. In practice, testing is a mostly manual process within software development that accounts for a large fraction of the total time and maintenance costs. Automated software testing aims to improve on the effectiveness and efficiency of software testing. Many techniques have been developed to automate different aspects of software testing. In particular, this thesis is devoted to the automation of a crucial part of the testing process: the generation of test cases.

Test Case Generation (TCG) is the process of automatically generating a collection of test cases, known as a *test suite*, which can then be used by testing tools to validate the correctness of the software under development. A wide variety of approaches to TCG can be found in the literature. A recent survey [ABC+13] considers the most prominent ones to be: structural TCG using symbolic execution, model-based TCG, combinatorial TCG, (adaptive)random TCG and search-based TCG. Among them, random TCG is plausibly the most popular approach. It is a black-box technique that requires almost no manual intervention, can be quickly implemented, and has been reported successful to find bugs in complex pieces of software [GKS05, CH00]. On the other hand, random TCG tends to generate plentiful test suites without being able to provide any coverage assurance to assist in determining the reliability and overall quality of the software under test. Therefore, random TCG is not the best alternative when the testing goal is to achieve high coverage, as it is the case, for instance, in *unit testing*. In this kind of scenario, and provided that the code is available, an structural approach seems more adequate.

This thesis focuses on *white-box* testing. That is, test cases are obtained from the concrete program (e.g., using its control flow graph) in contrast to *black-box* testing,

where they are deduced from a specification of the program. Also, our focus is on *static* testing, where we assume no knowledge about the input data, in contrast to *dynamic* approaches [FK96, GMS00] which execute the program to be tested for concrete input values.

The quality of a test suite is assessed by using code coverage criteria. A coverage criterion aims at measuring how well the program under test is exercised by a test suite. Examples of coverage criteria are: *statement coverage* which requires that each statement of the code is executed; *branch coverage* which requires all conditional statements in the program to be evaluated both to true and false; and *path coverage* which requires that every possible trace through a given part of the code is executed. These criteria are however not *finitely applicable* [ZHM97] due to infinite paths and infeasible statements (i.e., dead code). An alternative to path coverage, which is finitely-applicable is the *loop-k* coverage criterion, which requires every loop in the program to be executed zero times, once, twice, ..., and k times.

The standard approach to generating test cases statically is to perform a *symbolic execution* of the program [Kin76, Cla76, GBR00, Meu01, MLK04, EH07, DSV10b, TdH08]. Symbolic execution is a program analysis technique that has received a renewed interest in recent years, thanks in part to the increased availability of computational power and decision procedures [CS13]. White-box TCG is among the most studied applications of symbolic execution, with several tools available [CGK+11].

Symbolic execution consists in executing a program with the contents of its input arguments being symbolic values rather than concrete ones. Symbolic execution paths represent sets of concrete executions. A symbolic execution *tree* characterizes the set of execution paths explored during the symbolic execution of a program. During the course of symbolic execution, the values of the program's variables are represented as symbolic expressions over the input symbolic values and a *path condition* is maintained. Such a path condition is updated whenever a branch instruction is executed. That is, for each conditional statement in the program, symbolic execution explores both the "then" and the "else" branch, refining the path condition accordingly. The satisfiability of each of these branches is checked and symbolic execution stops exploring any path whose path condition becomes unsatisfiable, hence only feasible paths are followed. Since the number of feasible paths in in general infinite, a termination criterion must be imposed to ensure finiteness of the symbolic execution process. Such a termination criterion can be expressed in different forms. For instance, a computation time budget can be established, or a bound

on the depth of the symbolic execution tree can be imposed.

The outcome of symbolic execution is a set of equivalence classes of inputs, one for each symbolic execution path, each of them consisting of the *constraints* over the input variables that characterize the set of feasible concrete executions of the program that take the same path, i.e., the path condition. In a further step, off-the-shelf constraint solvers can be used to solve path conditions and generate concrete instantiations for these path conditions. This last step provides actual test inputs for the program, amenable to further validation by testing frameworks such as JUnit, which execute such test inputs and check that the output is as expected. Importantly, notice that we adopt a non-standard definition for the terminology "test case". Standard test cases are concrete, i.e., actual input values obtained in the last step of the process. In contrast, in this thesis we refer as *test cases*, or *test suite*, to the set of path conditions (and symbolic expressions for variables) that characterize all symbolic execution paths explored by symbolic execution using a particular termination criterion.

The idea of performing symbolic execution by relying on CLP has been subject of previous work [DO91, Meu01, CG10, GBR00, DSV10b, GZAP10]. The work presented in this thesis builds upon a CLP-based approach to TCG of *imperative object-oriented programs*[*] [AGZP09, GZAP10], which consists of three main ingredients:

(i) The imperative program is first translated into an equivalent CLP program, referred to as *CLP-translated program* in what follows. The translation can be performed by partial evaluation [GZAP09] or by traditional compilation.

(ii) Symbolic execution is performed on the CLP-translated program by relying on the standard evaluation mechanism of CLP, which provides backtracking and handling of symbolic expressions for free.

(iii) Special treatment to support the use of dynamic memory in object-oriented programming is provided by means of heap-related operations which are fully implemented in CLP. These operations not only allow constructing complex arbitrary data structures with unbounded data (e.g., recursive data structures) but also serve as an effective mechanism to explore all possible heap shapes due to aliasing of references.

[*]The application of this approach to TCG of logic programs must consider failure [DSV10a] and, to functional programs, should consider laziness, higher-order, etc. [FK07].

Finiteness of the symbolic execution process in our CLP-based approach is ensured by the use of the *loop-k* coverage criterion, which by specifying the number of loop iterations implicitly imposes an upper bound on the length of symbolic execution paths. The finite domain constraint solver CLP(FD) [Tri12] is used to check the satisfiability of path conditions during symbolic execution and to solve constraints and generate actual imput data (if needed) at the end of the process.

A main challenge of symbolic execution and white-box TCG [PV09] is to scale up to larger applications. As mentioned before, the symbolic execution tree is in general infinite due to loops or recursion in the program under test. It is well known that performing TCG by symbolic execution on large real-world programs can quickly become computationally impracticable due to the large number of paths that need to be explored and also to the size of their associated constraints [PV09]. Several techniques have been proposed with the aim of mitigating these scalability limitations. Notable examples are concolic execution [MS07], program abstraction [APV06], compositionality [SWP+09,God07,AGT08], search heuristics [CGP+08] and path merging [AEO+08]. In this thesis, we propose several novel techniques that aim at alleviating different scalability issues of symbolic execution for software testing in practice.

The first scalability issue that we tackle is the inter-procedural path explosion problem. We develop a compositional approach to symbolic execution and TCG. The main idea is to start by performing symbolic execution and TCG on the independent components of the program under test, e.g., functions or procedures, and then continue to incrementally execute larger portions of the program, according to a bottom-up traversal of its call graph, composing previously computed components results until finally whole-program results can be computed. The proposed methodology is first applied to our CLP-based TCG framework, where we construct so-called method summaries, which encode input/output relations for all feasible execution paths in each component. Then, we observe that encoding explicitly all input/output relations may hinder scalability. To alleviate this problem, we propose the use of *program specialization* [JGS93] (also known as *partial evaluation*) to build more succinct method summaries that lead to a more efficient compositional operation.

Another common limitation of symbolic execution in the context of TCG is that it tends to produce an unnecessarily large number of test cases for all but tiny programs. This limitation not only hinders scalability but also complicates human reasoning on the generated test cases. We explore scalability measures towards reducing the size of

the symbolic execution tree to be traversed, and therefore reducing the number of test cases to be generated. We first propose a methodology that builds resource consumption information [AAG+12] into the TCG process. Interestingly, we show that user-provided resource policies can be used as a selection criterion in order to generate test cases for paths of the program that adhere to such a policy, discarding those that violate it. Furthermore, we generalize this idea and propose a generic framework for guided TCG whose main motivation is to guide the construction of the symbolic execution tree towards the most interesting parts of the program with respect to given selection criteria, avoiding the exploration of parts of the program, which e.g., have already been tested. We show the potential applicability of this guided TCG framework in industrial software testing practices such as unit testing or selective testing.

The last widely acknowledge limitation of symbolic execution for TCG that we explore is the lack of dedicated decision procedures (constraint-solving optimizations) for the type of constraints generated during symbolic execution of real programs. Specifically, we propose a heap solver to enable efficient symbolic execution and TCG of heap-manipulating programs. We demonstrate that our heap solver greatly outperforms lazy initialization, the currently "de facto" standard technique, in allowing the systematic exploration of all possible heap shapes in programs that create and manipulate complex input data structures.

All the techniques presented in this thesis have been implemented in the PET system [AGZP10], an automatic TCG tool for Java bytecode. Effectiveness and efficiency have been validated empirically against relevant challenging benchmarks. In most cases, benchmarks are borrowed from the well-known net.datastructures package [GTZ10], an educational collection of Java interfaces and classes that implement fundamental datastructures (e.g., search trees and graphs) and algorithms (e.g. sorting and traversal) and which is publicly available online at `http://net3.datastructures.net`). Moreover, the compositional approach to TCG via program specialization was implemented and evaluated in the SPF tool.

1.1 Thesis Objectives

The global objective of this thesis is to develop novel techniques to alleviate the well-known scalability issues of symbolic execution in the context of TCG. The baseline to tackle this objective is the CLP-based TCG framework defined in [AGZP09, AGZP10, GZAP10]. The

research performed in this thesis responds to the following concrete objectives:

- To develop an scalable compositional TCG approach.
- To realise a methodology to guide TCG with resource consumption information and resource policies.
- To design and develop a generic framework to guide TCG according to selection criteria of interest.
- To define an effective and efficient mechanism to handle TCG of object-oriented heap-manipulating programs.

An important additional concrete objective, which is orthogonal to the aforementioned ones, is to implement all the proposed techniques and validate their effectiveness with respect to relevant case studies and benchmarks.

1.2 Contributions

In this section, we briefly describe the contributions of this thesis. We also report on the dissemination of the obtained results in international peer-reviewed venues.

Compositional Test Case Generation

Compositional reasoning is a general purpose methodology that has been successfully applied in the past to scale up static analysis and software verification techniques. Large programs can be handled more effectively by first analyzing their parts (such as methods, procedures, components, modules, libraries, etc.) separately and then by composing the results obtained for these parts to incrementally obtain results for the whole program. We propose compositional reasoning as an effective methodology for scaling up symbolic execution and TCG. Importantly, compositional reasoning not only allows to consider larger more realistic programs but also provides a practical solution to handle native code, which may be unavailable or written in a different programming language. Namely, we can model the behavior of a native method by means of so-called method summaries and compositional reasoning is able to use them when needed. Our compositional approach has been developed as an extension of our CLP-based TCG framework.
The results of this work have been published in:

Elvira Albert and Miguel Gómez-Zamalloa and José Miguel Rojas and Germán

Puebla. **Compositional CLP-based Test Data Generation for Imperative Languages.** In *LOPSTR 2010, 20th International Symposium on Logic-Based Program Synthesis and Transformation*, volume 6564 of *Lecture Notes in Computer Science*, pages 99–116. Springer, July 2010.

Furthermore, we also propose a novel compositional approach that relies on *program specialization* to compute more concise summaries which allow increasing the overall efficiency of the symbolic execution and TCG process, in comparison to previous approaches. This program specialization-based compositional approach has been designed, implemented and evaluated in Symbolic PathFinder [PMB+08], a symbolic execution engine built on top of the Java PathFinder model checker [VM05].
The results of this work have been published in:

José Miguel Rojas and Corina S. Păsăreanu. **Compositional Symbolic Execution through Program Specialization.** In *BYTECODE 2013, 8th Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, March 2013. An extended version of this work is, by the time of writing this dissertation, under consideration for publication in the Journal *Science of Computer Programming*.

Resource-driven Test Case Generation

Traditionally, Test Case Generation (TCG) has been used to automatically obtain *test inputs* which can then be used by a software testing tool to validate the *functional* behaviour of the program. In this chapter, we propose *resource-aware* TCG, whose purpose is to generate test cases (from which the test inputs are obtained) with associated *resource consumption* information. The framework is parametric with respect to the notion of resources (it can measure memory, steps, etc.) and allows using software testing to detect bugs related to non-functional aspects of the program. As a further step, we introduce *resource-driven* TCG, whose purpose is to guide the TCG process by taking resource consumption into account. Interestingly, given a *resource policy*, TCG is guided to generate test cases that adhere to the policy and avoid generating test cases which violate it.
The results of this work have been published in:

Elvira Albert and Miguel Gómez-Zamalloa and José Miguel Rojas. **Resource-driven CLP-based Test Case Generation.** In *LOPSTR 2011, 21st International Symposium on Logic-Based Program Synthesis and Transformation*,

volume 7225 of *Lecture Notes in Computer Science*, pages 25–41. Springer, July 2011.

Guided Test Case Generation

Another common limitation of symbolic execution for software testing the field is that it tends to produce an unnecessarily large number of test cases even for medium size programs. In an attempt to allow for a more controlled TCG process, in this chapter we propose a generic framework that lets the programmer decide on a selection criteria that can guide symbolic execution and thus TCG towards parts of the program that are more interesting to test. We show how guided TCG framework can help alleviate these scalability drawbacks that most symbolic execution-based TCG approaches endure.
The results of this work have been published in:

> José Miguel Rojas and Miguel Gómez-Zamalloa. **A Framework for Guided Test Case Generation in Constraint Logic Programming.** In *LOPSTR 2012, 22nd International Symposium on Logic-Based Program Synthesis and Transformation*, volume 7844 of *Lecture Notes in Computer Science*, pages 176–193. Springer, September 2012.

Heap Solver for TCG of Heap-manipulating Programs

Efficiently testing heap-manipulating programs is the last important challenge that we have tackled in this work. These programs often build complex, dynamically allocated data structures during execution and, to ensure reliability, the testing process needs to consider all possible shapes these data structures can take. Scalability issues arise since high (often exponential) numbers of shapes may be built due to the *aliasing* of references. In this chapter, we present a novel *CLP heap solver* for the TCG of heap-manipulating programs that is more scalable than previous proposals, thanks to the treatment of reference aliasing by means of *disjunction*, and to the use of advanced *back-propagation* of heap related constraints. In addition, the heap solver supports the use of *heap assumptions* to avoid aliasing of data that, though legal, should not be provided as input.
The results of this work have been published in:

> Elvira Albert and María García de la Banda and Miguel Gómez-Zamalloa and José Miguel Rojas and Peter Stuckey. **A CLP Heap Solver for Test**

Case Generation. In *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue*, 13(4-5):721–735. Cambridge University Press, July 2013.

1.3 Thesis Outline

The content of this thesis is structured in five chapters in addition to this introductory one. Chapter 2 introduces the CLP-based TCG framework in which the core of the thesis is developed. Chapter 3 develops an approach to compositional symbolic execution and TCG. First, compositional reasoning is proposed for the CLP-based framework introduced in Chapter 2. Second, a compositional approach that relies on program specialization is proposed for the model checking-based symbolic execution framework Symbolic PathFinder (SPF). In Chapter 4, we present a methodology to compute resource consumption information and integrate it into test cases such that CLP-based TCG can be used with user-provided resource policies. The work presented in Chapter 5 generalizes the methodology of Chapter 4 and develops a framework for guiding TCG by abstract trace generators. Chapter 6 presents a novel heap solver that efficiently handles heap-manipulating programs and overcomes the path explosion problem due to aliasing of references. Finally, the thesis concludes in Chapter 7, where we summarize the contributions, discuss final remarks and outline possible future research directions.

Chapter 2

CLP-based Test Case Generation

This chapter summarizes the Constraint Logic Programming-based approach to TCG for imperative languages introduced in [AGZP09] and extended to object-oriented languages with dynamic memory in [GZAP10]. This framework serves as the basis on top of which the core of this thesis develops.

2.1 CLP-based Test Case Generation

CLP-based Test Case Generation advocates the use of CLP technology to perform test case generation of imperative object-oriented programs. The process has two phases. In the first phase, the imperative object-oriented program under test is automatically transformed into an equivalent executable *CLP-translated* program. Instructions that manipulate heap-allocated data are represented by means of calls to specific *heap operations*. In the second phase, the CLP-translated program is symbolically executed using the standard CLP execution and constraint solving mechanism. The above-mentioned heap operations are also implemented in standard CLP, in a suitable way in order to support symbolic execution. The next two sections overview these two phases, which are also shown graphically in Figure 2.1.

The Imperative Object-Oriented Language Although our approach is not tied to any particular imperative object-oriented language, we consider as the source language a subset of Java. For simplicity, we leave out of such subset features like concurrency, bitwise operations, static fields, access control (i.e., the use of public, protected and private modifiers) and primitive types besides integers and booleans. Nevertheless, these features

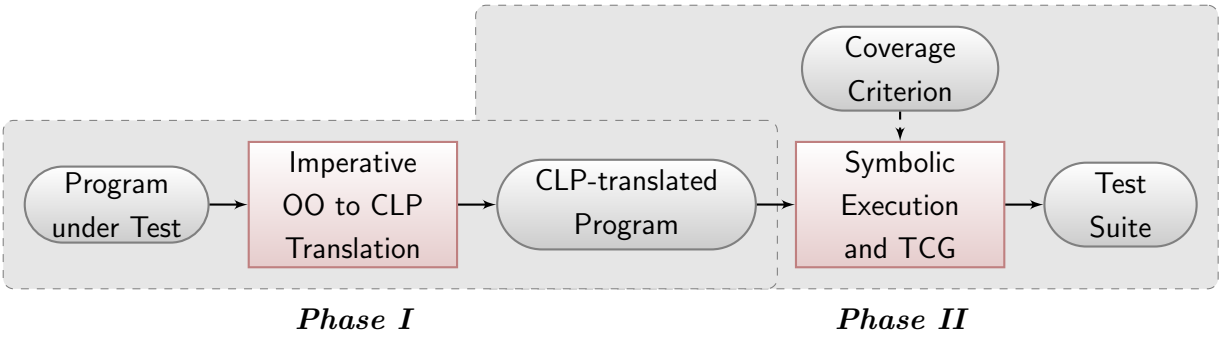


Figure 2.1: CLP-based Test Case Generation Framework

can be relatively easy to handle in practice by our framework, except for concurrency, which is well-known to pose further challenges to symbolic execution and its scalability.

2.1.1 CLP-translated Programs

The translation of imperative object-oriented programs into equivalent CLP-translated programs has been subject of previous work (see, e.g., [GZAP09, AAG+07]). Therefore, we will recap the features of the translated programs without going into deep details of how the translation is done. The translation is formally defined as follows:

Definition 2.1.1 (CLP-translated program)**.** *The CLP-translated program for a given method m from the original imperative object-oriented program consists of a set of predicates $m, m_1, \ldots, m_n$, each of them defined by a set of mutually exclusive rules of the form $m(In, Out, H_{in}, H_{out}, EF) : -[\bar{G},]b_1, \ldots, b_n$. where:*

1. *In and Out are, resp., the (possibly empty) list of input and output arguments.*
2. *H_{in} and H_{out} are, resp., the input and (possibly modified) output heaps to each predicate.*
3. *EF is an exception flag which indicates whether the execution of m ends normally or with an uncaught exception.*
4. *If predicate m is defined by multiple rules, the guards in each rule contain mutually exclusive conditions. We denote by m_i^k the $k-th$ rule defining m_i.*
5. *$\bar{G}$ is a set of constraint acting as guards of the rule (comparisons between numeric data or references, etc.)*

$$
\begin{array}{rl}
Clause ::= & Pred\ (Args_{in}, Args_{out}, H_{in}, H_{out}, ExFlag)\ \text{:-}\ [G,]B_1, B_2, \ldots, B_n. \\
G ::= & Num^*\ ROp\ Num^* \mid Ref_1^*\ \backslash == Ref_2^* \mid \mathsf{type}(H, Ref^*, T) \\
B ::= & Var\ \#= Num^*\ AOp\ Num^* \\
& \mid Pred\ (Args_{in}, Args_{out}, H_{in}, H_{out}, ExFlag) \\
& \mid \mathsf{new\_object}(H_{in}, C^*, Ref^*, H_{out}) \\
& \mid \mathsf{new\_array}(H_{in}, T, Num^*, Ref^*, H_{out}) \mid \mathsf{length}(H_{in}, Ref^*, Var) \\
& \mid \mathsf{get\_field}(H_{in}, Ref^*, FSig, Var) \mid \mathsf{set\_field}(H_{in}, Ref^*, FSig, Data^*, H_{out}) \\
& \mid \mathsf{get\_array}(H_{in}, Ref^*, Num^*, Var) \mid \mathsf{set\_array}(H_{in}, Ref^*, Num^*, Data^*, H_{out}) \\
Pred ::= & Block \mid MSig \qquad\qquad ROp ::= \ \#> \mid \#< \mid \#>= \mid \#=< \mid \#= \mid \#\backslash= \\
Args ::= & [] \mid [Data^* | Args] \qquad\qquad AOp ::= \ + \mid - \mid * \mid / \mid mod \\
Data ::= & Num \mid Ref \mid ExFlag \qquad\qquad T ::= bool \mid int \mid C \mid array(T) \\
Ref ::= & null \mid r(Var) \qquad\qquad FSig ::= C{:}FN \\
ExFlag ::= & ok \mid exc(Var) \qquad\qquad H ::= Var
\end{array}
$$

Figure 2.2: Syntax of CLP-translated programs

6. $b_1, \ldots, b_n$ *is a sequence of instructions including arithmetic operations, calls to other predicates and built-ins to operate on the heap, etc., as defined in Figure 2.2. As usual, an SSA transformation is performed [CFR+91].*

Specifically, CLP-translated programs adhere to the grammar in Figure 2.2. As customary, terminals start with lowercase (or special symbols) and non-terminals start with uppercase; subscripts are provided just for clarity. Non-terminals *Block*, *Num*, *Var*, *FN*, *MSig* and *C* denote, resp., the set of predicate names, numbers, variables, field names, method signatures, field signatures and class names. A clause indistinguishably defines either a method which appear in the original source program (*MSig*), or an additional predicate which correspond to an intermediate block in the control flow graph of original program (*Block*). A field signature *FSig* contains the class where the field is defined and the field name *FN*. An asterisk on a non-terminal denotes that it can be either as defined by the grammar or a (possibly constrained) variable (e.g., Num^*, denotes that the term can be a number or a variable). Heap references are written as terms of the form $r(Ref)$ or *null*. The operations that handle data in the heap are translated into built-in heap-related predicates.

Let us observe the following: There exists a one-to-one correspondence between blocks in the control flow graph of the original program and rules in the CLP-translated one. Mutual exclusion between the rules of a predicate is ensured either by means of mutually exclusive *guards*, or by information made explicit on the heads of rules, as usual in CLP. This makes the CLP-translated program deterministic, as well as it is the original imperative one (point 4 in Definition 2.1.1). Observe that the global memory (or heap) is explicitly represented in the CLP-translated program by means of logic variables. When a rule is invoked, the input heap $\mathtt{H_{in}}$ is received and, after executing its body, the heap might be modified, resulting in $\mathtt{H_{out}}$. The operations that modify the heap will be shown later. Note that the above definition proposes a translation to CLP as opposed to a translation to pure logic (e.g. to predicate logic or even to propositional logic, i.e., a logic that is not meant for "programming"). This is because we then want to execute the resulting translated programs to perform TCG and this requires, among other things, handling a constraint store and then generating actual data from such constraints. CLP is a natural paradigm to perform this task. Virtual method invocations are resolved at compile-time in the original imperative object-oriented language by looking up all possible runtime instances of the method. In the CLP-translated program, such invocations are translated into a choice of type instructions which check the actual object type, followed by the corresponding method invocation for each runtime instance. Exceptional behaviour is handled explicitly in the CLP-translated program.

Figure 2.3 summarizes the CLP implementation of the operations to create heap-allocated data structures (new_object and new_array) and to read and modify them (get_field, set_array, etc.) [GZAP10]. These operations rely on some auxiliary predicates (like deterministic versions of member member_det, of replace replace_det, and nth0 and replace_nth0 for arrays) which are quite standard and hence their implementation is not shown. For instance, a new object is created through a call to predicate new_object(H_{in},Class,Ref,H_{out}), where H_{in} is the current heap, Class is the new object's type, Ref is a unique reference in the heap for accessing the new object and H_{out} is the new heap after allocating the object. Read-only operations do not produce any output heap. For example, get_field(H_{in},Ref,FSig,Var) retrieves from H_{in} the value of the field identified by *FSig* from the object referenced by *Ref*, and returns its value in *Var* leaving the heap unchanged. Instruction set_field(H_{in},Ref,FSig,Data,H_{out}) sets the field identified by *FSig* from the object referenced by *Ref* to the value *Data*, and returns the modified heap H_{out}. The remaining operations are implemented likewise.

```
    new_object(H,C,Ref,H') :- build_object(C,Ob), new_ref(Ref), H' = [(Ref,Ob)|H].
   new_array(H,T,L,Ref,H') :- build_array(T,L,Arr), new_ref(Ref), H' = [(Ref,Arr)|H].

             type(H,Ref,T) :- get_cell(H,Ref,Cell), Cell = object(T,_).
           length(H,Ref,L) :- get_cell(H,Ref,Cell), Cell = array(_,L,_).

   get_field(H,Ref,FSig,V) :- get_cell(H,Ref,Ob), FSig = C:FN,
                              Ob = object(T,Fields), subclass(T,C),
                              member_det(field(FN,V),Fields).
      get_array(H,Ref,I,V) :- get_cell(H,Ref,Arr), Arr = array(_,_,Xs), nth0(I,Xs,V).

set_field(H,Ref,FSig,V,H') :- get_cell(H,Ref,Ob), FSig = C:FN,
                              Ob = object(T,Fields), subclass(T,C),
                              replace_det(Fields,field(FN,_),field(FN,V),Fields'),
                              set_cell(H,Ref,object(T,Fields'),H').
   set_array(H,Ref,I,V,H') :- get_cell(H,Ref,Arr), Arr = array(T,L,Xs),
                              replace_nth0(Xs,I,V,Xs'),
                              set_cell(H,Ref,array(T,L,Xs'),H').
```

```
   get_cell([(Ref',Cell')|_],Ref,Cell) :- Ref == Ref', !, Cell = Cell'.
             get_cell([_|RH],Ref,Cell) :- get_cell(RH,Ref,Cell).

    set_cell([(Ref',_)|H],Ref,Cell,H') :- Ref == Ref', !, H' = [(Ref,Cell)|H].
set_cell([(Ref',Cell')|H'],Ref,Cell,H) :- H = [(Ref',Cell')|H''],
                                          set_cell(H',Ref,Cell,H'').
```

Figure 2.3: Heap operations for ground execution [GZAP10]

The Heap term. The heaps generated by using these operations adhere to this grammar:

$$\begin{array}{rcl} Heap & ::= & [] \mid [Loc|Heap] \\ Cell & ::= & object(C^*,Fields^*) \mid array(T^*,Num^*,Args^*) \\ Loc & ::= & (Num^*,Cell) \\ Fields & ::= & [] \mid [f(FN,Data^*)|Fields^*] \end{array}$$

The heap is represented as a list of locations which are pairs formed by a unique reference and a cell. Each cell can be an object or an array. An object contains its type and its list of fields, each of which is made of its signature and data content. An array contains its type, its length and its list of elements.

Note that our CLP-translated programs manipulate the heap as a black-box through its associated operations.

```
1 class Node {
2   int data;
3   Node next;
4 }
5
6 class List {
7   Node first;
8   void remAll(List l) {
9     // block1
10    Node lf = l.first;
11    // loop1 -> cond1
12    while (lf != null) {
13      // block2
14      Node prev = null;
15      Node p = null;
16      Node next = first;
17      // loop2 -> cond2
18      while (next != null) {
19        // block3
20        prev = p;
21        p = next;
22        next = next.next;
23        // if1
24        if (p.data == lf.data)
25          // if2
26          if (prev == null) {
27            first = next;
28            p = null;
29          } else {
30            prev.next = next;
31            p = prev;
32          }
33      }
34      // block4
35      lf = lf.next;
36    }
37  }
38 }
```

(a) Java source code

```
remAll([r(Th),L],Hi,none,Ho,E) :-
  block1([Th,L],Hi,Ho,E).
block1([Th,r(L)],Hi,Ho,E) :-
  get_field(Hi,L,first,LfR,H2),
  loop1([Th,L,LfR],H2,Ho,E).
block1([Th,null],Hi,Ho,exc(E)) :-
  create_object(Hi,'NPE',E,Ho).
loop1([Th,L,LfR],Hi,Ho,E) :-
  cond1([Th,L,LfR],Hi,Ho,E).
cond1([Th,L,null],H,H,ok).
cond1([Th,L,r(Lf)],Hi,Ho,E) :-
  block2([Th,L,Lf],Hi,Ho,E).
block2([Th,L,Lf],Hin,Ho,E) :-
  get_field(Hi,Th,first,FR,H2),
  loop2([Th,L,Lf,null,null,FR],H2,Ho,E).
loop2([Th,L,Lf,Prev,P,FR],Hi,Ho,E) :-
  cond2([Th,L,Lf,Prev,P,FR],Hi,Ho,E).
cond2([Th,L,Lf,Prev,P,null],Hi,Ho,E) :-
  block4([Th,L,Lf],Hi,Ho,E).
cond2([Th,L,Lf,Prev,P,r(F)],Hi,Ho,E) :-
  block3([Th,L,Lf,P,F,F],Hi,Ho,E).
block3([Th,L,Lf,P,F,F],Hi,Ho,E) :-
  get_field(Hi,F,next,FRN,H2),
  get_field(H2,F,data,A,H3),
  get_field(H3,Lf,data,B,H4),
  if1([A,B,Th,L,Lf,P,F,FRN],H4,Ho,E).
if1([A,B,Th,L,Lf,Prev,P,FRN],Hi,Ho,E) :-
  #\=(A,B),
  loop2([Th,L,Lf,Prev,P,FRN],Hi,Ho,E).
if1([A,A,Th,L,Lf,Prev,P,FRN],Hi,Ho,E) :-
  if2([Th,L,Lf,Prev,P,FRN],Hi,Ho,E).
if2([Th,L,Lf,r(F),P,N],Hi,Ho,E) :-
  set_field(Hi,F,next,N,H2),
  loop2([Th,L,Lf,F,F,N],H2,Ho,E).
if2([Th,L,Lf,null,P,N],Hi,Ho,E) :-
  set_field(Hi,Th,first,N,H2),
  loop2([Th,L,Lf,null,null,N],H2,Ho,E).
block4([Th,L,Lf],Hi,Ho,E) :-
  get_field(Hi,Lf,next,LfRN,H2),
  loop1([Th,L,LfRN],H2,Ho,E).
```

(b) CLP-translation

Figure 2.4: CLP-based TCG example

Example 1 Figure 2.4a shows the Java source code of class `List`, which implements

a singly-linked list. The class contains one attribute `first` of type `Node`. As customary, `Node` is a recursive class with two attributes: `data` of type `int` and `next` of type `Node`. Method `remAll` takes as argument an object `l` of type `List`, traverses it (outer while loop) and for each of its elements, traverses the `this` object and removes all their occurrences (inner loop). Figure 2.4b shows the equivalent (simplified and pretty printed) CLP-translated code for method `remAll`. Let us observe some of the main features of the CLP-translated program. The `if` statement in line 24 is translated into two mutually exclusive rules (predicate `if1`) guarded by an arithmetic condition. Similarly, the `if` statement in line 26 is translated into predicate `if2`, implemented by two rules whose mutual exclusion is guaranteed by terms `null` and `r(_)` appearing in each rule head. Observe that iteration in the original program (`while` constructions) is translated into recursive predicates. For instance, the head of the inner `while` loop is translated into predicate `loop2`, its condition is guarded by the rules of predicate `cond2` (`null` or `r(_)`), and recursive calls are made from predicates `if1` (first rule) and `if2` (both rules). Finally, exception handling is made explicit in the CLP-translated program; the second rule of predicate `block1` encodes the runtime null pointer exception (`'NPE'`) that raises if the input argument `l` is `null`. □

2.1.2 Semantics of CLP-translated Programs

The standard CLP execution mechanism suffices to execute the CLP-translated programs if the target imperative language does not feature dynamic (heap-allocated) data structures [AGZP09]. However, our goal here is to target realistic object-oriented languages with dynamic memory.

