DISSERTATION

Query-Driven Program Testing

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften

unter der Leitung von

Univ. Prof. Dipl.-Ing. Dr. techn. Helmut Veith
E184/4
Institut für Informationssysteme

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Michael Tautschnig
0928914
Girardigasse 4/25
1060 Wien
Österreich

Wien, im Februar 2011

Michael Tautschnig

Kurzfassung

Testen von Software bezeichnet die Ausführung von Programmen mit dem Ziel, Fehler zu finden. Abdeckungskriterien, wie etwa die Abdeckung von Entscheidungen oder multiple condition/decision coverage (MC/DC), bestimmen dabei die Eignung einer Menge von Testfällen bei der Suche nach Fehlern. Testen, sowie die Messung der durch eine Testreihe induzierte Abdeckung des Programms, spielen eine entscheidende Rolle in aktuellen Softwareenwicklungsprozessen: Richtlinien für die Zertifizierung, wie etwa DO-178B, fordern als Nachweis der Systemsicherheit Testreihen, die bestimmte Abdeckungskriterien erfüllen. Trotz dieser herausragenden Bedeutung für sicherheitskritische Systeme bleibt Testen nach wie vor eine informale und manuelle Tätigkeit. Das führt zu erhöhter Fehleranfälligkeit der Systeme und hohen Kosten nebst schlechter Prognostizierbarkeit, und verhindert eine höhere Automatisierung. Große Hürden auf dem Weg zu einer umfangreicheren Automatisierung sind insbesondere das Fehlen formaler Strukturen zur Beschreibung von Abdeckungskriterien und, als Folge davon, die fehlende Flexibilität von bestehenden Testfallgeneratoren.

In dieser Dissertation wird ein neues Verfahren zur vollautomatischen Testfallgenerierung beschrieben, das Testfälle zu formalen Beschreibungen von Abdeckungskriterien erzeugt. Die Spezifikation der Testfälle erfolgt dabei etwa durch Testingenieure. Der Kern dieses Verfahrens ist ein formales Gerüst, das die Beschreibung von Abdeckungskriterien ermöglicht. Darauf aufbauend wird die deklarative Testfallspezifikationssprache FQL (FShell Query Language) beschrieben. Dazu passend wurde ein Backend entwickelt, das Testfälle zu FQL Anfragen erzeugt. Diese Architektur wurde in Analogie zu Datenbanken gewählt und daher wird der Ansatz als *query-driven program testing* bezeichnet. Die Umsetzung erfolgte für C Programme und führte zum Werkzeug FShell, das auf Teilen des C Bounded Model Checkers (CBMC) aufbaut.

Die Spezifikationssprache für Testreihen, FQL, wurde als einfach anwendbare und klar strukturierte Sprache mit formaler Semantik konzipiert. Mögliche Anwendungen sind sowohl die Testfallgenerierung als auch die Messung der Abdeckung – insbesondere auch in agilen Entwicklungsprozessen und manueller Programmexploration, wo ausgewählte Programmteile untersucht werden müssen. Neben solchen speziellen Szenarien werden aber auch Standard-Abdeckungskriterien wie die Abdeckung von basic blocks, multiple

condition coverage oder predicate complete coverage unterstützt. Auf Grund ihrer Ausdrucksstärke ist FQL auch dazu geeignet, die Kluft zwischen Model Checking und Testen zu verringern.

Das Backend basiert auf bounded model checking. Mit Hilfe von Komponenten von CBMC wird Unterstützung für volles ANSI C erreicht. Die Verwendung von Model Checkern für die Testfallgenerierung wurde schon mehrfach diskutiert, aber dennoch unterscheiden sich die die Anforderungen für Model Checking und Testen deutlich. Für eine effiziente Testfallgenerierung zu einer FQL Anfrage wurde daher die von CBMC erzeugte SAT Formel durch Übersetzungen in Aussagenlogik der aus FQL Anfragen erzeugten Automaten erweitert. Mittels *iterative constraint strengthening*, des wichtigsten algorithmischen Beitrags dieser Dissertation, kann eine FQL Anfrage in nur einem Aufruf des Model Checkers und mit einer neuartigen Form von inkrementellem SAT solving gelöst werden: Für jede vom SAT solver bestimmte Lösung erfolgt ein Vergleich mit dem Abdeckungskriterium um dann die Klauseldatenbank des SAT solvers so zu erweitern, dass redundante Lösungen ausgeschlossen werden.

Zur experimentellen Evaluierung wird für eine Menge von informellen Anforderungen gezeigt, wie diese in FQL übersetzt werden können. Außerdem werden mit diesen Spezifikationen Experimente durchgeführt und es wird die Skalierbarkeit von FShell in Experimenten mit Gerätetreibern, Steuerungssoftware aus dem Automobilbereich und open source Software gezeigt.

Abstract

Software testing is the process of executing a program to discover errors. Coverage criteria, such as decision coverage or multiple condition/decision coverage (MC/DC), define the adequacy of a test suite for finding errors. Tests, and coverage criteria in particular, play a crucial role in today's software development processes: certification guidelines, such as DO-178B, require test suites satisfying certain coverage criteria as evidence of system safety. Despite the importance for safety of mission-critical systems, software testing remains a largely informal and manual task. This makes testing an error-prone, costly and unpredictable process, and hinders higher degrees of automation. Major obstacles are the lack of formal frameworks for code coverage criteria and, as a consequence thereof, missing versatility of existing automated test case generators.

In this dissertation we describe a new method for fully automatic test case generation following formal specifications given by test engineers. We build upon a well-defined mathematical core that captures the semantics of coverage criteria. On top of this framework we define the declarative test specification language FQL, the FShell query language. These formal specifications are supplemented with an engine that generates test cases in response to FQL queries. We chose this overall design of a mathematical core, a query language and an efficient back end in analogy to databases and hence refer to our method as *query-driven program testing*. The full workflow is implemented for ANSI C programs in a tool called FShell, which uses components of the C Bounded Model Checker (CBMC).

FQL is designed to be simple and concise in daily use, features a precise semantics, leverages suitable engines to compute matching test suites, and assess the coverage achieved by a test suite. Equipped with sufficient expressive power, FQL helps to close the gap between testing and model checking; it also supports agile software development and manual program exploration by realizing coverage criteria which are complex but narrowly aimed at specific program properties. Our query language subsumes standard coverage criteria ranging from simple basic block coverage all the way to predicate complete coverage and multiple condition coverage, but also facilitates on-the-fly requests for test suites specific to the code structure, to external requirements, or to ad hoc needs arising in program understanding/exploration.

To perform automated test case generation we build upon bounded model

checking. We re-use components of CBMC to gain support for full ANSI C syntax and semantics. Although the principal analogy between counterexample generation and white-box testing has been repeatedly addressed, the usage patterns and performance requirements for software testing are quite different from formal verification. For efficient test case generation with respect to an FQL query we augment the SAT formula generated by bounded model checking with an additional propositional encoding of query-derived automata. Our main algorithmic contribution is a method called *iterative constraint strengthening* which enables us to solve a query for an arbitrary coverage criterion by a single call to the model checker and a novel form of incremental SAT solving: Whenever the SAT solver finds a solution, our algorithm compares this solution to the coverage criterion, and strengthens the SAT solver's clause database with additional clauses which exclude redundant new solutions.

To evaluate the language, we show how to express a list of informal requirements in FQL. Moreover, we perform practical experiments with the sample specifications. We demonstrate the scalability of FShell and its ability to compute compact test suites with experiments involving device drivers, automotive controllers, and open source projects.

All we have to decide is what to do with the time that is given to us.

J. R. R. Tolkien,
The Fellowship of the Ring

Acknowledgments

First of all I would like to thank my adviser Helmut Veith for guiding me through this labyrinth of graduate research. In those years of being a PhD student I learned far more than what one could expect of doing a PhD. Part of this experience was moving twice, first from Munich to Darmstadt, and two years later to Vienna. Despite the overhead of moving, this enriched my life with knowledge about different academic cultures and forms of organization, and of course extended social networks.

Next I would like to thank Daniel Kroening – not only for accepting to review my dissertation, but especially for his passion in building and improving the CPROVER framework, without which most of the work presented in this thesis never would have happened.

Most parts of this work were developed in a team, together with Andreas Holzer, Christian Schallhart and Helmut Veith. I am grateful for all the time and energy they invested in this project, for which this dissertation shall only be a first summary. Christian, thank you for teaching me all the details about the C++ programming language. Andi, I would also like to thank you for our discussions at any time of the day (others might call some of it "night"). The project was further inspired by our project partners Sven Bünte and Michael Zolda, whom I would also like to thank for their invaluable feedback on the implementation.

Our team, again, was embedded in a larger and evolving group, including my office mates Mohammad Khaleghi, Visar Januzaj and Florian Zuleger, and Johannes Kinder and Stefan Kugele. Despite being always busy, many of them offered help in reviewing parts of this thesis. Especially Johannes and Andi went to great lengths to provide detailed feedback – thank you! Part of the time that I formally was a PhD student was spent on industrial collaborations. Thanks to Stefan Kugele and Wolfgang Haberl those times in Munich were a lot of fun.

Of course, given this work was performed in parts at three different universities, many more people were involved directly or implicitly in this work. I would like to wholeheartedly extend my gratitude to all of you!

Even though such an academic life already stretches far beyond a formal 34-40.5 working hours job, such work can only be performed successfully when provided with tireless support by friends and family. I would like to thank my parents and Samira's parents for giving me a warm and cosy home

at any time, which becomes even more important when one doesn't quite know anymore which town to call one's home. I would like to thank my brother and my friends for helping me to build nice and comfortable places wherever I moved. Samira, thank you for your loving support and all your patience, and for following me to any place.

Thank you!

München Darmstadt Wien Innsbruck Oxford, 2006–2011
Michael Tautschnig

Contents

1 Introduction **1**
- 1.1 Software Testing . . . 2
  - 1.1.1 Terminology . . . 3
  - 1.1.2 Applications of Software Testing . . . 4
- 1.2 Query-Driven Program Testing . . . 6
- 1.3 Overview of Realization . . . 8
  - 1.3.1 Mathematical Framework . . . 9
  - 1.3.2 Query Language . . . 9
  - 1.3.3 Test Case Generation Back End . . . 10
- 1.4 Advantages and Limitations of the Implementation . . . 12
- 1.5 Contributions . . . 14
- 1.6 Related Work . . . 15

2 Requirements for the Design of FQL **19**

3 A Primer on Query-Driven Program Testing **23**
- 3.1 FQL Language Concept . . . 23
  - 3.1.1 Path Patterns: Regular Expressions . . . 25
  - 3.1.2 Coverage Specifications: Quoted Regular Expressions . 25
  - 3.1.3 Target Graphs and Filter Functions . . . 27
  - 3.1.4 Target Alphabet: CFA Edges, Nodes, Paths . . . 29
  - 3.1.5 Full FQL Specifications . . . 31
- 3.2 Example Specifications . . . 32
- 3.3 Disambiguating Specifications using FQL . . . 35
- 3.4 Tool Support for Query-Driven Program Testing: FShell . . . 38

4 A Mathematical Model for White-box Program Testing **42**
- 4.1 Intermediate Representation: Control Flow Automata . . . 42

4.2 Concrete Program Semantics: Transition Systems 45
4.3 Predicates and Coverage Criteria 46

5 FQL – the FShell Query Language 58
5.1 FQL Design Overview 58
5.2 FQL Elementary Coverage Patterns 60
5.2.1 Semantics of Elementary Coverage Patterns 61
5.2.2 Interpretation of Path Patterns as Path Predicates . . 62
5.3 Target Graphs and CFA Transformers 63
5.4 Filter Functions for ANSI C 66
5.4.1 ANSI C Specific Terminology 67
5.4.2 Detailed Specification of Filter Functions 69
5.5 FQL Specifications 71
5.6 Full FQL Specifications 74
5.7 Example of FQL Query Evaluation 78
5.8 Expressive Power and Usability 80
5.8.1 Scenario 1: Structural Coverage Criteria 80
5.8.2 Scenario 2: Data Flow Coverage Criteria 81
5.8.3 Scenario 3: Constraining Test Cases 81
5.8.4 Scenario 4: Customized Test Goals 83
5.8.5 Scenario 5: Seamless Transition to Verification 84
5.9 Discussion 85

6 FShell 87
6.1 Overview of CBMC's Architecture 87
6.2 Tool Architecture 90
6.3 Front End and Query Parsing 92
6.3.1 Command Line Options 93
6.3.2 Interactive Shell, Control Commands, and Macros . . . 97
6.3.3 Processing FQL Queries 99
6.3.4 Running Example 101
6.4 Computing Target Graphs 101
6.4.1 Example 103
6.4.2 Predicates over Program Variables 103
6.5 Trace Automata 104
6.5.1 Construction of Trace Automata 107
6.5.2 Program Traces 111

6.6 Integrating Trace Automata . . . 112
6.6.1 Program Instrumentation . . . 112
6.6.2 Propositional Encoding of Trace Automata . . . 121
6.7 Efficient Test Case Enumeration . . . 126
6.7.1 Overview of CDCL/DPLL SAT Solving . . . 126
6.7.2 Guided SAT Enumeration . . . 127
6.7.3 Coverage Analysis . . . 134
6.8 Test Suite Minimization . . . 137
6.9 Computing Test Inputs . . . 139
6.10 Test Harness Generation . . . 140

7 Evaluation **142**
7.1 Uses of Query-Driven Program Testing . . . 142
7.1.1 Measurement-based Execution Time Analysis . . . 143
7.1.2 Model/Implementation Consistency Checking . . . 143
7.1.3 Coverage Evaluation . . . 143
7.1.4 Reasoning on Coverage Criteria . . . 144
7.1.5 Test Case Generation . . . 145
7.1.6 Discussion . . . 145
7.2 Expressiveness . . . 145
7.3 Experimental Evaluation . . . 147
7.3.1 Efficient Evaluation of Complex Queries . . . 150
7.3.2 Applicability to Real-World Software . . . 153
7.3.3 Comparing to other Test Case Generation Approaches 154
7.3.4 Scalability . . . 157
7.4 Comparison of Instrumentation vs. Native SAT Encoding . . . 159
7.5 Minimization of Test Suites . . . 162

8 Conclusions **164**

Bibliography **167**

Curriculum Vitae **188**

Beware of bugs in the above code;
I have only proved it correct, not
tried it.

Donald E. Knuth

Chapter 1

Introduction

Software testing is performed ever since the first programs were written. As of today, testing is a key technique to provide evidence for correct operation of a system with respect to requirements. Hence testing is an integral part of software development processes. This includes development processes for industrial and safety critical systems, such as automotive systems [Int98], avionics [RTC92], or medical devices [Int03, Int06]. Yet there are two main problems that go hand-in-hand, as we shall see. First, there is a lack of suitable formalizations of what has to be tested. Second, testing remains one of the most expensive parts of systems development. Excessive cost is largely determined by the labor-intensiveness of manual testing, as we shall discuss first:

Cost. At the time where first programs were written, "program checkout" (testing and debugging) was the most labor-intensive step [Bak57]. The situation had neither changed quarter a century later, where the general rule of thumb was that 50% of development expenses are spent on testing [MSBT04], nor has it changed more than 50 years later: talking to industry confirms that testing accounts for at least 50% of development costs. Given today's cost pressure this is hardly affordable anymore. To improve on this, testing must be increasingly automated and be made as efficient as possible, for example with respect to the following criterion stated by Myers [MSBT04]:

> What subset of all possible test cases has the highest probability of detecting the most errors?

Clearly, naïve random testing – although highly automated, and in many cases the only automated procedure that is employed – can hardly yield a small set of test cases matching the above criterion. Hence more elaborate approaches for automated test case generation are sought for. Automation, however, requires a precise description of what is to be tested. Hence it is hampered by the lack of formalization:

Formalization. Test adequacy criteria, such as code coverage metrics, determine whether a set of test cases is suitable for detecting a high number of errors. For efficient automated test case generation these criteria must be formalized in a suitable way, such that test case generators can use them as guidance in computation of test cases. Although formalizations of selected criteria have been described in the literature (cf. Section 1.6), there is still no single framework that consistently captures a majority of the proposed coverage criteria. Therefore automated test case generation is always limited to a small set of hard-coded coverage criteria.

We therefore first developed a formal framework for describing coverage criteria, and, on top of that, an efficient technique for automated white-box program testing. Through proper formalization we therefore achieve automation to a much higher degree, which again shall help to reduce cost. Before we give a detailed description of our approach, we describe the general setting.

1.1 Software Testing

Myers [MSBT04] defines software testing as "... the process of executing a program with the intent of finding errors." This clearly distinguishes software testing from techniques such as static analysis [NNH99], abstract interpretation [CC77], or model checking [CE81, QS82], which aim at proving programs correct w.r.t. a specification without actually executing the program.

Testing could only establish correctness if all possible executions of a program had been shown to be error-free. This, however, is infeasible in all practical cases. Therefore, testing can only establish confidence that certain errors are not present, while it tells nothing about other potential errors. Its bug-finding capabilities, the ease of use and its scalability, however, made testing remain an integral part of software development. Furthermore all proofs of correctness will have to be complemented by tests as only tests show

the effects of actual executions, whereas proofs are bound to the correctness of assumptions on the environment (cf. [GG75] for a more extensive discussion). More even than a formal correctness proof, a well chosen suite of test cases is known to be extremely valuable for the working programmer: it helps to understand program behavior, it gives immediate feedback about code changes, it helps detect compiler errors (a particularly important issue in the context of off-the-beaten-track embedded processors), it enables the physical measurement of execution time and power consumption [WRKP05], and it can grow as the program develops. The last years have also seen novel formal verification techniques which integrate dynamic analysis with software model checking and static analysis [BCH+04b, GHK+06].

1.1.1 Terminology

As formal definitions are given no earlier than Chapter 4, we give an informal description of several important terms here.

- **Test Input.** Test input is program input that causes a unique sequence of steps in the execution of the program.
- **Test Case.** A test case describes an execution of the program. To denote such an execution, often only test input and the expected output is given.
- **Test Suite.** A test suite is a set of test cases.
- **Coverage Criterion.** A coverage criterion defines a metric over test suites. A coverage criterion measures the adequacy of a test suite for finding a certain set of errors. Well known coverage criteria include statement coverage, condition coverage or path coverage. A coverage criterion itself can also help to discover errors, e.g., dead code if full coverage of statements cannot be achieved.
- **Test Goal.** A test goal is a property of a test case. Coverage criteria induce test goals: for instance, statement coverage yields one test goal per statement. Each such property then states that one of the statements must be reached. Test goals are also known as test targets or test obligations.

- **Test Harness.** A test harness is a program fragment that executes the program under test with a specific test input.

In software testing we distinguish two main areas:

- In *black-box testing* the description of the system under test is limited to its interface and test inputs are therefore solely selected according to this interface description or other requirements specifications. To increase the likelihood of finding errors, testers choose boundary values or other specific values that likely yield errors. While computing specific inputs is easy to accomplish, error-free execution upon these inputs yields no information about even only slightly varied input data.

- *White-box testing*, in contrast, includes information about the logic of the program, its structure, and its implementation. First, this permits systematic study of parts of the program deemed critical. Second, interesting combinations of input values can be determined from the source code and reasoning about equivalence classes of input parameters becomes possible.

In the remainder of this dissertation we are solely concerned with white-box testing.

1.1.2 Applications of Software Testing

Applications of white-box testing range from ad hoc debugging to software certification of safety-critical systems:

1. For debugging, we need program specific ad hoc test suites that cover, e.g., certain lines of code or functions, or enforce a precondition in the execution of a function high up in the call stack. For reproducing stack traces we further need to model sequences of function calls, possibly including conditions over function arguments.

2. For requirement-based testing [WRHM06, UL06], we need test suites which reflect the intended system behavior. Alternatively, also violation of requirements must be tested by explicitly describing faulty behavior.

3. For certification, we need test suites that ensure standard coverage criteria, like condition coverage, in connection with industry standards such as DO-178B [RTC92].

4. In most practical cases the situation is even more complex: For instance, while a system is still under development, we want to assure, e.g., condition coverage, but avoid test cases invoking certain unimplemented functions.

It is therefore interesting to note that there is relatively little support for testing by formal methods and test case generation tools in today's incremental development processes. In particular, there is a strong need for a tool chain which allows the non-expert user to specify the test goals to be covered and the program paths permissible as test cases in the suite to be generated from this specification and source code.

Current best practice therefore requires a lot of tedious manual work to find test suites. Manual test case generation incurs both high costs and imprecision. Even though heuristic automated test case generation techniques such as random testing or directed testing [BM83, CDE08, God07, GKS05, GHK+06, SMA05] are very useful for general debugging, they can usually not achieve the coverage goals discussed here. Furthermore they are incompatible both with today's incremental development processes and certification standards. Incremental development of the product is accompanied by incrementally developing unit tests, either by a dedicated test engineer or the product developers themselves.

Working with industry and in several software projects we have seen test suites that achieve high coverage, e.g., 90% statement coverage, without being constructed with explicit coverage goals. Such test suites are built manually or using variants of random testing. In general, the most common use cases are covered, but often special cases and error paths remain untested. Manually deriving test cases for such scenarios is tedious and, as going from single functions to larger code chunks in integration testing, increasingly hard to do. Cheap random testing is not an option in such cases as it unnecessarily bloats a possibly well engineered test suite.

Instead of working towards automatic one-shot test suite generation, we intend to help test engineers and developers in these manual, labor-intensive steps with a tool that (a) simplifies the manual work needed to derive test cases and (b) can efficiently compute test cases missing in a desired test suite.

At the very core of these two goals lies a need for a means to specify test cases in a declarative way. Such specifications enable automatic generation of high quality test suites as defined by the developer. Because of its declarative style and an analogy to databases that we outline below, we refer to this approach as *query-driven program testing*.

1.2 Query-Driven Program Testing

Approaching testing from a model checking background, we were quite surprised that the literature contains a rich taxonomy and discussions of test coverage criteria (cf. Section 1.6), but is lacking a systematic framework for their specification. We believe that such a framework helps to reason about specifications and build tools which are working towards common goals.

History of computer science has shown that the introduction of temporal logic was essential to model checking, similarly as SQL/relational algebra was to databases. In particular, a formal and well-designed language helps to separate the problem specification from the algorithmic solution. Taking this analogy one step further, we developed query-driven program testing. This method, with its main constituents outlined in Figure 1.1, follows the main concepts of database systems: given a database and a declarative query, the back end efficiently processes these as black-box, and returns a result set.

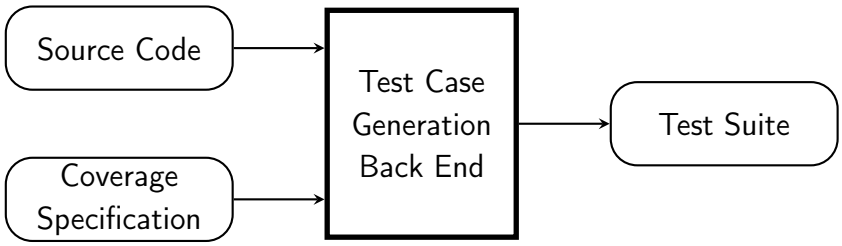


Figure 1.1: Query-driven program testing

In query-driven program testing we view the program as a database. Given a declarative coverage specification over such a program, the test case generation back end processes these inputs in a black-box fashion, and returns a set of test cases. We emphasize the advantage of the black-box style of the back end: we can apply optimizations and even use completely different solving strategies without changing the interface to the user.

Query-driven program testing is a method that provides both improvements in industrial practice as well as it opens up new research directions. We list some of these:

- **Test Case Generation.** Query-driven program testing enables us to compute test suites *according to user specified coverage criteria*. This feature is a crucial difference to directed testing which aims at good program coverage as a push button tool but has no explicit coverage goals. In particular, it enables the programmer to do intelligent and adaptive unit testing, even for unfinished code.

- **Requirement-driven Testing.** We can translate informal requirements into rigorous test specifications, and generate a covering test suite. When we evaluate the resulting test suite against a generic coverage criterion such as decision coverage, we understand whether the requirements contain sufficient detail to guide the implementation.

- **Certification.** We can formulate precise criteria for code certification in our query language and evaluate them on the source code. The lack of formal test specifications (even in standards such as DO-178B [RTC92]) has lead to inconsistent tool support. To illustrate the problem, we have used the four commercial test tools CoverageMeter [CMe], CTC++ [CTC], BullseyeCoverage [Bul], and Rational Test RealTime (RTRT) [RTR] to check for condition coverage on the C program shown in Listing 1.1.

```
1 void foo(int x) {
2   int a = x > 2 && x < 5;
3   if (a) { 0; } else { 1; }
4 }
```

Listing 1.1: Sample program

We compiled the C program using the tool chain of each coverage analysis tool and ran the programs with the two test cases `x = 1` and `x = 4`. Here, CoverageMeter and CTC++ reported 100% coverage but the other two tools returned a mere 83%. The difference occurs because BullseyeCoverage and RTRT treat not only the variable `a` in line 3 as condition but also `x>2` and `x<5` in line 2.

- **Coverage Evaluation.** We can determine coverage with respect to a query achieved by other test methods, e.g., directed, model-based, or manual testing. A clear understanding of coverage enables us to combine existing testing techniques in a precise manner. For instance, we can derive concise specifications of *missing* test cases to perform automated coverage completion with a more powerful tool such as the query-driven test case generator presented in this dissertation.

- **Systematic Reasoning about Test Specifications.** We believe that a rigorously defined test specification language and the underlying mathematical framework gives us a clean and simple basis to study fundamental issues about test specifications such as equivalence and subsumption of specifications, normal forms, distribution of specifications to multiple test servers etc.

- **Independent Development of Back Ends.** The test specification language precisely defines the interface available to the user of a query-driven program testing tool chain. Therefore back end optimizations or even completely new back ends can be developed and later be added as drop-in replacements. A first back end is presented in this dissertation, but there is also already ongoing work on a second back end which, internally, uses almost orthogonal concepts.

1.3 Overview of Realization

To implement query-driven program testing we decompose the task into three main building blocks:

1. Analogously to relational algebra begin a foundation for databases, we define a mathematical framework to give a semantics of coverage criteria.

2. On top of this framework we define a declarative language that can be used by practitioners – "SQL for program testing."

3. We develop back ends that facilitate efficient generation of test cases for real-world programs.

Our solution culminates in a tool – FShell[1] – which processes C programs as input source code, accepts queries in FQL (FShell Query Language), and computes corresponding test suites.

1.3.1 Mathematical Framework

The mathematical core of this dissertation is a slink framework that suffices to model standard coverage criteria. We use control flow automata as syntactic and transition systems as semantic representation of programs. Thereupon we define three levels of predicates: state predicate, path predicates, and path set predicates. These types of predicates are used to describe the test goals induced by coverage criteria. Therefore we are then able to give a formal definition of coverage criteria. To the best of our knowledge, this is the first model that captures all kinds of coverage criteria in a single mathematical framework. The resulting abstraction layer encompassing all practical coverage criteria is essential for the definition of a query language.

1.3.2 Query Language

Despite the practical importance of a test specification language, there was very little previous work on this topic. With the aim of developing an automatic test case generation tool for measurement-based execution time analysis, we first developed a prototype that permitted specifications of paths in terms of a sequence of lines of code [HSTV08]. This early prototype helped to explore capabilities of automatic test case generation, but the semantics of its specification language was only described in terms of an implementation, which often led to non-obvious results. We consequently developed the mathematical model sketched above, which we first described in [HSTV09]. Such proper foundations enabled us to develop efficient algorithms in the test case generation back end. The query language FQL built on top of this mathematical model was presented in [HSTV10]. Specifications in FQL enable the user to formulate test specifications which range from local code-specific requirements ("cover all decisions in function `foo` using only calls from function `bar` to `foo`") to generic code-independent requirements ("condition coverage"). We have designed FQL as a specification language which

[1] Originally *FORTAS shell*, as it was developed within the research project FORTAS – Formal Timing Analysis Suite for Real Time Programs

is easy to read – it is based on regular expressions – but has an expressive and precise semantics.

The main challenge was to find a language that enables us to work towards these goals, but is simple enough to be used by practitioners, and clean enough to facilitate a clear semantics. The role models for our language were languages such as LTL and SQL. We believe that our language FQL is a valuable first step towards a test specification language bearing the quality of these classics. It is *easy* to find a complicated very rich test specification language, but the challenge was to find a simple and clean one. The main difficulty we were facing in the design of FQL stems from the need to talk about both structural and syntactic elements of the code, and the semantics of the program under test *in one formalism*.

1.3.3 Test Case Generation Back End

Another major contribution of this dissertation is an efficient query engine which integrates our theoretical framework, code instrumentation, bounded model checking, and SAT enumeration into a tool of high efficiency. Our query engine employs and adapts the software model checking framework of Kroening's C bounded model checker (CBMC) [CKL04]. CBMC handles full ANSI C and translates such programs to Boolean formulas in a bit-precise manner. A SAT solver then computes satisfying assignments, which correspond to counterexamples of a safety property. We adapted and extended the code based to employ it for efficient test case generation. Given an FQL query, our tool performs the following conceptual steps, cf. Figure 1.2:

(1) We use the code base of CBMC to first obtain an intermediate representation of the program under test and later a SAT instance whose solutions correspond to the feasible program paths.

(2) With information about the intermediate representation we can translate the FQL query into automata over statements in the intermediate representation. If the query contains predicates over program variables, instrumentation is used to embed evaluation of predicates into the program. The SAT instance computed by CBMC is augmented with propositional encodings of the automata. Furthermore we keep mapping information, such that for each SAT solution, we can easily determine which test goals are covered.

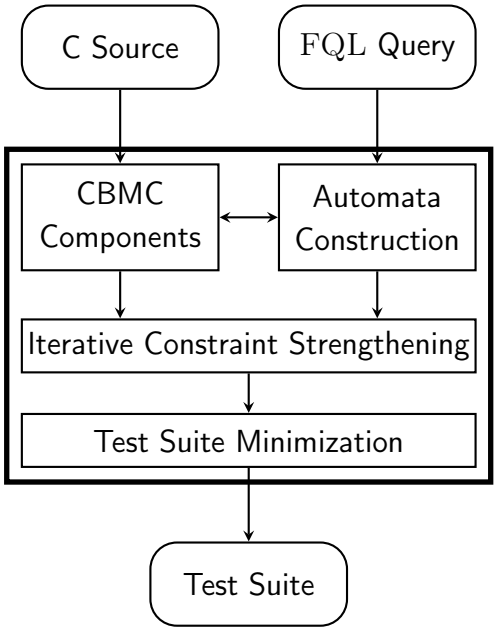


Figure 1.2: Query processing

(3) We use the SAT solver to enumerate test cases as solutions to the SAT instance until we satisfy the coverage criterion defined by the query. The *iterative constraint strengthening* technique used in this step is discussed below.

(4) To remove redundant test cases, we perform a test suite minimization. This problem is an instance of the minimum set cover problem, which we reduce to a series of SAT instances.

Iterative Constraint Strengthening (ICS). A naïve implementation of step (3) above would either use SAT enumeration to compute an enormous number of test cases until the test goals are reached, or it would call the SAT solver for each query goal anew. In iterative constraint strengthening, we circumvent both problems by modifying the clause database of the SAT solver on-the-fly. Whenever the SAT solver halts to output a solution, we compare the test case obtained from this solution against the test goals. Then we add new clauses to the clause database in such a way that the next solution is guaranteed to satisfy at least one test goal that had not yet been covered. In this way, we exploit incremental SAT solving to quickly enumerate a test suite of high quality: since we only add new clauses to the clause database,

the SAT solver is able to reuse information learned in prior invocations. We further refine this strategy in *groupwise constraint strengthening (GCS)*: coverage criteria such as Cartesian combinations of basic block coverage or predicate complete coverage nominally have an exponential number of test goals. For efficient enumeration we partition the goals into a small number of groups characterized by a common compound goal.