Let us now focus on the concrete execution of CLP-translated programs by assuming that all input parameters of the predicate to be executed (i.e., In and H_{in}) are fully instantiated in the initial input state. We assume familiarity with the basic notions of CLP.

Let M be a method in the original imperative program, m be its corresponding predicate in the CLP-translated program P, and P' be the union of P and the predicates in Figure 2.3. The operational semantics of the CLP program P', can be defined in terms of *derivations*. A derivation is a sequence of reductions between states $S_0 \rightarrow_p S_1 \rightarrow_P \ldots \rightarrow_P S_n$, also denoted $S_0 \rightarrow_P S_n$, where a *state* $\langle \mathcal{G} \mid \theta \rangle$ consists of a goal $\mathcal{G}$ and a constraint store θ. The concrete execution of m with input θ is the derivation

$S_0 \rightarrow S_n$, where $S_0 = \langle m(In, Out, H_{in}, H_{out}, ExFlag) \mid \theta \rangle$ and θ initializes In and H_{in} to be fully ground. If the derivation successfully terminates, then $S_n = \langle \epsilon \mid \theta' \rangle$ and θ' is the *output state* (ϵ denotes the empty goal).

This definition of concrete execution relies on the correctness of the translation algorithm, which must guarantee that the CLP-translated program captures the same semantics of the original imperative one [GZAP09, AAG+07].

2.1.3 Symbolic Execution

When the source imperative language does not support dynamic memory, symbolic execution of the CLP-translated programs is attained by using the standard CLP execution mechanism *with all arguments being free variables*. The inherent constraint solving and backtracking mechanisms of CLP allow to keep track of path conditions, failing and backtracking when unsatisfiable constraints are hit, hence discarding such execution paths; and succeeding when satisfiable constraints lead to a terminating state in the program, which in the context of TCG implies that a new test case is generated.

However, in the case of heap-manipulating programs, the heap-related operations presented in Figure 2.3 fall short to generate arbitrary heap-allocated data structures and all possible heap shapes when accessing symbolic references. This is a well-known problem in TCG by symbolic execution. A naive solution to this problem could be to fully initialize all the reference parameters prior to symbolic execution. However, this would require imposing bounds on the size of input data structures, which is highly undesirable. Doing so would circumscribe the symbolic search space, hence jeopardizing the overall effectiveness of the technique.

Lazy Initialization. Lazy initialization [KPV03] is the *de facto* standard technique to enable symbolic execution to systematically handle arbitrary input data structures, and to explore all possible heap shapes that can be generated during the process, including those produced due to aliasing of references. The main idea is that symbolic execution starts with no knowledge about the program's input arguments and, as the program symbolically executes and accesses object fields, the components of the program's inputs are initialized on an "as-needed" basis. The intuition is as follows. To symbolically execute method `m` of class `C`, a new object `o` of class `C` with all its fields uninitialized is created (the `this` object in Java). When an unknown field of primitive type is read, a fresh unconstrained variable is created for that field. When a reference field `f` of type `T` is accessed in `m` for the first

time, all aliasing possibilities are considered. That is, `f` is non-deterministically set to: (a) `null`; (b) a new symbolic object of type `T` with uninitialized fields; and (c) an alias to a previously initialized object of type `T`. Such non-deterministic choices are materialized into branches in the symbolic execution tree. As a result, the heap associated with any particular execution path is built using only the constraints induced by the visited code.

Interestingly, a straightforward generalization of predicates get_cell and set_cell in Figure 2.3 provide a simple and flexible solution to the aforementioned problem, and constitutes a quite natural implementation of the *lazy initialization* technique in our CLP-based framework. Figure 2.5 shows the new implementation of these operations.

The intuitive idea is that the heap during symbolic execution contains two parts: the *known part*, with the cells that have been explicitly created during symbolic execution appearing at the beginning of the list, and the *unknown part*, which is a logic variable (tail of the list) in which new data can be added. Importantly, the definition of get_cell/3 distinguishes two situations when searching for a reference: (i) It finds it in the known part (second clause), meaning that the reference has already been accessed earlier (note the use of syntactic equality rather than unification, since references at execution time can be variables); or (ii) It reaches the unknown part of the heap (a logic variable), and it allocates the reference (in this case a variable) there (first clause). The third clause of get_cell/3 allows to consider all possible configurations of aliasing between references. In essence, get_cell/3 is therefore a CLP implementation of *lazy initialization*.

To conclude this section, let us now provide a definition for symbolic execution in terms of the CLP derivation tree of the CLP-translated program extended with built-in operations to handle dynamic memory:

```
                get_cell(H,Ref,Cell) :- var(H), !, H = [(Ref,Cell)|_].
  get_cell([(Ref',Cell')|_],Ref,Cell) :- Ref == Ref', !, Cell = Cell'.
  get_cell([(Ref',Cell')|_],Ref,Cell) :- var(Ref), var(Ref'), Ref = Ref', Cell = Cell'.
          get_cell([_|RH],Ref,Cell) :- get_cell(RH,Ref,Cell).

            set_cell(H,Ref,Cell,H') :- var(H), !, H' = [(Ref,Cell)|H].
   set_cell([(Ref',_)|H],Ref,Cell,H') :- Ref == Ref', !, H' = [(Ref,Cell)|H].
set_cell([(Ref',Cell')|H'],Ref,Cell,H) :- H = [(Ref',Cell')|H''],
                                          set_cell(H',Ref,Cell,H'').
```

Figure 2.5: Redefining heap operations for symbolic execution [GZAP10]

Definition 2.1.2 (Symbolic Execution)**.** *Let M be a method, m be its corresponding predicate from its associated CLP-translated program P, and P' be the union of P and the set of predicates in Figure 2.3. The symbolic execution of m is the CLP derivation tree, denoted as $\mathcal{T}_m$, with root $m(In, Out, H_{in}, H_{out}, E)$ and initial constraint store $\theta = \{\}$ obtained using P'.*

2.1.4 Test Case Generation

When handling realistic programs, it is well-known that the symbolic execution tree to be explored is in general infinite. This is because iterative constructs such as loops and recursion, whose number of iterations depend on input arguments, usually induce an infinite number of execution paths when executed with symbolic input values. It is therefore essential to establish a *termination criterion*. In the context of TCG, termination is usually ensured by a *coverage criterion*.

Coverage Criteria

A *coverage criterion* guarantees that the set of generated paths remains finite and serves as a measure to determine the adequacy or quality of the generated test suite. Since our focus is on white-box(i.e., structural) TCG, the coverage criteria that we will consider are also *structural*.

Let us review some of the most relevant structural coverage criteria (for a throughout survey, see e.g., [ZHM97]).

- *Statement coverage.* Also known as line coverage or segment coverage, it requires each instruction of the program to be covered by the test suite. In spite of its popularity, this criterion presents severe limitations regarding conditional and loop statements.

- *Branch coverage.* Also known as decision coverage or edge coverage, this criterion requires all conditional statements in the program to be evaluated both to *true* and *false*. Although finer than statement coverage, this criterion is still weak regarding composed conditional statements.

- *Path coverage.* Requires that every possible trace through a given part of the code is executed. This criterion is finer than the previous ones. However, full path coverage,

due to infinite number of paths produced by loops or recursion, is usually not finitely applicable, and therefore impractical.

In this thesis, we focus on yet another coverage criterion, which resembles *path coverage* but unlike the latter, ensures finite applicability by imposing a limit on the number and on the length of the paths to be considered:

- *Block-k coverage.* Requires that every possible trace through a given part of the code is executed, with each block in the control flow graph of the program being visited *at most k* times. The block-k coverage is a refinement of the classical *loop-k* coverage more suitable for bytecode languages (see [AGZP09] for details).

Later on, in Chapter 5 we will further refine these coverage criteria with the notion of *selection*.

Finite symbolic execution tree, test case, and TCG

Let us now establish definitions for key concepts of our approach:

Definition 2.1.3 (Finite symbolic execution tree, test case, and TCG)**.** *Let m be the corresponding predicate for a method M in a CLP-translated program P, and let $\mathcal{C}$ be a termination criterion.*

- *$\mathcal{T}_m^{\mathcal{C}}$ is the finite and possibly incomplete symbolic execution tree of m with root $m(In, Out, H_{in}, H_{out}, EF, T)$ w.r.t. $\mathcal{C}$. Let B be the set of successful (terminating) paths of $\mathcal{T}_m^{\mathcal{C}}$.*
- *A test case for m w.r.t. $\mathcal{C}$ is a 6-tuple of the form: $\langle \sigma(In), \sigma(Out), \sigma(H_{in}), \sigma(H_{out}), \sigma(EF), \sigma(T), \theta \rangle$, where σ and θ are, resp., the set of bindings and the constraint store associated to b.*
- *TCG is the process of generating the set of test cases obtained for all paths in B.*

Each *test case* produced by TCG represents a class of inputs that will follow the same execution path.In a subsequent stage, it is possible to produce actual values from the obtained constraint stores (e.g., by using labeling mechanisms in standard *clpfd* domains) therefore obtaining concrete and executable test cases. However, this is not an issue of this work and we will hence comply with the above abstract definition of *test case*.

Table 2.1: Test cases for method `remAll`

N	Input	Output	Constraint	EF
1	this *(free)* l.first = null	this *(free)* l.first = null	$\emptyset$	ok
2	this.first = null l.first→(A)→ null	this.first = null l.first→(A)→ null	$\emptyset$	ok
3	this.first →(A)→ null l.first →(B)→ null	this.first →(A)→ null l.first →(B)→ null	$\{A \neq B\}$	ok
4	this.first →(A)→ null l.first →(A)→ null	this.first = null l.first →(A)→ null	$\emptyset$	ok
5	this *(free)* l → null	-	$\emptyset$	exc
6	this.first →(A)→ null l = this	this.first = null l = this	$\emptyset$	ok
7	this.first→(A)→ null l.first	this.first = null l.first →(A)→ null	$\emptyset$	ok

Example 2 The test suite generated for method `remAll` for a block-2 coverage criterion is shown in Table 2.1. The first 5 cases are generated without considering aliasing of references, and by doing so, the last two cases are also generated. Let us explain a couple of these cases. Case 3 in Table 2.1 corresponds to the path in which the `this` list contains one element, and so does the input argument list `l`. The constraint $\{A \neq B\}$ indicates that fields `this.first.data` and `l.first.data` can not be equal. The output state of this symbolic execution path indicates that the heap remains unchanged. Let us observe now case 4 in the table. The input state is the almost same as in case 3, but here, the symbolic variables corresponding to `this.first.data` and `l.first.data` are unified (variable A), meaning that `this.first` and `l.first` are aliased. In the output state, notice that one node from the `this` list has been removed. Finally, as mentioned before, by solving the constraint system and applying labeling on the variables involved, concrete inputs can be obtained. □

2.1.5 The PET System

PET (Partial Evaluation-based Test case generator) is a system that implements the CLP-based TCG framework described in this chapter. It is is fully implemented in SWI-Prolog [WSTL12] and uses the CLP(FD) library [Tri12] (Constraint Logic Programming over Finite Domains) as constraint solver. The system is available for download and for online use through its web interface at `http://costa.ls.fi.upm.es/pet`. Moreover, an Eclipse plugin called jPET [ACFM+11] supports full sequential Java and provides important features like interactive test case visualization, trace highlighting and parsing of method preconditions, e.g., written in JML.

Chapter 3

Compositional Test Case Generation

This chapter incorporates compositional reasoning into the CLP-based TCG framework introduced in Chapter 2. Moreover, a novel compositional approach that leverages program specialization is presented. Our results have been published in:

> Elvira Albert and Miguel Gómez-Zamalloa and José Miguel Rojas and Germán Puebla. **Compositional CLP-based Test Data Generation for Imperative Languages.** In *LOPSTR 2010, 20th International Symposium on Logic-Based Program Synthesis and Transformation*, volume 6564 of *Lecture Notes in Computer Science*, pages 99–116. Springer, July 2010.

> José Miguel Rojas and Corina S. Păsăreanu. **Compositional Symbolic Execution through Program Specialization.** In *BYTECODE 2013, 8th Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, March 2013. An extended version of this work is, by the time of writing this dissertation, under consideration for publication in the Journal *Science of Computer Programming*.

3.1 Introduction

Scalability is a major challenge in symbolic execution and test case generation [PV09, CGK+11]. The large number of paths that need to be explored and the large size of the

constraints that must be checked often compromise the effectiveness of symbolic execution for software testing in practice. Compositional reasoning is a general purpose methodology that has been used with success in the past to scale up static analysis and software verification techniques. The main idea is to analyze each elementary unit (methods or procedures) in the program separately, stowing the results in method or procedure *summaries*. Whole-program results are obtained by incrementally composing and re-utilizing the summaries obtained for each of its parts.

While compositional reasoning has been applied in many areas of static analysis and software verification to alleviate scalability problems, it is less widely used in TCG (some notable exceptions in the context of dynamic testing are [God07, AGT08]). In this chapter, we propose a compositional approach for the CLP-based TCG framework for imperative languages introduced in Chapter 2. Moreover, we also propose a novel partial evaluation-based compositional approach for the symbolic execution engine Symbolic PathFinder [PMB+08]. Both approaches share a *context-insensitive composition strategy* and a basic notion of *method summary*. However, they differ in the algorithms that compute such method summaries, the information that is stored in them and the way they are incrementally composed.

In symbolic execution for TCG, compositionality means that when a method m invokes another method p, for which TCG has already been performed, the execution can *compose* the *test cases* available for p (also known as *method summary* for p) with the current execution state and continue the process, instead of having to symbolically execute p again. By test cases (or method summary), we refer to the set of path constraints obtained by symbolically executing p. Notice that since the symbolic execution tree is in general infinite, a *termination criterion* is essential to ensure finiteness of the process. Then, a method summary is a *finite* set of *summary cases*, one for each terminating path through the symbolic execution tree of the method. Intuitively, a summary can be regarded as a complete specification of the method for a certain termination criterion, but it is still a partial specification of the method in general.

Compositional TCG has several advantages over traditional non-compositional TCG. First, it avoids repeatedly performing TCG of the same method. Second, components can be tested with higher precision when they are chosen small enough. Third, since separate TCG is done on parts and not on the whole program, total memory consumption may be reduced. Fourth, separate TCG can be performed in parallel on independent computers and the global TCG time can be reduced as well. Furthermore, having a

compositional TCG approach in turn facilitates the handling of *native code*, i.e., code which is implemented in a different language. This is achieved by modeling the behavior of native code as a method summary which can be composed with the current state during symbolic execution in the same way as the test cases inferred automatically by the testing tool are. By treating native code, we overcome one of the inherent limitations of symbolic execution (see [PV09]).

3.2 Compositional CLP-based TCG

The goal of this section is to study the compositionality of the CLP-based approach to TCG of imperative languages presented in the Chapter 2. For simplicity, we do not take aliasing of references into account and simplify the language by excluding inheritance and virtual invocations. However, these issues are orthogonal to compositionality and our approach could be applied to the complete framework of Chapter 2.

3.2.1 Running Example

Figure 3.1 shows the Java source of our running example and Figure 3.2 shows the CLP-translated version of method `simp` obtained from the Java bytecode. The main features that can be observed from the translation are: (1) All clauses contain input and output arguments and heaps, and an exception flag. Reference variables are of the form `r(V)` and we use the same variable name `V` as in the program. E.g., argument `Rs` of `simp` corresponds to the method input argument (and can be `null` or a valid reference). (2) Java exceptions are made explicit in the translated program, e.g., the second clauses for predicates `simp` and `loopbody2` capture the null-pointer exception (`NullPointerException`) . (3) Conditional statements and iteration in the source program are transformed into guarded rules and recursion in the CLP program, respectively, e.g., the for-loop has been converted to the recursive predicate `loop`. (4) Methods (like `simp`) and intermediate blocks (like `r1`) are uniformly represented by means of predicates and are not distinguishable in the translated program.

Heap Normalization

An important point to note is that, in the remainder of this section, we assume that all fields of an object are present in the heap and appear in the same order as they are

```
class Rational {
    int n; int d;
    void simplify() {
        int gcd = Arithmetics.gcd(n,d);
        n = n/gcd;
        d = d/gcd;
    }
    Rational[] simp(Rational[] rs) {
        int length = rs.length;
        Rational[] oldRs = new Rational[length];
        arraycopy(rs,oldRs,length);
        for (int i = 0; i < length; i++)
            rs[i].simplify();
        return oldRs;
    }
}
class Arithmetics {
    static int abs(int x) {
        if (x >= 0) return x;
        else return -x;
    }
    static int gcd(int a,int b) {
        int res;
        while (b != 0) {
            res = a%b;
            a = b;
            b = res;
        }
        return abs(a);
   }
}
```

Figure 3.1: Compositional TCG Example: Java source code

declared in the program; we say that the heap is "normalized". This is accomplished by updating calls to predicate normalize/2 within get_field/3 and set_field/4 (as shown in Figure 3.3), which initializes the list of fields producing the corresponding template list if it has not been initialized yet. Note that this initialization is only produced the first time a call to get_field/3 or set_field/4 is performed on an object. In contrast, in [GZAP10] the list of fields of an object can be partial (not all fields are present but just those that have been already accessed in the program) and is not ordered (fields occur in the order they are accessed during the corresponding execution). The need for this normalization is further motivated in the next section.

```
simp([r(Rs)],[Ret],H0,H3,EF) :-
    length(H0,Rs,Length),
    Length #>= 0,
    new_array(H0,'Rational',Length,OldRs,H1),
    arraycopy([r(Rs),r(OldRs),Length],[],H1,H2,EFp),
    r1([EFp,r(Rs),r(OldRs),Length],[Rt],H2,H3,EF).
simp([null],_,Hin,Hout,exc(ERef)) :-
    new_object(Hin,'NullPointerException',ERef,Hout).
r1([ok,Rs,OldRs,Length],[Ret],H1,H2,EF) :-
    loop([Rs,OldRs,Length,0],[Ret],H1,H2,EF).
r1([exc(ERef),_,_,_],_,H,H,exc(ERef)).
loop([_,OldRs,Length,I],[OldRs],H,H,ok) :-
    I #>= Length.
loop([Rs,OldRs,Length,I],[Ret],H1,H2,EF) :-
    I #< Length,
    loopbody1([Rs,OldRs,Length,I],[Ret],H1,H2,EF).
loopbody1([r(Rs),OldRs,Length,I],[Ret],H1,H2,EF) :-
    length(H1,Rs,Length),
    Length #>= 0,
    I #< Length,
    get_array(H1,Rs,I,RSi),
    loopbody2([r(Rs),OldRs,Length,I,Rsi],[Ret],H1,H2,EF).
loopbody2([Rs,OldRs,Length,I,r(Rsi)],[Ret],H1,H3,EF) :-
    simplify([r(Rsi)],[],H1,H2,EFp),
    loopbody3([EFp,Rs,OldRs,Length,I],[Ret],H2,H3,EF).
loopbody2([_,_,_,_,null],_,H1,H2,exc(ERef)) :-
    new_object(H1,'NullPointerException',ERef,H2).
loopbody3([ok,Rs,OldRs,Length,I],[Ret],H1,H2,EF) :-
    Ip #= I+1,
    loop([Rs,OldRs,Length,Ip],[Ret],H1,H2,EF).
loopbody3([exc(ERef),_,_,_,_],_,H,H,exc(ERef)).
```

Figure 3.2: Compositional TCG Example: CLP-translated code

Symbolic Execution of Running Example

Let us show with an example how symbolic execution of our running example works. Notice that for simplicity, we do not take aliasing of references into account and simplify the language by excluding inheritance and virtual invocations. However, these issues are orthogonal to compositionality and our approach could be applied to the complete framework of [GZAP10].

Consider the branch of the symbolic execution tree of method `simp` which starts from $\texttt{simp(A}_{\text{in}}\texttt{,A}_{\text{out}}\texttt{,H}_{\text{in}}\texttt{,H}_{\text{out}}\texttt{,EF)}$, the empty state $\phi_0 = \langle \emptyset, \emptyset \rangle$ and which (by ignoring the call to `arraycopy` for simplicity) executes the predicates $\texttt{simp}_1 \rightarrow \mathsf{length} \rightarrow \geq \rightarrow \mathsf{new\_array} \rightarrow$

```
   get_field(H,Ref,FSig,V) :- get_cell(H,Ref,Ob), FSig = C:FN,
                              Ob = object(T,Fields), subclass(T,C),
                              normalize(C,Fields), member_det(field(FN,V),Fields).
set_field(H,Ref,FSig,V,H') :- get_cell(H,Ref,Ob), FSig = C:FN, Ob = object(T,Fields),
                              subclass(T,C), normalize(C,Fields),
                              replace_det(Fields,field(FN,_),field(FN,V),Fields'),
                              set_cell(H,Ref,object(T,Fields'),H').
```

Figure 3.3: Heap normalization. Updated `get_field` and `set_field` from Figure 2.3

$\texttt{r1}_1 \rightarrow \texttt{loop}_1 \rightarrow \geq \rightarrow \texttt{true}$. The subindex 1 indicates that we pick up the first rule defining a predicate for execution. As customary in CLP, a state ϕ consists of a set of bindings σ and a constraint store θ. The final state of the above derivation is $\phi_f = \langle \sigma_f, \theta_f \rangle$ with $\sigma_f = \{\mathsf{A_{in}} = [\mathsf{r(Rs)}],\ \mathsf{A_{out}} = [\mathsf{r(C)}],\ \mathsf{H_{in}} = [(\mathsf{Rs}, \mathsf{array(T, L, \_)})|\_],\ \mathsf{H_{out}} = [(\mathsf{C}, \mathsf{array('R', L, \_)})|\mathsf{H_{in}}],\ \mathsf{EF} = \mathsf{ok}\}$ and $\theta_f = \{\mathsf{L} = 0\}$. This can be read as "if the array at location Rs in the input heap has length 0, then it is not modified and a new array of length 0 is returned". This derivation corresponds to the first test case in Table 3.3 where a graphical representation for the heap is used. For readability, in the table we have applied the store substitution to both $Heap_{in}$ and $Heap_{out}$ terms.

3.2.2 Method Summaries in CLP-based TCG

The termination of CLP-based TCG is guaranteed with the use of a so-called *coverage criterion* (see Section 2.1.4). Given a method m and a coverage criterion $\mathcal{C}$ we denote by SYMBOLIC-EXECUTION$(m, \mathcal{C})$ the process of generating the minimal execution tree which guarantees that the test cases obtained from it will meet the given coverage criterion. Then, a *method summary* corresponds to the finite representation of its symbolic execution for a certain coverage criterion, as defined next.

Definition 3.2.1 (method summary in CLP-based TCG)**.** *Let $\mathcal{T}_m^C$ be the finite symbolic execution tree of method m obtained by using a coverage criterion C. Let B be the set of successful branches in $\mathcal{T}_m^C$ and $m(Args_{in}, Args_{out}, H_{in}, H_{out}, EF)$ be its root.* A method summary *for m w.r.t. C, denoted $\mathcal{S}_m^C$, is the set of 6-tuples associated to each branch $b \in B$ of the form: $\langle \sigma(In), \sigma(Out), \sigma(H_{in}), \sigma(H_{out}), \sigma(EF), \theta \rangle$, where σ and θ are the set of bindings and constraint store, resp., associated to b.*

Each tuple in a summary is said to be a *(test) case* of the summary, denoted c, and its associated *state* ϕ_c comprises its corresponding σ and θ, also referred to as *context* ϕ_c. Intuitively, a method summary can be seen as a *complete specification* of the method for the considered coverage criterion, so that each summary case corresponds to the *path constraints* associated to each finished path in the corresponding (finite) execution tree. Note that, though the specification is complete for the criterion considered, it will be, in general, a *partial specification* for the method, since the finite tree may contain incomplete branches which, if further expanded, may result in (infinitely) many execution paths.

Example 3 Table 3.1 shows the summary obtained by symbolically executing method `simplify` using the **block-2** coverage criterion: The summary contains 5 cases, which

Table 3.1: Summary of method `simplify`

A_{in}	A_{out}	$Heap_{in}$	$Heap_{out}$	EF	$Constraints$
r(A)		A → (F / 0)	A → (M / 0)	ok	F<0, N=-F, M=F/N
r(A)		A → (F / 0)	A → (1 / 0)	ok	F>0
r(A)		A → (0 / 0)	A → (0 / 0), B → AE	exc(B)	
r(A)		A → (F / G)	A → (M / N)	ok	G<0, F *mod* G=0, K=-G, M=F/K, N=G/K
r(A)		A → (F / G)	A → (M / 1)	ok	G>0, F *mod* G=0, M=F/G

correspond to the different execution paths induced by calls to methods `gcd` and `abs`. For the sake of clarity, we adopt a graphical representation for the input and output heaps. Heap locations are shown as arrows labeled with their reference variable names. Split-circles represent objects of type `R` and fields `n` and `d` are shown in the upper and lower part, respectively. Exceptions are shown as starbursts, like in the special case of the fraction "0/0", for which an arithmetic exception (`AE`) is thrown due to a division by zero. In the method summary examples of Tables 3.2 and 3.3, split-rectangles represent arrays, with the length of the array in the upper part and its list of values (in Prolog syntax) in the lower one. □

In a subsequent stage, it is possible to produce actual values from the obtained path constraints (e.g., by using labeling mechanisms in standard *clpfd* domains) therefore ob-

```
compose_summary(Call) :-
    Call =..[M,A_in,A_out,H_in,H_out,EF],
    summary(M,SA_in,SA_out,SH_in,SH_out,SEF,σ),
    SA_in = A_in, SA_out = A_out, SEF = EF,
    compose_hin(H_in,SH_in),
    compose_hout(H_in,SH_out,H_out),
    load_store(σ).
compose_hin(_,SH) :- var(SH), !.
compose_hin(H,[(R,Cell)|SH]) :-
    get_cell (H,R,Cell'), Cell' = Cell,
    compose_hin(H,SH).

compose_hout(H,SH,H) :- var(SH), !.
compose_hout(H_in,[(Ref,Cell)|SH_out],H_out) :-
    set_cell (H_in,Ref,Cell,H'),
    compose_hout(H',SH_out,H_out).
```

Figure 3.4: Compositional CLP-based TCG: The composition operation

taining executable test cases. However, this is not a goal of this work and therefore we will rely only on method summaries in what follows.

3.2.3 Compositional Symbolic Execution

Let us assume that during the symbolic execution of a method m, there is a method invocation to p (i.e. a predicate call $p(In, Out, H_{in}, H_{out}, E)$) within a state ϕ. In the context of our CLP approach, the challenge is to define a composition operation so that, instead of symbolically executing p, its previously computed summary $\mathcal{S}_p$ can be reused. For this, TCG for m should produce the same results regardless of whether we use a summary for p or we symbolically execute p within TCG for m, in a non-compositional way.

Figure 3.4 shows such a composition operation (predicate compose_summary/1). The idea is therefore to replace, during symbolic execution, every method invocation to p by a call compose_summary($p(\ldots)$) when there is a summary available for it. Intuitively, given the variables of the call to p, with their associated state ϕ, compose_summary/1 produces, on backtracking, a branch for each *compatible* case $c \in \mathcal{S}_p$, composes its state ϕ_c with ϕ and produces a new state ϕ' to continue the symbolic execution with. We

assume that the summary for a method p is represented as a set of facts of the form summary(p,SA_{in},SA_{out},SH_{in},SH_{out},SEF,θ). Roughly speaking, state ϕ_c is *compatible* with ϕ if: 1) the bindings and constraints on the arguments can be conjoined, and 2) the structures of the input heaps match. This means that, for each location which is present in both heaps, its associated cells match, which in turn requires that their associated bindings and constraints can be conjoined. Note that compatibility of a case is checked on the fly, so that if ϕ is not compatible with ϕ_c some call in the body of compose_summary/1 will fail.

As it can be observed by looking at the code of compose_summary/1, the input and output arguments, and the exception flags are simply unified, while the constraint store θ is trivially incorporated by means of predicate load_store/1. However, the heaps require a more sophisticated treatment, mainly due to the underlying representation of sets (of objects) as Prolog lists. Predicate compose_hin/2 composes the input heap of the summary case SH_{in} with the current heap H_{in}, producing the composed input heap in H_{in}. To accomplish this, compose_hin/2 traverses each cell in SH_{in}, and: 1) if its associated reference is not present in H_{in} (first rule of get_cell/3 succeeds), it is added to it, 2) if is is present in H_{in} (second rule of get_cell/3 succeeds) then the cells are unified. This is possible since we are assuming that every object that arises in the heap during symbolic execution has its list of fields normalized. This allows using just unification (Cell = Cell') for the aim of matching cells.

Similarly, compose_hout/3 composes the output heap of the summary case SH_{out} with the current heap H_{in}, producing the composed output heap in H_{out}. As can be seen in Figure 3.4, compose_hout/3 traverses each cell in SH_{out} and, if its associated reference is not present in H (first rule of set_cell/4 succeeds), then it is added to it. Otherwise (second rule of set_cell/4 succeeds) it overwrites the current cell. In both cases, set_cell/4 produces a new heap H' which is passed as first argument to the recursive call to compose_hout/3. This process continues until there are no more cells in SH_{out}, in which case the current heap is returned. Again this is possible thanks to the normalization of object fields.

As noticed before, further features of imperative languages not considered in this work, such as inheritance and pointer aliasing, can be handled by compose_summary/1 for free by just using the corresponding extensions of get_cell/3 and set_cell/4 defined in Chapter 2.

Example 4 When symbolically executing `simp`, the call simplify(A_{in},A_{out},H_{in},H_{out},EF) arises in one of the branches with state σ = {A_{in}=[r(E0)], A_{out}=[], H_{in}=[(0, array('R', L, [E0|_]))),

Table 3.2: Summary of method `arraycopy`

A_{in}	A_{out}	$Heap_{in}$	$Heap_{out}$	EF	$Constraints$
[X,Y,0]		H	H	ok	$\emptyset$
[r(A),null,Z]		A→ L [V|_]	A→ L [V|_] B → NPE	exc(B)	Z>0, L>0
[null,Y,Z]		H	A → NPE	exc(A)	Z>0
[X,Y,Z]		H	A → AE	exc(A)	Z<0
[r(A),r(B),1]		A→ L1 [V|_] B→ L2 [V2|_]	A→ L1 [V|_] B→ L2 [V|_]	ok	L1>1, L2>0

$(\mathsf{Rs}, \mathsf{array}('\mathsf{R}', \mathsf{L}, [\mathsf{E0}|\_]))|\mathsf{RH_{in}}]\}$ and $\theta = \{\mathsf{L} \geq 0\}$. The composition of this state with the second summary case of `simplify` succeeds and produces the state $\sigma' = \sigma \cup \{\mathsf{E0} = \mathsf{B}, \mathsf{RH_{in}} = [(\mathsf{B}, \mathsf{ob}('\mathsf{R}', [\mathsf{field}(\mathsf{n}, \mathsf{F}), \mathsf{field}(\mathsf{d}, 0)]))|\_], \mathsf{H_{out}} = [\ldots, (\mathsf{B}, \mathsf{ob}('\mathsf{R}', [\mathsf{field}(\mathsf{n}, 1), \mathsf{field}(\mathsf{d}, 0)]))|\_]\}$ and $\theta' = \{\mathsf{L} \geq 0, \mathsf{F} > 0\}$. The dots in $\mathsf{H_{out}}$ denote the rest of the cells in $\mathsf{H_{in}}$. □

3.2.4 Approaches to Compositional TCG

In order to perform compositional TCG, two main approaches can be considered:

Context-sensitive. Starting from an entry method m (and possibly a set of preconditions), TCG performs a top-down symbolic execution such that, when a method call p is found, its code is executed from the actual state ϕ. In a context-sensitive approach, once a method is executed, we store the summary computed for p in the context ϕ. If we later reach another call to p within a (possibly different) context ϕ', we first check if the stored context is sufficiently general. In such case, we can adapt the existing summary for p to the current context ϕ' (by relying on the operation in Figure 3.4). At the end of each execution, it can be decided which of the computed (context-sensitive) summaries are stored for future use. In order to avoid the problems of computing summaries which end up being not sufficiently general, in the rest of this chapter we focus in the context-insensitive approach presented below.

Context-insensitive. Another possibility is to perform the TCG process in a context-insensitive way. Algorithm 1 presents this strategy, by abstracting some implementation-related details. Intuitively, the algorithm proceeds in the following steps. First, it com-

putes the call graph (line 3) for the entry method $m_{\mathcal{P}}$ of the program under test, which gives us the set of methods that must be tested. The strongly connected components (SCCs for short) for such graph are then computed in line 4. SCCs are then traversed in reverse topological order starting from an SCC which does not depend on any other (line 4). The idea is that each SCC is symbolically executed from its entry m_{scc} w.r.t. the most general context (i.e., *true*) (line 29). If there are several entries to the same SCC, the process is repeated for each of them. Hence, it is guaranteed that the obtained summaries can always be adapted to more specific contexts.

Algorithm 1 Context-insensitive compositional TCG

Input: Program $\mathcal{P}$, Coverage criterion $\mathcal{C}$

Output: Test suite $\mathcal{T}$ for program $\mathcal{P}$ w.r.t. $\mathcal{C}$

```
1:  procedure BOTTOM-UP-TCG(P, C)
2:      Let m_P be the entry method of P
3:      G ← callGraph(m_P)
4:      SCC ← stronglyConnectedComponents(G)
5:      SCC' ← buildTopologicalOrderList(SCC)
6:      for all scc ∈ SCC' do
7:          for all m_scc ∈ entryMethods(scc) do
8:              S_c^{m_scc} ← SYMBOLIC-EXECUTION(m_scc, C)
9:          end for
10:     end for
11: end procedure
```

In general terms, the advantages of the context-insensitive approach are that composition can always be performed and that only one summary needs to be stored per method. However, since no context information is assumed, summaries can contain more test cases than necessary and can be thus more expensive to obtain. In contrast, the context-sensitive approach ensures that only the required information is computed, but it can happen that there are several invocations to the same method that cannot reuse previous summaries (because the associated contexts are not sufficiently general). In such case, it is more efficient to obtain the summary without assuming any context.

3.2.5 Handling Native Code during Symbolic Execution

An inherent limitation of symbolic execution is the handling of native code, i.e., code implemented in another (lower-level) language. Symbolic execution of native code is not possible since the associated code is not available and it can only be handled as a black box. In the context of hybrid approaches to TCG which combine symbolic and concrete execution, a solution is concolic execution [GKS05] where concrete execution is performed on random inputs and path constraints are collected at the same time; native code is executed for concrete values. Although we believe that such approach could be also adapted to our CLP framework, we concentrate here on a purely symbolic approach. In this case, the only possibility is to model the behavior of the native code by means of specifications. Such specifications can be in turn treated as summaries for the corresponding native methods. They can be declared by the code provider or automatically inferred by a TCG tool for the corresponding language. Interestingly, the composition operator uses them exactly in the same way as it uses the summaries obtained by applying our own symbolic execution mechanism. Let us see an example.

Example 5 Assume that method `arraycopy` is native. A method summary for `arraycopy` can be provided, as shown in Table 3.2, where we have (manually) specified five cases: the first one for arrays of length zero, the second and third ones for null arrays, the fourth one for a negative length, and finally a normal execution of non-null arrays. Now, by using our compositional reasoning, we can continue symbolic execution for `simp` by composing the specified summary of `arraycopy` within the actual context. In Table 3.3, we show the entire summary of method `simp` for a **block-**2 coverage criterion obtained by relying on the summaries for `simplify` and `arraycopy` shown before. □

A practical question is how method summaries for native code should be provided. A standard way is to use assertions (e.g., in JML in the case of Java) which could be parsed and easily transformed into our Prolog syntax.

3.2.6 Compositionality of Different Coverage Criteria

Though in Section 3.2.3 we have presented a mechanism for reusing existing summaries during TCG, not all coverage criteria behave equally well w.r.t. compositionality. A coverage criterion C is *compositional* if whenever performing TCG of a method m w.r.t.

Table 3.3: Summary of method `simp`

A_{in}	A_{out}	$Heap_{in}$	$Heap_{out}$	EF	$Constraints$
r(A)	r(B)	A→[0|▯]	A→[0|▯] B→[0|▯]	ok	∅
null	X	H	A→NPE	exc(A)	∅
r(A)	r(C)	A→[1|r(B)] B→(F|0)	A→[1|r(B)] B→(M|0) C→[1|r(B)]	ok	F<0, K=-F, M=F/K
r(A)	r(C)	A→[1|r(B)] B→(F|0)	A→[1|r(B)] B→(1|0) C→[1|r(B)]	ok	F>0
r(A)	X	A→[1|r(B)] B→(0|0)	A→[1|r(B)] B→(0|0) C→[1|r(B)] D→AE	exc(D)	∅
r(A)	r(C)	A→[1|r(B)] B→(F|G)	A→[1|r(B)] B→(M|N) C→[1|r(B)]	ok	G<0, F *mod* G=0, K=-G, M=F/K, N=G/K
r(A)	r(C)	A→[1|r(B)] B→(F|G)	A→[1|r(B)] B→(M|1) C→[1|r(B)]	ok	G>0, F *mod* G=0, M=F/G
r(A)	X	A→[1|[null]]	A→[1|[null]] C→[1|[null]] B→NPE	exc(B)	∅

C, if we use a previously computed summary for a method p w.r.t. C in a context which is sufficiently general, the results obtained for m preserve criterion C. In other words, if a criterion is compositional, we do not lose the required coverage because of using summaries.

Unfortunately, not all coverage criteria are compositional. For example, statement coverage is not compositional, as illustrated in the example below.

Example 6 Consider the following simple method:

```
p(int a,int b) { if (a > 0 || b > 0) S; }
```

where `S` stands for any statement, and the standard shortcut semantics for Java boolean expressions is used. This means that as soon as the boolean expression has a definite *true* or *false* value, it is not further evaluated. In our case, once the subexpression `a > 0` takes the value *true*, the whole condition definitely takes the value *true*, the subexpression `b > 0` is not evaluated, and `S` is executed.

If assuming the top (most general) context, a summary with a single case containing $a > 0$ is sufficient to achieve statement coverage. Consider now that `p` is called from an outer scope with a more restricted context in which $a \leq 0$ holds. Then, using such summary instead of performing symbolic execution of `p` does not preserve statement

coverage, since it is not guaranteed that statement `S` is visited. It depends on the particular value picked for `b` for testing, which is unconstrained in the summary. If the value for `b` is picked to be greater than zero, statement coverage is satisfied, but not otherwise. Note that by considering a context where $a \leq 0$ from the beginning, a summary with a single case with constraint $\{b > 0\}$ would be computed instead. □

As this example illustrates, a challenge in compositional reasoning is to preserve coverage when using summaries previously computed for a context ϕ which is sufficiently general, but not identical to ϕ', the one which appears during the particular invocation of the method. More precisely, compositionality of coverage criteria requires that the following property holds: given a summary S obtained for p in a context ϕ w.r.t. C, a summary S' for a more restricted context ϕ' can be obtained by removing from S those entries which are incompatible with context ϕ'.

For instance, the **block-k** coverage criterion used in the examples of this chapter is compositional. This is because there is a one to one correspondence between entries in the summary and non-failing branches in the symbolic execution tree obtained for ϕ. If we are now in a more restricted context ϕ', those branches which become failing branches are exactly those whose precondition is incompatible with ϕ'. Therefore, we obtain identical results by working at the level of the symbolic execution tree or that of the entries in the summary.

In fact, we can classify coverage criteria into two categories: local and global. A criterion is *local* when the decision on whether the path should be included or excluded in the summary can be taken by looking at the corresponding path only. A criterion is *global* when we need to look at other paths before determining whether to include a given path in the summary or not. As examples, both block-k and depth-k are local, whereas statement coverage is global: a given path is not needed if it does not visit statements not covered by any of the previously considered paths.

In general, local criteria are compositional, whereas global criteria are not. The reason for such non-compositionality is that we may decide not to include certain paths which are not required for achieving the criterion in a context ϕ but which are needed in a more specific context ϕ', since other paths which achieved the criterion are now incompatible with the context and have been removed from the summary.

3.2.7 Reusing Summaries Obtained for Different Criteria

In the discussion about compositionality presented in Section 3.2.6, we assumed that the same criterion C is used both for m and the summary of p. Another interesting practical question is: given a summary computed for p w.r.t. a criteria C', can we use it when computing test cases for m w.r.t. a criteria C?

As an example, by focusing on **block-k**, assume that C' corresponds to $k = 3$ and C to $k = 2$. We can clearly adapt the summaries obtained for $k = 3$ to the current criteria $k = 2$. Even more, if one uses the whole summary for $k = 3$, the required coverage $k = 2$ is ensured, although unnecessary test cases are introduced. On the contrary, if C' corresponds to $k = 2$ and C to $k = 3$ the coverage criterion is not preserved for m. However, this can be acceptable when we would like to perform TCG of different levels to different parts of the code, depending on their size, relevance, level of trust, etc. For instance, code which is safety-critical can be more exhaustively tested using coverage criteria that ensures a higher degree of coverage. In contrast, code which is more stable (e.g., library methods) can be tested using more lightweight coverage criteria.

Another issue is what happens when the existing summary is for a completely unrelated criterion. There, it is not possible to guarantee that the criterion for m is guaranteed. Nevertheless, such summaries can be used for obtaining information on the output states of p in order to be able to continue the symbolic execution of m after the calls to p terminate. This is especially relevant when p is native, since in that case performing symbolic execution instead of using the summary is not an alternative.

3.2.8 Experimental results

We have implemented our proposed compositional approach using the context-insensitive algorithm in Section 3.2.4 within PET (see Section 2.1.5). In this section we report on some experimental results which aim at demonstrating the applicability and effectiveness of compositional TCG. As benchmarks, we consider a set of classes implementing some traditional data structures ranging from simple stacks and queues to sorted and non-sorted linked lists (both singly and doubly linked lists), priority queues and binary search trees. Some benchmarks are taken from the net.datastructures package. Table 3.4 shows the list of benchmarks we have used, together with some static information: the number of methods for which we have generated test cases (column **NMs**); the number of reachable classes, methods and Java bytecode instructions (columns **RCs**, **RMs** and **RIs**) (excluding code

Table 3.4: Compositional CLP-based TCG: Benchmarks

Benchmark	**NMs**	**RCs**	**RMs**	**RIs**	$\mathbf{T}_{dec}$
NodeStack	6	3	12	94	32
ArrayStack	7	3	11	103	39
NodeQueue	6	3	15	133	48
NodeList	19	9	33	449	277
DoublyLinkedList	13	2	20	253	107
SortedListInt	6	2	8	155	58
SLPriorityQueue	12	14	42	515	330
BinarySearchTree	14	15	54	717	620
SLIntMaxNode	9	2	12	232	99
SearchTreeInt	9	1	10	189	75

of the Java libraries) and the time taken by PET to decompile the bytecode to CLP ($\mathbf{T}_{dec}$), including parsing and loading all reachable classes. All times are in milliseconds and are obtained as the arithmetic mean of five runs on an Intel(R) Core(TM)2 Quad CPU Q9300 at 2.5GHz with 1.95GB of RAM, running Linux 2.6.26 (Debian lenny).

Table 3.5 aims at comparing the performance and effectiveness of the compositional approach against the non-compositional one by using two coverage criteria, **block-**2 and **block-**3. In general the latter one expands much further the symbolic execution tree, thus allowing us to compare scalability issues. For each run, we measure, for both the compositional and non-compositional approaches, the time taken by PET to generate the test cases (column $\mathbf{T}_{tcg}$), the number of obtained test cases (**N**), the number of unfolding steps performed during the symbolic execution process (**US**), and the *code coverage* (**CC**). In the case of the compositional approach we also measure the number of generated summaries (**SG**) and summary compositions performed (**SC**). The code coverage measures, given a method, the percentage of bytecode instructions which are exercised by the obtained test cases, among all reachable instructions (including all transitively called methods). This is a common measure in order to reason about the effectiveness of the TCG. As expected, the code coverage is the same in both approaches, and so is the number of obtained test cases. Otherwise, this would indicate a bug in the implementation. The last column ($\Delta\mathbf{T}_{tcg}$) shows the speedup of the compositional approach computed as X/Y where X is the $\mathbf{T}_{tcg}$ value of the non-compositional approach and Y the $\mathbf{T}_{tcg}$ value

Table 3.5: Compositional CLP-based TCG: Experimental results

Benchmark	**Cov.**	**Non-Compositional**				**Compositional**						**Gains**
	Crit.	$\mathbf{T}_{tcg}$	**N**	**US**	**CC**	$\mathbf{T}_{tcg}$	**N**	**US**	**CC**	**SG**	**SC**	$\Delta\mathbf{T}_{tcg}$
NodeStack	**block-**2	6.7	9	112	100%	10.3	9	94	100%	11	12	0.65
	block-3	6.7	9	112	100%	6.7	9	94	100%	12	11	1.00
ArrayStack	**block-**2	13.0	15	203	100%	13.0	15	161	100%	10	10	1.00
	block-3	13.3	15	203	100%	13.0	15	161	100%	10	10	1.03
NodeQueue	**block-**2	10.3	15	160	100%	13.0	15	149	100%	13	13	0.79
	block-3	10.3	15	160	100%	13.3	15	149	100%	13	13	0.78
NodeList	**block-**2	120.0	102	1856	96%	103.7	102	684	96%	52	28	1.16
	block-3	130.0	102	1856	96%	110.3	102	684	96%	28	52	1.18
DoublyLinkedList	**block-**2	160.3	116	5060	99%	130.3	116	1967	99%	39	12	1.23
	block-3	200.0	133	6068	99%	160.3	133	2164	99%	12	51	1.25
SortedListInt	**block-**2	133.3	49	2781	100%	77.0	49	775	100%	22	5	1.73
	block-3	4233.7	447	63593	100%	1713.0	447	6542	100%	5	176	2.47
SLPriorityQueue	**block-**2	533.3	266	5828	94%	566.7	266	1195	94%	114	62	0.94
	block-3	786.7	350	7910	94%	856.3	350	1369	94%	62	128	0.92
BinarySearchTree	**block-**2	693.0	307	9210	96%	706.3	307	5203	96%	463	63	0.98
	block-3	1526.7	559	21162	96%	1486.7	559	11143	96%	63	733	1.03
SLIntMaxNode	**block-**2	299.7	109	5012	100%	230.0	109	1172	100%	27	8	1.30
	block-3	28150.3	2957	328096	100%	20419.7	2957	34770	100%	8	520	1.38
SearchTreeInt	**block-**2	279.7	90	5707	100%	143.0	90	726	100%	33	8	1.96
	block-3	45003.0	5140	553536	100%	20236.7	5140	12980	100%	8	232	2.22

of the compositional one.

By looking at the gains, we observe that the compositional approach outperforms the non-compositional one in most benchmarks. Let us observe also that, in general, the further the symbolic execution tree is expanded (i.e., when the **block-**3 criterion is used), the higher the gains are. There are, however, cases where the performance of the compositional approach is equal to, or even worse than, that of the non-compositional one. Importantly, those cases usually correspond to very simple methods whose complexity is not enough so that the overhead of applying the compositional scheme pays off.

After a careful study of the obtained results, we conclude that there can be many

factors that influence the performance of the compositional approach. The most important ones are: the complexity of the program under test (especially that of its call graph and its strongly connected components), the constraint solving library, and, the kind of constraint-based operations performed and, in particular, whether they are arithmetic constraints or heap related operations. In this direction, we have carried out the following experiment. Given a method p which simply calls repeatedly (three or four times) method q, we consider two versions of it: q_1 which performs both heap and arithmetic operations, and q_2 with arithmetic operations only. This allows us to detect whether the kind of constraint-based operations performed influences the performance of compositional TCG. As expected, with q_1 compositional TCG improves notably (two or even three times faster) over non-compositional TCG. Surprisingly, with q_2 the performance of compositional TCG is basically the same (or even worse). This explains the lack of improvements in some of our benchmarks. The reason is that the cost of the TCG process is totally dominated by the constraint solving operations, in this case by the `clpfd` solver. Interestingly, if we simplify by hand the constraints on the summary of q_2 we do get significant improvements with compositional TCG. This illustrates the flexibility of the approach in the sense that, provided that a summary is available for a method, it can be worth spending resources in simplifying the constraint stores. Once this is done, they will be used every time a call to the method is found and thus producing a performance improvement. On the other hand, this demonstrates that compositional TCG can significantly benefit from more efficient constraint solving libraries.

Overall, our experimental results support our claim that compositional TCG improves over non-compositional TCG in terms of scalability. Let us observe that, in general, the benchmarks where the improvements are higher correspond to those for which a larger number of unfolding steps (column **US**) is required. We have seen also that such improvement could be higher by using a more efficient constraint solving library. It remains as future work to experiment in such direction.

3.3 Compositional TCG through Program Specialization

Although compositional reasoning has been previously applied quite effectively in the context of symbolic execution and TCG, the technique can become expensive in the presence

of program operations that update the heap. These operations can not be easily encoded and checked as logical constraints, as in the summaries proposed in [AGT08, God07], where they are not treated. On the other hand, in the work presented in Section 3.2, we explicitly encode the input and output heap in the procedure summaries, which entails a very complex and expensive compositional operator that not only checks compatibility at invocation point, as all compositional techniques do, but also needs to synthesize the new program state (with the new heap) to continue execution.

We now propose an alternative approach to compositional symbolic execution in the presence of heap updates, which is based on *partial evaluation* (PE) [JGS93]. PE, also known as *program specialization*, is a well known technique for automatically specializing a program with respect to some of its inputs. In our approach a method summary consists of a set of summary cases, corresponding to all the symbolic paths through the method; each summary case contains the path condition and heap constraints that enable a particular symbolic execution path, together with a "path-specialized" version of the method code, obtained through PE. Thus the operation of composing a method summary with the actual calling context consists of re-executing a summary case according to the specialized code which naturally reconstructs the heap without the need of keeping an explicit representation in the summary.

Symbolic execution and program specialization have been used together in previous work. In [CPPGM91], symbolic execution is used to achieve program specialization of Ada programs. In contrast, our approach uses program specialization to achieve compositional symbolic execution of object-oriented Java Bytecode. In [BHJ10], symbolic execution and partial evaluation are interleaved to speed up a logic-based verification framework. However that work does not address compositionality in symbolic execution, which is the focus here.

Using program specialization techniques helps optimize the symbolic execution process by enabling preemptive pruning of unfeasible branches and allowing deeper exploration of the symbolic execution state space. Last but not least, a byproduct of our program specialization-based compositional approach is that the code stored in each method summary case can be further used as a specialized, more efficient version of the original method.

We have implemented our compositional analysis in Symbolic PathFinder [PMB+08] for the symbolic execution of Java bytecode programs. We report on encouraging results on three case studies; more experimental evaluation is planned for future work.

3.3.1 Symbolic PathFinder

Symbolic PathFinder (SPF, [PR10]) is a symbolic execution framework built on top of the Java PathFinder (JPF) model checking toolset for Java bytecode analysis [VM05]. SPF implements a bytecode interpreter that replaces the standard, concrete execution semantics of bytecodes with a non-standard symbolic execution. Non-deterministic choices in branching conditions are handled by means of JPF's *choice generators*. JPF's listeners are used to monitor and influence the symbolic execution and to collect and print its results. Symbolic execution of looping programs may result in an infinite symbolic execution tree; for this reason, SPF is run with a user-specified bound on the search depth. Mixed concrete-symbolic solving allows supporting native code. Moreover, SPF relies on off-the-shelf decision procedures to check satisfiability of path conditions. It backtracks if the path condition is unsatisfiable and otherwise it can generate solutions for satisfiable path conditions to be used as test inputs. Finally, SPF uses *lazy initialization* (see Section 2.1.3) to handle dynamic input data structures (e.g., lists and trees), which enables the systematic exploration of all possible heap configurations during symbolic execution.

Example 7 Figure 3.5 shows the Java source code and bytecode for a method `m`, together with the symbolic execution tree (sketched). The example consists of two nested conditional statements. The first one checks the nullity of a reference argument `x` and the second one evaluates the value of an integer field `a` of `x`. The tree at the right of the figure shows the three possible execution paths of `m`, where transitions are labeled with the set of conditions (initially empty) that need to hold for the execution to follow each path. □

3.3.2 Program Specialization

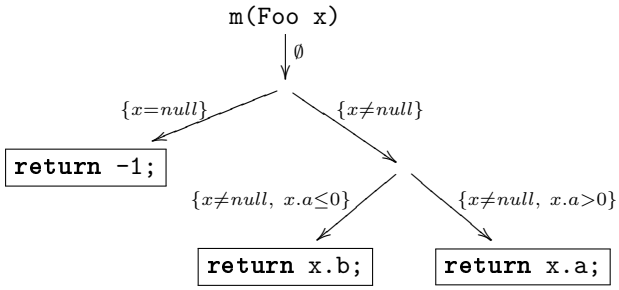
Program Specialization (also known as Partial Evaluation [JGS93]) is a program optimization technique that, given a program p, can produce a so-called residual program $p*$, which is a specialized version of p with respect to some of its inputs. The main benefit of program specialization is speed of execution: the specialized program $p*$ tends to be much faster than the original program p. The technique has many applications, including compiler optimization and program transformation. Figure 3.6 shows a traditional example of partial evaluation. There, a recursive function `f`, which originally takes two arguments x and n and computes x^n, is specialized for $n = 5$. While partial evaluation

```
int m(Foo x) {
  if (x != null)
    if (x.a > 0)
      return x.a;
    else
      return x.b;
  else return -1;
}
```

(a) Java source code

```
 0: aload x
 1: ifnull 11
 2: aload x
 3: getfield a
 4: ifle 8
 5: aload x
 6: getfield a
 7: ireturn
 8: aload x
 9: getfield b
10: ireturn
11: iconst -1
12: ireturn
```

(b) Java bytecode



(c) Symbolic execution (sketch)

Figure 3.5: Example of symbolic execution in Symbolic PathFinder

```
int f(int x, int n) {
  if (n == 0)
    return 1;
  else
    if (even(n))
      return pow(f(x,n/2),2);
    else
      return x * f(x,n-1);
}
```

(a) Original function `f` (computes x^n)

```
int f5(int x) {
  return x * pow(pow(x,2),2);
}
```

(b) Function `f` specialized for $n = 5$ (computes x^5)

Figure 3.6: Program specialization example

is traditionally used by fixing a subset of its input arguments, in this work we *specialize* programs with respect to a particular path condition. To the best of our knowledge, this work is the first attempt to apply program specialization in this way in the context of symbolic execution for software testing.

3.3.3 Compositional Symbolic Execution

We now propose compositional approach for Symbolic PathFinder. We take a context-insensitive strategy, where the methods of a program are processed in an order corresponding to a bottom-up traversal of the program's call graph, starting with the ones

that invoke no other methods and incrementally processing methods whose sub-methods have already been processed until the whole program is analyzed. For each processed method, we use symbolic execution and partial evaluation to compute a *method summary*, which is a succinct representation of all the symbolic paths through the method, and which can be re-used whenever that method is invoked from another method – we say that the two methods are *composed*.

As discussed earlier, when deciding on the order to analyze the methods in a given system, two main strategies can be followed. A context-sensitive or top-down approach may be adequate if one wants to compute only the strictly necessary information (method summaries). However, this approach does not guarantee that reusability of summaries is always possible. On the other hand, a context-insensitive or bottom-up approach ensures that the computed summaries can always be reused, at the price of computing summaries larger than necessary in some cases. As we did in the previous section, we now continue to follow the latter strategy. We leave recursion out of the scope of this chapter.

3.3.4 Method Summaries in SPF

In SPF, the finiteness of the TCG process and hence that of method summaries is ensured by imposing a depth limit in the symbolic execution tree. A method summary is defined as follows.

Definition 3.3.1 (method summary in SPF)**.** *A* method summary *is a finite set of* summary cases *of the form* $\langle PC, HPC, C, S\rangle$*, where PC is the path condition, HPC is the heap path condition, C is a specialized version of the bytecode of* `m`*, and S is the associated composition schedule.*

Let us now describe each of the components of a summary case.

Path Condition The path condition is a conjunction of constraints over the symbolic numeric input arguments of the method or numeric fields in the heap. These constraints are generated and checked for satisfiability (using an off-the-shelf solver) during the symbolic execution of conditional bytecode instructions such as `ifle`, `if_icmpeq`, `if_acmpeq`, `dcmpg`, etc.

Heap Path Condition The heap path condition is a conjunction of constraints over the heap allocated objects in the input data structure. These constraints are generated

by lazy initialization during the symbolic execution of instructions `aload`, `getfield` and `getstatic`. The constraints can have the following form:

- $Ref = null$. Reference Ref points to $null$;
- $Ref \neq null$. Reference Ref is a new object, different from $null$ and not aliased with any other object in the heap, with all its fields made symbolic;
- $Ref_1 = Ref_2$. References Ref_1 and Ref_2 are aliased, i.e., point to the same object in the heap.

These constraints are sufficient to express all the possible aliasing scenarios in the input data structures [KPV03]. By construction, the conjunction of the path condition together with the heap path condition characterizes the inputs that follow the respective path, *including* the behaviour contributed by all the other methods invoked along the path.

Specialized Code The specialized code corresponds to the code executed along a particular path in the method. In the specialized code, all the forms of non-determinism are resolved and the specialized code is guaranteed to contain no branching conditions (see Section 3.3.5). Thus, when a summary is (re-)used during compositional symbolic execution, no expensive constraint solving needs to be performed *during* the execution of the specialized code.

Composition Schedule The composition schedule is a sequence of numbers specifying which summary case to use for each method invocation that appears in the specialized code. It allows for incremental, deterministic composition of method summaries by determining, for each invoke instruction in the specialized code, which case from the invoked method's summary must be composed.

Example 8 Figure 3.7 presents the summary for method `m` in Figure 3.5. The first element corresponds to the case in which `x` is lazily initialized to `null` and the path condition remains empty. The second and third cases correspond to `x` being lazily initialized to a new object, for different PCs. Namely, the second case represents the path corresponding to the *then* branch of the inner if statement and the third case represents the *else* branch. Since `m` does not invoke any other methods, the composition schedules are empty. Notice that the specialized bytecode produced for each summary case is much shorter than that of the original bytecode of `m`. □

Case	PC	HPC	Code	Schedule
0	$\emptyset$	$\{x = null\}$	[`iconst -1, ireturn`]	[ ]
1	$\{x.a > 0\}$	$\{x \neq null\}$	[`aload x, getfield a, ireturn`]	[ ]
2	$\{x.a \leq 0\}$	$\{x \neq null\}$	[`aload x, getfield b, ireturn`]	[ ]

Figure 3.7: Method summary for example from Figure 3.5

Example 9 Consider method `comp` in Figure 3.8, which has an invocation to method `m` in its `if-then` branch. Note that at this point we have already computed the summary for `m`. When building the summary case for `comp` along that particular path, we will record in its composition schedule the particular summary case of `m` that must be used at that invocation point. As in this example, there may be multiple summary cases of `m` that can be used at an invocation point, each one with different (heap)path conditions. All such summary cases for `m` are systematically composed, resulting in multiple summary cases for `comp` as well, with the path conditions for `comp` encoding the constraints along the path in `comp` conjoined with the path conditions in `m`'s relevant summary cases. To wit, the composition schedule of summary case 0 (resp. 1) of `comp` indicates that when the invocation to `m` in the specialized code is reached, the execution will proceed with the specialized code of `m` stored in its 1st (resp. 2nd) summary case. For more clarity, we add the name of the target method as a subscript to each element in the composition schedule.

```
int comp(Foo x) {
  if (x != null) return m(x);
  else throw new Exception();
}
```

Case	PC	HPC	Code	Schedule
0	$\{x.a > 0\}$	$\{x \neq null\}$	[`aload this, aload x, invokevirtual m, ireturn`]	$[1_m]$
1	$\{x.a \leq 0\}$	$\{x \neq null\}$	[`aload this, aload x, invokevirtual m, ireturn`]	$[2_m]$
2	$\emptyset$	$\{x = null\}$	[`new Exception, dup, invokespecial init, athrow`]	[ ]

Figure 3.8: Source code and summary for method `comp`

□

3.3.5 Program Specialization during Symbolic Execution

Algorithm 2 formalizes how we use symbolic execution and partial evaluation to generate method summaries. Procedure SPECIALIZATION is applied to each instruction in the method, during symbolic execution. Conditional instructions (`ifle`, `ifnull`, ...) are left out of the specialized code and the instruction(s) that pushed their operand(s) are also sliced away (function SLICECODE). The intuition is that the conditions in these instructions will be captured and fixed by the path conditions encoded in the summary; these path conditions will be checked only once when the summary is (re-)used. Importantly, notice that the effect of applying function SLICECODE when the last executed instruction is a `getfield` is that the complete ascending reference chain will be followed and sliced away from the specialized code.