1.4 Advantages and Limitations of the Implementation

Query-driven program testing is a method that is applicable for all kinds of programming languages and software. Yet, in our implementation we restricted ourselves to C programs and had to take several decisions. Our choice of CBMC and bounded model checking as a query solving back end has advantages which come at a price: On the one hand, we achieve excellent performance and have the guarantee that the model-checker respects ANSI C, which is important for low level code, our primary application area. On the other hand, a bounded model checking approach is *unable to compute certain test cases* involving paths larger than the constant bound. It is easy to come up with examples where this situation will happen, but it is detectable by CBMC and accounted for in our implementation; in our evaluation we had to find suitable bounds, which we state for each experiment.

The following additional issues have to be considered when working with C programs, and we describe how they are addressed:

- **Bit-precise Reasoning.** C programs, especially those used in embedded systems, make frequent use of bit-wise operators for efficiency reasons. Consequently any program analysis must take these operators plus architecture-specific properties such as bit-widths of data types, overflow, and endianness into account. CBMC uses bit-precise modeling and includes support for architecture configuration.

- **Pointers and Dynamic Memory Allocation.** Pointers, including function pointers, are part of any non-trivial C program. CBMC fully supports pointers to data objects and resolves function pointers to the resulting function calls. Pointer arithmetic and dynamic memory allocation are supported as well.

- **Arrays and Data Structures.** Through pointers and dynamic memory allocation unbounded data structures can be built at runtime. CBMC can analyze these up to a given bound. For test case generation in cases where the entry function takes pointers to data structures as input parameters, however, additional stubs have to be written at present. While this will be addressed in future work, we currently assume that such pointers reference invalid memory (a worst-case assumption).

- **Behavior left undefined in ANSI C.** The standard defining ANSI C [Ame99a] leaves several aspects to an implementation, such as the representation of floating point numbers or order of evaluation for arguments of arithmetic operations. For floating point numbers, CBMC already includes support for standardized representations, which may become part of future versions of FShell. Other aspects left undefined are resolved in an arbitrary way.

- **Concurrency and Atomicity.** Multi-threaded code is not yet properly supported. Basic support is currently being developed, but testing concurrent software requires extensions of FQL as well. This will be addressed in future work.

- **Library Calls.** Calls to functions where no source code is available are assumed to return nondeterministic values. For many functions of the C library, CBMC includes stubs that approximate the behavior of these functions. For test case generation, we consider the return values of undefined functions as part of the test input.

- **Inline assembly.** Neither FShell nor CBMC addresses inline assembly in any way, other than skipping over it. If future versions of CBMC gain support for embedded assembly code, possibly by combining CBMC with analysis frameworks for binaries [KV08], FShell would readily no longer ignore it. For proper testing, however, extensions of FQL will likely become necessary.

- **Test Input.** When arbitrary programs are tested it remains to define what is considered as input parameters. In FShell we include as test input all parameters of the entry function, undefined local variables, and return values of undefined functions. Optionally, global variables

can be marked as test input – by default, and in accordance with the ANSI C standard, CBMC zero-initializes global variables.

- **Test Execution.** As a result of the lack of a well-defined input-interface test execution is a non-trivial task. Variables and functions marked as test input by FShell have to be set to the computed values. At present, only a prototypical Perl script is provided that edits (copies of) the source code to include input values.

1.5 Contributions

To summarize the approach sketched above, this dissertation contributed to improvements in the state of the art in the following ways:

- Query-driven program testing is an entirely new approach towards testing. In this approach the test engineer uses a declarative query language (FQL) to specify the test goals to be covered. The design of this query language follows requirements (Chapter 2) that were collected while working in industrial projects. As this approach for software testing is new, we give a tutorial-style introduction that describes the main concepts of FQL and several use cases. Furthermore we showcase the practical use of FQL in the automated test case generation tool FShell (Chapter 3).

- The core of query-driven program testing is a mathematical framework to describe coverage criteria (Chapter 4). We show that this formal framework is sufficiently general to describe both well-established coverage criteria and new coverage criteria defined in an ad hoc manner.

- FQL is the first declarative language that allows programmers to specify coverage criteria in a concise and flexible manner (Chapter 5). The specification language builds upon the above mathematical framework, with the restriction to elementary coverage criteria.

- The main contribution of this dissertation is the efficient implementation of query-driven program testing using FQL in the tool FShell (Chapter 6). FShell adopts components of the C bounded model checker to enable practical automated test case generation for ANSI C

programs. Efficient computation of test suites is achieved by employing SAT solvers in several ways: for test case enumeration, for coverage analysis, and for test suite minimization. We show the effectiveness of our approach on several case studies and dedicated experiments (Chapter 7).

1.6 Related Work

The idea of using code coverage as a measurement for test adequacy was first presented in [MM63]. Structural coverage criteria, e.g., basic block coverage, condition coverage, or path coverage, and data flow coverage criteria such as all-definitions are well studied [How75, Hua75, Pai75, McC76, Her76, WHH80, LK83, Gou83, RW85, Nta88, FW88, Het88, CPRZ89, HS91, BM93, UY93, ZHM97, AOX08], albeit with different names and a notable lack of precise definitions. Publications on specific coverage criteria tend to establish formalisms that suffice to describe the specific set of criteria only, making use of set-based [FW93] or graph-based [PC90] approaches. There is, however, only little work on mathematical frameworks that generalize to all these criteria, and even less work for making coverage criteria applicable in test case generation.

Vilkomir and Bowen [VB08] present a comprehensive formalization using the Z notation. They build a series of schemata to compare several well known coverage criteria. Although such an approach enables proper reasoning about different coverage criteria, it is not suitable for test case generation: They do not consider specifics of programming languages nor are program semantics modeled in any way. Hence instantiation of the schemata for a given program such as to discuss test case generation is not possible.

A framework targeted at automatic test case generation was described by Lee et al. [HLSU02, TSL04], who use CTL to formalize coverage criteria. They apply CTL as a formal framework to describe both structural and data flow coverage criteria. Using these specifications, they apply model checkers for automatic test generation. For representing the system under test they use extended finite state machines, which naturally induce Kripke structures for capturing semantics. Apart from different notations, these foundational definitions only differ from the concepts described in Sections 4.1 and 4.2 in the lack of annotations, which we attach to CFA edges. Structural code coverage criteria, however, are defined on source code. Hence these annotations

are a key capability to retain the relation to source code and only these enable a formal definition of standard coverage criteria. Ammann et al. [AOX08] use graphs as basic concept and define several coverage criteria on top of these. Yet they are also missing a relation to source code. While they do argue that, e.g., node coverage on graphs equals statement coverage on source code, this does not easily generalize to other white-box coverage criteria.

Although standard coverage criteria are a key aspect in the design of query-driven program testing in general and the query language in particular, we strive for a more versatile specification language. Hence we also studied work that allows to describe tests beyond standard coverage criteria. Many of these publications are found in context of model-based testing. Most existing formalisms for test specifications focus on the description of test *data*, e.g., TTCN-3 [Din04] and the UML 2.0 Testing Profile [SDGR03], but none of them allows to describe structural coverage criteria. Friske et al. [FSWl08] have presented coverage specifications using OCL constraints. Although OCL provides the necessary operations to speak about UML models, OCL constraints can yield hard to read expressions for complex coverage criteria. There exist several approaches that cover specifically the test input generation part for UML models: In the tradition of automata-theoretic methods, the most common [DNSVT07] approaches employ UML state machines [WS07, CTF01] and interaction diagrams [NS09], respectively.

The basic principles behind model-based testing were described by Chow in 1978 [Cho78], the term model-based testing was coined and further refined by Dalal et al. [DJK+99]. Their work includes automated test input generation and focuses on boundary value testing. Hessel et al. [BHJP04] present a specification language for coverage criteria at model level that uses parameterized observer automata. Test suites for specified coverage criteria can be automatically generated using the tool Uppaal Cover [HLM+08]. In [BHM+09], the generation of test inputs for Simulink models is realized via a translation of models to C code. This code is subsequently processed by a tool that – like FShell – is built upon CBMC. One of the most advanced model-based testing tool chains is Spec Explorer [VCG+08]. It combines model-based testing with various techniques for automated test case generation. Spec Explorer works on Spec# models and .Net code and uses AsmL [BGN+03] as formal foundation. Building upon Spec Explorer, Kicillof et al. [KGTB07] describe an approach that combines model-level

black-box testing with parametrized white-box unit testing. Black-box approaches, such as input/output conformance (also known as "ioco") testing as performed in the TorX framework [TB03], require different specifications. Query-driven program testing could even be applied in such cases, albeit only using a fraction of its power. Briones et al. [BBS06] investigate coverage measures considering the semantics of a functional specification and weighted fault models to arrive at minimal test suites in a black-box setting.

Random testing [BM83], fuzz testing [GLM08], and the use of genetic algorithms [Hol92] for test case generation [JSE96, PHP99] are applicable in both black-box and white-box settings, but cannot flexibly incorporate coverage specifications. Directed testing, and further symbolic execution based approaches aim at achieving a high code coverage with respect to standard criteria like basic block or path coverage [CDE08, God07, GKS05, GHK+06, SMA05, TS05, TS06]. These approaches are *not* tailored towards flexible and customized coverage criteria, and are therefore orthogonal to our work. It is an interesting question for future research, however, which FQL specifications can be solved efficiently by directed testing.

The use of model checking for test case generation was first proposed in [GH99]. They did, however, not consider coverage as guidance. Test case generation with model checkers following coverage specifications was proposed in [RH01a]. This was further refined in [HLSU02, TSL04, FW06, HLM+08, RH01b, HRV+03], as in parts already discussed above. The notion of coverage, however, must be taken with a grain of salt: Some of these publications focus on coverage of specifications rather than structural coverage, cf. [FWA09] for a survey. They use NuSMV [CCGR99], SAL2 [HdMR04], and Java PathFinder [VPK04] as model checking back ends. Consequently none of these approaches support ANSI C syntax and semantics.

Prior to our work, Beyer et al. [BCH+04b] presented a test case generation engine for C programs that supports "target predicate coverage", i.e., every program location has to be visited by some test case that enters the location such that predicate `p` evaluates to `true`. In FQL, this coverage criterion is given by the specification `cover {p}.NODES(ID)`. For test case generation Beyer et al. use an extended version of the C model checker BLAST [HJMS02, BHJM07]. Note that BLAST uses the database analogy in a different way than we do. BLAST uses a query language [BCH+04a] to process and access reachability information from the software model checker.

However, the BLAST query language is not well suited for specifying complex coverage criteria: (i) Specifications have to be stated in a combination of two formalisms, one for an observer automaton, and the other for a relational query. (ii) The BLAST language misses concise primitives for coverage criteria; for instance, path coverage can only be achieved by creating an individual observer automaton for each program path. (iii) The encoding of FQL's passing clause into a BLAST observer automaton is in general nontrivial for the test engineer.

> . . . the tools we are trying to use and the language or notation we are using to express or record our thoughts, are the major factors determining what we can think or express at all!
>
> *Edsger W. Dijkstra*

Chapter 2

Requirements for the Design of FQL

Bearing in mind what Dijkstra said in "The Humble Programmer" [Dij72] (see quote above), the design of the query language will be a key to success of query-driven program testing. The language must balance expressiveness versus an easy-to-use syntax and its underlying concepts must be rigorous yet easily accessible to the programmer. In engineering our query language FQL we therefore considered the possible use cases described in Section 1.2 to end up with the following list of requirements that have to be addressed:

(a) **Simplicity and Code Independence.** Simple coverage specifications should be expressed by simple FQL specifications. For example, well-known coverage criteria should be expressed with short queries. At the same time, however, it should be avoided to require a new keyword for each coverage criterion. The resulting small set of reserved words must be chosen in a way that makes queries easy to read. Furthermore, to facilitate early test goal specifications and their reuse throughout a project, FQL specifications should be maximally code independent; for instance, a specification referring to a procedure should not depend on line numbers.

(b) **Encapsulation of Language Specifics.** Specifications given in FQL necessarily refer to elements of source code. Nevertheless FQL should reduce the specifics of a programming language, such as ANSI C, to a minimum. To this end, FQL should provide a clear and concise interface with the underlying programming language.

(c) **Precise Semantics.** FQL specifications should have a simple and unambiguous semantics.

(d) **Expressive Power.** FQL should be based on a small number of orthogonal concepts. These must allow to express coverage specifications ranging from standard coverage criteria to ad hoc coverage requests arising during systems development, as discussed in Section 1.1.2. As examples of the wide variety of possible coverage specifications we list 24 possible queries in Figures 2.1–2.3.

(e) **Tool Support for Real World Code.** FQL must have a good trade-off between expressive power and feasibility. In particular, common coverage specifications should lend themselves naturally to efficient test case generation algorithms.

Scenario 1: Structural Coverage Criteria. The certification of critical software systems often requires coverage criteria such as basic block, condition or decision coverage [MSBT04] which refer to entities present in all source code. This results in our first specifications.

[**Q1-2** — *"Standard Coverage Criteria"*] Basic block coverage and condition coverage.

Assuming that **Q2** refers to BullseyeCoverage and RTRT's interpretation of condition coverage, one must also be able to express the competing criterion:

[**Q3** — *"Alternative Condition Coverage"*] Condition coverage as defined by CoverageMeter and CTC++ (see Section 1.2).

For intensive testing a developer will employ a variant of path coverage [Nta88], but restrict it to local coverage due to high costs:

[**Q4** — *"Acyclic Path Coverage"*] Cover all acyclic paths through functions `main` and `insert`.

[**Q5** — *"Loop-Bounded Path Coverage"*] Cover all paths through `main` and `insert` which pass each statement at most twice.

Figure 2.1: Twenty-four examples of informal test case specifications

Scenario 2: Data Flow Coverage Criteria. We give three examples of typical data flow coverage criteria.

[**Q6** — *"Def Coverage"*] Cover all statements defining a variable `t`.

[**Q7** — *"Use Coverage"*] Cover all statements that use the variable `t` as right hand side value.

[**Q8** — *"Def-Use Coverage"*] Cover all def-use pairs of variable `t`.

Scenario 3: Constraining Test Cases. During development and for code exploration, it is often important to achieve the desired coverage with test cases which, for instance, avoid a call to an unimplemented function. Below we list five examples of this group.

[**Q9** — *"Constrained Program Paths"*] Basic block coverage with test cases that satisfy the assertion $j > 0$ after executing line 2.

[**Q10** — *"Constrained Calling Context"*] Condition coverage in a function `compare` with test cases which call `compare` from inside function `sort` only.

[**Q11** — *"Constrained Inputs"*] Basic block coverage in function `sort` with test cases that use a list with 2 to 15 elements.

[**Q12** — *"Recursion Depth"*] Cover function `eval` with condition coverage and require each test case to perform three recursive calls of `eval`. [**Q13** — *"Avoid Unfinished Code"*] Cover all calls to `sort` such that `sort` never calls `unfinished`. That function is allowed to be called outside `sort` – assuming that only the functionality of `unfinished` that is used by `sort` is not testable.

[**Q14** — *"Avoid Trivial Cases"*] Cover all conditions and avoid trivial test cases, i.e., require that `insert` is called twice before calling `eval`.

Figure 2.2: Twenty-four examples of informal test case specifications (*cont.*)

Scenario 4: Customized Test Goals. Complementary to the constraints on test *cases* of Scenario 3, we also want to modify the set of test *goals* to be achieved by the test cases.

[**Q15** — *"Restricted Scope of Analysis"*] Condition coverage in a function `partition` with test cases that reach line 7 at least once.

[**Q16** — *"Condition/Decision Coverage"*] Condition/decision coverage (the union of condition and decision coverage) [MSBT04].

To study interactions of two program parts, it is not sufficient to cover the *union* of the test goals induced by each part; tests must cover their *Cartesian product*:

[**Q17** — *"Interaction Coverage"*] Cover all possible pairs between conditions in function `sort` and basic blocks in function `eval`, i.e., cover all possible interactions between `sort` and `eval`.

In a similar spirit, we can also approximate path coverage by covering pairs, triples, etc. of basic blocks:

[**Q18-20** — *"Cartesian Block Coverage"*] Cover all pairs, triples, and quadruples of basic blocks in function `partition`.

Scenario 5: Seamless Transition to Verification. When full verification by model checking is not possible, testing can be used to approximate model checking. For instance, we can specify to cover all assertions.

[**Q21** — *"Assertion Coverage"*] Cover all assertions in the source.

[**Q22** — *"Assertion Pair Coverage"*] Cover each pair of assertions with a single test case passing both of them.

We can finally use test specifications to provoke unintended program behavior, effectively turning a test case into a counterexample. In the following examples, we check the presence of an erroneous calling sequence and the violation of a postcondition:

[**Q23** — *"Error Provocation"*] Cover all basic blocks in `eval` without reaching label `init`.

[**Q24** — *"Verification"*] Ask for test cases which enter function `main`, satisfy the precondition, and violate the postcondition.

Figure 2.3: Twenty-four examples of informal test case specifications (*cont.*)

Quality is never an accident; it is always the result of intelligent effort.

John Ruskin

Chapter 3

A Primer on Query-Driven Program Testing

Automated test case generation techniques such as random testing or directed testing can help to quickly uncover a series of bugs when run on a previously untested program. It is rarely the case, however, that a program is put together in one atomic step and only being tested afterwards. Instead, today's agile development style [BBvB+01] emphasizes incremental development, where testing is supposed to happen in intermediate steps as well. But even in an idealized industrial development process, where only completed implementations are tested, these tests have to follow an engineer's specifications instead of randomly walking the code. *Query-driven program testing* aims to fill this void: We use declarative specifications of test suites – *queries* – and provide suitable back ends that automatically generate suitable test cases. In the following we start from very simple specifications and describe how the programmer expresses them using our query language FQL. We continue to more complex usage scenarios and describe subtle differences in specifications, which can be clearly distinguished using FQL. We end with an example session of our tool, FShell, and briefly describe its usage.

3.1 FQL Language Concept

It is natural to specify a single program execution – *a test case* – on a *fixed given program* by a regular expression. For instance, to obtain a test case which leads through line number 4 (*covers* line 4) of the program, we can

write a regular expression `ID* . @4 . ID*`, where '`ID`' denotes a wildcard. We will refer to such regular expressions as *path patterns*. Equipped with a suitable alphabet which involves statements, assertions and other program elements, path patterns are the backbone of our language.

Writing path patterns as test specifications is simple and natural, but has a principal limitation: it only works for individual tests, and not for test suites. Let us discuss the problem on the example of basic block coverage. Basic block coverage requires a test suite where

> *"for each basic block in the program there is a test case in the test suite which covers this basic block."*

It is clear that basic block coverage can be achieved manually by writing one path pattern for each basic block in the program. The challenge is to find a specification language from which the path patterns can be automatically derived. This language should work not only for simple criteria such as basic block coverage, but, on the contrary, facilitate the specification of complex coverage criteria, such as those described in the introduction. To understand the requirements for the specification language, let us analyze the above verbal specification:

A The specification requires a *test suite*, i.e., multiple test cases, which together have to achieve coverage.

B The specification contains a *universal quantifier*, saying that each basic block must be covered by a test case in the test suite.

C Referring to entities such as "basic blocks" the specification assumes *knowledge about program structure.*

D The specification has a meaning which is independent of the concrete program under test. In fact, it can be translated into a set of path patterns only *after the program under test is fixed.*

The challenge is to find a language with a syntax, expressive power, and usability appropriate to the task. Our solution is to evolve regular expressions into a richer formalism (FQL) which is able to address the issues A-D.

In the following we will discuss the main features of FQL. We use the C code snippet shown in Listing 3.1 to explain basic aspects of FQL. To exemplify more complex test specifications and their FQL counterparts we will augment this snippet with additional program code.

```
1 int cmp(int x, int y) {
2   int value = 0;
3   if (x > y)
4     value = 1;
5   else if (x < y)
6     value = −1;
7   return value;
8 }
```

Listing 3.1: C source code of function `cmp`

3.1.1 Path Patterns: Regular Expressions

FQL is a natural extension of regular expressions. To cover line 4, we just write

```
> cover "ID* . @4 . ID*"
```

The quotes indicate that this regular expression is a path pattern for which we request a matching program path. We use the operators '`+`', '`*`', '`.`' for alternative, Kleene star and concatenation. Note that the regular expressions can contain combinations of conditions and actions, as in

```
> cover "ID* . { x==42 } . @4 . ID*"
```

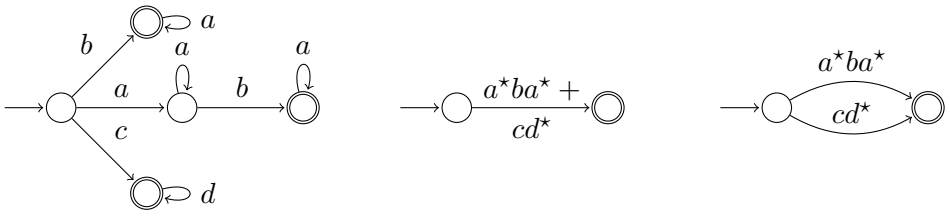
which requests a test where the value of variable `x` is 42 at line 4. For the first query a suitable pair of inputs is, e.g., `x = 1`, `y = 0`, whereas the second query requires `x = 42` and a value of variable `y` smaller than 42, such as `y = 0`.

3.1.2 Coverage Specifications: Quoted Regular Expressions

Using the regular alternative operator '`+`' we can build a path pattern matching all basic block entries in Listing 3.1. These map to line numbers 2, 4, 6, and 7. Consequently we can describe the basic block entries using the path pattern `@2 + @4 + @6 + @7` and use a query

```
> cover "ID* . (@2 + @4 + @6 + @7) . ID*"
```

to request *one* matching test case. For basic block coverage, however, we are interested in multiple test cases covering *all* of these four lines – a *test suite*.



(a) `a*.b.a* + c.d*` (b) `"a*.b.a* + c.d*"` (c) `"a*.b.a*"+"c.d*"`

Figure 3.1: Automata resulting from expansion of path patterns and coverage specifications

Note that no single assignment to the parameters `x` and `y` can cause execution of both assignments in lines 4 and 6. As path patterns are well suited to describe single test cases, we introduce *coverage specifications*, which describe a *finite* language *over path patterns*, where *each word* defines one *test goal*. We emphasize finiteness: An infinite number of test goals would lead test case generation ad absurdum. To tackle the infinite nature of an expression like `ID*` we introduce *quoting*: The coverage specification `"ID*"` queries for one test case that matches a word in the language of `ID*`. The same holds true for expressions such as `"ID* . (@2 + @4 + @6 + @7) . ID*"` as used above, where we implicitly used quoting. We refer to this extension of regular expressions as *quoted regular expressions*. Together with concatenation and alternative, but not Kleene star, FQL combines quoted regular expressions into coverage specifications for test suites. We illustrate the effect of quoting on a simple example: A path pattern `a*.b.a* + c.d*` describes an *infinite* language

$$\mathcal{L}(\texttt{a*.b.a* + c.d*}) = \{b, c, ab, cd, aba, cdd, aab, aaba, aabaa, \ldots\}$$

with a corresponding finite automaton shown in Figure 3.1(a). If we enclose this pattern into quotes, then the expansion of the regular expression will be blocked. Thus, `"a*.b.a* + c.d*"` defines a *finite* language

$$\mathcal{L}(\texttt{"a*.b.a* + c.d*"}) = \{a^\star ba^\star + cd^\star\}$$

and the automaton shown in Figure 3.1(b). If only the two subexpressions are quoted, i.e., `"a*.b.a*"+"c.d*"` is used as coverage specification, we obtain two words: Formally, we treat the quoted regular expressions `"a*.b.a*"` and `"c.d*"` as temporary alphabet symbols x and y and obtain *all* words in the

resulting regular language $x + y$ with $\mathcal{L}(x + y) = \{x, y\}$, cf. Figure 3.1(c):

$$\mathcal{L}(\texttt{"}a\texttt{*}.b.a\texttt{*"+"}c.d\texttt{*"}) = \{a^\star ba^\star, cd^\star\}.$$

The words of coverage specification are the path patterns which the test suite has to satisfy. As we will see more clearly below, this feature equips FQL with the power for universal quantification. To specify a test suite achieving basic block coverage we hence write

```
> cover "ID*" . (@2 + @4 + @6 + @7) . "ID*"
```

which is tantamount to a list of four path patterns:

```
> cover "ID* . @2 . ID*"
> cover "ID* . @4 . ID*"
> cover "ID* . @6 . ID*"
> cover "ID* . @7 . ID*"
```

3.1.3 Target Graphs and Filter Functions

For a fixed given program, coverage specifications using `ID` and line numbers such as `@7` are useful to give ad hoc coverage specifications. For program independence and generality, FQL has support to access additional natural program entities such as basic blocks, files, decisions, etc. We call these functions *filter functions*.

For instance, in the above example, the filter function `@BASICBLOCKENTRY` is essentially a shorthand for the regular expression `@2+@4+@6+@7`. Thus, the query

```
> cover "ID*" .@BASICBLOCKENTRY. "ID*"
```

will achieve basic block coverage. To make this approach work in practice, of course we have to do more engineering work. It is only for simplicity of presentation that we identify program locations with line numbers.

Towards this goal, we represent programs as *control flow automata* (CFA). Henzinger et al. [HJMS02] proposed CFAs as a variant of control flow graphs where statements are attached to edges instead of nodes. The nodes then correspond to program locations. In Figure 3.2 the CFA for Listing 3.1 is shown; for illustration, we use line numbers as program locations. This CFA contains assignments, a function return edge, and assume edges: bracketed expressions describe assumptions resulting from Boolean conditions.

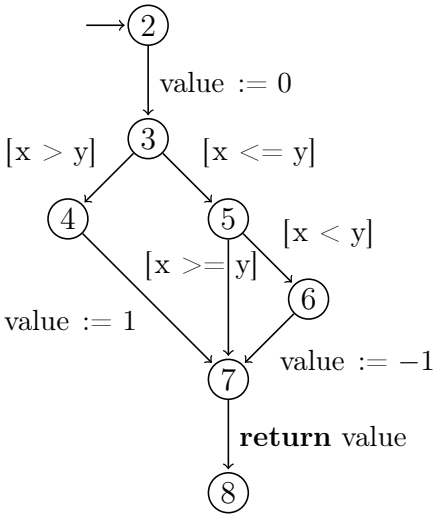


Figure 3.2: Control flow automaton for Listing 3.1

We define *target graphs* is a subgraphs of the CFA. Filter functions are used to extract different target graphs from a given program. For instance, we have filter functions for individual program lines such as `@4`, basic blocks (`@BASICBLOCKENTRY`), functions (as in `@FUNC(sort)`), etc. To consider another example, the filter function `@CONDITIONEDGE` refers to the true/false outcomes of all atomic Boolean conditions in the source code.

Thus, filter functions and target graphs provide the link to the individual programming language. The evaluation of filter functions to target graphs is the only language-dependent part of FQL.

Let us return to our running example: The filter function `ID` selects the entire CFA as target graph. For the program in Listing 3.1 with the CFA of Figure 3.2 an expression `@2` selects the edge $(2, 3)$, and `@BASICBLOCKENTRY` yields the edges $(2, 3)$, $(4, 7)$, $(6, 7)$ and $(7, 8)$. For `@CONDITIONEDGE` we obtain the subgraph consisting of the edges $(3, 4)$, $(3, 5)$, $(5, 6)$ and $(5, 7)$; the same result could have been obtained by combining the target graphs of `@3` (edges $(3, 4)$, $(3, 5)$) and `@5` (edges $(5, 6)$, $(5, 7)$), using set union: FQL provides functionality to extract and manipulate target graphs from programs, for instance the operations '`&`' and '`|`' for intersection and union of graphs, or '`NOT`' for complementation. For example, to extract the conditions *of function* `cmp` *only*, we intersect the target graphs of `@FUNC(cmp)`, which yields all edges in function `cmp`, and `@CONDITIONEDGE`. In FQL, we write this intersection as `@FUNC(cmp) & @CONDITIONEDGE`.

3.1.4 Target Alphabet: CFA Edges, Nodes, Paths

In our test specifications, we can interpret target graphs via their edges, their nodes or their paths. In most cases, it is most natural to view them as sets of edges. In the above examples, we implicitly interpreted a target graph resulting from the application of a filter function `@BASICBLOCKENTRY` as a set of edges: for Listing 3.1 we obtained four edges.

In fact, expressions such as `@BASICBLOCKENTRY`, which we used throughout the section, are shorthands for regular expressions constructed from the set of CFA edges, which can be made explicit by stating `EDGES(@BASICBLOCKENTRY)`. By default, FQL will interpret every target graph as a set of edges.

The target graph, however, can also be understood as a set of nodes – or even as a description of a set of finite paths. Let us study these three cases on practical examples of coverage requirements for the program in Listing 3.1.

- *Edges.* In FQL, `EDGES(@FUNC(cmp))`, or simply `@FUNC(cmp)`, yields the expression $(2,3)+(3,4)+(3,5)+(4,7)+(5,7)+(5,6)+(6,7)+(7,8)$. Hence the coverage specification of a query

  ```
  > cover "EDGES(ID)*" . EDGES(@FUNC(cmp)) . "EDGES(ID)*"
  ```

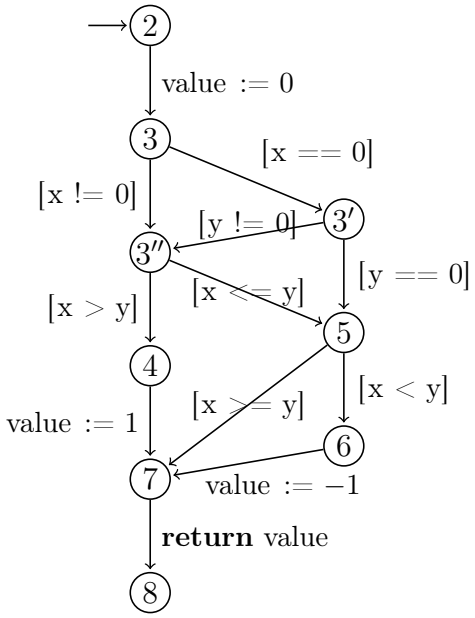
  has eight goals, one for each edge. Three different test inputs, e.g., (`x = 0`, `y = -1`), (`x = 0`, `y = 0`), and (`x = -1`, `y = 1`), are required to cover all edges.

- *Nodes.* Statement coverage requires that each program statement is covered by some test case. In this case, it is *not* necessary to cover each edge of a CFA, which would yield branch coverage; for an `if (cond) stmt;` without `else` it suffices to reach the CFA nodes with outgoing edges for `cond` and `stmt`. Hence, to request statement coverage of function `cmp` we use `NODES(@FUNC(cmp))`, which yields the expression $2+3+4+5+6+7+8$. Consequently the corresponding query

  ```
  > cover "ID*" . NODES(@FUNC(cmp)) . "ID*"
  ```

  yields only seven test goals (words). In this case, two pairs of test inputs suffice, e.g., (`x = 0`, `y = -1`) and (`x = -1`, `y = 1`).