For invoke instructions, the summary for the invoked method is used to update the path condition and the composition schedule, using summary composition as described in the next section. Only the invoke instruction is appended to the specialized code.

Return instructions are appended to the specialized code and lead to the completion of a new summary case, which is added to the method summary. The remaining instructions, except the `goto` instruction which is ignored, are just appended to the specialized code.

Table 3.6: Specialized code

Java Bytecode instruction	**Specialized code**
0: aload x	[aload x]
1: ifnull 11	[]
2: aload x	[aload x]
3: getfield a	[aload x, getfield a]
4: ifle 8	[]
8: aload x	[aload x]
9: getfield b	[aload x, getfield b]
10: ireturn	[aload x, getfield b, ireturn]

Example 10 Let us exemplify the algorithm by constructing the specialized code from the third summary case in Figure 3.7. Table 3.6 shows to the left the bytecode instructions

Algorithm 2 Specialization during Symbolic Execution

Input: insn:Instruction, currentState ≡ ⟨ pc, hpc, code, sched ⟩

```
 1: procedure SPECIALIZATION
 2:     switch TYPE(insn) do
 3:         case ConditionalInstruction
 4:             code ← SLICECODE(code,insn)
 5:         case InvokeInstruction
 6:             COMPOSESUMMARY(getInvokedMethod(insn),true)
 7:             code ← APPEND(code,insn)
 8:         case ReturnInstruction
 9:             code ← APPEND(code,insn)
10:             STORESUMMARYCASE(pc,hpc,code,sched)
11:         case GotoInstruction
12:             IGNORE
13:         default
14:             code ← APPEND(code,insn)
15: end procedure
16: function SLICECODE(code:InstructionSequence, insn:Instruction)
17:     if TYPE(insn)≡UnaryConditionalInstruction then
18:         return REMOVELASTPUSH(code)
19:     else                                   ▷ BinaryConditionalInstruction
20:         return REMOVELASTPUSH(REMOVELASTPUSH(code))
21:     end if
22: end function
23: function REMOVELASTPUSH(code:InstructionSequence)
24:     lastInsn ← POP(code)            ▷ removes last instruction from code
25:     if lastInsn ∈ {getfield, arraylength } then
26:         return REMOVELASTPUSH(code)          ▷ follow reference chain
27:     else
28:         return code
29:     end if
30: end function
```

executed along the path and to the right the effect of each instruction in the specialized code. First, observe that `aload`, `getfield` and `ireturn` instructions are merely appended to the specialized code (last case in the algorithm). The second instruction is a conditional one, thus, by the first case of Algorithm 2, the previous *pushing* instruction is cut off. Later on, when the conditional `ifle 8` is executed, the first case of the algorithm activates

a special case in function SLICECODE. Namely, since the previous pushing instruction is a `getfield`, the algorithm traverses the specialized code backwards to cut off the complete reference chain (recursive call in function REMOVELASTPUSH). Section 3.3.9 reports on case studies that unveil the potential of this mechanism to produce shorter and more efficient specialized programs. □

3.3.6 Composing Summaries

When computing method summaries we re-use the method summaries of the invoked methods to update the path conditions and composition schedules as described below. Recall that the the invoked methods already have computed summaries since we adopt a bottom-up approach to summary composition.

Procedure *ComposeSummary* Let us assume that a method `comp` invokes method `m`. By construction, the existence of a method summary for `m` is assumed. Procedure COMPOSESUMMARY in Algorithm 3 executes every time a method invocation (e.g., `invokevirtual`) is reached. The procedure distinguishes two cases, according to parameter *mode*.

- When *mode* is set to *true* ("during specialization" from Algorithm 2), it means we are in the middle of creating a summary for method `comp` and an invocation to method `m` for which a summary exists is reached. Here, for each case in the method summary for `m`, its composition schedule is established and the composition operation (procedure COMPOSECASE) is applied.
- Alternatively, when *mode* is set to *false* it means that we are in the middle of executing specialized code (from an already computed method summary case) and we reached a method invocation. Therefore, a deterministic composition schedule that specifies which summary case to use for the invoked method has already been set. E.g., during execution of a summary case of method `comp`, an invocation to method `m` is reached; relying on the composition schedule, we can uniquely select the summary case of `m` that must be composed. Thus, there is no branching during the execution of a summary case for `comp`.

Procedure *ComposeCase* The COMPOSECASE procedure first analyzes the compatibility of the heap path conditions (procedure CHECKANDSET). This analysis implies: a)

checking that the heap constraints from the summary case hold in the current concrete heap; and b) firing lazy initialization for all the heap constraints that operate on symbolic heap elements. This has the effect of "populating" the heap with all necessary elements that are accessed during the execution of the the summary case in `m`. Therefore, non-determinism (due to lazy initialization) will *not* happen during the symbolic re-execution of the summary case in `m`.

If checkAndSet succeeds, the algorithm takes the path condition from the summary case, it instantiates it for the current calling context in method `comp` and it conjoins it into the current path condition. The resulting path condition is checked for satisfiability. If the new path condition passes the satisfiability check, the original code of the invoked method is replaced with the specialized code from the summary case and symbolic execution proceeds; note that in this case *mode* is *false* (we are not during specialization), so on an invoke, the second case of composeSummary will be used.

Example 11 Consider case study in Figure 3.10. The 5th summary case obtained for method `q` is as follows: $\langle\{y.next.a \geq 0,\ y.next.a \neq 0\},\ \{x = null, y \neq null,\ y.next \neq null\}$, [`aload y, getfield next, getfield a, invoke abs, ireturn`], $[0_{abs}]\rangle$. As explained in Section 3.3.4, the composition schedule $[0_{abs}]$ indicates that when processing the invocation to `abs`, the first case (index 0) of `abs`'s summary must be deterministically chosen for composition. Now, let us consider building the summary for method `r` and let us look at the symbolic execution path of `r` in which arguments `x`, `y` and `z` are lazily initialized to `null` and method `q` is invoked with argument `w` lazily initialized to a new object with symbolic fields. The composition of the 5th summary case for `q` enables us to generate the following summary case for `r`: $\langle\{w.next.a \neq 0, w.next.a \geq 0\}, \{x = null, y = null, z = null, w \neq null, w.next \neq null\}$,[`iconst 0`, `istore sum`, `iconst 3`, `anewarray`, `dup`, `iconst 0`, `aload x`, `aastore`, `dup`, `iconst 1`, `aload y`, `aastore`, `dup`, `iconst 2`, `aload z`, `aastore`, `astore arr`, `iconst 0`, `istore i`, `iload sum`, `aload arr`, `iload i`, `aaload`, `aload w`, `invoke q`, `iadd`, `istore sum`, `iinc i 1`, `iload sum`, `aload arr`, `iload i`, `aaload`, `aload w`, `invoke q`, `iadd`, `istore sum`, `iinc i 1`, `iload sum`, `aload arr`, `iload i`, `aaload`, `aload w`, `invoke q`, `iadd`, `istore sum`, `iinc i 1`, `iload sum`, `ireturn`], $[5_q, 0_{abs}, 5_q, 0_{abs}, 5_q, 0_{abs}]\ \rangle$ (110th case in Figure 3.12). Observe that `w.next` and `w.next.a` are constrained according to the (heap) path condition from the composed summary case. Moreover, notice that the composition schedule stored in the new summary case for `r` specifies which summary cases to be se-

Algorithm 3 Composition Operations

```
1: procedure composeSummary(m,mode)
2:     if mode = true then                          ▷ mode = duringSpecialization
3:         S ← getSummary(m)
4:         for all case ∈ S do
5:             setCompositionSchedule(case.getCompSched())
6:             composeCase(case)
7:         end for
8:     else                                         ▷ mode ≠ duringSpecialization
9:         S ← getSummary(m)
10:        caseIndex ← compositionSchedule.getNext()
11:        case ← getSummaryCase(S,caseIndex)
12:        composeCase(case)
13:    end if
14: end procedure
15: procedure composeCase(case)
16:    heapPC ← case.getHeapPC()
17:    projectActualParameters(heapPC)
18:    if checkAndSet(currentHeapPC,heapPC) then
19:        pc ← case.getPC()
20:        projectActualParameters(pc)
21:        currentPC ← currentPC ∪ pc
22:        if satisfy(currentPC) then
23:            ReplaceCode(invokedMethod,case.getCode())
24:            ContinueSymbolicExecution
25:        else
26:            Backtrack
27:        end if
28:    else
29:        Backtrack
30:    end if
31: end procedure
```

lected in further calls. Namely, during execution of this particular summary case for `r`, when each call to method `q` is reached, its summary case with index 5 will be deterministically selected; and in turn, when the calls from `q` to `abs` are reached, its summary case with index 0 will be chosen. □

3.3.7 Discussion

We argue here the correctness of our approach. Consider a method `m` that does not invoke any other methods. Let us first note that the "plain" symbolic execution of a method `m` explores the same behaviors as the symbolic execution of the summary cases for `m`. This follows from the fact that each summary case is a representation of a terminating path in the symbolic execution tree and, by construction, the path condition stored in each summary case characterizes all the possible inputs which follow that path. We also note that since the path condition already precisely characterizes the inputs that follow a particular path through the code, the conditional instructions along that path are no longer necessary; this is why they are removed in the specialized code that is kept in the summary case.

Consider now the case where method `comp` invokes method `m`, for which we have computed the summaries. Whenever `m` is invoked inside `comp` we check to see which summary case from `m` can be used in the current context, by checking the satisfiability of the path conditions stored in each summary case. It may be the case that multiple summary cases from `m` can be used in the current calling context from `comp`: they are all used systematically, and the method schedule is updated with each case number accordingly. Note that each summary case of `m` that is used will contribute a *new* path in the symbolic execution of `comp`, with a new path condition and a new schedule, and hence to a new summary case in `comp` as well. Since all the cases that apply are used from `m` we conclude that there is no loss of information from the composition operator.

3.3.8 Error Summaries

Error detection is among the most studied applications of symbolic execution, with several tools available [CGK+11]. So far we have described how to compute summaries as finite sets of summary cases, using bounded symbolic execution and partial evaluation. The computation of each summary case completes when either a `return` for the analyzed method is encountered or an error is detected. We have extended our work to specifically target error detection. In particular, we compute *error summaries* which are sets of *error summary cases*, where each *error summary* is completed when an error is encountered during symbolic executions (e.g. assert violations or run-time errors). The advantage of this approach is that the error summaries can be much smaller than the full summaries leading to a quick detection of errors in large programs (if such errors exist). For instance,

let us consider again method `comp` from Figure 3.8. Its `if-else` branch leads to a runtime error, hence, if we were to focus on error detection, the error summary for `comp` would consist *only* of the last summary case from its method summary.

Different scenarios can occur when targeting compositional symbolic execution on error detection. In particular, we distinguish three alternatives. Let us explain them by means of an example.

```
comp() { ... mist(); ... buggy();};
mist(){...};
buggy(){... assert false; ...};
```

Consider the above code fragment that contains three methods. We are interested in analyzing method `comp` for errors only. Now, let us assume that an assertion violation is reachable within method `buggy`. We have defined the following three composition strategies.

a) Method `mist` is executed symbolically and a method summary is created for it. This is essentially the strategy we have followed so far. The system is analyzed in a bottom-up fashion. That is, first a method summary is computed for `mist`, then, an error summary is computed for `buggy` and finally, an error summary is computed for `comp`.

b) Method `mist` is executed symbolically but no summary is created for it. We start by computing the error summary for `buggy`. Later, when symbolic execution of `comp` encounters the call to `mist`, its code is explored only to find the first terminating branch that further steers the execution to an error summary case in `buggy`.

c) Method `mist` is skipped from the analysis. This strategy is only possible under the assumption that `mist` is a side-effect free method, which could be annotated or previously inferred by, e.g., static analysis. Again, we start by computing an error summary for `buggy` followed by the compositional symbolic execution of `comp`. Here, when the call to `bar` is reached, a symbolic variable holding its return value is created and execution proceeds to create the error summary for `comp`. While this strategy may be very fast, the disadvantage is that spurious error summary cases may be generated. The problem can be addressed as follows. For each reported error case of the top-most method, a "plain" symbolic execution is performed, as dictated by the schedule, to validate the errors. However, we leave such validation for future work.

3.3.9 Experience and Results

Our compositional approach has been implemented as an extension of SPF through two listeners that perform specialization and composition by monitoring the symbolic execution and reacting upon execution of method calls and returns. The call graph and method ordering for our bottom-up approach are computed using the COSTA tool [RDCP11]. We have applied our implementation to three case studies:

Case Study 1 Our first case study (Figure 3.9) stresses the use of linear integer constraints and is a variant of the example used before in Section 3.2. It contains two classes. Class `Arithmetics` implements two methods that compute the absolute value of an integer number (`abs`) and the greatest common divisor of two integers (`gcd`), for which 2 and 13 summary cases are generated, respectively. Notice that `gcd` invokes `abs`. Class `Rational` implements methods `simplify`, which calls `Arithmetics.gcd` and `simp`, which in turn invokes `simplify` for each of the elements of an array of `Rational` objects.

Case Study 2 The second case study (Figure 3.10) illustrates the potential impact of our approach on object-oriented programs; the code was created to illustrate the code slicing in the presence of heap updates. Method `r` takes as input four `Foo` objects, creates an array with the first three arguments and traverses it invoking method `q` in each iteration. Method `q` returns an integer value after evaluating heap constraints over its two arguments of type `Foo`. Method `abs` is invoked in one of the paths of `q`.

Case Study 3 Our third case study is borrowed from the net.datastructures package. We selected method `swapElements` from class `NodePositionList`. For brevity, we only show its call graph in Figure 3.11 (source code is publicly available online). Method `swapElements` takes as input two `DNode` objects, checks that they are valid positions in the list (method `checkPosition`) and swaps their elements. Notice that methods `checkPosition` and `element` may raise runtime exceptions. In total, six methods are analyzed in this example.

Results

We have applied our compositional analysis techniques to the three case studies described above. For case study 1, 14 summary cases are generated for `simplify` and 2744 for `simp`. For case study 2, a total of 18 and 18013 summary cases are generated for methods `q`

```
class Arithmetics {
  static int abs(int x){
    if (x >= 0) return x;
    else return -x;
  }
  static int gcd(int x, int y) {
    if (x == 0) return abs(y);
    while ((y != 0)  && (i<2)) {
      if (x > y)  x = x-y;
      else y = y-x;
      if (i==2) return -1;
      i++;
    }
    return abs(x);
  }
}
class Rational {
  int num, den;
  void simplify(int a, int b){
    int gcd = Arithmetics.gcd(a,b);
    if (gcd != 0) {
      num = num/gcd; den = den/gcd;
    }
  }
  Rational[] simp(Rational[] rs){
    Rational[] oldRs = new Rational[rs.length];
    arraycopy(rs,oldRs,length);
    for (int i=0;i < length;i++)
      rs[i].simplify(rs[i].num, rs[i].den);
    return oldRs;
  }
}
```

Figure 3.9: Case Study 1: source code

and `r`, respectively. Finally, column **All** of Table 3.8 shows the number of summary cases generated for each of the methods in case study 3.

Table 3.7 summarizes the results obtained with standard (SPF) and our compositional symbolic execution (CompSPF). The results confirm that our compositional framework greatly outperforms non-compositional SPF both in computation time, number of explored states and number of instructions executed. Notice that the gains in number of instructions executed is quite different between the first case study and the latter ones; the specialized programs computed in Section 3.3.5 tend to be much shorter in the presence of heap constraints, which are heavily used in case studies 2 and 3. This is due to the

```
class Foo {
  int a;
  Foo next;
  int q(Foo x, Foo y){
      if (x != null) {
        if ((x.next != null) &&
            (x.next.next != null) &&
            (x.next.next.a != 0))
          return x.next.next.a;
        else
          return 1;
      } else if ((y != null) &&
                 (y.next != null) &&
                 (y.next.a != 0))
               return abs(y.next.a);
             else
               return 2;
  }
  int r(Foo x, Foo y, Foo z, Foo w){
      int sum = 0;
      Foo[] arr = new Foo[]{x,y,z};
      for (int i=0;i<arr.length;i++)
          sum = sum + q(arr[i],w);
      return sum;
  }
}
```

Figure 3.10: Case Study 2: source code

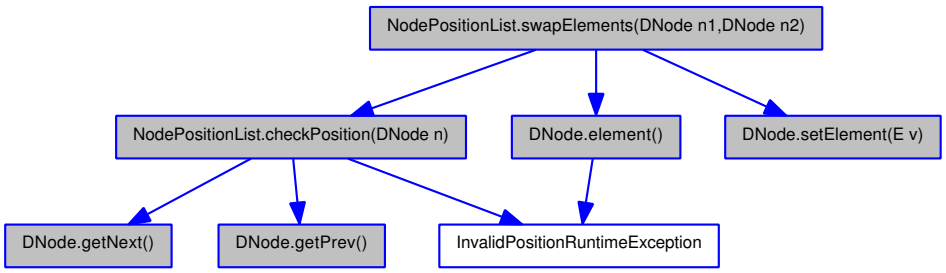


Figure 3.11: Case Study 3: call graph

removal of many instructions by cutting off complete reference chains when specializing conditionals over reference fields. For instance, let us observe method `q` from case study 2, whose original code consists of 39 bytecode instructions, whereas the sizes of the spe-

cialized versions that are stored in its summary (Figure 3.12) range from 2 to 5 bytecode instructions only. Nevertheless, there are other methods, e.g., method `r`, for which the specialized code is the same or even longer than the original one. This may happen with code that imposes no constraints on the input arguments or in the presence of loops, which are just unrolled during symbolic execution, hence leading to longer specialized code.

We further notice that our approach is more expensive in terms of memory consumption; this is not surprising given the large number of cases that need to be stored. In the future we plan to work on optimizing our implementation to alleviate this cost, e.g., by storing only bytecode references in the summaries and by simplifying the numeric constraints.

	Case Study 1		**Case Study 2**		**Case Study 3**	
	SPF	**CompSPF**	**SPF**	**CompSPF**	**SPF**	**CompSPF**
Time	02:50	01:02	00:59	00:14	00:34	00:05
States	24899	13928	165230	63526	34211	11918
Instructions	145908	139992	2132113	465229	175011	68504
Max. Memory	106MB	170MB	156MB	390MB	314MB	358MB

Table 3.7: Experimental results

Finally, let us use case study 3 to illustrate the benefits of computing *error* summaries instead of full summaries when the goal is to detect bugs. We now want to focus the analysis on the errors raised in method `element`. To do this, we store only paths finishing in `return` statements for methods `setElement`, `getNext`, `getPrev` and `checkPosition`. *Error* summaries are computed for the method of interest (`element`) and the entry method `swapElements`. The result of running compositional SPF with this configuration is shown in the third column of Table 3.8. We could even further refine the analysis to exclusively consider paths leading to runtime errors in any of the methods in the case study. The fourth column of Table 3.8 shows the results obtained with this configuration. These two configurations are instances of case a) of Section 3.3.8. Finally, the last column shows the results for methods `element` and `swapElements`, skipping the rest of methods, thus exemplifying case c) of Section 3.3.8. Overall, we believe that these results constitute evidence of the efficiency and applicability of our compositional framework.

Summary for method `abs`

Case	PC	HPC	Code	Schedule
0	$\{x \geq 0\}$	$\emptyset$	[`iload x, ireturn`]	[ ]
1	$\{x < 0\}$	$\emptyset$	[`iload x, ineg, ireturn`]	[ ]

Selected summary cases for method `q`

Case	PC	HPC	Code	Schedule
0	$\emptyset$	$\{x = null, y = null\}$	[`iconst 2, ireturn`]	[ ]
1	$\{y.a \geq 0, y.a \neq 0\}$	$\{x = null, y \neq null, y.next = y\}$	[`aload y, getfield next, getfield a, invoke abs, ireturn`]	$[0_{abs}]$
2	$\{y.a < 0, y.a \neq 0\}$	$\{x = null, y \neq null, y.next = y\}$	[`aload y, getfield next, getfield a, invoke abs, ireturn`]	$[1_{abs}]$
3	$\{y.a = 0\}$	$\{x = null, y \neq null, y.next = y\}$	[`iconst 2, ireturn`]	[ ]
3				
4	$\emptyset$	$\{x = null, y \neq null, y.next = null\}$	[`iconst 2, ireturn`]	[ ]
5	$\{y.next.a \geq 0, y.next.a \neq 0\}$	$\{x = null, y \neq null, y.next \neq null\}$	[`aload y, getfield next, getfield a, invoke abs, ireturn`]	$[0_{abs}]$
6	$\{y.next.a < 0, y.next.a \neq 0\}$	$\{x = null, y \neq null, y.next \neq null\}$	[`aload y, getfield next, getfield a, invoke abs, ireturn`]	$[1_{abs}]$

Selected summary cases for method `r` *

Case	PC	HPC	Schedule
0	$\emptyset$	$\{x = null, y = null, z = null, w = null\}$	$[0_q, 0_q, 0_q]$
1	$\{w.a \neq 0, w.a \geq 0\}$	$\{x = null, y = null, z = null, w \neq null, w.next = w\}$	$[1_q, 0_{abs}, 1_q, 0_{abs}, 1_q, 0_{abs}]$
110	$\{w.next.a \neq 0, w.next.a \geq 0\}$	$\{x = null, y = null, z = null, w \neq null, w.next \neq null\}$	$[5_q, 0_{abs}, 5_q, 0_{abs}, 5_q, 0_{abs}]$
857	$\{y.a \neq 0, y.next.a \neq 0, y.next.a < 0\}$	$\{x = null, y \neq null, z = y, w = y, y.next \neq null, y.next.next = y\}$	$[6_q, 1_{abs}, 13_q, 13_q]$
8005	$\{w.a \neq 0, w.a \geq 0, x.next.a \neq 0\}$	$\{x \neq null, y = null, z \neq null, w \neq null, x.next \neq null, x.next.next = x.next, w.next = w, z.next = null\}$	$[11_q, 1_q, 0_{abs}, 10_q]$

* The specialized code for all these summary cases is identical to that shown in Example 11, thus the column is omitted.

Figure 3.12: Case Study 2: results

3.4 Conclusions

Compositional reasoning is a widely used methodology in static analysis and software verification but notably less common in the field of symbolic execution and TCG. Compositional symbolic execution for TCG has been proposed with different flavours. The main difference between the proposed approaches is the information that is stored in the so-called method summaries. In [God07, AGT08] (demand-driven) compositionality is developed in the context of dynamic test generation. In this case, a method summary is a

Method	All	Errors in `element`	Only Errors	Skip others
`setElement`	4	2	2	-
`getNext`	4	3	1	-
`getPrev`	4	3	1	-
`element`	16	12	12	12
`checkPosition`	40	20	5	-
`swapElements`	1685	618	6	230
Total	1753	658	27	242
Time	00:05	00:03	00:01	00:03

Table 3.8: Case Study 3: results

disjunction of logical formulae, each of which represents a feasible path in the program. However, heap constraints and heap composition are not supported. In our CLP-based approach to compositionality (Section 3.2), the stored method summaries include not only the path condition, but also explicit representation of the input and output heap. The advantage of this approach is that all the effects of the computation are stored, and therefore there is no need to continue symbolic execution once a summary is found. However, such advantage comes at a high price of the composition operation, which not only has to check compatibility at the invocation point, but also has to synthesize the new state to continue with after the composition of each summary case. Our work on compositional symbolic execution through program specialization, presented in Section 3.3, aims at overcoming this limitation by building specialized code into the so-called method summaries. Namely, a method summary case not only stores the path condition for a particular symbolic execution path, but also includes an specialized version of the method with respect to such path condition.

To sum up, we have developed two approaches to compositionality for symbolic execution and testing. We have extended our CLP-based TCG framework with compositional reasoning, making the framework more scalable and enabling testing of programs with calls to native code. We have also described the program specialization-based compositional approach to symbolic execution that we have developed in SPF, reporting on experimental results that demonstrate the benefits of using partial evaluation as an enabling technique to realise compositional reasoning in symbolic execution. In future work, we plan to extend our approach to multithreaded/concurrent programs [WTH+12,QW04],

where the path explosion problem is even more acute mainly due to the large number of thread interleavings that must be considered.

Chapter 4

Resource-driven TCG

This chapter presents a novel framework that allows using resource consumption information to guide symbolic execution and thus TCG towards parts of the program under test that comply with a given resource policy. This work has been published in:

> Elvira Albert and Miguel Gómez-Zamalloa and José Miguel Rojas. **Resource-driven CLP-based Test Case Generation.** In *LOPSTR 2011, 21st International Symposium on Logic-Based Program Synthesis and Transformation*, volume 7225 of *Lecture Notes in Computer Science*, pages 25–41. Springer, July 2011.

4.1 Introduction

Reasoning about non-functional aspects of programs, such as resource consumption, is often more challenging than reasoning about functional ones. Non-functional properties are commonly assessed with dynamic program analysis techniques, such as *profiling*. Profiling tools execute a program for concrete inputs to estimate the associated resource consumption of the program. Profilers can be parametric w.r.t. the notion of resource which often includes cost models like time, number of instructions, memory consumed, number of invocations to methods, etc. The purpose of profiling is usually to find out which parts of a device or software contribute most to its poor performance and find bugs related to the resource consumption.

In this chapter, we propose *resource-aware TCG* which strives to build performance into test cases by additionally generating their resource consumption, thus enriching stan-

dard TCG with non-functional properties. The main idea is that, during the TCG process, we keep track of the exercised instructions to obtain the test case. Then, in a simple post-process we map each instruction into a corresponding cost, we obtain for each class of inputs a detailed information of its resource consumption (including the aforementioned resources). Our approach is not reproducible by first applying TCG, then instantiating the test cases to obtain concrete inputs and, finally, performing profiling on the concrete data. This is because, for some cost criteria, resource-aware TCG is able to generate symbolic (i.e., non-constant) costs. E.g., when measuring memory usage, the amount of memory might depend on an input parameter (e.g., the length of an array to be created is an input argument). The resource consumption of the test case will be a symbolic expression that profilers cannot compute.

An interesting aspect of resource-aware TCG is that resources can be taken into account in order to filter out test cases which do not consume more (or less) than a given amount of resources, i.e, one can consider a *resource policy*. This leads to the idea of resource-*driven* TCG, i.e., a new heuristics which aims at guiding the TCG process to generate test cases that adhere to the resource policy. The potential interest is that we can prune the symbolic execution tree and produce, more efficiently, test cases for inputs which otherwise would be very expensive (and even impossible) to obtain.

Our approach to resource-driven CLP-based TCG consists of two phases. First, in a pre-process, we obtain (an over-approximation of) the set of *traces* in the program which lead to test cases that adhere to the resource policy. We sketch several ways of automatically inferring such traces, starting from the simplest one that relies on the call graph of the program to more sophisticated ones that enrich the abstraction to reduce the number of unfeasible paths. We refer to this abstraction as a *trace generator*. Second, using as an input argument a trace produced by the *trace generator*, executing standard CLP-based TCG generates a test case that satisfies the resource policy (or it fails if the trace is unfeasible). Observe that the *resource policy* acts essentially as a *selection criterion* to reduce the size of the test suite generated by TCG. In Chapter 5 we generalize this idea to propose a generic framework for guiding TCG with respect to any given selection criterion.

4.2 CLP-based Test Case Generation with Traces

This section extends the framework proposed in Section 2.1 to incorporate *traces* in the CLP programs that will be instrumental later to define the resource-aware framework.

4.2.1 CLP-Translation with Traces

In this section, we define the notion of *trace term* and update Definition 2.1.1 to add a trace term as an additional argument to each rule of the CLP-translated program, which enables us to keep track of the sequence of rules that are symbolically executed.

Definition 4.2.1 (CLP-translated program with traces)**.** *Given the rule of Definition 2.1.1, its CLP-translation with trace is:* $m(In, Out, H_{in}, H_{out}, EF, T) :- g, b'_1, \ldots, b'_n$." *where:*

- *In, Out,* H_{in}*,* H_{out} *and EF remain as in Definition 2.1.1.*
- T *is the trace term for* m *of the form* $m(k, P, \langle T_{c_i}, \ldots, T_{c_m} \rangle)$*, where*
  - P *is the (possibly empty) list of trace parameters, i.e., the subset of the variables in rule* m^k *on which the resource consumption depends.*
  - $c_i, \ldots, c_m$ *is the (possibly empty) subsequence of method calls in* $b_1, \ldots, b_n$*.*
  - T_{c_j} *is a free logic variable representing the trace term associated to the call* c_j*.*
- *Calls in the body of the rule are extended with their corresponding trace terms, i.e., for all* $1 \leq j \leq n$*, if* $b_j \equiv p(I_p, O_p, H_{in_p}, H_{out_p})$*, then* $b'_j \equiv p(I_p, O_p, H_{in_p}, H_{out_p}, T_{c_j})$*; otherwise* $b'_j \equiv b_j$*.*

Notice that trace terms are not cardinal components in the translated program, but rather a supplementary argument with a central role in this chapter.