- *Paths.* The operator `PATHS(`T`,`k`)` extracts the target graph computed by a filter function T such that no node occurs more than k times. For a



x	y	x > y
true	∗	true
true	∗	false
false	true	true
false	true	false
false	false	∗

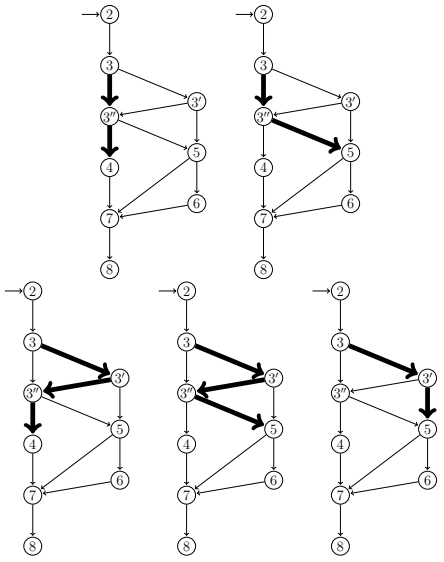


Figure 3.3: Multiple condition coverage of `(x || y) && x > y`

practical example assume we replace the condition `x > y` in line 3 with `(x || y) && x > y` to additionally test for at least one of `x` or `y` to be non-zero. The CFA for the modified function `cmp` is shown in Figure 3.3. To exercise this complex condition with multiple condition coverage we have to test all Boolean combinations of atomic conditions. Owing to short-circuit semantics only five cases remain to be distinguished, as described by the table in Figure 3.3. These five cases exactly correspond to the *paths* of the target graph computed by the filter function `@3`, i.e., the edges corresponding to line 3 of the program. In FQL we use `PATHS(@3, 1)` to describe the *bounded* paths in this target graph, i.e., $(3, 3'', 4) + (3, 3'', 5) + (3, 3', 3'', 4) + (3, 3', 3'', 5) + (3, 3', 5)$, as illustrated with bold edges in the CFAs at the right side of Figure 3.3. The query

```
> cover "ID*" . PATHS(@3, 1) . "ID*"
```

gives rise to five test goals. We require a bound to be specified, which in this case is 1, as cyclic target graphs would otherwise yield an infinite number of paths, and hence an infinite number of test goals.

3.1.5 Full FQL Specifications

General FQL specifications have the form

$$\texttt{in}\ G\ \texttt{cover}\ C\ \texttt{passing}\ P$$

where both `in` G and `passing` P can be omitted.

- The clause '`in` G' states that all filter functions in the `cover` clause are applied to a target graph resulting from first applying the filter function G. In practice, this is often used as

  ```
  in @FUNC(foo) cover EDGES(@DEF(x))
  ```

  which is equivalent to the specification

  ```
  cover EDGES(COMPOSE(@DEF(x),@FUNC(foo)))
  ```

- To restrict testing to a certain area of interest, FQL contains *passing clauses*, i.e., path patterns which *every* test case has to satisfy. For instance, by writing

  ```
  > cover "ID*" . @BASICBLOCKENTRY . "ID*"
    passing ^NOT(@CALL(unimplemented))*$
  ```

  we request basic block coverage with test cases restricted to paths where function `unimplemented` is never called. Such specifications enable testing of unfinished code, where only selected parts will be exercised. Furthermore, we can use passing clauses to specify invariants: Using the query

  ```
  > cover "ID*" . @BASICBLOCKENTRY . "ID*"
    passing ^(ID.{x >= 0})*$
  ```

  we request basic block coverage through a test suite where variable `x` never becomes negative. Note that the passing clause contains only path patterns and does not contain quotes. The symbols '`^`' and '`$`' are explained below.

FQL also contains syntactic sugar to simplify test specifications. In particular, `->` stands for `.ID*.` (or `."ID*".` when used in coverage specifications). Moreover, as stated above, `EDGES` is assumed as default target alphabet constructor. Therefore the above query for not calling function `unimplemented` expands to

```
> cover "EDGES(ID)*" . EDGES(@BASICBLOCKENTRY) . "EDGES(ID)*"
  passing EDGES(NOT(@CALL(unimplemented)))*
```

In addition, `"EDGES(ID)*"` is by default added before and after a coverage specification in the `cover` clause of an FQL query; for the `passing` clause we add the unquoted version:

```
> cover "EDGES(ID)*" . EDGES(@BASICBLOCKENTRY) . "EDGES(ID)*"
  passing EDGES(ID)*.EDGES(NOT(@CALL(unimplemented)))*.EDGES(ID)*
```

To avoid this implicit prefix/suffix being added, Unix `grep` style anchoring using '`^`' and '`$`' can be used. As shown above, this is mainly necessary when required invariants are specified, which have to hold for the entire path, or to ensure that the example function `unimplemented` is *never* called.

3.2 Example Specifications

In the preceding sections we described the framework and basic concepts of FQL. In the following we give a number of practical usage scenarios and resulting FQL queries.

- *Statement coverage.* This standard coverage criterion requires a set of program runs such that every statement in the program is executed at least once. To specify a test suite achieving statement coverage for the entire program at hand we use the FQL query

  ```
  > cover NODES(ID)
  ```

- *Basic block coverage with invariant.* FQL makes it easy to modify standard coverage criteria. Consider for instance basic block coverage with the additional requirement that the variable `errno` should remain zero at all times:

  ```
  > cover @BASICBLOCKENTRY passing ^(ID.{errno==0})*$
  ```

- *Multiple condition coverage in specified scope.* It is often desirable to apply automated test input generation to a restricted scope only. This situation comes in two flavors: First we consider coverage for a certain function only. In FQL we use the query

  ```
  > in @FUNC(foo) cover @CONDITIONEDGE
  ```

  to request condition coverage for decisions in function `foo` only. The second interesting restriction is to avoid execution of parts of the program, e.g., unfinished code. The following query achieves condition coverage and uses the *passing* clause to disallow calls to a function named `unfinished`:

  ```
  > cover @CONDITIONEDGE passing ^NOT(@CALL(unfinished))*$
  ```

  To achieve *multiple* condition coverage, all feasible Boolean combinations of atomic conditions must be covered. This corresponds to all paths in the control flow graphs of the decisions in the program. In FQL, this is expressed as follows:

  ```
  > cover PATHS(@CONDITIONGRAPH, 1)
    passing ^NOT(@CALL(unfinished))*$
  ```

- *Combining coverage criteria.* When full path coverage is not achievable, we can either choose to approximate it, or to restrict it to the most critical program parts and use, e.g., basic block coverage elsewhere. As an approximation of path coverage we suggest covering all pairs (or triples) of basic blocks in the program. This is easily expressed using the following queries

  ```
  > cover @BASICBLOCKENTRY->@BASICBLOCKENTRY
  > cover @BASICBLOCKENTRY->@BASICBLOCKENTRY->@BASICBLOCKENTRY
  ```

  for pairs and triples, respectively. If path coverage is a must, but can be restricted to function `critical`, we use a query

  ```
  > cover PATHS(@FUNC(critical), 3) + @BASICBLOCKENTRY
  ```

  to achieve basic block coverage for the entire program and path coverage with an unwinding bound of 3 in function `critical` only. If necessary, this procedure can be repeated for other important functions.

- *Predicate complete coverage.* Ball suggested predicate complete coverage [Bal04] as a new coverage criterion that subsumes several standard coverage criteria, except for path coverage. Given a set of predicates, e.g., $x \geq 0$ and $y = 0$, we state the query

  ```
  > cover ({x>=0}+{x<0}).({y==0}+{y!=0}).EDGES(ID)
    .({x>=0}+{x<0}).({y==0}+{y!=0})
  ```

  It is not difficult to extend FQL with features to automatically extract lists of predicates.

- *Testing recent changes.* In incremental software development we often want to assess the effects of changes to the software. Assume that in a recent change lines 5, 6, and 7 were modified, and that the code in line 8 calls a function `bar`. We would therefore like to systematically consider the effects of lines 5, 6, and 7 on function `bar`. In FQL this is easily done using the query

  ```
  > cover (@5+@6+@7) -> (@CONDITIONEDGE&@FUNC(bar))
  ```

  which for each of lines 5, 6, and 7 requests condition coverage inside `bar`.

- *Reproducing stack traces.* During program debugging it is easy to obtain a call stack of current execution state. It is, however, a lot harder to reproduce the same call stack to understand the cause of a problem. With FQL, this task is simple. Given a call stack of `foo`, `bar`, `foo` we turn this into a query

  ```
  > cover @CALL(foo) -> @CALL(bar) -> @CALL(foo)
  ```

  Note that this query may be too imprecise if, e.g., `foo` can be left such that `bar` is called outside `foo`. Therefore the query can be refined to

  ```
  > cover @CALL(foo)."NOT(@EXIT(foo))*".@CALL(bar)
    ."NOT(@EXIT(bar))*".@CALL(foo)
  ```

- *Testing according to requirements.* In industrial development processes test cases are often specified on a model rather than the source code. These specifications may be given for instance as a sequence diagram

which describes a series of events. Once these events are translated to code locations, e.g., "call function `foo`", "reach line 42", "call function `bar`", we can use an FQL query

```
> cover @CALL(foo) -> @42 -> @CALL(bar)
```

to express this requirement.

3.3 Disambiguating Specifications using FQL

Assume we are given the following – informal – test suite description:

> *"Cover (1) all calls to* `cmp` *and (2) all conditions inside* `cmp`*."*

To study the differences of possible interpretations of such a description, we use function `cmp` from Listing 3.1 and append as calling function the code shown in Listing 3.2, which performs multiple calls to `cmp` after reading three integers as input.

```
10 int main() {
11     int x, y, z, xy, yz, xz;
12     x = input(); y = input(); z = input();
13     xy = cmp(x, y);
14     yz = cmp(y, z);
15     xz = cmp(x, z);
16     return 0;
17 }
```

Listing 3.2: C source code of function `main` with multiple calls to `cmp`

The added value of FQL lies in the ability to precisely define test goals in a simple yet expressive language. We study several valid interpretations of the above informal description as well-defined FQL queries. For the above test suite request we will use the filter function `@CALL(cmp)` to specify "(1) all calls to `cmp`" (lines 13–15 of Listing 3.2) and `@CONDITIONEDGE & @FUNC(cmp)` for "(2) all conditions inside `cmp`". It remains to refine the meaning of the "and" connecting the two parts of the description. There are two major choices for this combination: either we want to cover the union of all call positions and all condition outcomes and write one of the – equivalent – queries

```
> cover @CALL(cmp) | (@CONDITIONEDGE & @FUNC(cmp))
```

```
> cover @CALL(cmp) + (@CONDITIONEDGE & @FUNC(cmp))
```

or we want to cover all possible *combinations* of calls to `cmp` and subsequent condition outcomes inside `cmp`:

```
> cover @CALL(cmp) -> (@CONDITIONEDGE & @FUNC(cmp))
```

In this query we require a more complex coverage: We want test cases in which all Cartesian combinations of calls to `cmp` and condition outcomes in that function occur in the test suite. To see this, just note that the first two queries give rise to 7 path patterns (three calls plus four condition outcomes), while the third amounts to 12 path patterns (three calls *times* four condition outcomes).

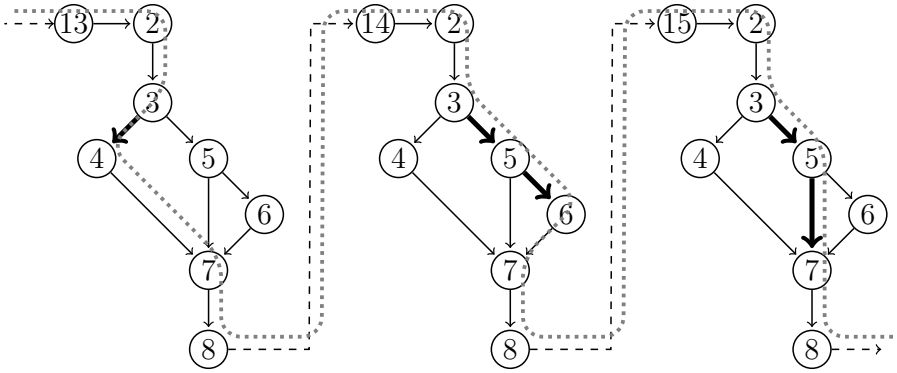
This can be further refined to ensure that *for each call site* each of the condition outcomes is reached before function `cmp` is left. In this case we write

```
> cover @CALL(cmp)."NOT(@EXIT(cmp))*".(@CONDITIONEDGE & @FUNC(cmp))
```

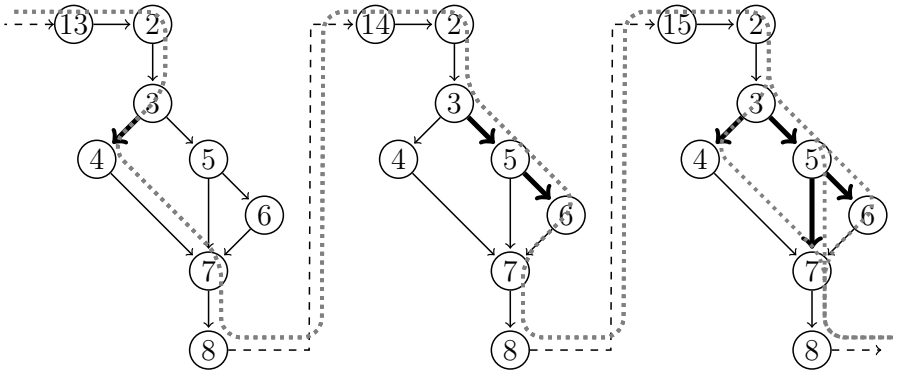
with the filter function `@EXIT(cmp)` describing the return edges of function `cmp`; complementing the corresponding target graph yields all edges not leaving function `cmp`. Further note the implicit `"ID*"` that is prepended and appended: Any of the three calls may be matched by the path patterns that serve as test goals.

Solutions to these queries are test suites. For brevity we denote these as sets of triples of input values for the variables `x`, `y`, and `z`. In Figure 3.4 we illustrate the resulting program runs as dotted gray lines. In this figure we inline the CFA of function `cmp` three times, once for each call site. Bold edges denote covered evaluations of conditions.

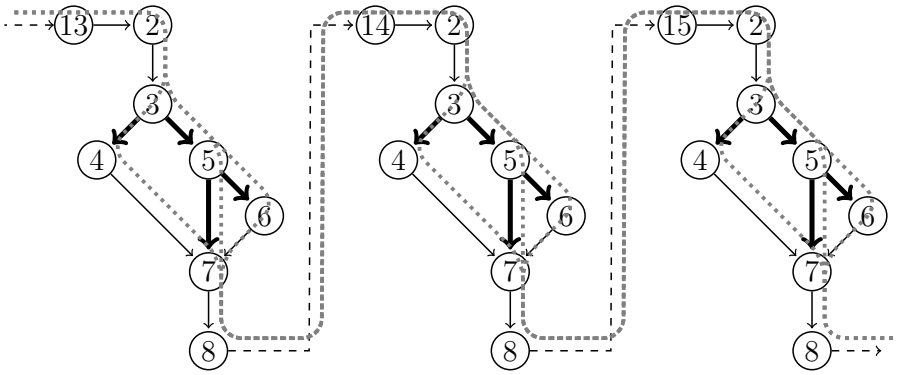
For the first two queries, the singleton test suite $\{(1, 0, 1)\}$ covers all calls and condition outcomes. Figure 3.4(a) shows the resulting program execution. One test case can cover all test goals of this specification since calls to `cmp` are not tied to test goals for condition coverage. For the third and fourth query possible solutions are $\{(1, 0, 1), (2, 0, 1), (1, 0, 2)\}$ (program runs shown in Figure 3.4(b)) and $\{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$ (Figure 3.4(c)) respectively. Note that the first test suite does not satisfy the later two queries and the second suite does not satisfy the last query; the third solution, however, satisfies all three queries.



(a) Test suite $\{(1,0,1)\}$ satisfying
`cover @CALL(cmp) | (@CONDITIONEDGE & @FUNC(cmp))`



(b) Test suite $\{(1,0,1),(2,0,1),(1,0,2)\}$ satisfying
`cover @CALL(cmp) -> (@CONDITIONEDGE & @FUNC(cmp))`



(c) Test suite $\{(1,0,0),(0,1,0),(0,0,1)\}$ satisfying
`cover @CALL(cmp)."NOT(@EXIT(cmp))*".(@CONDITIONEDGE & @FUNC(cmp))`

Figure 3.4: Test suites satisfying the different queries

3.4 Tool Support for Query-Driven Program Testing: FShell

FQL queries are processed by our tool FShell. Our tool supports both interactive and scripted test case generation for real world C code. As back end, FShell uses components of CBMC [CKL04], which enables support for full C syntax and semantics. FShell is available for download in binary form for the most popular platforms at `http://code.forsyte.de/fshell`.