Definition 4.2.2 (Function *instr*)**.** *Given a rule* m_i^k*, we denote by* $instr(m_i^k)$ *the sequence of instructions in the original program that have been translated into rule* m_i^k*.*

4.2.2 Test Case Generation with traces

We now revisit our definition of test case and TCG (Definition 2.1.3) to incorporate the notion of *trace*.

```
class Vector {
  int[] elems;
  int size;
  int cap;
  Vector(int iCap) throws Exception{
    if (iCap > 0){
      elems = new int[iCap];
      cap = iCap;
      size = 0;
    } else
      throw new Exception();
  }
  void add(int x){
    if (size >= cap)
      realloc();
    elems[size++] = x;
  }
  void realloc(){
    int nCap = cap*2;
    int[] nElems = new int[nCap];
    for (int i=0; i<cap; i++) {
      nElems[i] = elems[i];
    }
    cap = nCap;
    elems = nElems;
  }
}
```

Figure 4.1: Resource-driven example 1: Java source code

Definition 4.2.3 (Test case with trace and TCG)**.** *Given a method m, a termination criterion $\mathcal{C}$ and a successful (terminating) path b in the symbolic execution tree $\mathcal{T}_m^C$ with root $m(In, Out, H_{in}, H_{out}, EF, T)$, a* test case with trace *for m w.r.t. $\mathcal{C}$ is a 6-tuple of the form: $\langle \sigma(In), \sigma(Out), \sigma(H_{in}), \sigma(H_{out}), \sigma(EF), \sigma(T), \theta \rangle$, where σ and θ are, resp., the set of bindings and the constraint store associated to b. TCG generates the set of test cases with traces obtained for all successful paths in $\mathcal{T}_m^C$.*

Example 12 Our example in Figure 4.1 shows class `Vector`, that contains a reference to an array of integers `elems` and two integer fields to keep track of its size and capacity (`size` and `cap`). The initial capacity of the array is set by the constructor (method `init`). The interesting aspect of class `Vector` is that, when adding an element using method `add`, if the size has already reached the maximum capacity determined by field

cap the size of the array is duplicated (by method realloc) before actually adding the new element. Figure 4.2 shows the (simplified and pretty-printed) CLP-translated program obtained from the bytecode associated to class Vector. Observe that method add is transformed into predicates add, if and addc, method realloc is transformed into predicates realloc, loop and cond, and the constructor of the class is translated into predicate init. For brevity, we have omitted the predicates that model exceptional behavior. Function *instr* (Definition 4.2.2) keeps the mapping between rules and bytecode instructions. For instance, $instr(\mathsf{init}^1)$=⟨iload icap, ifgt, aload this, iload icap, newarray int, putfield elems, aload this, aload icap, putfield cap, aload this, iconst 0, putfield size, return⟩ is the sequence of bytecode instructions that have been translated into rule init. Notice that a trace term is made up by the predicate name and number, the set of input arguments on which the cost depends (e.g., rule realloc and its trace parameter NCap) and it recursively includes the trace terms for the predicates it calls.

□

Example 13 Let us now consider loop-1 as coverage criterion. In our example, loop-1 forces the array in the input vector to be at most of length 1. Note that we include the reference to the This object as an explicit input argument in the CLP-translated program. The symbolic execution tree of $\mathtt{add(In,Out,H_{in},H_{out},T)}$ will contain two successful derivations (ignoring exceptions) corresponding to the following situations:

1. If the size of the Vector object is less than its capacity, then the argument X is directly inserted in elems.

2. If the size of the Vector object is greater than or equal to its capacity, then method realloc is invoked before inserting X.

Figure 4.3 shows in detail the second test case. Heaps are graphically represented by using rounded boxes for arrays (the array length appears to the left and the array itself to the right) and square boxes for Vector objects (field elems appears at the top, fields size and cap to the left and right bottom of the square, resp.). The trace term T contains the rules that were executed along the derivation. At the bottom of the figure, an (executable) instantiation of this test case is shown. □

```
add([r(This),X],[],H,H1,add(1,[],[T])) :-
    get_field(H,This,size,Size),
    get_field(H,This,cap,Cap),
    if([Size,Cap,r(This),X],[],H,H1,T).
if1([Size,Cap,r(This),X],[],H,H1,if(1,[],[T])) :-
    Size #< Cap,
    addc([r(This),X],[],H,H1,T).[0.
if2([Size,Cap,r(This),X],[],H,H2,if(2,[],[T1,T2])) :-
    Size #>= Cap,
    realloc([r(This)],[],H,H1,T1),
    addc([r(This),X],[],H1,H2,T2).
addc([r(This),X],[],H,H2,addc(1,[],[])) :-
    get_field(H,This,elems,r(Es)),
    get_field(H,This,size,Size),
    set_array(H,Es,Size,X,H1),
    NSize #= Size+1,
    set_field(H1,This,size,NSize,H2).
realloc([r(This)],[],H,H2,realloc(1,[NCap],[T])) :-
    get_field(H,This,cap,Cap),
    NCap #= Cap*2,
    new_array(H,int,NCap,NEs,H1),
    loop([r(This),NCap,r(NEs),0],[],H1,H2,T).
loop([r(This),NCap,r(NEs),I],[],H,H1,loop(1,[],[T])) :-
    get_field(H,This,cap,Cap),
    cond([Cap,I,r(This),NCap,r(NEs)],[],H,H1,T).
cond1([Cap,I,r(This),NCap,r(NEs)],[],H,H2,cond(1,[],[])) :-
    I #>= Cap,
    set_field(H,This,cap,NCap,H1),
    set_field(H1,This,elems,r(NEs),H2).
cond2([Cap,I,r(This),NCap,r(NEs)],[],H,H2,cond(2,[],[T])) :-
    I #< Cap,
    get_field(H,This,elems,r(Es)),
    get_array(H,Es,I,E),
    set_array(H,NEs,I,E,H1),
    NI #= I+1, loop([r(This),NCap,r(NEs),NI],[],H1,H2,T).
init1([r(This),ICap],[],H,H4,init(1,[ICap],[])) :-
    ICap #> 0,
    new_array(H,int,ICap,E,H1),
    set_field(H1,This,elems,r(E),H2),
    set_field(H2,This,cap,ICap,H3),
    set_field(H3,This,size,0,H4).
init2([r(This),ICap],[],H,H1,init(2,[ICap],[])) :-
    ICap #=< 0,
    new_object(H,'Exception',E,H1).
```

Figure 4.2: Resource-driven example 1: CLP-translation

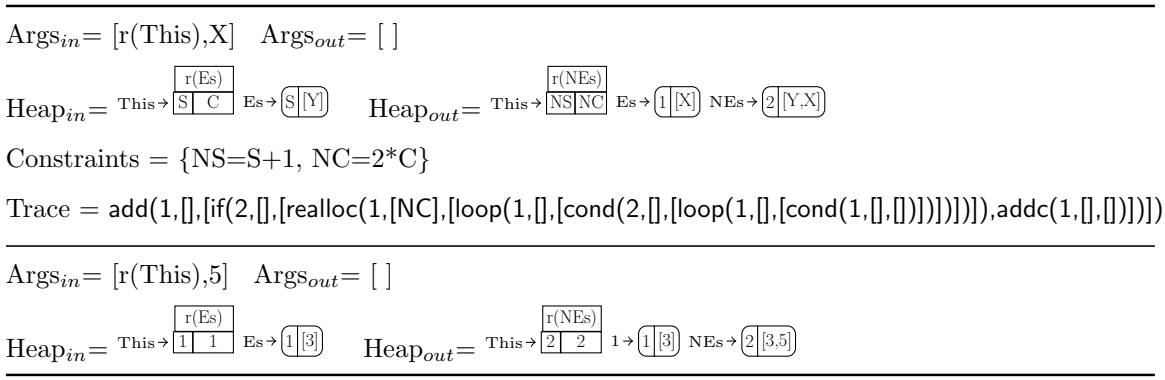


Figure 4.3: Example of test case (up) and test input (down) for add with loop-1

4.3 Resource-aware Test Case Generation

In this section, we present the extension of the TCG framework of Section 2.1 to build resource consumption into the test cases. First, in Section 4.3.1 we describe the cost models that we will consider in the present work. Then, Section 4.3.2 presents our approach to resource-aware TCG.

4.3.1 Cost Models

A cost model defines how much the execution of an instruction costs. Hence, the resource consumption of a test case can be measured by applying the selected cost model to each of the instructions exercised to obtain it.

Number of Instructions. The most traditional model, denoted $\mathcal{M}_{ins}$, is used to estimate the number of instructions executed. In our examples, since the input to our system is the bytecode of the Java program, we count the number of bytecode instructions. All instructions are assigned cost 1.

Memory Consumption. Memory consumption can be estimated by counting the actual size of all objects and arrays created along an execution [AGGZ07].

$$\mathcal{M}_{mem}(b) = \begin{cases} size(Class) & if\ b \equiv \texttt{new}\ Class \\ S_{ref} * Length & if\ b \equiv \texttt{anewarray}\ Class\ Length \\ S_{prim} * Length & if\ b \equiv \texttt{newarray}\ PrimType\ Length \\ 0 & otherwise \end{cases}$$

We denote by S_{prim} and S_{ref}, resp., the size of primitive types and references. In the examples, by assuming a standard JVM implementation, we set both values to 4 bytes. The size of a class is the sum of the sizes of the fields it defines. Note that, if one wants to consider garbage collection when assessing memory consumption, then the behaviour of the garbage collection should be simulated during the generation of test cases. In this work, we assume that no garbage collection is performed.

Number of calls. This cost model, $\mathcal{M}_{call}$, counts the number of invocations to methods. It can be specialized to $\mathcal{M}^m_{call}$ to count calls to a specific method m which, for instance, can be one that triggers a billable event (e.g. send SMS).

Example 14 The application of the cost models $\mathcal{M}_{ins}$, $\mathcal{M}_{mem}$ and $\mathcal{M}_{call}$ to the sequence of instructions in rule `init` (i.e., $instr(\texttt{init})$ of Example 12) results in, resp., 14 bytecode instructions, $4 * ICap$ bytes and 0 calls. □

4.3.2 Resource-aware TCG

Given the test cases with trace obtained in Definition 4.2.3, the associated cost can be obtained as a simple post-process in which we apply the selected cost models to all instructions associated to the rules in its trace.

Definition 4.3.1 (test case with cost)**.** *Consider a test case with trace* $Test \equiv \langle Args_{in}, Args_{out}, H_{in}, H_{out}, \text{Trace}, \theta \rangle$*, obtained in Definition 4.2.3 for method* m *w.r.t.* C*. Given a cost model* $\mathcal{M}$*, the cost of* $Test$ *w.r.t.* $\mathcal{M}$*, is defined as:*

$$C(Test, \mathcal{M}) \quad = \quad \texttt{cost}(\text{Trace}, \mathcal{M})$$

```
static Vector[] multiples(int[] ns, int div, int icap){
    Vector v = new Vector(icap);
    for (int i=0; i<ns.length; i++)
        if (ns[i]%div == 0)
            v.add(ns[i]);
    return r;
}
```

Figure 4.4: Resource-driven TCG example 2: Java source code

where function `cost` *is recursively defined as:*

$$\texttt{cost}(\texttt{m}(\texttt{k},\texttt{P},\texttt{L}),\mathcal{M}) \;=\; \begin{cases} \sum_{\forall i \in instr(m^k)} \mathcal{M}(i) & if\ \texttt{L} = [\,] \\ \sum_{\forall i \in instr(m^k)} \mathcal{M}(i) + \sum_{\forall l \in \texttt{L}} \texttt{cost}(l,\mathcal{M}) & otherwise \end{cases}$$

For the cost models in Section 4.3.1, we define the test case with cost *as a tuple of the form* $\langle Test, C(Test,\mathcal{M}_{ins}), C(Test,\mathcal{M}_{mem}), C(Test,\mathcal{M}_{call})\rangle$.

This could also be done by profiling the resource consumption of the execution of the test case. However, observe that our approach goes beyond the capabilities of TCG + profiling, as it can also obtain *symbolic* (non-constant) resource usage estimations while profilers cannot. Besides, it saves us from the non trivial implementation effort of developing a profiler for the language.

Example 15 We use a slightly more complex example from now on. Figure 4.4 shows method `multiples`, which receives an input array of integers `ns` and outputs an object of type `Vector`, created with initial capacity `icap`, containing all the elements of `ns` that are multiples of the second input argument `div`. Let us consider the TCG of the CLP-translated program obtained from method `multiples`. By using loop-4 as coverage criterion, we get 54 test cases, which correspond to all possible executions for input arrays of length not greater than 4, i.e., at most 4 iterations of the *for* loop. Figure 4.5 shows three test cases. The upper one corresponds to the test case that executes the highest number of instructions, in which method `realloc` is executed 2 times (worst case for $\mathcal{M}_{call}^{\texttt{realloc}}$ as well). The one in the middle corresponds to one of the paths with the highest parametric memory consumption (for brevity, only the trace for this case is shown), and the one at the bottom corresponds to that with the highest constant memory consumption. In the middle one, the input array `Ns` is of length 2, both elements in the

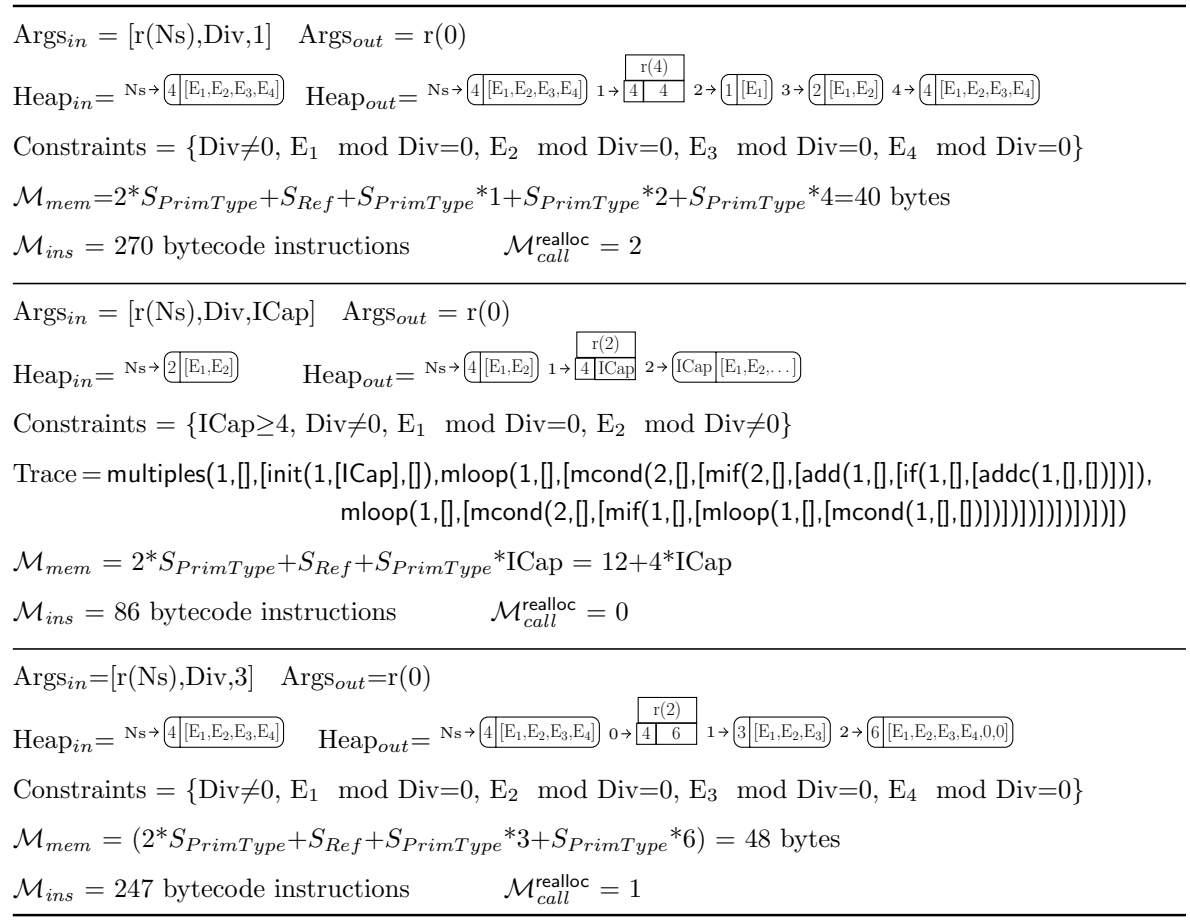


Figure 4.5: Selected test cases with cost for method multiples with loop-4

array are multiples of `Div`, and the initial capacity is constrained by `ICap` ≥ 4. With such input configuration, the array is fully traversed and its two elements are inserted in the resulting `Vector` object. By applying the cost model $\mathcal{M}_{mem}$ to the trace in the figure, we obtain a symbolic heap consumption which is parametric on `ICap` (observe that `ICap` is a parameter of the second and third calls in the trace). Importantly, this parameter remains as a variable because method `realloc` is not executed. Symbolic execution of `realloc` would give a concrete value to `ICap` when determining the number of iterations of its loop. The test case at the bottom in contrast executes `realloc` once, as the vector runs out of capacity at the fourth iteration of the loop. Hence, its capacity is duplicated. □

Resource-aware TCG can be useful as a mechanism to detect bugs related to an excessive consumption of resources.Interestingly, we observe that it can also be used in combination with a *resource policy* in order to filter out test cases which do not adhere to the policy. The resource policy can state that the resource consumption of the test cases must be larger (or smaller) than a given threshold so that one can focus on the (potentially problematic) test cases which consume a certain amount of resources.

Example 16 Let us recall that in Example 15 we had obtained 54 test cases. For instance, by using a resource policy to focus on those cases that consume more than 48 bytes, we filter out 23 test cases. In a realistic scenario, the user must provide the testing framework with resource consumption parameters. For instance, by setting the amount of memory available in the resource policy, TCG could help us detect (potentially buggy) behaviours of the program under test which exceed the memory limit. □

Furthermore, one can display to the user the test cases ordered according to the amount of resources they consume. For instance, for the cost model $\mathcal{M}_{mem}$, the test cases in Example 15 would be shown first. It is easy to infer the condition `ICap` > 9, which determines when the parametric test case is the most expensive one. Besides, one can implement a *worst-case* resource policy which shows to the user only the test case that consumes more resources among those obtained by the TCG process (e.g., the one at the top together with the previous condition for $\mathcal{M}_{mem}$), or display the n test cases with highest resource consumption (e.g., the two cases in Figure 4.5 for $n = 2$).

4.4 Resource-driven TCG

This section introduces *resource-driven TCG* , a novel heuristics to guide the symbolic execution process which improves, in terms of scalability, over the resource-aware approach, especially in those cases where restrictive resource policies are supplied. The main idea is to try to avoid, as much as possible, the generation of paths during symbolic execution that do not satisfy the policy. If the resource policy imposes a maximum threshold, then symbolic execution can stop an execution path as soon as the resource consumption exceeds it. However, it is often more useful to establish resource policies that impose a minimum threshold. In such case, it cannot be decided if a test case adheres to the policy until it is completely generated.

An advantage of relying on a CLP-based TCG approach is that the trace argument of our CLP-transformed programs can be used not only as an output, but also as an input argument. Moreover, traces can also be partially defined (not fully instantiated). This allows guiding, completely or partially, the symbolic execution towards specific paths. In the first case, i.e., when the trace is fully instantiated, symbolic execution is deterministic and thus as efficient as ground execution. In the second case, when the trace is partial (i.e., it includes free variables), a set of concrete paths can be explored, whose traces are instances of the partial trace. Our heuristics to avoid the unnecessary generation of test cases that violate the resource policy is based on this idea: 1) in a pre-process, we look for traces corresponding to potential paths (or sub-paths) that adhere to the policy, and 2) we use such traces to guide the symbolic execution.

4.4.1 Trace-guided TCG

Definition 4.4.1 (trace-guided TCG)**.** *Given a method m, a coverage criterion $\mathcal{C}$, and a (possibly partial) trace π, trace-guided TCG generates the set of test cases with traces, denoted $tgTCG(M, \mathcal{C}, \pi)$, obtained for all successful branches in $\mathcal{T}_m^C$ with root $m(Args_{in}, Args_{out}, H_{in}, H_{out}, E, \pi)$.*

Observe that the TCG guided by one trace π generates: (a) exactly one test case if π is complete and corresponds to a feasible path; (b) none if π is unfeasible; or (c) possibly several test cases if π is partial. In the latter case the traces of all test cases are instantiations of the partial trace.

Example 17 Let us consider the partial trace `multiples(1,[],[init(1, [ICap],[]), mloop(1,[], [mcond(2, [], [mif(2,[],[A1, mloop(1, [],[mcond(2,[],[mif(2,[], [A2, mloop(1,[], [mcond(2, [], [mif(2, [], [A3, mloop(1, [], [mcond(2,[], [mif(2,[], [A4, mloop(1,[], [mcond(1, [], [])...])`, which represents the paths that iterate four times in the *for* loop of method `multiples` (rules `mloop` [1] and `mcond` [2] in the CLP translated program), always following the *then* branch of the *if* statement (rule `mif` [2]), i.e. invoking method `add`. The trace is partial since it does not specify where the execution goes after method `add` is called (in other words, whether method `realloc` is executed or not). This is expressed by the free variables (`A1`, `A2`, `A3` and `A4`) in the trace term arguments. The symbolic execution guided by such trace produces four test cases which differ on the constraint on `ICap`, which is resp. `ICap` =1, `ICap` =2, `ICap` =3

and `ICap` $\geq$4. The first and the third test cases are the ones shown at the top and at the bottom resp. of Fig 4.5. All the executions represented by this partial trace finish with the evaluation to false of the loop condition (rule `mcond` [1]). □

4.4.2 Resource-driven TCG

By relying on a trace generator $\mathbb{G}$ that provides the traces, we now define resource-driven TCG as follows.

Definition 4.4.2 (resource-driven TCG)**.** *Given a method m, a coverage criterion C and a resource-policy R, resource-driven TCG generates the set of test cases with traces defined by*

$$\bigcup_{i=1}^{n} tgTCG(m, C, \pi_i)i$$

where $\{\pi_1, \ldots, \pi_n\}$ is the set of traces computed by a trace generator $\mathbb{G}$ w.r.t R and C.

Parallelization. An interesting observation is that the resource-driven TCG process could be parallelized. In the context of symbolic execution, there is an inherent need of carrying out a constraint store over the input variables of the program. When the constraint store becomes unsatisfiable, symbolic execution must discard the current execution path and backtrack to the last branching point in the execution tree. Therefore, in general it is not possible to parallelize the process. This is precisely what we gain with deterministic resource-guided TCG. Because the test cases are computed as the union of independent executions of $tgTCG$, they can be parallelized. Experimenting on a parallel infrastructure remains as future work.

The definition of resource-driven TCG above relies on a generic trace generator. We will now sketch different techniques for defining specific trace generators. Ideally, a trace generator should be *sound*, *complete* and *effective*. A trace generator is sound if every trace it generates satisfies the resource policy. It is complete if it generates all traces that satisfy the policy. Effectiveness is related to the number of unfeasible traces it generates. The larger the number, the less effective the trace generator and the less efficient the TCG process. For instance, assuming a worst-case resource policy, one can think of a trace generator that relies on the results of a static cost analyzer [AAG+07] to detect the methods with highest cost. It can then generate partial traces that force the execution go through such costly methods (combined with a terminating criterion). Such

trace generator can produce a trace as the one in Example 17 with the aim of trying to maximize the number of times method `add` (the potentially most costly one) is called. This kind of trace generator can be quite effective though it will be in general unsound and incomplete.

4.4.3 Sound and Complete Trace Generators

In the following we develop a concrete scheme of a trace generator which is sound, complete, and parametric w.r.t. both the cost model and the resource policy. Intuitively, a trace generator is complete if, given a resource policy and a coverage criterion, it produces an over-approximation of the set of traces (obtained as in Definition 4.2.3) satisfying the resource policy and coverage criterion. We first propose a naive way of generating such an over-approximation which is later improved.

Definition 4.4.3 (trace-abstraction of a program)**.** *Given a CLP-translated program with traces P, its trace-abstraction is obtained as follows: for every rule of P, (1) remove all atoms in the body of the rule except those corresponding to rule calls, and (2) remove all arguments from the head and from the surviving atoms of (1) except the last one (i.e., the trace term).*

The trace-abstraction of a program corresponds to its control-flow graph, and can be directly used as a trace generator that produces a superset of the (usually infinite) set of traces of the program. The coverage criterion is applied in order to obtain a concrete and finite set of traces. Note that this is possible as long as the coverage criterion is structural, i.e., it only depends in the program structure (like loop-k). The resource policy can then be applied over the finite set: (1) in a post-processing where the traces that do not satisfy the policy are filtered out or (2) depending on the policy, by using a specialized search method.

As regards soundness, the intuition is that a trace generator is sound if the resource consumption for the selected cost model is *observable* from the traces, i.e, it can be computed and it is equal to the one computed after the guided TCG.

Definition 4.4.4 (resource observability)**.** *Given a method m, a coverage criterion C and a cost-model $\mathcal{M}$, we say that $\mathcal{M}$ is* observable *in the trace-abstraction for m, if for every feasible trace π generated from the trace-abstraction using C, we have that $\texttt{cost}(\pi, \mathcal{M}) = \texttt{cost}(\pi', \mathcal{M})$, where π' is a corresponding trace obtained for $tgTCG(m, C, \pi)$.*

Observe that π can only have variables in trace parameters (second argument of a trace term). This means that the only difference between π and π' can be made by means of instantiations (or associated constraints) performed during the symbolic execution on those variables. Trivially, $\mathcal{M}_{ins}$ and $\mathcal{M}_{call}$ are observable since they do not depend on such trace parameters. Instead, $\mathcal{M}_{mem}$ can depend on trace parameters and is therefore non-observable in principle on this trace-abstraction, as we will discuss later in more detail.

Trace-abstraction Refinement

Unfortunately the trace generator proposed so far is in general very far from being effective since trace-abstractions can produce a huge amount of unfeasible traces. To solve this problem, we propose to enhance the trace-abstraction with information (constraints and arguments) taken from the original program. This can be done at many degrees of precision, from the empty enhancement (the one we have seen) to the full one, where we have the original program (hence the original resource-aware TCG). The more information we include, the less unfeasible traces we get, but the more costly the process is. The goal is thus to find heuristics that enrich sufficiently the abstraction so that many unfeasible traces are avoided and with the minimum possible information.

A quite effective heuristic is based on the idea of adding to the abstraction those program variables (input arguments, local variables or object fields) which get instantiated during symbolic execution (e.g., field `size` in our example). The idea is to enhance the trace-abstraction as follows. Let us start with a set of variables V initialized with those variables (this can be soundly approximated by means of static analysis). For every $v \in V$, we add to the program all occurrences of v and the guards and arithmetic operations in which v is involved. The remaining variables involved in those guards are added to V and the process is repeated until a fixpoint is reached. Figure 4.6 shows the trace-abstraction with the proposed enhancement for our working example, in which variables Size and Cap (fields), ICap (input argument) and I (local variable) are added.

In the next chapter (Section 5.5), we further discuss the importance of trace-abstraction refinements and provide a heuristics to generate them by means of static analysis and program transformation.

Resource Observability for $\mathcal{M}_{mem}$. As already mentioned, $\mathcal{M}_{mem}$ is in general non-observable in trace-abstractions. The problem is that the memory consumed by the

```
multiples(ICap,multiples(1,[],[T1,T2])) :-
    init(ICap,Size,Cap,T1),
    mloop(Size,Cap,T2).
mloop(Size,Cap,mloop(1,[],[T])) :-
    mcond(Size,Cap,T).
mcond1(_,_,mcond(1,[],[])).
mcond2(Size,Cap,mcond(2,[],[T])) :-
    mif(Size,Cap,T).[.5ex]
mif1(Size,Cap,mif(1,[],[T])) :-
    mloop(Size,Cap,T).
mif2(Size,Cap,mif(2,[],[T1,T2])) :-
    add(Size,Cap,NSize,NCap,T1),
    mloop(NSize,NCap,T2).
add(Size,Cap,NSize,NCap,add(1,[],[T])) :-
    if(Size,Cap,NSize,NCap,T).
if1(Size,Cap,NSize,Cap,if(1,[],[T])) :-
    Size #\= Cap,
    addc(Size,NSize,T).
if2(Size,Cap,NSize,NCap,if(2,[],[T1,T2])) :-
    Size #= Cap,
    realloc(Cap,NCap,T1),
    addc(Size,NSize,T2).
addc(Size,NSize,addc(1,[],[])) :-
    NSize #= Size+1.
realloc(Cap,NCap,realloc(1,[NCap],[T])) :-
    NCap #= Cap*2,
    loop(Cap,0,T).
loop(Cap,I,loop(1,[],[T])) :- cond(Cap,I,T).
cond1(Cap,I,cond(1,[],[])) :- I #>= Cap.
cond2(Cap,I,cond(2,[],[T])) :- I #< Cap,
    NI #= I+1,
    loop(Cap,NI,T).
init1(ICap,0,ICap,init(1,[ICap],[])).
init2(ICap,0,ICap,init(2,[ICap],[])).
```

Figure 4.6: Trace-abstraction refinement

creation of arrays depends on dynamic values which might be not present in the trace-abstraction. Again, this problem can be solved by enhancing the trace-abstraction with the appropriate information. In particular, the enhancement must ensure that the variables involved in the creation of new arrays (and those on which they depend) are added to the abstraction. This information can be statically approximated [AAG+08, LV05, LS96].

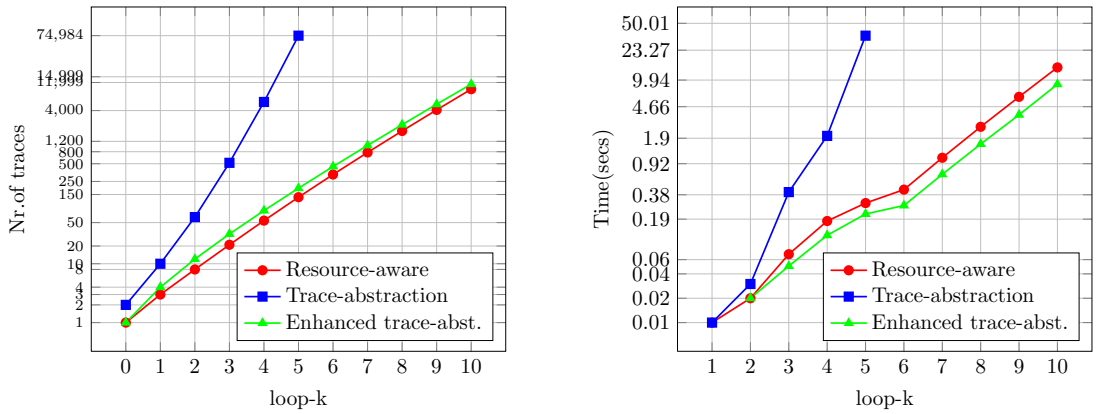
Instances of Resource-driven TCG. The resource-driven scheme has been deliberately defined as generic as possible and hence it could be instantiated in different ways for particular resource policies and cost-models producing more effective versions of it. For instance, for a worst-case resource policy, the trace generator must generate all traces in order to know which is the one with maximal cost. Instead of starting a guided symbolic execution for all of them, we can try them one by one (or k by k in parallel) ordered from higher to lower cost, so that as soon as a trace is feasible the process stops. By correctness of the trace generator, the trace will necessarily correspond to the feasible path with highest cost.

Theorem 4.4.5 (correctness of trace-driven TCG)**.** *Given a cost model $\mathcal{M}$, a method m, a coverage criterion C and a sound trace generator $\mathbb{G}$ on which $\mathcal{M}$ is observable, resource-driven TCG for m w.r.t. C using $\mathbb{G}$ generates the same test cases as resource-aware TCG w.r.t. C for the cost model $\mathcal{M}$.*

Soundness is trivially entailed by the features of the trace generator.

4.4.4 Performance of Resource-driven TCG

We have performed some preliminary experiments on our running example using different values for k for the loop-k coverage criterion (X axis) and using a worst-case resource policy for the $\mathcal{M}_{ins}$ cost model. Our aim is to compare resource-aware TCG with the two instances of resource-driven TCG, the one that uses the naive trace-abstraction and the enhanced one. Figure 4.7a depicts the number of traces which will be explored in each case. It can be observed that the naive trace-abstraction generates a huge number of unfeasible traces and the growth is larger as k increases. Indeed, from $k = 6$ on, the system runs out of memory when computing them. The enhanced trace-abstraction reduces drastically the number of unfeasible traces and besides the difference w.r.t. this number in resource-aware is a (small) constant. Figure 4.7b shows the time to obtain the worst-case test case in each case. The important point to note is that resource-driven TCG outperforms resource-aware TCG in all cases, taking in average half the time w.r.t. the latter. We believe our results are promising and suggest that the larger the symbolic execution tree is (i.e., the more exhaustive TCG aims to be), the larger the efficiency gains of resource-driven TCG are. Furthermore, in a real system, the different test cases for resource-driven TCG could be computed in parallel and hence the benefits would be potentially larger.



(a) Number of trace

(b) Time

Figure 4.7: Resource-driven TCG: Experimental results

4.5 Conclusions

In this chapter, we have proposed resource-aware TCG, an extension of standard TCG with resources, whose purpose is to build resource consumption into the test cases. Resource-aware TCG can be lined up in the scope of performance engineering, an emerging software engineering practice that strives to build performance into the design and architecture of systems. Resource-aware TCG can serve different purposes. It can be used to test that a program meets performance criteria up to a certain degree of code coverage. It can compare two systems to find which one performs better in each test case. It could even help finding out what parts of the program consume more resources and can cause the system to perform badly. In general, the later a defect is detected, the higher the cost of remediation. Our approach allows thus that performance test efforts begin at the inception of the development project and extend through to deployment.

Previous work also considers extensions of standard TCG to generate resource consumption estimations for several purposes (see [ZC02, ANV08, HJK09] and their references). However, none of those approaches can generate symbolic resource estimations, as our approach does, neither take advantage of a resource policy to guide the TCG process. The most related work to our resource-driven approach is [RMV09], which proposes

to use an abstraction of the program in order to guide symbolic execution and prune the execution tree as a way to scale up. An important difference is that our trace-based abstraction is an over-approximation of the actual paths which allows us to select the most expensive paths. In contrast, their abstraction is an under-approximation which tries to reduce the number of test cases that are generated in the context of concurrent programming, where the state explosion problem is even more acute. Besides, our extension to infer the resources from the trace-abstraction and the idea to use it as a heuristics to guide the symbolic execution is new.

Chapter 5

Guided TCG

In this chapter we propose *Guided TCG* , a framework that allows the use of *selection* criteria to *guide* symbolic execution in CLP-based TCG. The following publication supports this work.

> José Miguel Rojas and Miguel Gómez-Zamalloa. **A Framework for Guided Test Case Generation in Constraint Logic Programming.** In *LOPSTR 2012, 22nd International Symposium on Logic-Based Program Synthesis and Transformation*, volume 7844 of *Lecture Notes in Computer Science*, pages 176–193. Springer, September 2012.

5.1 Introduction

Guided TCG can serve different purposes. It can be used to discover bugs in a program, to analyze reachability of certain parts of a program, to lead symbolic execution to stress more interesting parts of the program, etc. This chapter targets selective and unit testing. Selective testing aims at testing only specific paths of a program. Unit testing is a widely used software engineering methodology, where units of code (e.g. methods) are tested in isolation to validate their correctness. Incorporating the notion of selection criteria in our TCG framework represents one step towards fully supporting both unit and integration testing, a different methodology, where all the pieces of a system must be tested as a single unit.

Our Guided TCG is a heuristics that aims at steering symbolic execution, and thus TCG, towards specific program paths to generate more relevant test cases and filter out

less interesting ones with respect to a given selection criterion. The goal is to improve on scalability and efficiency by achieving a high degree of control over the coverage criterion and hence avoiding the exploration of unfeasible paths. In particular, we develop two instances of the framework: one for covering all the local paths of a method, and the other to steer TCG towards a selection of program points in the program under test. Both instances have been implemented and we provide experimental results to substantiate their applicability and effectiveness.

Observe that Guided TCG is inspired on, extends and generalizes the methodology presented in Chapter 4, where TCG computes resource consumption information and can be guided by resource policies. Thereby, a so-called resource policy was used to *select* certain paths of the program for which test cased are to be generated. Namely, resource-driven generated only test cases that adhere to the given resource policy, filtering out those that violate it.

The structure of the chapter is as follows. In the remainder of this section we describe our running example for this chapter. Section 5.2 introduces the generic framework for guided TCG. Section 5.3 presents an instantiation of the framework based on trace-abstractions and targeting structural coverage criteria. Section Section 5.4 reports on the implementation and empirical evaluation of the approach. Section 5.5 discusses a complementary strategy to further optimize the framework. Finally, Section 5.6 situates our work in the existing research space, sketches ongoing and future work and concludes.

5.1.1 Running Example

Figure 5.1 shows a Java program made up of three methods: `lcm` calculates the least common multiple of two integers, `gcd` calculates the greatest common divisor of two integers, and `abs` returns the absolute value of an integer. The right side of the figure shows the equivalent CLP-translated program. Method `lcm` is translated into predicates `lcm`, `cont`, `try` and `div`. As per Section 2.1.1, the translation preserves the control flow of the program and transforms iteration into recursion (e.g. method `gcd`). Note that the example has been chosen deliberately small and simple to ease comprehension. For readability, the actual CLP code has been simplified, e.g., input and output heap arguments are not shown, since they do not affect the computation. Nevertheless, recall that our current implementation (see Section 2.1.5) supports full sequential Java.

```
int lcm(int a,int b) {
  if (a < b) {
    int aux = a;
    a = b;
    b = aux;
  }
  int d = gcd(a,b);
  try {
    return abs(a*b)/d;
  } catch (Exception e) {
    return -1;                    ⓤ
  }
}

int gcd(int a,int b) {
  int res;
  while (b != 0) {
    res = a\%b;
    a = b;
    b = res;
  };
  return abs(a);
}

int abs(int a) {
  if (a >= 0)
    return a;                     ⓚ
  else
    return -a;
}
```

(a) Java source code

```
lcm([A,B],[R],_,_,E,lcm(1,[T])) :-
    A #>= B,
    cont([A,B],[R],_,_,E,T).
lcm([A,B],[R],_,_,E,lcm(2,[T])) :-
    A #<= B,
    cont([B,A],[R],_,_,E,T).
cont([A,B],[R],_,_,E,cont(1,[T,V])) :-
    gcd([A,B],[G],_,_,E,T),
    try([A,B,G],[R],_,_,E,V).
try([A,B,G],[R],_,_,E,try(1,[T,V])) :-
    M #= A*B,
    abs([M],[S],_,_,E,T),
    div([S,G],[R],_,_,E,V).
try([A,B,G],[R],_,_,exc,try(2,[])).
div([A,B],[R],_,_,ok,div(1,[])) :-
    B #\= 0,
    R #= A/B.
div([A,0],[-1],_,_,catch,div(2,[])).   ⓤ
gcd([A,B],[D],_,_,E,gcd(1,[T])) :-
    loop([A,B],[D],_,_,E,T).
loop([A,0],[F],_,_,E,loop(1,[T])) :-
    abs([A],[F],_,_,E,T).
loop([A,B],[E],_,_,G,loop(2,[T])) :-
    B #\= 0,
    body([A,B],[E],_,_,G,T).
body([A,B],[R],_,_,E,body(1,[T])) :-
    B #\= 0,
    M #= A mod B,
    loop([B,M],[R],_,_,E,T).
body([A,0],[R],_,_,exc,body(2,[])).
abs([A],[A],_,_,ok,abs(1,[])) :-
    A #>= 0.                                ⓚ
abs([A],[-A],_,_,ok,abs(2,[])) :-
    A #< 0.
```

(b) CLP-translation

Figure 5.1: Guided TCG Example: Java (left) and CLP-translated (right) programs.

5.2 A Generic Framework for Guided TCG

The TCG framework as defined so far has been used in the context of coverage criteria only consisting of a termination criterion. In order to incorporate a selection criterion, one can employ a post-processing phase where only the test cases that are sufficient to satisfy the selection criterion are selected by looking at their traces. This is however not

an appropriate solution in general due to the exponential explosion of the paths that have to be explored in symbolic execution. Instead, we now aim at using the selection criterion to drive the TCG process towards satisfying paths, stressing to avoid as much as possible the exploration of irrelevant and redundant ones. The key idea that allows us to guide the TCG process is to use the trace terms of our CLP-translated program as input arguments. Let us recall that we could either supply fully or partially instantiated traces, the latter ones represented by including free logic variables within the trace terms. This allows guiding, completely or partially, the symbolic execution towards specific paths.

5.2.1 Redefining Coverage Criteria

As shown in Definition 2.1.3, so far we have been interested in covering *all* feasible paths of the program under test w.r.t. a termination criterion. Now, our goal is to improve on scalability by taking into account a *selection criterion* as well. First, let us define a *coverage criterion* as a pair of two components $\langle TC, SC \rangle$. TC is a *termination criterion* that ensures finiteness of symbolic execution. This can be done either based on execution steps or on loop iterations. In this chapter, we adhere to loop-k, which limits to a threshold k the number of allowed loop iterations and/or recursive calls (of each concrete loop or recursive method). SC is a *selection criterion* that steers TCG to determine which paths of the symbolic execution tree will be explored. In other words, SC decides which test cases the TCG must produce. In the rest of this chapter we focus on the following two coverage criteria:

- all-local-paths: It requires that all *local* execution paths within the method under test are exercised up to a loop-k limit. This has a potential interest in the context of unit testing, where each method must be tested in isolation.

- program-points(P): Given a set of program points P, it requires that all of them are exercised by at least one test case up to a loop-k limit. This criterion is the most appropriate choice for bug-detection and reachability verification purposes. A particular case of it is *statement coverage* (up to a limit), where all statements in a program or method must be exercised.

5.2.2 Trace-guided TCG

We rely on the definition of trace-guided TCG (refDefdef:tgtcg) from Chapter 4.

Given a method m, a coverage criterion $\mathcal{C}$, and a (possibly partial) trace π, trace-guided TCG generates the set of test cases with traces, denoted $tgTCG(M, \mathcal{C}, \pi)$, obtained for all successful branches in $\mathcal{T}_m^C$ with root $m(Args_{in}, Args_{out}, H_{in}, H_{out}, E, \pi)$.

Let us recall that TCG guided by a trace $\pi \equiv m(k, P, \langle \pi_{c_i}, \ldots, \pi_{c_m} \rangle)$ generates exactly none, one, or several test cases depending on whether π is feasible or not, and complete or partial. In the remainder of this chapter, we will not use, and therefore not show, the argument P in the trace terms.

For convenience, let define the firstOf-tgTCG(M, TC, π) to be the set corresponding to the leftmost successful branch in $\mathcal{T}_m^{TC}$.

Now, relying on trace-guided TCG and on the existence of a *trace generator* we define a generic scheme of *guided TCG.*

Definition 5.2.1 (guided TCG)**.** *Given a method m; a coverage criterion $CC = \langle TC, SC \rangle$; and a trace generator $TraceGen$, that generates, on demand and one by one, (possibly partial) traces according to CC. Guided TCG is defined as the following algorithm:*

```
Input:  M, and ⟨TC, SC⟩
TestCases = {}
while TraceGen has more traces and TestCases does not satisfy SC
   Ask TraceGen to generate a new trace in Trace
   TestCases = TestCases ∪ firstOf-tgTCG(M, TC, Trace)
Output:  TestCases
```

The intuition is as follows: The trace generator generates a trace, possibly using for that SC, TC and the current $TestCases$. If the generated trace is feasible, then the first solution of its trace-guided TCG is added to the set of test cases. The process finishes either when SC is satisfied, or when the trace generator has already generated all traces up to TC. If the trace generator is complete (see below), this means that SC cannot be satisfied within the limit imposed by TC. Observe that for some selection criteria, e.g., all-local-paths, the calls to firstOf-tgTCG can be computed in parallel (as discussed in Section 4.4.2).

Example 18 Let us consider the TCG for method `lcm` with program-points for points μ and κ as selection criterion. Observe the correspondence of these program points in

both the Java and CLP code of Figure 5.1. Let us assume that the trace generator starts generating the following two traces:

t_1 : `lcm(1,[cont(1,[G,check(1,[A,div(2,[])])])])`
t_2 : `lcm(2,[cont(1,[G,check(1,[A,div(2,[])])])])`

The first iteration does not add any test case since trace t_1 is unfeasible. Trace t_2 is proved feasible and a test case is generated. The selection criterion is now satisfied and therefore the process finishes. The obtained test case is shown in Example 23. □

On Soundness, Completeness and Effectiveness. Intuitively, a concrete instantiation of the guided TCG scheme is *sound* if all test cases it generates satisfy the coverage criterion and *complete* if it never reports that the coverage criterion is not satisfied when it is indeed satisfiable. *Effectiveness* is related to the number of iterations the algorithm performs. Those three features depend completely on the trace generator. As discussed in Section 4.4.2, hereby we refer to trace generators as being sound if every trace it generates satisfies the coverage criterion, and complete if it produces an over-approximation of the set of traces satisfying it, and effective, in terms of the number of unfeasible traces it generates: the larger the number, the less effective the trace generator.

5.3 Trace Generators for Structural Coverage Criteria

This section presents a general approach for building sound, complete and effective trace generators for structural coverage criteria by means of program transformations. We then instantiate the approach for the all-local-paths and program-points coverage criteria and propose concrete Prolog implementations of the guided TCG scheme for both of them. Let us first recall the definition of the *trace-abstraction* of a program (Definition 4.4.3), which will be the basis for defining our trace generators.

> *Given a CLP-translated program with traces P, its trace-abstraction is obtained as follows: for every rule of P, (1) remove all atoms in the body of the rule except those corresponding to rule calls, and (2) remove all arguments from the head and from the surviving atoms of (1) except the last one (i.e., the trace term).*

Example 19 Figure 5.2 shows the trace-abstraction of our CLP-translated program of Figure 5.1. Let us observe that it basically corresponds to its control-flow graph.

```
lcm(lcm(1,[T])) :- cont(T).
lcm(lcm(2,[T])) :- cont(T).
cont(cont(1,[T,V])) :- gcd(T), try(V).
try(try(1,[T,V])) :- abs(T), div(V).
try(try(2,[])).
div(div(1,[])).
div(div(2,[])).
gcd(gcd(1,[T])) :- loop(T).
loop(loop(1,[T])) :- abs(T).
loop(loop(2,[T])) :- body(T).
body(body(1,[T])) :- loop(T).
body(body(2,[])).
abs(abs(1,[])).
abs(abs(2,[])).
```

Figure 5.2: Trace-abstraction

□

The trace-abstraction can be directly used as a trace generator as follows: (1) Apply the termination criterion in order to ensure finiteness of the process. (2) Select, in a post-processing, those traces that satisfy the selection criterion. Such a trace generator produces on backtracking a superset of the set of traces of the program satisfying the coverage criterion. Note that, this can be done as long as the criteria are structural. The obtained trace generator is by definition sound and complete. However, it can be very ineffective and inefficient due to the large number of unfeasible and/or unnecessary traces that it can generate. In the following, we propose two concrete, and more effective, instantiations for the all-local-paths and program-points coverage criteria. In both cases, this is done by taking advantage of the notion of partial traces and the implicit information on the concrete coverage criteria.

5.3.1 An Instantiation for the all-local-paths Coverage Criterion

Let us start from the trace-abstraction program and apply a syntactic program slicing which removes from it the rules that do not belong to the considered method.

Definition 5.3.1 (slicing for all-local-paths coverage criterion)**.** *Given a trace-abstraction program P and an entry method M:*

1. *Remove from P all rules that do not belong to method M.*
2. *In the bodies of remaining rules, remove all calls to rules which are not in P.*

The obtained sliced trace-abstraction, together with the termination criterion, can be used as a trace generator for the all-local-paths criterion for a method. The generated traces will have free variables in those trace arguments that correspond to the execution of other methods, if any.

```
lcm(lcm(1,[T])) :- cont(T).
lcm(lcm(2,[T])) :- cont(T).
cont(cont(1,[G,T])) :- try(T).
try(try(1,[A,T])) :- div(T).
try(try(2,[])).
div(div(1,[])).
div(div(2,[])).
```

```
lcm(1,[cont(1,[G,try(1,[A,div(1,[])])])])
lcm(1,[cont(1,[G,try(1,[A,div(2,[])])])])
lcm(1,[cont(1,[G,try(2,[])])])
lcm(2,[cont(1,[G,try(1,[A,div(1,[])])])])
lcm(2,[cont(1,[G,try(1,[A,div(2,[])])])])
lcm(2,[cont(1,[G,try(2,[])])])
```

Figure 5.3: Slicing of method `lcm` for all-local-paths criterion.

Example 20 Figure 5.3 shows on the left the sliced trace-abstraction for method `lcm`. On the right is the finite set of traces that is obtained from such trace-abstraction for any loop-K termination criterion. Observe that the free variables `G`, resp. `A`, correspond to the sliced away calls to methods `gcd` and `abs`. □

Let us define the predicates: `computeSlicedProgram(M)`, that computes the sliced trace-abstraction for method M as in Definition 5.3.1; `generateTrace(M,TC,Trace)`, that returns in its third argument, on backtracking, all partial traces computed using such sliced trace-abstraction, limited by the termination criterion `TC`; and `traceGuidedTCG(M,TC,Trace,TestC` which computes on backtracking the set tgTCG(M,Trace,TC) in Definition 4.4.1, failing if the set is empty, and instantiating on success `TestCase` and `Trace` (in case it was partial). The guided TCG scheme in Definition 5.2.1, instantiated for the all-local-paths criterion,

can be implemented in Prolog as follows:

```
(1) guidedTCG(M,TC) :-
(2)     computeSlicedProgram(M),
(3)     generateTrace(M,TC,Trace),
(4)     once(traceGuidedTCG(M,Trace,TC,TestCase)),
(5)     assert(testCase(M,TestCase,Trace)),
(6)     fail.
(7) guidedTCG(_,_).
```

Intuitively, given a (possibly partial) trace generated in line `(3)`, if the call in line `(4)` fails, then the next trace is tried. Otherwise, the generated test case is asserted with its corresponding trace which is now fully instantiated (in case it was partial). The process finishes when `generateTrace/3` has computed all traces, in which case it fails, making the program exiting through line `(7)`.

Example 21 The following test cases are obtained for the all-local-paths criterion for method `lcm`:

Constraint store	*Trace*
{A>=B}	lcm(1,[cont(1,[gcd(1,[loop(1,[abs(1,[])])]), try(1,[abs(1,[]),div(1,[])])])])
{A=B=0,Out=-1}	lcm(1,[cont(1,[gcd(1,[loop(1,[abs(1,[])])]), try(1,[abs(1,[]),div(2,[])])])])
{B>A}	lcm(2,[cont(1,[gcd(1,[loop(1,[abs(1,[])])]), try(1,[abs(1,[]),div(1,[])])])])

This set of 3 test cases achieves full code and path coverage on method `lcm` and is thus a perfect choice in the context of unit-testing. In contrast, the original, non-guided, TCG scheme with loop-2 as termination criterion produces 9 test cases. □

5.3.2 An Instantiation for the program-points Coverage Criterion

Let us start by considering a simplified version of the program-points criterion so that only one program point is allowed, denoted as program-point. Starting again from the trace-abstraction program, we propose a syntactic program slicing that filters away the part of the program which is not relevant for the paths that do not visit the program point of interest.

Definition 5.3.2 (slicing for program-point coverage criterion)**.** *Given a trace-abstraction program P, a program point of interest pp, and an entry method M, the sliced program P' is computed as follows:*

1. *Initialize P' to be the empty program, and a set of clauses L with the clause corresponding to pp.*
2. *For each c in L which is not the clause for M, add to L all clauses in P whose body has a call to the predicate of clause c, and iterate until the set L stabilizes.*
3. *Add to P' all clauses in L.*
4. *Remove all calls to rules which are not in P' from the bodies of the rules in P'.*

The obtained sliced program, together with the termination criterion, can be used as a trace generator for the program-point criterion. The generated traces can have free variables representing parts of the execution which are not related (syntactically) to the paths visiting the program point of interest.

Example 22 Figure 5.4 shows on the left the sliced trace-abstraction program (using Definition 5.3.2) for method `lcm` and program point Ⓤ from Figure 5.1, i.e. the `return` statement within the `catch` block. On the right of the same figure, the traces obtained from such slicing using loop-2 as termination criterion. □

```
lcm(lcm(1,[T])) :- cont(T).
lcm(lcm(2,[T])) :- cont(T).
cont(cont(1,[G,T])) :- try(T).
try(try(1,[A,T])) :- div(T).
div(div(2,[])).
```

```
lcm(1,[cont(1,[G,try(1,[A,div(2,[])])])])
lcm(2,[cont(1,[G,try(1,[A,div(2,[])])])])
```

Figure 5.4: Slicing for program-point coverage criterion with pp=Ⓤ from Figure 5.1.

Consider again predicates `computeSlicedProgram/2`, `generateTrace/4` and `traceGuidedTCG/4` with the same meaning as in Section 5.3.1, but being the first two now based on Definition 5.3.2 and extended with the program-point argument `PP`. The

guided TCG scheme in Definition 5.2.1, instantiated for the program-points criterion, can be implemented in Prolog as follows:

```
(1) guidedTCG(M,[],TC) :- !.
(2) guidedTCG(M,[PP|PPs],TC) :-
(3)     computeSlicedProgram(M,PP),
(4)     generateTrace(M,PP,TC,Trace),
(5)     once(traceGuidedTCG(M,Trace,TC,TestCase)), !,
(6)     assert(testCase(M,TestCase,Trace)),
(7)     removeCoveredPoints(PPs,Trace,PPs'),
(8)     guidedTCG(M,PPs',TC).
(9) guidedTCG(M,[PP|_],TC) :- .
```

Intuitively, given the first remaining program point of interest `PP` (line 2), a trace generator is computed and used to obtain a (possibly partial) trace that exercises `PP` (lines 3–4). Then, if the call in line 5 fails, another trace for `PP` is requested on backtracking. When there are not more traces (i.e., line 4 fails) the process finishes through line 9 reporting that `PP` is not reachable within the imposed `TC`. If the call in line 5 succeeds, the generated test case is asserted with its corresponding trace (now fully instantiated in case it was partial), the remaining program points which are covered by `Trace` are removed obtaining `PPs'` (line 7), and the process continues with `PPs'`. Note that a new sliced program is computed for each program point in `PPs'`. The process finishes through line 1 when all program points have been covered.

The above implementation is valid for the general case of program-points criteria with any finite set size. The trace generator, instead, has been deliberately defined for just one program point since this way the program slicing can be more aggressive, hence saving the generation of unfeasible traces.

Example 23 The following test case is obtained for the program-points criterion for method `lcm` and program points ⓜ and ⓚ. This particular case illustrates specially well how guided TCG can reduce the number of produced test cases through adequate control of the selection criterion.

Constraint store	*Trace*
`{A=B=0,Out=-1}`	`lcm(1,[cont(1,[gcd(1,[loop(1,[abs(1,[])])]),` `try(1,[abs(1,[]),div(2,[])])])])`

□

Method Info			Standard TCG			Guided TCG			
Class.Name	**BCs**	**Tt**	**T**	**N**	**CC**	**Tg**	**Ng**	**CCg**	**GT/UT**
Seq.elemAt	98	45	18	24	100%	9	5	100%	6/1
Seq.insertAt	220	85	41	39	100%	14	6	100%	8/2
Seq.removeAt	187	76	35	36	100%	10	4	100%	5/1
Seq.replaceAt	163	66	35	36	100%	9	4	100%	5/1
PQ.insert	357	144	148	109	100%	10	3	100%	4/1
PQ.remove	158	69	8	12	100%	20	7	100%	15/8
BST.addAll	260	125	1491	379	100%	22765	18	100%	151/133
BST.find	228	113	76	62	100%	82	5	100%	7/2
BST.findAll	381	178	1639	330	100%	1266	4	100%	6/2
BST.insert	398	184	2050	970	100%	1979	9	100%	18/9
BST.remove	435	237	741	365	98%	3443	26	98%	204/178
HPQ.insert	322	132	215	43	100%	26	5	100%	6/1
HPQ.remove	394	174	1450	40	100%	100	8	100%	19/11

Table 5.1: Experimental results for the all-local-paths criterion

5.4 Experimental Evaluation

We have implemented the guided TCG schemes for both all-local-paths and program-points coverage criteria as proposed in Section 5.3, and integrated them within PET (Section 2.1.5). In this section we report on some experimental results which aim at demonstrating the applicability and effectiveness of guided TCG. The experiments have been performed using as benchmarks a selection of classes from the net.datastructures package. In particular, we have used as "methods-under-test" the most relevant public methods of the classes *NodeSequence*, *SortedListPriorityQueue*, *BinarySearchTree* and *HeapPriorityQueue*, abbreviated respectively as *Seq*, *PQ*, *BST* and *HPQ*.

Table 5.1 aims at demonstrating the effectiveness of the guided TCG scheme for the all-local-paths coverage criterion. This is done by comparing it to standard way of implementing the all-local-paths coverage criterion, i.e., first generating all paths up to the termination criterion using standard TCG by symbolic execution, and then applying a filtering so that only the test cases that are necessary to meet the all-local-paths selection

criterion are kept. Each row in the table corresponds to the TCG of one method using standard TCG vs. using guided TCG. For each method we provide: The number of reachable bytecode instructions (**BCs**) and the time of the translation of Java bytecode to CLP (**Tt**), including parsing and loading all reachable classes ; the time of the TCG process (**T**), the number of generated test cases before the filtering (**N**), and the code coverage achieved using standard TCG (**CC**); and the time of the TCG process (**Tg**), the number of generated test cases (**Ng**), the code coverage achieved (**CCg**), and the number of generated/unfeasible traces using guided TCG (**GT/UT**). All times are in milliseconds and are obtained as the arithmetic mean of five runs on an Intel(R) Core(TM) i5-2300 CPU at 2.8GHz with 8GB of RAM, running Linux Kernel 2.6.38. The code coverage measures, given a method, the percentage of its bytecode instructions which are exercised by the obtained test cases. This is a common measure in order to reason about the quality of the obtained test cases. As expected, the code coverage is the same in both approaches, and so is the number of obtained test cases. Otherwise, this would indicate a bug in the implementation.

Let us observe that the gains in time are significant for most benchmarks (column **T** vs. column **Tg**). There are however three notable exceptions for methods PQ.remove, BST.addAll and BST.remove, for which the guided TCG scheme behaves worse than the standard one, especially for BST.addAll. This happens in general when the control-flow of the method is complex, hence causing the trace generator to produce an important number of unfeasible traces (see last column). Interestingly, these cases could be statically detected using a simple syntactic analysis which looks at the control flow of the method. Therefore the system could automatically decide which methodology to apply. Moreover, Section 5.5 presents a trace-abstraction refinement that will help in improving guided TCG for programs whose control-flow is determine mainly by integer linear constraints. Other classes of programs, e.g. BST.addAll, require a more sophisticated analysis, since their control-flow are strongly determined by object types and dynamic dispatch information. This discussion and further refinement is left out of the scope of this chapter.

Table 5.2 aims at demonstrating the effectiveness of the guided TCG scheme for the program-points coverage criterion. For this aim, we have implemented the support in the standard TCG scheme to check the program-points selection criterion dynamically while the test cases are generated, in such a way that the process terminates when all program points are covered. Note that, in the worst case this will require generating the whole symbolic execution tree, as the standard TCG does. Table 5.2 compares the effectiveness

Method Info	Standard TCG			Guided TCG			
Class.Name	**T**	**N**	**CC**	**Tg**	**Ng**	**CCg**	**GT/UT**
Seq.elemAt	9	10	100%	6	3	100%	3/0
Seq.insertAt	39	36	100%	8	3	100%	3/0
Seq.removeAt	19	16	100%	8	3	100%	3/0
Seq.replaceAt	19	16	100%	8	3	100%	3/0
PQ.insert	149	109	100%	9	3	100%	3/0
PQ.remove	9	12	100%	5	3	100%	3/0
BST.addAll	1501	379	100%	284	2	100%	4/2
BST.find	77	62	100%	10	3	100%	3/0
BST.findAll	1634	330	100%	8	3	100%	3/0
BST.insert	2197	969	100%	35	3	100%	3/0
BST.remove	238	104	98%	61	3	98%	28/25
HPQ.insert	209	43	100%	24	3	100%	3/0
HPQ.remove	1385	38	100%	15	3	100%	3/0

Table 5.2: Experimental results for the program-points criterion

of this methodology against that of the guided TCG scheme. Again, each row in the table corresponds to the TCG of one method using standard TCG vs. using guided TCG, providing for both schemes the time of the TCG process (**T** vs **Tg**), the number of generated test cases (**N** vs **Ng**), the code coverage achieved (**CC** vs **CCg**), and the number of generated/unfeasible traces using guided TCG (**GT/UT**). We have selected three program points for each method with the aim of covering as much code as possible. In all cases, such selection of program points allows obtaining the same code coverage as with the standard TCG even without the selection criterion (i.e. 100% coverage for all methods except 98% for BST.remove because of dead code). Let us observe that the gains in time are huge (column **T** vs. column **Tg**), ranging from one to two orders of magnitude, except for the simplest methods, for which the gain, still being significant, is not so notable. These results are witnessed by the low number of unfeasible traces that are obtained (column **GT/UT**), hence demonstrating the effectiveness of the trace generator defined in Section 5.3.2.

Overall, we believe our experimental results support our initial claims about the potential interest of guiding symbolic execution and TCG by means of trace-abstractions. With the exception of some particular cases that deserve further study, our results demonstrate that we can achieve high code coverage without having to explore many unfeasible paths, with the additional advantage of discovering high quality (less in number and better selected) test cases.

5.5 Trace-Abstraction Refinement

As the above experimental results suggest, there are still cases where the trace-abstraction as defined in Definition 4.4.3 may still compromise the effectiveness of the guided TCG, because of the generation of too many unfeasible paths. This section discusses a complementary strategy to further optimize the framework. In particular, we propose a heuristics that aims to refine the trace-abstraction with information taken from the original program that will help reduce the number of unfeasible paths at symbolic execution. The goal is to reach a balanced level of refinement in between the original program (full refinement) and the trace-abstraction (empty refinement). Intuitively, the more information we include, the less unfeasible paths symbolic execution will explore, but the more costly it becomes.