The design of FShell comprises two main parts. The *front end* handles user interaction with a command line interface. There, control commands such as loading source files, macro definitions, and FQL queries are entered by the user. The *back end* performs the actual test case generation. The user of FShell is almost exclusively concerned with the front end. The only exception to this are command line options controlling the operation of the back end. FShell has several additional command line options to control output and operation; the complete list of currently supported options can be obtained by running `fshell --help`. We explain these options in Section 6.3. The complete set of commands accepted by the interactive shell is shown after typing `help`.

```
1 int foo(int x, int y) {
2   if (x > y)
3     return x;
4   else
5     return y + 10;
6 }
```

Listing 3.3: Source code (`bar.c`)

We will now describe the use of FShell on the C program of Listing 3.3. Assume we are interested in a test suite achieving condition coverage. As discussed in Section 3.2, the corresponding FQL query is `cover @CONDITIONEDGE`. For effective test case generation for this coverage criterion we need to relate it to a program. In our case, we want to use function `foo` of source file bar.c.

We therefore start FShell, load the source, set the function, and state the query:

```
$ fshell --verbosity 2 --outfile tests.txt --tco-location
FShell2>
```

FShell is now ready for user interaction. It has been started with status information limited to errors and warnings and test data being written to a file "tests.txt". Details about the command line switches (including the meaning of `--tco-location`) are discussed in Section 6.3. As next step we load the source code:

```
FShell2> add sourcecode 'bar.c'
FShell2>
```

The file name to be loaded is given in single quotes. On Unix systems tab completion can be used to find and complete the file name with less typing. As bar.c does not have a `main` function, we must tell FShell the name of the program's entry function. In our case this is function `foo`:

```
FShell2> set entry foo
FShell2>
```

Now that FShell has all the information to process the program we can pass in the FQL query to start test case generation:

```
FShell2> cover @CONDITIONEDGE
FShell2> quit
```

As we have no other test case requirements in this session, we have immediately left FShell after the test case generation step. The test data has been written to tests.txt, which we will inspect in a minute. Before doing so we want to execute the program with the computed test cases. We will also measure the coverage that is achieved by this test suite using `gcov -b`, which measures branch coverage. Even though `gcov` does not understand FQL syntax and semantics, this coverage check will give us a good idea whether test case generation was for condition coverage successful.

To execute the program on the inputs of the computed test suite we first have to turn the test suite into a test harness. This test harness is a C fragment that calls function `foo` with the computed inputs. We generate the test harness from tests.txt using a script and compile and execute the harness together with the original program (cf. Section 6.10 for details):

```
$ TestEnvGenerator.pl < tests.txt
$ make -f tester.mk BUILD_FLAGS="-fprofile-arcs -ftest-coverage"
./tester 1
./tester 2
$ gcov -b bar.c.mod.c
```

```
File 'bar.c.mod.c'
Lines executed:100.00% of 4
Branches executed:100.00% of 2
Taken at least once:100.00% of 2
No calls
bar.c.mod.c:creating 'bar.c.mod.c.gcov'
```

Coverage measurement using `gcov` confirms that the computed test suite achieved the intended coverage as 100% of branches were taken.

We will now inspect the output of test cases as produced by FShell. We run FShell once again, but omit the command line options controlling test case output:

```
$ fshell --verbosity 2
FShell2> add sourcecode 'bar.c'
FShell2> set entry foo
FShell2> cover @CONDITIONEDGE
Test Suite 0:
IN:
 x=301989888
 y=268435457

IN:
 x=-2013265917
 y=7

FShell2> quit
```

In this second session we can use FShell's command history to repeat the commands of the previous sessions by pressing the cursor-up key. The resulting test suite consists of two test cases, marked with a leading '`IN:`'. Each test case is described as a list of assignments to variables that denote the test inputs. In our case we have `x` and `y` as variables and FShell has computed the inputs `x = 301989888`, `y = 268435457` and `x = -2013265917`, `y = 7`. As FShell employs a SAT solver for computing input values, which has no knowledge about small or even minimal values for inputs, the values computed by FShell are arbitrarily chosen from the domain determined by the variable type but are guaranteed to satisfy the FQL specification.

In tests.txt, where the results of the first session were written to, we find

addition information (because of the `--tco-location` option):

```
Test Suite 0:
IN:
  ENTRY foo(x,y)@[file bar.c line 1]
  int x@[file bar.c line 1]=301989888
  int y@[file bar.c line 1]=268435457

IN:
  ENTRY foo(x,y)@[file bar.c line 1]
  int x@[file bar.c line 1]=-2013265917
  int y@[file bar.c line 1]=7
```

Here, all assignments include the location where an assignment of input values has to take place. In addition, the name of the entry function together with its argument names is printed. This data is necessary for automatic test harness generation, as explained in Section 6.10.

It is impossible to make anything foolproof because fools are so ingenious.

Corollary to Murphy's Law

Chapter 4

A Mathematical Model for White-box Program Testing

In testing, we face the problem of studying the relation between test suite specifications on the one hand, e.g., basic block coverage, which are concerned with structural properties of the program, and program executions on the other hand. The core of query-driven program testing is a mathematical model which allows to reason about this relationship. This model encompasses both a formal description of the program under scrutiny and the mathematical foundations of test suite specifications. Based on these notions, we formalize the notion of coverage criteria. We will therefore first introduce our representation of programs: we use control flow automata as an intermediate language. The semantics of programs represented as control flow automata are given by transition systems.

4.1 Intermediate Representation: Control Flow Automata

We use *control flow automata* (CFA) [HJMS02] as intermediate representation of programs. A representation of programs as CFAs makes programs both easier to handle formally than the program source code and is a representation independent of the programming language. In contrast to control flow *graphs*, control flow *automata* have statements attached to edges instead of nodes. CFAs are more suitable for our modeling than control flow graphs, because (i) they model programs using a small number of different kinds of

```
1 int bar(int x, int y) {
2   int ret = 0;
3   if (x > y)
4     ret = 1;
5   return ret;
6 }
```

```
7 int foo(int l, int a) {
8   int t;
9   while(l > 0) {
10    t = bar(l, a);
11    a += t;
12    --l;
13  }
14  return a;
15 }
```

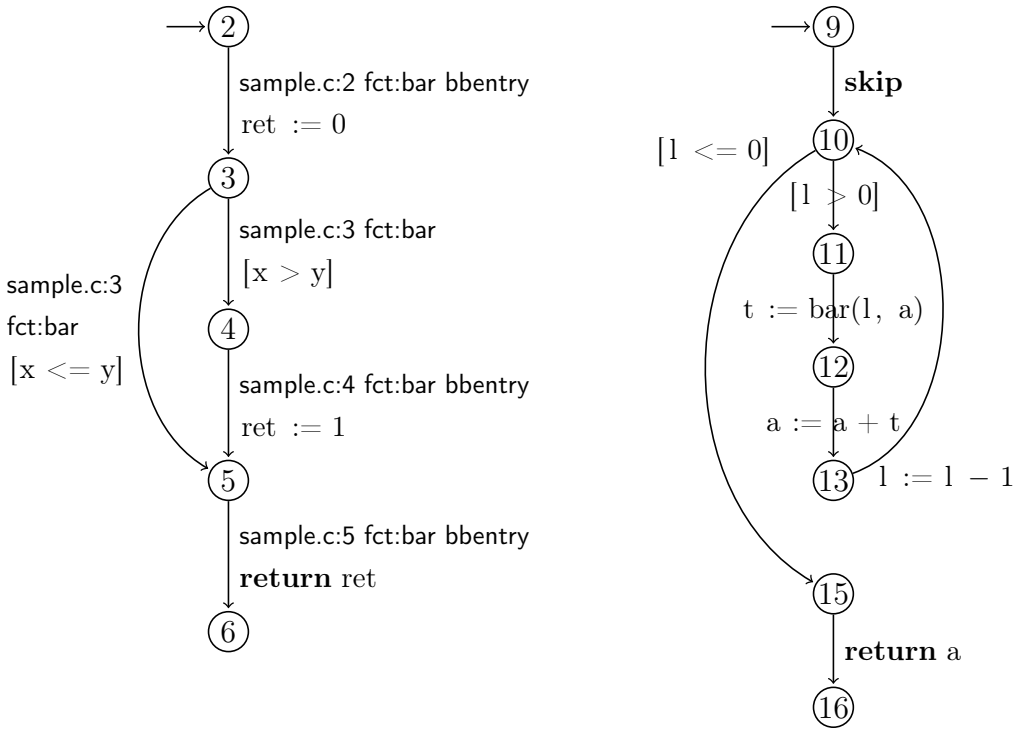


Figure 4.1: Sample C source code and corresponding control flow automata

statements, (ii) CFAs allow to model nondeterminism which can be used in modeling environments, and (iii), as we discuss in detail in Section 4.2, CFAs very naturally induce a transition system. In addition to the original definition of CFAs we annotate edges with parsing information, as in testing we still oftentimes refer to source code, e.g., when specifying coverage of certain lines of code.

CFA nodes represent program counter values drawn from the natural number $\mathbb{N}$; edges are labeled with operations and annotations, drawn from

finite sets Op and An, respectively. An operation op ∈ Op is either a skip statement, assignment, assumption (modeling conditional statements), function call, or function return. Annotations include parsing information such as line numbers, file names, function names, labels, etc. For example, Figure 4.1 shows the C source code of two functions `bar` and `foo` and the corresponding control flow automata involving a skip statement, several assignments, a function call to `bar`, return statements in each function, and assume statements marked with brackets: an `if (x > y)`-statement translates to its true- and false outcomes [x > y] and [x <= y], respectively. In the CFA for function `bar` we furthermore included annotations, shown in sans-serif font. Each statement is annotated with file name and line number, such as sample.c:3 for line 3 of file sample.c, the function name (fct:bar), and bbentry for edges corresponding to a new basic block in the source program.

We note that the sets Op and An are only finite for a fixed given program. The underlying domains may be infinite as these include, e.g., the set of all finite strings for variable names.

Definition 4.1 Control Flow Automaton (CFA)
A control flow automaton *(CFA) is a tuple* $\langle L, E, I\rangle$*, where* $L \subset \mathbb{N}$ *is a finite set of program locations,* $E \subseteq L \times \mathsf{Lab} \times L$ *is a set of edges that are labeled with pairs of operations and annotations from* $\mathsf{Lab} = \mathsf{Op} \times 2^{\mathsf{An}}$*, and* $I \subseteq L$ *is a set of initial locations. We denote the* set of CFAs *with* CFA*.*

We write $L_{\mathcal{A}}$, $E_{\mathcal{A}}$, and $I_{\mathcal{A}}$ to refer to the set of program locations, the set of edges, and the set of initial locations of a CFA $\mathcal{A}$, respectively. We define union '$\cup$', intersection '$\cap$', and difference '$\setminus$' as operations on CFAs:

$$
\begin{aligned}
\langle L_1, E_1, I_1\rangle \cup \langle L_2, E_2, I_2\rangle &= \langle L_1 \cup L_2, E_1 \cup E_2, I_1 \cup I_2\rangle \\
\langle L_1, E_1, I_1\rangle \cap \langle L_2, E_2, I_2\rangle &= \langle L_1 \cap L_2, E_1 \cap E_2, I_1 \cap I_2\rangle \\
\langle L_1, E_1, I_1\rangle \setminus \langle L_2, E_2, I_2\rangle &= \langle L', E', I'\rangle \text{ where} \\
&\qquad E' = E_1 \setminus E_2, \\
&\qquad L' = (L_1 \setminus L_2) \cup \{\ell, \ell' \mid (\ell, l, \ell') \in E'\}, \text{ and} \\
&\qquad I' = I_1 \cap L'.
\end{aligned}
$$

Note that the last condition for difference, $I' = I_1 \cap L'$, ensures that the required condition $I' \subseteq L'$ also holds for the resulting CFA.

4.2 Concrete Program Semantics: Transition Systems

Complementary to the CFA view on the program, which primarily captures the syntactic information of a program, we study the transition system induced by a CFA. The transition system describes the concrete program semantics in terms of reachable states and transitions between them. The states of a transition system are best thought of as snapshots of machine configurations, consisting of (at least) valuations of the program counter and the memory. The domain of states is then defined by the architecture of a – possibly abstract – machine.

Definition 4.2 Transition System
A transition system $\langle \mathcal{S}, \mathcal{R}, \mathcal{I} \rangle$ *consists of a state space* $\mathcal{S}$*, a transition relation* $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$*, and a nonempty set of initial states* $\mathcal{I} \subseteq \mathcal{S}$*. A* state *in* $\mathcal{S}$ *consists of a program counter value and a description of the memory. We denote with* $\mathcal{L}(\mathcal{T})$ *the* set of paths $\pi = \langle s_0 \dots s_m \rangle$ *such that* $s_0 \in \mathcal{I}$ *and* $(s_i, s_{i+1}) \in \mathcal{R}$*, for* $0 \leq i < m$*.*

In order to relate a CFA $\mathcal{A} = \langle L, E, I \rangle$ to a corresponding transition system $\mathcal{T} = \langle \mathcal{S}, \mathcal{R}, \mathcal{I} \rangle$ we fix the following functions:

- We consider the operation $\mathsf{op} \in \mathsf{Op}$ as a function $\mathsf{op} : \mathcal{S} \to 2^{\mathcal{S}}$ that takes a program state and determines a finite set of successor states. This definition caters for the fact that there may be no successor state, as in case of final states, or there can be one or more successor states. The latter case models nondeterminism.

- By $\mathsf{pc} : \mathcal{S} \to L$ we denote the function that, given a program state s, yields its program location $\mathsf{pc}(s)$.

- By $\mathsf{post} : E \times \mathcal{S} \to 2^{\mathcal{S}}$ we denote a function that, given a CFA edge $(\ell, (\mathsf{op}, \mathsf{an}), \ell') \in E$ and a program state s, returns the set $\{s' \mid \mathsf{pc}(s) = \ell, \mathsf{pc}(s') = \ell', s' \in \mathsf{op}(s)\}$.

A CFA $\mathcal{A}$ naturally induces a transition system $\mathcal{T}_{\mathcal{A}}$:

Definition 4.3 Induced Transition System
Given a CFA $\mathcal{A}$*, we define the* induced transition system $\mathcal{T}_{\mathcal{A}} = \langle \mathcal{S}, \mathcal{R}, \mathcal{I} \rangle$ *where* $\mathcal{S}$ *contains all possible program states,* $\mathcal{R} = \{(s, s') \in \mathcal{S} \times \mathcal{S} \mid \exists e \in E_{\mathcal{A}}. s' \in \mathsf{post}(e, s)\}$*, and* $\mathcal{I} = \{s \in \mathcal{S} \mid \mathsf{pc}(s) \in I_{\mathcal{A}}\}$*.*

This framework for defining program semantics is parameterized in the domain of the state space and the semantics of the programming language that is used to interpret $\mathsf{op} \in \mathsf{Op}$. The function post defines, for instantiations of op for a concrete programming language, a forward collecting semantics [DS90].

4.3 Predicates and Coverage Criteria

Let $\mathcal{T} = \langle \mathcal{S}, \mathcal{R}, \mathcal{I} \rangle$ be a transition system. With $\mathcal{S}^\star \supseteq \mathcal{L}(\mathcal{T})$ we denote the set of all sequences of states, including the empty sequence $\langle\rangle$. For a sequence $\pi = \langle s_0 s_1 \dots s_m \rangle \in \mathcal{S}^\star$ and $i \leq j$ we write $\pi^{i \dots j}$ to denote the substring (but not an arbitrary subsequence) $\langle s_i \dots s_j \rangle$. For a state $s \in \mathcal{S}$, we write $s \in \pi$, iff $s = s_i$ holds for some $0 \leq i \leq m$.