The refinement algorithm consists of two steps: First, in a fixpoint analysis we approximate the instantiation mode of the variables in each predicate of the CLP-translated program. In other words, we infer which variables will be constrained or assigned a concrete value at symbolic execution time. In a second step, by program transformation, the trace-abstraction is enriched with clause arguments corresponding to the inferred variables, and with those goals in which they are involved.

5.5.1 Approximating instantiation modes

We develop a static analysis, similar to [CGLH05, Deb89], to soundly approximate the instantiation mode of the input argument variables in the program at symbolic execution time. The analysis is implemented as a fixpoint computation over the simple abstract domain $\{static, dynamic\}$. Namely, *dynamic* means that nothing was inferred about a variable and it will therefore remain a free unconstrained variable during symbolic execution; and *static* means that the variable will unify with a concrete value or will be constrained during symbolic execution. The analysis's result is a set of assertions in the

form $\langle P, \mathcal{V} \rangle$ where P is a predicate name and $\mathcal{V}$ is the set of variables in P, each associated with an abstract value from the domain.

This analysis receives as input a CLP-translated program and a set of initial entries (predicate names). An event queue $\mathcal{Q}$ is initialized with this set of initial entries. The algorithm starts to process the events of $\mathcal{Q}$ until no more events are scheduled. In each iteration, an event p is removed from $\mathcal{Q}$ and processed as follows: Retrieve previously stored information $\psi \equiv \langle p, \mathcal{V} \rangle$ if any exists; else set $\psi \equiv \langle p, \emptyset \rangle$. For each rule r defining p, a new $\mathcal{V}_r$ is obtained by evaluating the body of r. The joint operation on the underlying abstract domain is performed to obtain $\mathcal{V}' \Leftarrow joint(\mathcal{V}, \mathcal{V}_r)$. If $\mathcal{V} \not\equiv \mathcal{V}'$ then set $\mathcal{V} \Leftarrow \mathcal{V}'$ and reschedule every predicate that calls p; else, if $\psi' \equiv \psi$ there is no need to recompute the calling predicates and the algorithm continues. That will ensure backward propagation of approximated instantiation modes. To propagate forward, the evaluation of r will schedule one event per call within its body. The process continues until a fixpoint is reached.

5.5.2 Constructing the trace-abstraction refinement

This is a syntactic program transformation step of the refinement. It takes as input the original CLP-program and the instantiation information inferred in the first step and outputs a trace-abstraction refinement program. For each rule r of a predicate p in the program, the algorithm retrieves $\langle p, \mathcal{V} \rangle$. We denote $\mathcal{V}_s$ the projection of all variables in $\mathcal{V}$ whose inferred abstract value is *static*. The algorithm adds to the trace-abstraction refinement a new rule r' whose list of arguments is $\mathcal{V}_s$. The body of r' is constructed by traversing the body $b_1, \ldots, b_n$ of r and including 1) all guards and arithmetic operations b_i involving $\mathcal{V}_s$, and 2) all calls to other predicates, with the corresponding projection of $\mathcal{V}_s$ over the arguments of the calls.

Example 24 Consider the Java example of Fig. 5.5 (left side). Function `pow` implements a exponentiation algorithm for positive integer exponents. Its CLP counterpart is shown at the right of the figure. The instantiation modes inferred by the first stage of our algorithm is presented at the right-bottom part of the figure. One can observe that variable `B` (the base of the exponentiation) remains *dynamic* all along the program, because it is never assigned any concrete value nor constrained by any guard. On the other hand, variable `E`'s final abstract value is *static*, since it is constrained by `0` and the also *static* variable `I` in rules `if` and `loop`. The following is the refined trace-abstraction that our algorithm

```
void arraypow(int a[],int e)
{
  int i=0;
  int n=a.length;
  for (i=0; i<n; i++)
    if (i%2==0)
      a[i]=pow(a[i],e);
}
int pow(int b, int e){
  if (e >= 0) {
    int pow = 1;
    while (i <= e) {
      pow *= b;
      i++;
    }
    return pow;
  } else return -1;
}
```

(a) Java source code

```
pow([B,E],[R],_,_,F,pow(1,[T])) :-
     if([B,E],[R],_,_,F,T).
if([B,E],[-1],_,_,F,if(1,[])) :-
     E #< 0.
if([B,E],[R],_,_,F,if(2,[T])) :-
     E #>= 0,
     loop([B,E,1,1],[R],_,_,F,T).
loop([B,E,I,P],[P],_,_,ok,loop(1,[])) :-
     I #> E.
loop([B,E,I,P],[R],_,_,F,loop(2,[T])) :-
     I #=< E,
     Pp #= P*B,
     Ip #= I+1,
     loop([B,E,Ip,Pp],[R],_,_,F,T).
```

(b) CLP-translation

$$\langle \texttt{pow}, \{\texttt{B}= \textit{dynamic}, \texttt{E}= \textit{static}\}\rangle$$
$$\langle \texttt{if}, \{\texttt{B}= \textit{dynamic}, \texttt{E}= \textit{static}\}\rangle$$
$$\langle \texttt{loop}, \{\texttt{B}= \textit{dynamic}, \texttt{E}= \textit{static}, \ \texttt{I}= \textit{static}, \texttt{P}= \textit{dynamic}\}\rangle$$

(c) Inferred instantiation modes

Figure 5.5: Trace-abstraction refinement

constructs:

```
pow([E],pow(1,[T])) :- if([E],T).
if(E],if(1,[])) :- E #< 0.
if([E],if(2,[T])) :- E #>= 0), loop([E,1,1],T).
loop([E,I],loop(1,[])) :- I #> E.
loop([E,I],loop(2,[T])) :- I #=< E, Ip #= I+1, loop([E,Ip],T).
```

To illustrate how the trace-abstraction refinement can improve on effectiveness of the guided TCG, let us observe method `arraypow`. It iterates over all the elements of an input array `a` and calls function `pow` to update all even positions of the array by raising their values to the power of the integer input argument `e`. We report on the following TCG performance results for this example and a coverage criterion $\langle \mathsf{loop\text{-}2}, \{\}\rangle$:

- Standard non-guided TCG of this example generates 11 test cases.
- Trace-abstraction guided TCG with the empty refinement generates 497 possibly (un)feasible traces to be tested.
- Trace-abstraction guided TCG with our trace-abstraction refinement reduces the number of possibly (un)feasible traces to be tested to 161.

□

These preliminary, yet promising, results unveil the potential integration of the trace-abstraction refinement algorithm presented in this section with the general guided TCG framework developed in this chapter. The refinement is complementary to the slicings presented in Section 5.3 without any modification. Unfortunately, the slicings could produce a loss of important information added by the refinement. This could be however improved by means of simple syntactic analyses on the sliced parts of the program. A deeper study of these issues remains as future work.

5.6 Conclusions

Previous work also uses abstractions to guide symbolic execution and TCG by several means and for different purposes. Fundamentally, abstraction aims to reduce large data domains of a program to smaller domains [LBBO01]. One of the most relevant to ours is [Bal03], where predicate abstraction, model checking and SAT-solving are combined to produce abstractions and generate test cases for C programs, with good code coverage, but depending highly on an initial set of predicates to avoid infeasible program paths. Rugta *et al.* [RMV09] also proposes to use an abstraction of the program in order to guide symbolic execution and prune the execution tree as a way to scale up. Their abstraction is an under-approximation which tries to reduce the number of test cases that are generated in the context of concurrent programming, where the state explosion is in general problematic.

The main contribution of this chapter is the development of a methodology for Guided TCG that allows to guide the process of test generation towards achieving more selective and interesting structural coverage. Implicit is the improvement in the scalability of TCG by guiding symbolic execution by means of trace-abstractions, since we gain more control

over the symbolic execution state space to be explored. Moreover, whereas the main goal of our CLP-based TCG framework has been the exhaustive testing of programs, our new Guided TCG framework unveil new potential applications areas. Namely, the all-local-paths and program-points Guided TCG schemes we have presented in this chapter, enable us to explore on the automation of other interesting software testing practices, such as selective and unit testing, goal-oriented testing and bug detection.

The effectiveness and applicability of Guided TCG is substantiated by an implementation within PET and encouraging experimental results. Nevertheless, our current and future work involves a more thorough experimental evaluation of the framework and the exploration of the new application areas in software testing. In a different line, a particularly challenging goal has been triggered which consists in developing static analysis techniques to achieve optimal refinement levels of the trace-abstraction programs. Last but not least, we plan to further study the generalization and integration of other interesting coverage criteria to our Guided TCG framework.

Chapter 6

Heap Solver

This chapter presents a novel approach to efficiently handling heap-manipulating programs in symbolic execution and TCG. The work presented in this chapter has been published in:

> Elvira Albert and María García de la Banda and Miguel Gómez-Zamalloa and José Miguel Rojas and Peter Stuckey. **A CLP Heap Solver for Test Case Generation.** In *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue*, 13(4-5):721–735. Cambridge University Press, July 2013.

6.1 Introduction

One of the main challenges in symbolic execution for software testing today is to efficiently handle heap-manipulating programs [PV09]. These are programs that create and use dynamically heap-allocated data structures. In order to ensure reliability, the testing process needs to consider all possible shapes these data structures can take. This creates a significant scalability issue for symbolic execution, since an exponential number of shapes may be built due to the *aliasing* of references.

To handle unknown heap structures, existing systems such as PET (Section 2.1.5) and SPF (Section 3.3.1), use *lazy initialization*. As described in Section 2.1.3, lazy initialization enables symbolic execution to non-deterministically consider all aliasing configurations. The motivation of the work presented in this chapter stems from the observation that

branching due to *aliasing* choices can be made "more lazily" than in lazy initialization. As we will see, delaying aliasing choices is crucial for the scalability of TCG.

Let us motivate our approach by symbolically executing the `m` method appearing on Figure 6.1a, assuming that the executions of `call_a` and `call_b` do not modify the heap. The symbolic derivation tree computed using standard lazy initialization (as in, e.g., PET and SPF) is shown in Figure 6.2a. Note that before a field is accessed, the execution branches if it can alias with previously accessed fields. For example, the second field access `z.f` branches in order to consider the possible aliasing with the previously accessed `x.f`. Similarly, the write access to `y.f` must consider all possible aliasing choices with the two previous accessed fields `x.f` and `z.f`. This ensures that the effect of the field access is known within each branch. For example, in the leftmost branch the statement `y.f=x.f+1` assigns -4 to `x.f`, `y.f` and `z.f`, since in that branch all these objects are aliased. The advantage of this approach is that (at least for this program) by the time we reach the `if` statement we know the result of the test, since each variable is fixed. However, such early branching creates a combinatorial explosion problem since, for example, `call_a` is symbolically executed in two branches and `call_b` in five.

Our challenge is to be able to execute symbolically as shown in Figure 6.2b, where branching only occurs due to explicit branching in the program, rather than to aliasing. For this purpose, we present a *heap solver* that handles the *disjunction* due to aliasing of references. In particular, at instruction 6 the solver will carry the following conditional information for `x.f`' (the current value of field `f` of `x`): $x = z \rightarrow x.f' = z.f \wedge x \neq z \rightarrow x.f' = x.f$ indicating that if `x` and `z` are aliased, then `x.f`' will take its value from `z.f` and, otherwise, from `x.f`. Once the conditional statement at 8 is executed and we learn that `x` and `z` are aliased (in the `then` branch), we need to look up *backwards* in the heap and propagate this unification so that instruction 6 can be fully executed. This will allow the symbolic execution of `call_d(y.f)` with a known value for `y.f`. Our heap solver works on a novel internal representation of the heap that encodes the disjunctive information and easily allows looking up backwards in the heap. In addition, it is possible to provide *heap assumptions* on non-aliasing, non-sharing and acyclicity of heap-allocated data in the initial state. The heap solver can take these assumptions into account to discard aliasing that is known not to occur for some input data. Importantly, our heap solver can be used by any TCG tool for imperative languages through its interface heap operations.

We have integrated our solver in PET and performed an experimental evaluation on methods from the net.datastructures package. Our results demonstrate that our approach

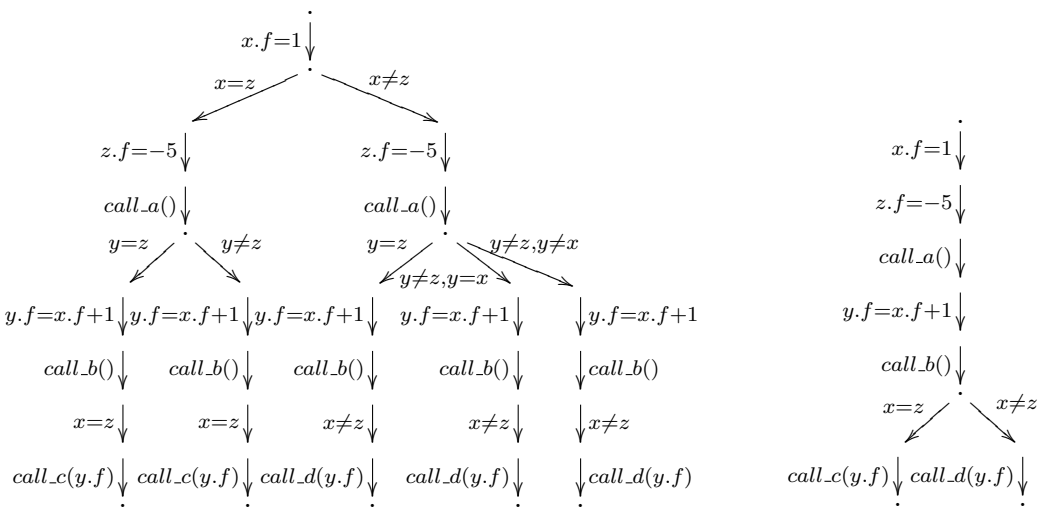
```
void m(Ref x, Ref y, Ref z)
{
  x.f=1;
  z.f=-5;
  call_a();
  y.f=x.f+1;
  call_b();
  if (x==z)
    call_c(y.f);
  else
    call_d(y.f);
}
```

(a) Java source code

```
m_init([r(X),r(Y),r(Z)],[],H_in,H_out) :-
    m([X,Y,Z],[],H_in,H_out).
m([X,Y,Z],[],H_in,H_out) :-
     set_field(H_in,X,f,1,H_1),
     set_field(H_1,Z,f,-5,H_2),
     call_a([],[],H_2,H_3),
     get_field(H_3,X,f,Xf,H_4),
     #=(Yf,Xf+1),
     set_field(H_4,Y,f,Yf,H_5),
     call_b([],[],H_5,H_6),
     cond([X,Y,Z],[],H_6,H_out).
cond([X,Y,Z],[],H_6,H_out) :-
     ref_eq(H_6,X,Z,H_7),
     get_field(H_7,Y,f,Yf,H_8),
     call_c([Yf],[],H_8,H_out).
cond([X,Y,Z],[],H_6,H_out) :-
     ref_neq(H_6,X,Z,H_7),
     get_field(H_7,Y,f,Yf,H_8),
     call_d([Yf],[],H_8,H_out).
```

(b) CLP-translation

Figure 6.1: Heap Solver: Motivating example



(a) Lazy Initialization

(b) Heap Solver

Figure 6.2: Symbolic Execution Trees: Lazy Initialization and Heap Solver

can handle heap-manipulating programs efficiently.

The structure of this chapter is as follows. Section 6.1.1 describes a motivating example and revisits some notions of our CLP-based TCG framework. Section 6.2 develops our heap solver, providing the details of its internal representation, handled operations and propagation mechanisms. Section 6.3 extends the heap solver with support for heap assumptions. Section 6.4 reports on implementation and validation results; and finally, Section 6.5 concludes and summarizes related work.

6.1.1 Motivating Example

Figure 6.1b shows the CLP-translated code of method m, which was obtained automatically from the bytecode of m. For simplicity, we have omitted the exception flag (*ExFlag*) argument and consequently all rules capturing exceptional behaviour when the references are null, since they do not require any special treatment in the heap solver. Thus, our initial predicate, m_{init}, assumes that the three input references are non-null (since they match r(_)) and invokes m. We have also omitted the implicit parameter this of all non-static methods as it does not play any role in the example. Furthermore, notice that we add the output heap as an argument to predicate get_field (whose syntax was defined in Figure 2.2); the reason for this will become apparent later, when we define the new internal representation of the heap and its operations. Lastly, notice that we redefine the notion of *guard* from the syntax of our CLP-translated programs to incorporate explicit reference equality and disequality:

$$G ::= Num\ ROp\ Num \mid \mathsf{ref\_eq}(H_{in}, Ref, Ref', H_{out}) \mid \mathsf{ref\_neq}(H_{in}, Ref, Ref', H_{out})$$

Guards now might contain comparisons between numeric data and between references. In particular, instructions ref_eq and ref_neq check whether *Ref* and *Ref'* are equal and different, respectively. They receive the heap as an explicit parameter since, as we will see later, their execution might modify the heap's contents. Since subclasses inherit the fields of the class they extend, new_object needs to access the set of classes partially ordered with respect to the *subclass* relation, and the fields declared by each class. Hence, we assume this information is available. Since virtual invocations do not add any complexity to the heap solver, we do not consider them here.

As in previous examples, all clauses contain input and output arguments and heaps. The heap is accessed using the heap operations set_field and get_field. Instruction 6 in the source code is translated into the three CLP instructions 7, 8 and 9. This is because the

CLP-translated code is obtained from the bytecode, where addition is performed using three operations: pushing the field value to the stack, increasing it by one and then putting the value again to the heap. Conditional statements in the source program are translated into guarded rules (e.g., cond). Methods (like m) and intermediate blocks (like cond) are uniformly represented by means of predicates and are not distinguishable in the CLP-translated program.

As mentioned before, CLP-translated programs can be executed by using the standard execution mechanism of CLP, given a suitable heap solver. The execution is performed on symbolic values, often represented as *constraint variables*, which are accumulated into path conditions (also called *path constraints*). The path constraints in feasible paths provide pre-conditions on the input data that guarantee the corresponding path will be executed at run-time. Whenever a path constraint is updated, it is checked for satisfiability by the solver. If the path constraint is unsatisfiable, the procedure backtracks. Otherwise, execution continues until a solution (representing a test case) is produced. Collecting all solutions returns the entire set of tests generated for the original program.

In previous work [GZAP10], and in the preceding sections of this thesis, the heap in the CLP-translated programs was represented explicitly as a list of locations, each being a pair made of a unique reference and a cell. This chapter presents a novel approach where the heap is treated as a black-box through its associated operations, which are handled more efficiently by means of a heap solver. As a result, our heap is always represented by a variable. Other constraint-based approaches [CBG09, DSV10a] also represent the heap as a variable, but they differ from us on the definition of the heap operations (see Section 6.5 for a more detailed comparison).

6.2 The Heap Solver

This section presents our heap solver. In particular, it provides the internal representation of the heap, presents the heap operations, proposes an advanced method for the back-propagation of constraints that allows pruning unfeasible branches earlier, and discusses a simple extension of the heap solver to handle arrays.

6.2.1 Internal Representation

The heap is internally represented by the heap solver as a tuple $\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle$, where:

- $\mathcal{S}$ is a recursive term that stores every read/write access performed on object fields that have not been explicitly created during symbolic execution, in the order in which such accesses occurred. $\mathcal{S}$ can be seen as a stack of field accesses (the reason for this will become apparent later). Formally, it is defined as $\mathcal{S} ::= getF(Ref, FSig, Var_{\{\mathcal{R}\}}, \mathcal{S}) \mid setF(Ref, FSig, Data, \mathcal{S}) \mid \varnothing$, where Ref, $FSig$ and $Data$ have the same meaning as in the get_field and set_field instructions introduced before, Var is an attributed variable and $\mathcal{R}$ is its attribute, that is, a set of rules representing possible aliasing configurations. Each rule in attribute $\mathcal{R}$ is a conjunction of constraints of the form $\bigwedge_{i=0}^{k-1} Ref \neq Ref_i \wedge Ref = Ref_k \rightarrow Var = Var_k$, corresponding to the aliasing configuration in which if Ref is *only* aliased with Ref_k, then $Var = Var_k$.
- $\mathcal{N}$ is a dictionary that maps fresh numeric references to new objects explicitly created by new_object during the symbolic execution, where an object is a list of fields.
- $\mathcal{RC}$ is a set of disequality constraints over references.

Note that references to objects explicitly created by new_object will appear in $\mathcal{N}$, while references to all other objects will appear in $\mathcal{S}$.

6.2.2 Heap Operations

The heap solver is accessed by means of its heap operations, get_field, set_field, new_object, ref_eq and ref_neq, which are invoked directly from the CLP-translated program, and by the solve operation which is only invoked at the end of a symbolic execution branch to get one or more concrete solutions. Heap operations, which update the input heap $H_{in} = \langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle$ to obtain $H_{out} = \langle \mathcal{S}', \mathcal{N}', \mathcal{RC}' \rangle$, denoted as $H_{in} \rightsquigarrow H_{out}$ (the conditions for the update will appear over the transition), are handled by the solver as follows:

set_field$(H_{in}, Ref, FSig, Data, H_{out})$. If Ref maps to an object O in $\mathcal{N}$, then this operation updates field $FSig$ in O. Otherwise, it adds a new $setF$ element to $\mathcal{S}$, indicating that field $Ref.FSig$ is set to value $Data$.

$$\frac{(Ref \mapsto O) \in \mathcal{N} \quad \mathcal{N}' \leftarrow \mathcal{N}[Ref \mapsto O[FSig \mapsto Data]]}{\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle \rightsquigarrow \langle \mathcal{S}, \mathcal{N}', \mathcal{RC} \rangle} \qquad \frac{(Ref \mapsto O) \notin \mathcal{N} \quad \mathcal{S}' \leftarrow setF(Ref, FSig, Data, \mathcal{S})}{\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle \rightsquigarrow \langle \mathcal{S}', \mathcal{N}, \mathcal{RC} \rangle}$$

get_field*(H_in, Ref, FSig, Var_{R}, H_out).* If Ref maps to an object O in $\mathcal{N}$, then this operation simply accesses O, gets the value of field $FSig$ in $Var_{\{\mathcal{R}\}}$, and sets $H_{out} = H_{in}$. Otherwise, a new $getF$ element is added to $\mathcal{S}$ and $Var_{\{\mathcal{R}\}}$ is a fresh variable whose $\mathcal{R}$ attribute and domain are calculated by function ψ.

$$\frac{(Ref \mapsto O) \in \mathcal{N} \quad Var_{\{\mathcal{R}\}} \leftarrow O[FSig]}{\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle \rightsquigarrow \langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle} \qquad \frac{(Ref \mapsto O) \notin \mathcal{N} \quad \langle \mathcal{R}, dom(Var) \rangle \leftarrow \psi(\mathcal{S}, Ref, FSig, Var, true) \quad \mathcal{S}' \leftarrow getF(Ref, FSig, Var_{\{\mathcal{R}\}}, \mathcal{S})}{\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle \rightsquigarrow \langle \mathcal{S}', \mathcal{N}, \mathcal{RC} \rangle}$$

where $\psi(\mathcal{S}, Ref, FSig, Var, \varphi) =$

$$\begin{cases} \langle \{\varphi \rightarrow Var = F\}, dom(F) \rangle \quad (F\ fresh) & \mathcal{S} = \varnothing \quad (1) \\ \langle \{\varphi \rightarrow Var = F\}, dom(F) \rangle & \mathcal{S} = [g|s]etF(Ref, FSig, F, \mathcal{S}_i) \quad (2) \\ \langle \{\varphi \wedge Ref = Ref_i \rightarrow Var = F\} \cup \mathcal{R}_i, dom(F) \cup D_i \rangle & \mathcal{S} = getF(Ref_i, FSig, F, \mathcal{S}_i) \quad (3) \\ \quad where\ \langle \mathcal{R}_i, D_i \rangle = \psi(\mathcal{S}_i, Ref, FSig, Var, \varphi) & \\ \langle \{\varphi \wedge Ref = Ref_i \rightarrow Var = F\} \cup \mathcal{R}_i, dom(F) \cup D_i \rangle & \mathcal{S} = setF(Ref_i, FSig, F, \mathcal{S}_i) \quad (4) \\ \quad where\ \langle \mathcal{R}_i, D_i \rangle = \psi(\mathcal{S}_i, Ref, FSig, Var, \varphi \wedge Ref \neq Ref_i) & \\ \psi(\mathcal{S}_i, Ref, FSig, Var, \varphi) & \mathcal{S} = [g|s]etF(Ref_i, FSig, F, \mathcal{S}_i) \quad (5) \end{cases}$$

As an optimization, if $\mathcal{R}$ contains only one single rule $r \equiv true \rightarrow Var = F$, meaning that no aliasing is possible, then $\mathcal{R}$ is emptied and $Var_{\{\mathcal{R}\}} = F$ is added to the store.

ref_eq*(H_in, Ref_1, Ref_2, H_out).* Propagates constraint $Ref_1 = Ref_2$.

$$\frac{t \equiv Ref_1 = Ref_2 \quad \mathcal{S}' \leftarrow propagate(\mathcal{S}, t)}{\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle \rightsquigarrow \langle \mathcal{S}', \mathcal{N}, \mathcal{RC} \rangle}$$

propagate returns $\mathcal{S}'$ by simplifying $\mathcal{S}$ w.r.t. a constraint t. Each rule r in attribute $\mathcal{R}$ of each $getF(Ref, FSig, Var_{\{\mathcal{R}\}}, \_)$ subterm in $\mathcal{S}$ is treated as follows:

(I) If $lhs(r)$ contains t, we remove t from $lhs(r)$. If the resulting $lhs(r)$ becomes $true$, we add the constraint $rhs(r)$ to the store.

(II) If $lhs(r)$ contains $\neg t$, then $lhs(r)$ can never hold and we remove r from $\mathcal{R}$.

Our CLP implementation of ref_eq simply unifies Ref_1 and Ref_2. This wakes up the constraints in the attributes involving these references so that the above simplification is performed. In addition, we recalculate $dom(Var)$ when rules are removed in (II).

ref_neq$(H_{in}, Ref_1, Ref_2, H_{out})$. Propagates constraint $Ref_1 \neq Ref_2$. The disequality is added to the $\mathcal{RC}$ store, since we may later try to add $Ref_1 = Ref_2$ and this must fail.

$$\frac{t \equiv Ref_1 \neq Ref_2 \quad \mathcal{S}' \leftarrow propagate(\mathcal{S}, t) \quad \mathcal{RC}' = \mathcal{RC} \cup \{t\}}{\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle \rightsquigarrow \langle \mathcal{S}', \mathcal{N}, \mathcal{RC}' \rangle}$$

new_object$(H_{in}, C, Ref, H_{out})$. Adds to $\mathcal{N}$ a fresh numeric reference mapped to the newly created object whose fields are initialized to default values (integers are initialized to 0 and references to `null`).

$$\frac{new(Ref) \quad createObject(C, O) \quad \mathcal{N}' = \mathcal{N} \cup \{Ref \mapsto O\}}{\langle \mathcal{S}, \mathcal{N}, \mathcal{RC} \rangle \rightsquigarrow \langle \mathcal{S}, \mathcal{N}', \mathcal{RC} \rangle}$$

Example 25 Consider the method `m` from Figure 6.1. Let us show part of the symbolic execution tree for `m([X,Y,Z],[ ],`H_{in}`,`H_{out}`)`, starting from the empty heap $H_{in} = \langle \varnothing, \emptyset, \emptyset \rangle$. The following shows, after each $\rightsquigarrow_{\mathtt{n}}$ arrow, the update performed to the heap by the execution of the instruction appearing in line `n`:

$$\begin{array}{rll}
H_{in} & \rightsquigarrow_4 & H_1 = \langle \mathcal{S}_1, \emptyset, \emptyset \rangle \text{ where } \mathcal{S}_1 = setF(X, \mathsf{f}, 1, \varnothing) \\
& \rightsquigarrow_5 & H_2 = \langle \mathcal{S}_2, \emptyset, \emptyset \rangle \text{ where } \mathcal{S}_2 = setF(Z, \mathsf{f}, -5, \mathcal{S}_1) \\
& \rightsquigarrow_6 & H_3 = \langle \mathcal{S}_3, \emptyset, \emptyset \rangle \text{ where } \mathcal{S}_3 = \mathcal{S}_2 \\
& \rightsquigarrow_7 & H_4 = \langle \mathcal{S}_4, \emptyset, \emptyset \rangle \text{ where } \mathcal{S}_4 = getF(X, \mathsf{f}, Xf_{R_4}, \mathcal{S}_3), \\
& & R_4 \equiv \{X = Z \rightarrow Xf = -5; X \neq Z \rightarrow Xf = 1\} \text{ and } dom(Xf) = \{-5, 1\} \\
& \rightsquigarrow_8 & dom(Yf) = \{-4, 2\} \\
& \rightsquigarrow_9 & H_5 = \langle \mathcal{S}_5, \emptyset, \emptyset \rangle \text{ where } \mathcal{S}_5 = setF(Y, \mathsf{f}, Yf, \mathcal{S}_4) \\
& \rightsquigarrow_{10} & H_6 = H_5 \\
& \rightsquigarrow_{13} & H_7 = H_6[Xf = -5, Yf = -4] \\
& \rightsquigarrow_{14} & H_8 = \langle \mathcal{S}_8, \emptyset, \emptyset \rangle \text{ where } \mathcal{S}_8 = getF(Y, \mathsf{f}, -4, \mathcal{S}_7) \ldots
\end{array}$$

Note that when executing the get_field in instruction 7, the value stored in the heap for the field is Xf_{R_4}, where Xf is a fresh variable with an associated attribute R_4 built according to the definition of get_field. Namely, we have traversed $\mathcal{S}_3$ down until reaching the $setF$ that has been set in step 4 and we have found two fields $\langle Z, -5 \rangle, \langle X, 1 \rangle$. Thus, we set up the domain of R_4 to $dom(Z) \cup dom(X)$ which results in $dom(Xf) = \{-5, 1\}$, and we add two rules which correspond to cases (4) and (2) in the definition. The first rule is obtained from $\langle Z, -5 \rangle$ and thus has the head $X = Z$, while the second rule is obtained from the negation of the head $X \neq Z$. Observe the role of $getF$ to record a local name for the

current value of the field (e.g., Xf for X.f) so that if we look up the same field twice in succession, we get the same name. Instruction 8 simply adds one to the gathered value and stores it in an intermediate variable Yf whose domain is obtained by adding one to $dom(Xf)$. The next interesting step occurs when executing the cond statement. The derivation shows only the branch that applies to the first definition of cond. The execution of the instruction 13 wakes up the definition R_4 and allows us to apply the first rule in R_4, namely as we know that $X = Z$, we can unify Xf to -5. This in turn propagates $Yf = -4$. These unifications are applied to H_6 denoted as $H_6[Xf = -5, Yf = -4]$. Finally, when we apply get_field on Y.f at instruction 14 and traverse the heap, we directly find a $setF$ for Y.f with the value $Yf = -4$. Thus, we do not traverse it further and call call_c with this fixed argument. Although not shown, the second rule for cond sets $X \neq Z$ and wakes the definition R_4 and sets $Xf = 1$, which in turn propagates $Yf = 2$. The instruction in line 18 retrieves $Yf = 2$ and then calls call_d with this fixed argument.

Assume now that we include the instruction x=new C(); as the first instruction of method m. It will be CLP-translated into new_object(H_{in}, C, X, H_1). According to the definition for new_object, the new object is stored in $\mathcal{N}$. Then, the get_field for x.f will be performed with X being a numeric reference and its value will be retrieved from $\mathcal{N}$. The next instructions will therefore not create (infeasible) aliasings of x with y and z. □

As we can see in the definition of get_field, we only add $Ref \neq Ref_i$ to the lhs if Ref_i arose from a $setF$ term. Let us explain this by means of an example. First, consider the fragment "int a=x.f; int b=y.f; int c=z.f". It is sound to have the following attributes associated to Zf in the third get_field instruction, $R \equiv \{Z = Y \rightarrow Zf = Yf; Z = X \rightarrow Zf = Xf\}$, i.e., it is not necessary to include $Z \neq Y$ in the second rule, nor have a rule with $Z \neq Y \wedge Z \neq X \rightarrow Zf = F$. The point is that as the simplified rules are not mutually exclusive, they both apply if $Z = Y = X$, which is correct. The advantage of having unitary guards is that if the body of the rule becomes unsatisfiable, we can negate the head of the rule and propagate this knowledge. This is illustrated in the next section. However, if we add a set_field instruction to the previous fragment "int a=x.f; w.f=a; int b=y.f; int c=z.f", then we must add the non-aliasing guard. We should then have the following attribute for Zf: $R \equiv \{Z = Y \rightarrow Zf = Yf; Z = W \rightarrow Zf = Wf; Z \neq W \wedge Z = X \rightarrow Zf = Xf; Z \neq W \rightarrow Zf = F\}$. This prevents us from applying the third unification of the values of Wf and Xf (which would be incorrect) when Z and W are aliased.

Regarding computational complexity, the heap solver asymptotic complexity is poly-

nomial in the program size. However, this asymptotic complexity is irrelevant since it is dwarfed by the exponential path complexity of symbolic execution. In practice, our experiments show that the practical complexity is acceptable, and more importantly has the effect we want of exponentially reducing the path complexity.

6.2.3 Backwards Propagation of Constraints

As described in the previous section, our heap solver uses information about equality and disequality of references to determine equality among the heap cells. This is done by propagating such information forwards in the rules of attributes. We can extend the solver straightforwardly to also propagate information backwards. Consider a rule r that defines field F and is part of some attribute $\mathcal{R}$. If $rhs(r)$ is $F = F'$ and the current store implies $F \neq F'$, then r cannot be applied. We can thus recalculate the domain of F by excluding $dom(F')$ from the calculation. Further, if all literals in $lhs(r)$ except one – say l – are known to be true in the current store, we can also assert $\neg l$.

Example 26 Consider the method m but with the condition of the if (in instruction 8) changed to "if (x.f == 1)". This would be translated as get_field(H_6,X,f,$Xf2$,H_7) followed by $Xf2 = 1$. The get_field operation creates propagation rules $X = Y \rightarrow Xf2 = Yf$ and $X \neq Y \rightarrow Xf2 = Xf$. When setting $Xf2 = 1$ the integer solver determines that $Xf2 = Yf$ cannot hold. This means that the solver can propagate $X \neq Y$, which then causes $Xf2 = Xf$. Since we know that $Xf = 1$, the rule $X = Z \rightarrow Xf = -5$ also back propagates to add the information $X \neq Z$. As a result, we recalculate $dom(Yf) = \{2\}$ and the call call_c is performed with that fixed value for y.f. □

6.2.4 Extension to Arrays

It is straightforward to extend our language to handle arrays, since we can use the same method as for handling object fields. In this case, the array indices play the role of the references that point to the heap-allocated data.

Example 27 Consider the following fragment of code: " x[k] = -1; x[i] = 2; x[j] = 5; if (x[k] > 0)... ". The heap access x[k] is CLP-translated into the operation getArrayElem($X, k, Xk_{\{\mathcal{R}\}}, H$) where $Xk_{\{\mathcal{R}\}}$ is an attributed variable built in a similar way to how $F_{\mathcal{R}}$ is built by the get_field instruction. In particular, $\mathcal{R}$ will have the propa-

gation rules $\{K = J \rightarrow Xk = 5; K \neq J \wedge K = I \rightarrow Xk = 2; K \neq J \wedge K \neq I \rightarrow Xk = -1\}$ and $dom(Xk) = \{5, 2, -1\}$. □

Note that this is basically an encoding of the SMT theory of arrays into the same chain of constraints. However, our encoding has an advantage over the SMT approach, since it allows us to later refine the domain of x[k]: if in the above example we later find that $k = i$, we can then refine $dom(Xk)$ to $\{5, 2\}$ (regardless of whether $i = j$ or not).

6.3 Testing with Heap Assumptions

As we have seen, the TCG process described so far assumes that all possible kinds of aliasing among heap-allocated (reference) input data of the same type can occur. However, it may be the case that while some of these aliasings might indeed occur, others might not (consider, for instance, aliased data structures that cannot be constructed using the public methods in the Java class). In order to avoid generating such inputs, we have extended our framework to handle *heap assumptions*, that is, assertions describing reachability, aliasing, separation and sharing conditions in the heap. We currently support three types of heap assumptions:

- *non-aliasing(a,b):* specifies that memory locations a and b are not the same.
- *non-sharing(a,b):* specifies disjointness, i.e., that references a and b do not share any common region in the heap.
- *acyclic(a):* specifies that a is an acyclic data structure.

Non-aliasing is implicit in the framework and implemented by simply using the constraint $a \neq b$. Non-sharing and acyclicity are properties of the initial heap. In order to implement them correctly we define a CHR predicate initial($\mathcal{S}$,*Ref*,F), which succeeds if field F of *Ref* refers to the original heap state $\varnothing$. Its implementation, shown in Figure 6.3, simply unrolls the $\mathcal{S}$ term ensuring that there is no set_field applied to *Ref* for field F. Both non-sharing and acyclicity need to track references to the original heap. We assume each call to get_field($\langle \mathcal{S}, N, RC \rangle$,*Ref*,$F$,*Cell*,$H'$) creates a call to track_get($\mathcal{S}$, *Ref*, F, *Cell*) which is implemented as a CHR predicate. Non-sharing of a and b is implemented via the nonsharing(a, b) CHR predicate of Figure 6.3, which simply checks that any field reachable in the initial heap from a is not the same as a field reachable from b. Acyclicity of

```
initial(∅,Ref,F)                    <=> true.
initial(getF(_,_,_,S), Ref, F)      <=> initial(S,Ref,F).
initial(setF(Ref,F,_,S), Ref, F)    <=> fail.
initial(setF(Ref1,F1,_,S), Ref, F) <=> Ref != Ref1 | initial(S,Ref,F).
initial(setF(Ref1,F1,_,S), Ref, F) <=> F != F1 | initial(S,Ref,F).
track_get(S,Ref,F,Cell)             <=> not initial(S,Ref,F) | true.

nonsharing(X,X)                          => fail.
nonsharing(X,Y), track_get(S,X,F,FX)     => initial(S,X,F) | nonsharing(FX,Y).
nonsharing(X,Y), track_get(S,Y,F,FY)     => initial(S,Y,F) | nonsharing(X,FY).

acyclic(Ref)                          <=> path(Ref, []).
path(Ref, Path), track_get(S,Ref,F,Cell) => initial(S, Ref, F), member(Ref,Path) | fail.
path(Ref, Path), track_get(S,Ref,F,Cell) => initial(S, Ref, F), notmember(Ref,Path)
                                          | path(Cell, [Ref|Path]).
```

Figure 6.3: Implementation of Heap Assumptions in CHR

a is implemented via the acyclic(a) CHR predicate, which checks that each path of fields reachable from a cannot reach back to itself.

Example 28 Consider the pre-condition nonsharing(X, Y) for the method starting with fragment

```
if (y.f == z && x.f = z && x == z.f) ...
```

where field f has reference type. The initial get_field on y creates track_get($\varnothing$, Y, f, Yf). This, together with the third rule for nonsharing, adds the constraint nonsharing(X, Yf). The first conjunct forces $Z = Yf$. The set_field of the second conjunct does not create a track_get and does not cause any CHR execution. The third statement creates track_get(S, Z ,f, Zf), which creates nonsharing(X,Zf) using nonsharing(X,Yf). Then $X = Zf$ and the first rule of nonsharing causes failure. Hence, the then branch will not be visited. □

6.4 Implementation and Experimental Results

We have implemented a prototype of the heap solver and integrated it within PET. The prototype makes use of its SWI Prolog's support for attributed variables and of its clpfd library [Wie10]. All experiments can be reproduced online at the PET web interface by

selecting the option "CLP Heap Solver". The implementation provides support for full sequential Java Bytecode. This includes exception handling, which is managed by the transformation from Java Bytecode to CLP and does not require extending the heap solver. Regarding inheritance/polymorphism, realistic Java code makes extensive use of type checks and castings. This is handled during symbolic execution by constraints of the form "subtype of", "not subtype of" and "instance of", over reference types. Our prototype includes a solver to handle such type constraints. In particular, get_field makes use of these type constraints and the solver checks whether the types of two references are compatible before considering their aliasing.

Our experiments aim at illustrating the scalability of our approach on realistic OO programs. They have been performed using as benchmarks a selection of classes from the net.datastructures package. In particular, we have used as "methods-under-test" the most relevant public methods of the classes *DoublyLinkedList*, *NodeSequence*, *SortedList-PriorityQueue*, *BinarySearchTree* and *HeapPriorityQueue*, abbreviated respectively as *DLL*, *Seq*, *PQ*, *BST* and *HPQ*. Table 6.1 compares the performance of the new approach (columns labeled with 3) against that of the standard approach based on lazy initialization, with and without considering reference aliasing (columns labeled with 2 and 1, respectively). For each run we provide the number of clauses in the CLP-translated program (**C**), the number of generated test cases (**N**), the time in milliseconds of the TCG process (**T**) and the number of (thousands of) derivation steps (**U**). For all runs we use the *loop-1* coverage criterion, which limits the number of iterations on loops to at most one. As customary, the test cases are obtained by means of a solve operation which is invoked at the end of each symbolic execution branch and gives a concrete solution. All times are obtained as the arithmetic mean of five runs on an Intel Core i5 CPU at 1.8GHz with 4GB of RAM, running OSX 10.8.2. We use '-' in column **T2** to indicate that the process has not finished within a timeout of 30 seconds. In those cases, **N2** and **U2** correspond to the accumulated numbers when the process is aborted. The figures on columns **N2**, **T2** and **U2** clearly show that the approach based on lazy initialization quickly blows up. By comparing those numbers with **N1**, **T1** and **U1**, where aliasing of references is not being taken into account, we confirm that the explosion on the number of branches is due to the aliasing. Looking at **N3**, **T3** and **U3**, we can observe that our heap solver does not suffer from this explosion problem. Indeed, in terms of speed, in general, our approach (**T3**) can be up to two orders of magnitude faster than the standard approach with aliasing support (**T2**), and similar to (or in some cases even faster than) the standard approach

Table 6.1: Experimental evaluation

Method	**C**	**N1**	**T1**	**U1**	**N2**	**T2**	**U2**	**N3**	**T3**	**U3**
DLL.add	139	33	36	1.8k	200	177	7.5k	33	42	2.2k
Seq.removeAt	213	36	42	1.3k	89	116	3.9k	37	50	1.5k
Seq.replaceAt	187	36	41	1.1k	39	47	1.3k	37	27	1.1k
PQ.insert	399	101	160	3.6k	3602	10672	196.7k	101	182	3.6k
PQ.remove	193	12	11	0.4k	86	68	1.2k	12	15	0.4k
BST.addAll	293	379	2019	115.3k	919	-	1832.6k	379	2535	115.3k
BST.find	269	62	108	4.8k	184	285	10.6k	62	98	4.8k
BST.findAll	428	330	2385	120.7k	1165	-	1655.7k	330	2538	120.7k
BST.insert	426	970	2527	84.8k	8365	-	758.5k	970	3924	84.8k
BST.remove	516	203	615	28.7k	2745	5587	200.6k	203	725	28.8k
HPQ.insert	349	61	283	5.4k	135	638	8.3k	95	163	5.2k
HPQ.remove	469	80	1814	69.7k	2021	-	824.8k	146	2378	66.5k

without aliasing support (**T1**). It should be noted that we achieve such speedup even if the back-propagation of constraints is currently only partially implemented. Importantly, observe that the more complex the structure of the program is (in terms of **C** or **N**) the more gain we get with our heap solver.

As regards the number of test cases, the figures in **N3** are in general only slightly greater than those in **N1**. This is because **N3** includes not only the paths of the program that can be reached when there is no aliasing (as **N1**), but also those that can only be reached when some objects are aliased. All additional branches obtained in **N2** are spurious, i.e., they do not lead to further coverage. This becomes apparent in the tree of Figure 6.2a which has five branches (and hence five test cases will be obtained from it), while we only need two branches to have full coverage.

6.5 Conclusions and Related Work

Ignoring aliasing in TCG leads to loss of coverage and, as a consequence, errors due to undesired aliasing of data go undetected. We have proposed a novel approach to TCG for heap-manipulating programs which (a) avoids the explosion that occurs when handling

aliasing of references and (b) allows specifying initial heap assumptions on acyclicity and disjointness. While we have presented our approach within the CLP-based framework to TCG, where the imperative program in translated to CLP, the heap solver at the core of our approach can indeed be used by other TCG tools by means of its interface heap operations. Therefore, our work is widely applicable.

Constraint-based testing approaches to TCG [CBG09, DSV10a] are closely related to our work. These approaches first extract a constraint system from the source code of the program under test, and then obtain concrete test cases by solving this constraint system. The solvers handle the two sources of non-determinism: the one associated with the control flow (conditional, while statements) and the one associated with the selection of concrete input data (the heap constraints). A main difference with our CLP-based approach is that we represent the control flow by means of a CLP program and handle the constraints associated to heap allocated data by means of the heap solver. This separation of concerns has the advantage that control decisions can be made as soon as possible, while the heap constraints can be lazily executed. Our definition of the symbolic heap operations is different from those of [CBG09, DSV10a]. In particular, [DSV10a] defines a heap solver using CHRs which is equivalent to the SMT theory of arrays. Since it does not track possible domains of cell lookups as we do, it propagates less information. Note that propagating less information on the heap cells may lead to many infeasible branches which are not pruned until the delayed operations are executed. This could degrade the efficiency. Also their framework is defined only for list-manipulating programs. While they discuss how to extend the approach to new data types, each new data type requires adding new reasoning capabilities, as opposed to our generic approach to memory. The approach of [CBG09] defines their own constraint operators, but suffers from the severe restriction of being unable to handle inter-procedural calls, which our approach handles transparently. They do not define heap assumption handling. [DSV10a] mention that their framework could incorporate heap assumptions. We have provided an implementation for the most common heap assumptions, which could be also used in their framework.

While heap assumptions are extensively used in software verification [PRW08], their use in software testing is less common. Notable exceptions are [VPK04, OL99]. In [VPK04], user-defined assumptions are given as Java methods that are executed during TCG. This can be inefficient since the pre-conditions are not given in the same language of the constraint solver (as ours are). [OL99] use a declarative language to specify preconditions. Our approach is the only one capable of specifying preconditions at the level of

the data on which they operate, rather than having heap operations that integrate such preconditions as in [GZAP10]. Finally, instead of using constraint operations, we could also use SMT solvers [PVL11]. It remains as future work to compare how an SMT-based approach (e.g., [TdH08]) compares to our pure CLP-based scheme. Note that while the $\mathcal{R}$ attributes mimic the solver for a theory of arrays, our solver keeps track of the disjunction of possible values for base types, without forcing equalities. Hence, it can propagate disequality more strongly than the SMT approach. The backwards propagation explained in Example 26 would not occur using an SMT approach (at least until $X \neq Y$ was decided by the solver after further decisions).

Chapter 7

Conclusions and Future Work

This chapter concludes the thesis by summarizing its contributions and discussing some potential future lines of research.

7.1 Conclusions

Although symbolic execution for software testing was introduced almost forty years ago and has been intensively studied in the last 10 years, well-known scalability limitations remain to hamper its applicability in software testing in practice. To provide an absolute solution to these limitations is unapproachable. However, much effort has been devoted to improve scalability and several techniques co-exist which, in a sense, complement each other. Probably, the most widely used is abstraction, a well-known technique to reduce large data domains of a program to smaller domains (see [LBBO01]). Another scaling technique which is closely related to abstraction is path merging [AEO+08, KP05], which consists in defining points where the merging of symbolic execution should occur.

In this thesis, we have developed several novel techniques aimed at alleviating some scalability issues of TCG by symbolic execution. The fundamental conclusions which can be drawn from this research can be summarized as follows.

- We have tackled the inter-procedural path explosion problem by proposing two compositional approaches to symbolic execution and TCG. These compositional approaches outperform their standard non-compositional counterparts in terms of computation time and space, hence scaling up to handle larger and more complex pieces of software.

- We have proposed a methodology, dubbed resource-driven TCG, to stress the generation of test cases towards non-functional aspects of the program under test, e.g., its resource usage. We devise the use of resource policies as selection criteria to decide which symbolic execution paths to explore and which to avoid.

- We have developed a generic framework for guided TCG, which is inspired on and generalizes the ideas of resource-driven TCG. This framework allows guiding TCG towards the most interesting parts of the program under test with respect to a given selection criterion. In doing so, not only the size of the symbolic execution tree is considerably reduced, but also the number of generates test cases is lower, and their relevance and quality higher.

- We have proposed a heap solver, whose purpose is to allow for a more efficient symbolic execution and TCG of heap-manipulating programs. We show how the proposed heap solver remarkably surpass the state-of-the-art technique lazy initialization, both in terms of effectiveness and efficiency.

All techniques have been implemented in the PET system (Compositional TCG has also been implemented in the SPF tool) and their effectiveness and efficiency have been validated through experimental evaluation using relevant challenging case studies and benchmarks.

7.2 Future Work

This section sketches some alternatives for future work in the lines of the research performed in this dissertation. They range from low level challenging problems, e.g., concolic execution and TCG of concurrent programs, to high level ones, such as the oracle problem or using symbolic execution and TCG for teaching purposes.

Concolic Execution. It is well known that static techniques, such as symbolic execution, are effective to achieve high code coverage, but often at a high computational cost. On the other hand, dynamic techniques allow quick and efficient generation of test cases that exercise complex parts of the program under test, however hardly achieving high code coverage. Concolic execution consists on interleaving concrete and symbolic execution. First, concrete execution is performed to collect the path condition of a deep state in the

execution tree, then, symbolic execution can be performed to explore the "neighborhood" of such concrete state.

Concurrent programs. Reasoning about the behaviour of concurrent programs is more complex than reasoning on that of sequential ones, due, among other factors, to features like preemption, scheduling, interleaving and shared data. Covering all possible execution paths, which is basically the goal of TCG, is even harder. However, static analysis techniques can be useful to infer some static information about the program which can then be used to guide the TCG process.

Heap Solver. Most TCG tools nowadays are parametric with respect to the decision procedure that is used to assess the satisfiability of path conditions. They often use several solvers, each with strength on different kinds of constraints (e.g., integer arithmetic constraints, string manipulation). We plan to provide a standalone implementation of the heap solver developed in Chapter 6. The fact that lazy initialization is the technique that most TCG systems currently use and the outstanding gains that we have observed from the use of our heap solver in our framework, lead us to believe that it could also have a remarkable impact on the efficiency of other TCG tools.

The Oracle problem. We plan to study the actual effectiveness of the generated test cases in real-world software development scenarios. The empirical evaluation of the impact of TCG in the productivity of programmers and the quality of software would also be an exciting field to explore in the near future. Moreover, there is a wide variety of techniques related to software testing that aim to help the programmer develop more reliable and trust-worthy programs, such as test case visualization and integration of test generation tools in industrial software development environments.

Teaching Support. The PET system, our TCG tool, is still in a prototypical phase of development. While we acknowledge that the limitations of the involved techniques prevent the use of PET in an industrial setting, we do believe that the academic world can be approached in the near future. In concrete, we plan to use PET as a supporting tool in programming, data structures and testing courses. A relevant example of this use of symbolic execution and TCG tool for teaching is PEX [TdH08], with its interactive online interface at http://research.microsoft.com/en-us/projects/pex/.

Bibliography

[AAG+07] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.

[AAG+08] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Removing Useless Variables in Cost Analysis of Java Bytecode. In *Proc. of SAC'08*, pages 368–375. ACM, 2008.

[AAG+12] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages)*, 413(1):142–159, 2012.

[ABC+13] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. 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, 2013.

[ACFM+11] Elvira Albert, Israel Cabañas, Antonio Flores-Montoya, Miguel Gómez-Zamalloa, and Sergio Gutiérrez. jPET: an Automatic Test-Case Generator for Java. In *18th Working Conference on Reverse Engineering (WCRE 2011)*, pages 441–442. IEEE Computer Society, October 2011.

[AEO+08] Tamarah Arons, Elad Elster, Shlomit Ozer, Jonathan Shalev, and Eli Singerman. Efficient Symbolic Simulation of Low Level Software. In *DATE'08*, pages 825–830. IEEE, 2008.

[AGGZ07] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *Proc. of ISMM'07*. ACM Press, 2007.

[AGT08] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-Driven Compositional Symbolic Execution. In *TACAS'08*, volume 4963 of *LNCS*, pages 367–381. Springer, 2008.

[AGZP09] Elvira Albert, Miguel Gómez-Zamalloa, and Germán Puebla. Test Data Generation of Bytecode by CLP Partial Evaluation. In *18th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'08)*, number 5438 in Lecture Notes in Computer Science, pages 4–23. Springer-Verlag, March 2009.

[AGZP10] Elvira Albert, Miguel Gómez-Zamalloa, and Germán Puebla. PET: A Partial Evaluation-based Test Case Generation Tool for Java Bytecode. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'10)*, pages 25–28. ACM Press, 2010.

[ANV08] J. Antunes, N. F. Neves, and P. Veríssimo. Detection and Prediction of Resource-Exhaustion Vulnerabilities. In *ISSRE'08*, pages 87–96. IEEE Computer Society, 2008.

[AO08] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[APV06] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. Symbolic Execution with Abstract Subsumption Checking. In *Model Checking Software*, volume 3925 of *LNCS*, pages 163–181. Springer, 2006.

[Bal03] T. Ball. Abstraction-guided Test Generation: A Case Study. Technical Report MSR-TR-2003-86, Microsoft Research, 2003.

[BHJ10] R. Bubel, R. Hähnle, and R. Ji. Interleaving Symbolic Execution and Partial Evaluation. In *FMCO'09*, pages 125–146. Springer-Verlag, 2010.

[CBG09] F. Charreteur, B. Botella, , and A. Gotlieb. Modelling Dynamic Memory Management in Constraint-Based Testing. *Journal of Systems and Software*, 82(11):1755–1766, 2009.

[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 Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[CG10] F. Charreteur and A. Gotlieb. Constraint-Based Test Input Generation for Java Bytecode. In *ISSRE 2010*, pages 131–140. IEEE Computer Society, 2010.

[CGK+11] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *ICSE'11*, pages 1066–1071. ACM, 2011.

[CGLH05] S.-J. Craig, J. P. Gallagher, M. Leuschel, and K. S. Henriksen. Fully Automatic Binding-Time Analysis for Prolog. In *LOPSTR'04*, LNCS 3573, pages 53–68. Springer, 2005.

[CGP+08] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.*, 12(2), 2008.

[CH00] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell programs. In *ICFP'00*, pages 268–279. ACM, 2000.

[Cla76] L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[CPPGM91] A. Coen-Porisini, F. De Paoli, C. Ghezzi, and D. Mandrioli. Software Specialization Via Symbolic Execution. *IEEE Trans. on Software Engineering*, 17(9):884–899, 1991.

[CS13] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, February 2013.

[Deb89] S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Trans. Program. Lang. Syst.*, 11(3):418–450, jul 1989.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991. ISSN: 0098-5589.

[DSV10a] F. Degrave, T. Schrijvers, and W. Vanhoof. Towards a Framework for Constraint-Based Test Case Generation. In *LOPSTR'09*, LNCS 6037, pages 128–142. Springer, 2010.

[DSV10b] François Degrave, Tom Schrijvers, and Wim Vanhoof. Towards a Framework for Constraint-Based Test Case Generation. In *LOPSTR'09*, volume 6037 of *LNCS*, pages 128–142. Springer, 2010.

[EH07] Christian Engel and Reiner Hähnle. Generating Unit Tests from Formal Proofs. In *TAP'07*, volume 4454 of *LNCS*, pages 169–188. Springer, 2007.

[FK96] R. Ferguson and B. Korel. The Chaining Approach for Software Test Data Generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[FK07] S. Fischer and H. Kuchen. Systematic Generation of Glass-Box Test Cases for Functional Logic Programs. In *PPDP'07*, pages 63–74. ACM, 2007.

[GBR00] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. A CLP Framework for Computing Structural Test Data. In *Computational Logic'00*, volume 1861 of *LNAI*, pages 399–413. Springer, 2000.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *PLDI'05*, pages 213–223. ACM, 2005.

[GMS00] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating Test Data for Branch Coverage. In *ASE'00*, pages 219–228. IEEE Computer Society, 2000.

[God07] Patrice Godefroid. Compositional Dynamic Test Generation. In *POPL'07*, pages 47–54. ACM, 2007.

[GTZ10] M.T. Goodrich, R. Tamassia, and R. Zamore. The net.datastructures Package, version 5.0. Available at http://net3.datastructures.net, 2010.

[GZAP09] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology*, 51(10):1409–1427, October 2009.

[GZAP10] Miguel Gómez-Zamalloa, Elvira Albert, and Germán Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming, 26th Int'l. Conference on Logic Programming (ICLP'10) Special Issue*, 10(4-6):659–674, July 2010.

[HJK09] A. Holzer, V. Januzaj, and S. Kugele. Towards Resource Consumption-Aware Programming. In *ICSEA'09*, pages 490–493. IEEE Computer Society, 2009.

[JGS93] N. D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[Kin76] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.

[KP05] Alfred Kölbl and Carl Pixley. Constructing Efficient Formal Models from High-Level Descriptions Using Symbolic Simulation. *International Journal of Parallel Programming*, 33(6):645–666, 2005.

[KPV03] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS'03*, volume 2619 of *LNCS*, pages 553–568. Springer, 2003.

[LBBO01] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 98–112, Genova, Italy, April 2001. Springer-Verlag.

[LS96] M. Leuschel and M.H. Sørensen. Redundant argument filtering of logic programs. In *Proc. of LOPSTR'96*. Springer-Verlag, 1996.

[LV05] M. Leuschel and G. Vidal. Forward Slicing by Conjunctive Partial Deduction and Argument Filtering. In *Proc. of ESOP'05*. Springer-Verlag, 2005.

[Meu01] C. Meudec. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.

[MLK04] Roger A. Müller, Christoph Lembeck, and Herbert Kuchen. A Symbolic Java Virtual Machine for Test Case Generation. In *IASTED Conf. on Software Engineering'04*, pages 365–371. IASTED/ACTA Press, 2004.

[MS07] Rupak Majumdar and Koushik Sen. Hybrid Concolic Testing. In *ICSE'07*, pages 416–426. IEEE Computer Society, 2007.

[OL99] A. Jefferson Offutt and Shaoying Liu. Generating Test Data from SOFL Specifications. *Journal of Systems and Software*, 49(1):49–62, 1999.

[PMB+08] C. S. Păsăreanu, P. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In *ISSTA'08*, pages 15–26. ACM, 2008.

[PR10] Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *ASE'10*, pages 179–180. ACM, 2010.

[PRW08] Andreas Podelski, Andrey Rybalchenko, and Thomas Wies. Heap Assumptions on Demand. In *CAV'08*, volume 5123 of *LNCS*, pages 314–327. Springer, 2008.

[PV09] Corina S. Păsăreanu and Willem Visser. A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, 2009.

[PVL11] J. Peleska, E. Vorobev, and F. Lapschies. Automated Test Case Generation with SMT-Solving and Abstract Interpretation. In *NFM'11*, LNCS 6617, pages 298–312. Springer, 2011.

[QW04] S. Qadeer and D. Wu. Kiss: Keep It Simple and Sequential. *SIGPLAN Not.*, 39(6):14–24, 2004.

[RDCP11] Diana Ramírez-Deantes, Jesús Correas, and Germán Puebla. Modular Termination Analysis of Java Bytecode and its Application to phoneME Core Libraries. In *Formal Aspects of Computer Software (FACS 2010)*, volume 6921 of *Lecture Notes in Computer Science*, pages 218–236. Springer, 2011.

[RMV09] N. Rungta, E.G. Mercer, and W. Visser. Efficient Testing of Concurrent Programs with Abstraction-Guided Symbolic Execution. In *SPIN'09*, LNCS 5578. Springer, 2009.

[SWP+09] T. Sun, Z. Wang, G. Pu, X. Yu, Z. Qiu, and B. Gu. Towards Scalable Compositional Test Generation. In *QSIC'09*, pages 353–358. IEEE Computer Society, 2009.

[TdH08] Nikolai Tillmann and Jonathan de Halleux. Pex: White Box Test Generation for .NET. In *TAP'08*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.

[Tri12] Markus Triska. The Finite Domain Constraint Solver of SWI-Prolog. In *FLOPS*, volume 7294 of *LNCS*, pages 307–316, 2012.

[VM05] Willem Visser and Peter C. Mehlitz. Model Checking Programs with Java PathFinder. In *SPIN 2005*, volume 3639 of *LNCS*, page 27. Springer, 2005.

[VPK04] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *ISSTA'04*, pages 97–107. ACM, 2004.

[Wie10] J. Wielemaker. *The SWI-Prolog User's Manual 5.9.9*, 2010. Available from `http://www.swi-prolog.org`.

[WSTL12] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[WTH+12] J. Wu, Y. Tang, G. Hu, H. Cui, and J. Yang. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In *PLDI'12*, pages 205–216. ACM, 2012.

[ZC02] J. Zhang and S.C. Cheung. Automated Test Case Generation for the Stress Testing of Multimedia Systems. *Softw., Pract. Exper.*, 32(15):1411–1435, 2002.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.