We use *state predicates* to describe properties of individual program states and we use *path* and *path set predicates* in the description of individual test targets and coverage criteria. Here, a predicate is a function mapping states, paths, or sets of paths to $\{\mathsf{true}, \mathsf{false}\}$ and is expressed in a suitable logic.

Definition 4.4 State, Path, & Path Set Predicates
A state predicate φ *is a predicate on the state space* $\mathcal{S}$*, a* path predicate ϕ *is a predicate over the set* $\mathcal{S}^\star$*, and a* path set predicate Φ *is a predicate over the set* $2^{\mathcal{S}^\star}$*. We write* $s \models \varphi$ *iff a state* $s \in \mathcal{S}$ *satisfies* φ*,* $\pi \models \phi$ *iff a path* $\pi \in \mathcal{S}^\star$ *satisfies* ϕ*, and* $\Gamma \models \Phi$ *iff a path set* $\Gamma \subseteq \mathcal{S}^\star$ *satisfies* Φ*.*

We call a state predicate φ, a path predicate ϕ, or a path set predicate Φ *feasible over* $\mathcal{T}$, iff, respectively, there exists a reachable state $s \in \mathcal{S}$ with $s \models \varphi$, a path $\pi \in \mathcal{L}(\mathcal{T})$ with $\pi \models \phi$, or a path set $\Gamma \subseteq \mathcal{L}(\mathcal{T})$ with $\Gamma \models \Phi$.

Frequently, we are looking for a path (path set) which *contains* a state (a path) which satisfies a given state (path) predicate – leading to an *implicit existential quantification:*

Definition 4.5 Implicit Existential Quantification
To evaluate a state predicate φ *over a path* π*, we implicitly interpret* φ *to be existentially quantified, i.e.,* $\pi \models \varphi$ *stands for* $\exists s \in \pi . s \models \varphi$*. Analogously, a path predicate* ϕ *is existentially evaluated over a path set* Γ*, i.e.,* $\Gamma \models \phi$ *iff* $\exists \pi \in \Gamma . \pi \models \phi$*.*

Remark 4.6 *Note that a path π can satisfy a state predicate φ* and *its negation $\neg\varphi$, if there exist two states $s, s' \in \pi$ with $s \models \varphi$ and $s' \models \neg\varphi$. Moreover, a state predicate φ can also be interpreted over a path set Γ in the natural way, i.e., $\Gamma \models \varphi$ iff $\exists \pi \in \Gamma . \exists s \in \pi . s \models \varphi$.*

We interpret the Boolean connectives $\wedge$, $\vee$, and $\neg$ on state, path, and path set predicates in the standard way. For path predicates ϕ_1 and ϕ_2 we define predicate concatenation $\phi_1 \cdot \phi_2$ where $\pi \models \phi_1 \cdot \phi_2$ holds iff

$$(\pi^{0\ldots n} \models \phi_1 \text{ and } \pi^{n\ldots|\pi|-1} \models \phi_2 \text{ for } \textit{some } 0 \leq n < |\pi|)$$
$$\textit{or } (\langle\rangle \models \phi_1 \text{ and } \pi \models \phi_2) \textit{ or } (\pi \models \phi_1 \text{ and } \langle\rangle \models \phi_2)$$

holds. Note that the last state of $\pi^{0\ldots n}$ is the first state of $\pi^{n\ldots|\pi|-1}$.

Definition 4.7 Test Case and Test Suite
Let $\mathcal{T}$ be a transition system. Then a test case *is a single path $\pi \in \mathcal{L}(\mathcal{T})$ and a* test suite *Γ is a finite subset $\Gamma \subseteq \mathcal{L}(\mathcal{T})$ of the paths in $\mathcal{L}(\mathcal{T})$.*

A *coverage criterion* imposes a predicate on test suites:

Definition 4.8 Coverage Criterion
A coverage criterion *Φ is a mapping from a CFA $\mathcal{A}$ to a path set predicate $\Phi^{\mathcal{A}}$. We say that $\Gamma \subseteq \mathcal{L}(\mathcal{T}_{\mathcal{A}})$ satisfies coverage criterion Φ on $\mathcal{T}_{\mathcal{A}}$, the transition system induced by $\mathcal{A}$, iff $\Gamma \models \Phi^{\mathcal{A}}$ holds.*

Our definition of coverage criteria is very general, but can readily be applied on a concrete example. We first show exemplary formalizations of coverage criteria for basic block coverage (Φ_{BB}) and condition coverage (Φ_{CC}):

$$\Phi_{BB}(\mathcal{A}) := \{\mathsf{true} \cdot b \cdot \mathsf{true} \mid b \in \mathsf{is\_bbentry}(\mathcal{A})\}$$
$$\Phi_{CC}(\mathcal{A}) := \{\mathsf{true} \cdot c \cdot \mathsf{true} \mid c \in \mathsf{is\_condedge}(\mathcal{A})\}$$

Both definitions describe a set of path predicates, built using predicate concatenation, and functions $\mathsf{is\_bbentry} : \mathsf{CFA} \to \mathsf{CFA}$ and $\mathsf{is\_condedge} : \mathsf{CFA} \to \mathsf{CFA}$, respectively. These functions are evaluated over a CFA to compute a sub-CFA thereof. For $\mathsf{is\_bbentry}$ we get the sub-CFA consisting of all edges annotated with "$\mathsf{bbentry}$". The function $\mathsf{is\_condedge}$ yields the sub-CFA with all assume-edges of the CFA. In Chapter 5 such functions will be formally introduced as *filter functions*.

We can now instantiate these coverage criteria, e.g., for the CFA of function `bar`, which was shown in Figure 4.1. We will refer to this CFA as A_{bar}. Thereby we arrive at path set predicates $\Phi_{BB}^{A_{bar}}$ and $\Phi_{CC}^{A_{bar}}$ for basic block and condition coverage, respectively:

$$\begin{aligned}
\Phi_{BB}^{A_{bar}} \equiv \{&\mathsf{true} \cdot (\ell, l, \ell') \cdot \mathsf{true} \wedge \mathsf{pc}(\ell) \equiv 2 \wedge \mathsf{pc}(\ell') \equiv 3,\\
&\mathsf{true} \cdot (\ell, l, \ell') \cdot \mathsf{true} \wedge \mathsf{pc}(\ell) \equiv 4 \wedge \mathsf{pc}(\ell') \equiv 5,\\
&\mathsf{true} \cdot (\ell, l, \ell') \cdot \mathsf{true} \wedge \mathsf{pc}(\ell) \equiv 5 \wedge \mathsf{pc}(\ell') \equiv 6\}\\
\Phi_{CC}^{A_{bar}} \equiv \{&\mathsf{true} \cdot (\ell, l, \ell') \cdot \mathsf{true} \wedge \mathsf{pc}(\ell) \equiv 3 \wedge \mathsf{pc}(\ell') \equiv 4,\\
&\mathsf{true} \cdot (\ell, l, \ell') \cdot \mathsf{true} \wedge \mathsf{pc}(\ell) \equiv 3 \wedge \mathsf{pc}(\ell') \equiv 5\}
\end{aligned}$$

Like the examples above, most coverage criteria used in practice – and all criteria expressible by FQL – are based on sets of *test goals* which need to be satisfied. Typically, test goals are path predicates, leading to the prototypical setting accounted for in the next definition. A coverage criterion, MC/DC, where test goals are *path set* predicates, is discussed at the end of this chapter.

Definition 4.9 Elementary Coverage Criterion
An elementary coverage criterion Φ *is a coverage criterion defined as follows:*

(i) There is a mapping $\Phi(\mathcal{A}) = \{\Psi_1, \ldots, \Psi_k\}$ *which maps a CFA* $\mathcal{A}$ *to a finite set of path predicates* $\{\Psi_1, \ldots, \Psi_k\}$*. We call* Ψ_i *a* test goal*.*

(ii) $\Phi(\mathcal{A})$ *induces the predicate* $\Phi^{\mathcal{A}}$ *such that* $\Gamma \models \Phi^{\mathcal{A}}$ *holds iff for each test goal* $\Psi_i \in \Phi(\mathcal{A})$ *which is feasible over* $\mathcal{T}_{\mathcal{A}}$*,* Γ *contains a test case* $\pi \in \mathcal{L}(\mathcal{T}_{\mathcal{A}})$ *with* $\pi \models \Psi_i$*:*

$$\Gamma \models \Phi^{\mathcal{A}} \quad \textit{iff} \quad \bigwedge_{i=1}^{k} \mathcal{L}(\mathcal{T}_{\mathcal{A}}) \models \Psi_i \Rightarrow \Gamma \models \Psi_i$$

Intuitively, an elementary coverage criterion Φ is a function that maps a CFA $\mathcal{A}$ to a finite set $\Phi(\mathcal{A})$ of path predicates. A test suite Γ satisfies an elementary coverage criterion Φ on program $\mathcal{A}$, if each path predicate in $\Phi(\mathcal{A})$ is matched by an element of the test suite Γ, except for those path patterns which are semantically impossible in the program (e.g., dead code).

Remark 4.10 *It is semi-decidable whether a test goal Ψ is feasible over $\mathcal{T}_\mathcal{A}$, i.e., whether $\mathcal{L}(\mathcal{T}_\mathcal{A}) \models \Psi$ is valid for a given CFA $\mathcal{A}$ and a path predicate Ψ.*
Proof. If $\mathcal{L}(\mathcal{T}_\mathcal{A})$ is finite, then $\mathcal{L}(\mathcal{T}_\mathcal{A}) \models \Psi$ is decidable by computing the finite disjunction

$$\bigvee_{\pi \in \mathcal{L}(\mathcal{T}_\mathcal{A})} \pi \models \Psi.$$

Conversely, if $\mathcal{L}(\mathcal{T}_\mathcal{A})$ if infinite, then it is recursively enumerable by applying Definition 4.3 as enumerator. Undecidability follows by reduction from the Halting problem with Ψ encoding the input to be accepted. □

Appropriateness of Elementary Coverage Criteria

Definition 4.9 completes our formal framework. However, it remains to study whether these definitions are actually appropriate. We therefore use the axioms described by Zhu and Hall [ZH93] to evaluate which of these axioms are guaranteed to be valid for elementary coverage criteria. As we will conclude in Remark 4.15, an elementary coverage criterion may violate two of these axioms, which is in fact desirable.

Weyuker [Wey86] presented a set of axioms that describe the characteristics of test adequacy criteria. Zhu and Hall [ZH93] formalized these axioms to arrive at a measurement theory for test adequacy. In the following we re-state the ten axioms given by Zhu and Hall and then study whether elementary coverage criteria are consistent with these axioms. The axioms of Zhu and Hall are based on a definition of a measure $M_P^S(C)$ called a *test data adequacy criterion* that, given a specification S, a program P, and a test set C (a test suite given in terms of test inputs), evaluates to a real number in the interval $[0, 1]$.

We give a variant of their definition that omits specifications (properties of the program) as we do not use such specifications in our framework. Furthermore we use CFAs instead of a generic concept of well formed programs and use sequences of states instead of test inputs as a description of test data. Therefore we arrive at the following definition:

Definition 4.11 Test Data Adequacy Criterion [ZH93]
A test data adequacy criterion *M is a mapping from the set of CFAs* CFA *and the powerset of state sequences $2^{\mathcal{S}^\star}$ to the real interval $[0, 1]$:*

$$M : \mathsf{CFA} \times 2^{\mathcal{S}^\star} \to [0, 1].$$

The axioms due to Zhu and Hall [ZH93] are summarized as follows. Note that finite additivity is only a stronger alternative to sub-additivity, hence Zhu and Hall speak of ten axioms only and we will not assign it its own number. For the ease of notation for the rest of this section we assume that $\mathcal{A} \in \mathsf{CFA}$, $\Gamma, \Gamma', \Gamma_1, \Gamma_2 \in 2^{\mathcal{S}^\star}$, and $r \in [0, 1]$.

$$\forall \mathcal{A}, \Gamma . 1 \geq M(\mathcal{A}, \Gamma) \geq 0 \qquad \text{(Normalization)}$$

$$\forall \mathcal{A} . M(\mathcal{A}, \emptyset) = 0 \qquad \text{(Inadequacy of empty test)}$$

$$\forall \mathcal{A} . M(\mathcal{A}, \mathcal{L}(\mathcal{T}_{\mathcal{A}})) = 1 \qquad \text{(Adequacy of exhaustive testing)}$$

$$\forall \mathcal{A}, r . 0 \leq r < 1 \Rightarrow \exists \Gamma . \Gamma \text{ is finite and } M(\mathcal{A}, \Gamma) \geq r \qquad \text{(Finite applicability)}$$

$$\forall \mathcal{A}, \Gamma, \Gamma' . \Gamma \subseteq \Gamma' \Rightarrow M(\mathcal{A}, \Gamma) \leq M(\mathcal{A}, \Gamma') \qquad \text{(Monotonicity)}$$

$$\forall \mathcal{A}, \Gamma_1, \Gamma_2 . M(\mathcal{A}, \Gamma_1 \cup \Gamma_2) \leq M(\mathcal{A}, \Gamma_1) + M(\mathcal{A}, \Gamma_2) \qquad \text{(Sub-additivity)}$$

$$\forall \mathcal{A}, \Gamma_1, \Gamma_2 . \Gamma_1 \cap \Gamma_2 = \emptyset \Rightarrow M(\mathcal{A}, \Gamma_1 \cup \Gamma_2) = M(\mathcal{A}, \Gamma_1) + M(\mathcal{A}, \Gamma_2) \qquad \text{(Finite additivity)}$$

$$\forall \mathcal{A}, \Gamma, \Gamma_1, \Gamma_2 . \Gamma_2 \subseteq \Gamma_1 \Rightarrow M(\mathcal{A}, \Gamma \cup \Gamma_2) - M(\mathcal{A}, \Gamma_2) \geq M(\mathcal{A}, \Gamma \cup \Gamma_1) - M(\mathcal{A}, \Gamma_1) \qquad \text{(Diminishing returns)}$$

$$\forall \mathcal{A}, \Gamma, \Gamma_1, \Gamma_2, \overset{\text{countably}}{\overset{\cdots}{\text{many}}} (\Gamma_1 \subseteq \Gamma_2 \subseteq \ldots \wedge \bigcup_{i=1}^{\infty} \Gamma_i = \Gamma \Rightarrow \lim\{M(\mathcal{A}, \Gamma_i) \mid i = 1, 2, \ldots\} = M(\mathcal{A}, \Gamma)) \qquad \text{(Convergence)}$$

$$\forall \mathcal{A}, \Gamma . M(\mathcal{A}, \Gamma) = M(\mathcal{A}, \Gamma \cap \mathcal{L}(\mathcal{T}_{\mathcal{A}})) \qquad \text{(Relevance)}$$

$$\forall \mathcal{A}, \Gamma . M(\mathcal{A}, \Gamma) = 1 \Rightarrow \Gamma \supseteq \mathcal{L}(\mathcal{T}_{\mathcal{A}}) \qquad \text{(Imperfect testing)}$$

Note that the axioms of Zhu and Hall are parametrized in the definition of the metric M. This metric can be chosen specifically for each coverage criterion. For a general analysis, however, we fix the following coverage measure that applies to all elementary coverage criteria and all CFAs, but uses the semi-decidable property to determine feasibility of test goals.

Definition 4.12 Elementary Coverage Measure
Let $\mathcal{A}$ be a program and let Φ be an elementary coverage criterion with $\Phi(\mathcal{A}) = \{\Psi_1, \ldots, \Psi_k\}$. We define the elementary coverage measure M_E^Φ *as*

$$\begin{array}{rcl} M_E^\Phi : \mathsf{CFA} \times 2^{\mathcal{S}^\star} & \to & [0,1] \\ \mathcal{A}, \Gamma & \mapsto & \dfrac{|\{\Psi \mid \Psi \in \Phi(\mathcal{A}) \wedge (\Gamma \cap \mathcal{L}(\mathcal{T}_\mathcal{A})) \models \Psi\}|}{|\{\Psi \mid \Psi \in \Phi(\mathcal{A}) \wedge \mathcal{L}(\mathcal{T}_\mathcal{A}) \models \Psi\}|} \end{array}$$

The elementary coverage measure M_E^Φ computes the ratio of satisfied test goals over all *feasible* test goals. Note that feasibility is only guaranteed to be decidable if $\mathcal{L}(\mathcal{T}_\mathcal{A})$ is finite, i.e., $\mathcal{A}$ always terminates after finitely many steps. We define the following functions for numerator (the number of covered goals) and denominator (the number of all goals) of M_E^Φ:

$$\mathsf{covered}(\mathcal{A}, \Gamma) := |\{\Psi \mid \Psi \in \Phi(\mathcal{A}) \wedge (\Gamma \cap \mathcal{L}(\mathcal{T}_\mathcal{A})) \models \Psi\}|$$
$$\mathsf{goals}(\mathcal{A}) := |\{\Psi \mid \Psi \in \Phi(\mathcal{A}) \wedge \mathcal{L}(\mathcal{T}_\mathcal{A}) \models \Psi\}|$$

which simplifies M_E^Φ to

$$M_E^\Phi(\mathcal{A}, \Gamma) = \frac{\mathsf{covered}(\mathcal{A}, \Gamma)}{\mathsf{goals}(\mathcal{A})}.$$

In Proposition 4.13 we will now show that there *exists* an elementary coverage criterion that satisfies all of Zhu and Hall's axioms under the (undecidable) assumption of program termination. We prove this claim by giving a concrete example: *exhaustive testing*. In exhaustive testing each feasible path yields one test goal. Hence the definition of elementary coverage criteria does not contradict Zhu and Hall's axioms. Existence of such an elementary coverage criterion, however, does not guarantee that all axioms are valid for *all* elementary coverage criteria, as we will show in Proposition 4.14.

Proposition 4.13 The Elementary Coverage Measure for Exhaustive Testing satisfies Zhu and Hall's Axioms
For all programs $\mathcal{A}$ terminating after finitely many steps, i.e., $\mathcal{L}(\mathcal{T}_\mathcal{A})$ is finite and non-empty, we define exhaustive testing Φ_{exh} *as an elementary coverage criterion*

$$\Phi_{exh}(\mathcal{A}) := \{\Psi \mid \pi, \pi' \in \mathcal{L}(\mathcal{T}_\mathcal{A}) \text{ with } \pi \neq \pi' \Leftrightarrow \pi \models \Psi \wedge \pi' \not\models \Psi\}.$$

The induced elementary coverage measure $M_E^{\Phi_{exh}}$ satisfies Zhu and Hall's axioms.

Proof. We first observe that $M_E^{\Phi_{exh}}$ simplifies to

$$M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma) = \frac{|\Gamma \cap \mathcal{L}(\mathcal{T}_\mathcal{A})|}{|\mathcal{L}(\mathcal{T}_\mathcal{A})|}$$

because each path in $\mathcal{L}(\mathcal{T}_\mathcal{A})$ induces exactly one test goals, and all of which are feasible. Furthermore we have $|\mathcal{L}(\mathcal{T}_\mathcal{A})| > 0$ for all $\mathcal{A}$ by assumption of non-emptiness.

1. Normalization.

$$\forall \mathcal{A}, \Gamma.\, |\Gamma \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| \leq |\mathcal{L}(\mathcal{T}_\mathcal{A})| \Rightarrow \forall \mathcal{A}, \Gamma.1 \geq M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma) \geq 0$$

2. Inadequacy of empty test.

$$\forall \mathcal{A}.\, |\emptyset \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| = 0 \Rightarrow \forall \mathcal{A}. M_E^{\Phi_{exh}}(\mathcal{A}, \emptyset) = 0$$

3. Adequacy of exhaustive testing.

$$\forall \mathcal{A}.\, |\mathcal{L}(\mathcal{T}_\mathcal{A}) \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| = |\mathcal{L}(\mathcal{T}_\mathcal{A})| \Rightarrow \forall \mathcal{A}. M_E^{\Phi_{exh}}(\mathcal{A}, \mathcal{L}(\mathcal{T}_\mathcal{A})) = 1$$

4. Finite applicability. Follows from 2. and 3., and the assumption that $|\mathcal{L}(\mathcal{T}_\mathcal{A})|$ is finite.

5. Monotonicity.

$$\forall \mathcal{A}, \Gamma, \Gamma'. \Gamma \subseteq \Gamma' \Rightarrow |\Gamma \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| \leq |\Gamma' \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| \Rightarrow \\ \forall \mathcal{A}, \Gamma, \Gamma'. \Gamma \subseteq \Gamma' \Rightarrow M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma) \leq M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma')$$

6. Sub-additivity.

$$\forall \mathcal{A}, \Gamma_1, \Gamma_2.\, |(\Gamma_1 \cup \Gamma_2) \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| \leq |\Gamma_1 \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| + |\Gamma_2 \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| \Rightarrow \\ \forall \mathcal{A}, \Gamma_1, \Gamma_2. M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma_1 \cup \Gamma_2) \leq M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma_1) + M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma_2)$$

   Finite additivity.

$$\forall \mathcal{A}, \Gamma_1, \Gamma_2. \Gamma_1 \cap \Gamma_2 = \emptyset \Rightarrow |(\Gamma_1 \cup \Gamma_2) \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| = \\ |\Gamma_1 \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| + |\Gamma_2 \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| \Rightarrow \\ \forall \mathcal{A}, \Gamma_1, \Gamma_2. \Gamma_1 \cap \Gamma_2 = \emptyset \Rightarrow M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma_1 \cup \Gamma_2) = \\ M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma_1) + M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma_2)$$

7. Diminishing returns.

$$\forall \mathcal{A}, \Gamma, \Gamma_1, \Gamma_2.\Gamma_2 \subseteq \Gamma_1 \Rightarrow \Gamma \setminus \Gamma_2 \supseteq \Gamma \setminus \Gamma_1 \Rightarrow M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma \setminus \Gamma_2) \geq M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma \setminus \Gamma_1)$$
$$\wedge \text{ for } i = 1, 2\text{: } \Gamma \cup \Gamma_i = (\Gamma \setminus \Gamma_i) \cup \Gamma_i \wedge (\Gamma \setminus \Gamma_i) \cap \Gamma_i = \emptyset \overset{\text{finite additivity}}{\Leftrightarrow}$$
$$\forall \mathcal{A}, \Gamma, \Gamma_1, \Gamma_2.\Gamma_2 \subseteq \Gamma_1 \Rightarrow M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma \cup \Gamma_2) - M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma_2) \geq M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma \cup \Gamma_1) - M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma_1)$$

8. Convergence.

$$\forall \mathcal{A}, \Gamma.\, |\Gamma \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| \leq |\mathcal{L}(\mathcal{T}_\mathcal{A})| \overset{\text{monotonicity}}{\Rightarrow}$$
$$\forall \mathcal{A}, \Gamma, \Gamma_1, \Gamma_2, \overset{\text{countably many}}{\cdots} (\Gamma_1 \subseteq \Gamma_2 \subseteq \ldots \wedge \bigcup_{i=1}^{\infty} \Gamma_i = \Gamma \Rightarrow \lim\{M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma_i) \mid i = 1, 2, \ldots\} = M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma))$$

9. Relevance.

$$\forall \mathcal{A}, \Gamma.\, |\Gamma \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| = |(\Gamma \cap \mathcal{L}(\mathcal{T}_\mathcal{A})) \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| \Rightarrow \forall \mathcal{A}, \Gamma.M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma) = M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma \cap \mathcal{L}(\mathcal{T}_\mathcal{A}))$$

10. Imperfect testing.

$$\forall \mathcal{A}, \Gamma.M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma) = 1 \Rightarrow |\Gamma \cap \mathcal{L}(\mathcal{T}_\mathcal{A})| = |\mathcal{L}(\mathcal{T}_\mathcal{A})| \Rightarrow \Gamma \supseteq \mathcal{L}(\mathcal{T}_\mathcal{A}) \Rightarrow \forall \mathcal{A}, \Gamma.M_E^{\Phi_{exh}}(\mathcal{A}, \Gamma) = 1 \Rightarrow \Gamma \supseteq \mathcal{L}(\mathcal{T}_\mathcal{A})$$

□

Therefore we conclude that there exists an elementary coverage criterion that satisfies all of Zhu and Hall's axioms. This is, however, not the case for all elementary coverage criteria, because finite additivity and imperfect testing cannot be guaranteed in general. We formalize this in the following proposition. As we discuss in Remark 4.15, the invalidness of these two axioms is in fact desirable – finite additivity would mean that any additional test cases, if satisfying any goals, satisfy additional test goals. Imperfect testing would enforce exhaustive testing as the only acceptable testing method.

Proposition 4.14 Elementary Coverage Criteria and Zhu and Hall's Axioms
Let Φ be an elementary coverage criterion. For all $\mathcal{A} \in \mathsf{CFA}$ where at least one test goal of $\Phi(\mathcal{A}) = \{\Psi_1, \ldots, \Psi_k\}$ is feasible, all of Zhu and Hall's axioms other than finite additivity and imperfect testing are valid for the above defined elementary coverage measure M_E^Φ.
Proof. By the (undecidable) assumption of at least one feasible test goal we have $\mathsf{goals}(\mathcal{A}) \geq 1$ and therefore M_E^Φ is well-defined.

1. Normalization.

$$\forall \mathcal{A}, \Gamma.(\Gamma \cap \mathcal{L}(\mathcal{T}_\mathcal{A})) \subseteq \mathcal{L}(\mathcal{T}_\mathcal{A}) \Rightarrow \\ \mathsf{covered}(\mathcal{A}, \Gamma) \leq \mathsf{goals}(\mathcal{A}) \Rightarrow \forall \mathcal{A}, \Gamma.1 \geq M_E^\Phi(\mathcal{A}, \Gamma) \geq 0$$

2. Inadequacy of empty test.

$$\forall \mathcal{A}.\mathsf{covered}(\mathcal{A}, \emptyset) = 0 \Rightarrow \forall \mathcal{A}.M_E^\Phi(\mathcal{A}, \emptyset) = 0$$

3. Adequacy of exhaustive testing.

$$\forall \mathcal{A}.\mathsf{covered}(\mathcal{A}, \mathcal{L}(\mathcal{T}_\mathcal{A})) = \mathsf{goals}(\mathcal{A}) \Rightarrow \forall \mathcal{A}.M_E^\Phi(\mathcal{A}, \mathcal{L}(\mathcal{T}_\mathcal{A})) = 1$$

4. Finite applicability. Follows from 2. and 3. with $\mathsf{goals}(\mathcal{A}) \leq k$ for $\Phi(\mathcal{A}) = \{\Psi_1, \ldots, \Psi_k\}$.

5. Monotonicity.

$$\forall \mathcal{A}, \Gamma, \Gamma'.\Gamma \subseteq \Gamma' \Rightarrow \mathsf{covered}(\mathcal{A}, \Gamma) \leq \mathsf{covered}(\mathcal{A}, \Gamma') \Rightarrow \\ \forall \mathcal{A}, \Gamma, \Gamma'.\Gamma \subseteq \Gamma' \Rightarrow M_E^\Phi(\mathcal{A}, \Gamma) \leq M_E^\Phi(\mathcal{A}, \Gamma')$$

6. Sub-additivity.

$$\forall \mathcal{A}, \Gamma_1, \Gamma_2.\mathsf{covered}(\mathcal{A}, \Gamma_1 \cup \Gamma_2) \leq \mathsf{covered}(\mathcal{A}, \Gamma_1) + \mathsf{covered}(\mathcal{A}, \Gamma_2) \Rightarrow \\ \forall \mathcal{A}, \Gamma_1, \Gamma_2.M_E^\Phi(\mathcal{A}, \Gamma_1 \cup \Gamma_2) \leq M_E^\Phi(\mathcal{A}, \Gamma_1) + M_E^\Phi(\mathcal{A}, \Gamma_2)$$

Finite additivity, however, does not hold in general: two paths $\pi, \pi' \in \mathcal{L}(\mathcal{T}_\mathcal{A})$ with $\pi \neq \pi'$ (and hence $\{\pi\} \cap \{\pi'\} = \emptyset$) could still satisfy the same test goal.

7. Diminishing returns. We define

$$\begin{aligned}\mathsf{common}(\Gamma,\Gamma') :=& \{\Psi \mid \Psi \in \Phi(\mathcal{A}) \wedge (\Gamma \cap \mathcal{L}(\mathcal{T}_\mathcal{A})) \models \Psi\} \cap \\ & \{\Psi \mid \Psi \in \Phi(\mathcal{A}) \wedge (\Gamma' \cap \mathcal{L}(\mathcal{T}_\mathcal{A})) \models \Psi\}\end{aligned}$$

to denote the set of test goals satisfied both by test suite Γ and by test suite Γ'. With sub-additivity we have for $i = 1, 2$:

$$M_E^\Phi(\mathcal{A}, \Gamma \cup \Gamma_i) = M_E^\Phi(\mathcal{A}, \Gamma) + M_E^\Phi(\mathcal{A}, \Gamma_i) - \frac{|\mathsf{common}(\Gamma, \Gamma_i)|}{\mathsf{goals}(\mathcal{A})}. \quad (4.1)$$

We apply this as follows:

$$\begin{aligned}&\forall \mathcal{A}, \Gamma, \Gamma_1, \Gamma_2.\Gamma_2 \subseteq \Gamma_1 \Rightarrow |\mathsf{common}(\Gamma, \Gamma_2)| \leq |\mathsf{common}(\Gamma, \Gamma_1)| \Leftrightarrow \\ &\qquad \forall \mathcal{A}, \Gamma, \Gamma_1, \Gamma_2.\Gamma_2 \subseteq \Gamma_1 \Rightarrow \\ &M_E^\Phi(\mathcal{A}, \Gamma) + M_E^\Phi(\mathcal{A}, \Gamma_2) - \frac{|\mathsf{common}(\Gamma, \Gamma_2)|}{\mathsf{goals}(\mathcal{A})} - M_E^\Phi(\mathcal{A}, \Gamma_2) \geq \\ &M_E^\Phi(\mathcal{A}, \Gamma) + M_E^\Phi(\mathcal{A}, \Gamma_1) - \frac{|\mathsf{common}(\Gamma, \Gamma_1)|}{\mathsf{goals}(\mathcal{A})} - M_E^\Phi(\mathcal{A}, \Gamma_1) \overset{\text{Equation (4.1)}}{\Leftrightarrow} \\ &\qquad \forall \mathcal{A}, \Gamma, \Gamma_1, \Gamma_2.\Gamma_2 \subseteq \Gamma_1 \Rightarrow M_E^\Phi(\mathcal{A}, \Gamma \cup \Gamma_2) - M_E^\Phi(\mathcal{A}, \Gamma_2) \geq \\ &\qquad\qquad M_E^\Phi(\mathcal{A}, \Gamma \cup \Gamma_1) - M_E^\Phi(\mathcal{A}, \Gamma_1)\end{aligned}$$

8. Convergence.

$$\begin{aligned}&\forall \mathcal{A}, \Gamma.\mathsf{covered}(\mathcal{A}, \Gamma) \leq \mathsf{covered}(\mathcal{A}, \mathcal{L}(\mathcal{T}_\mathcal{A})) \overset{\text{monotonicity}}{\Rightarrow} \\ &\qquad \forall \mathcal{A}, \Gamma, \Gamma_1, \Gamma_2, \overset{\text{countably many}}{\cdots} (\Gamma_1 \subseteq \Gamma_2 \subseteq \ldots \wedge \bigcup_{i=1}^{\infty} \Gamma_i = \Gamma \Rightarrow \\ &\qquad\qquad \lim\{M_E^\Phi(\mathcal{A}, \Gamma_i) \mid i = 1, 2, \ldots\} = M_E^\Phi(\mathcal{A}, \Gamma))\end{aligned}$$

9. Relevance.

$$\begin{aligned}&\forall \mathcal{A}, \Gamma.\mathsf{covered}(\mathcal{A}, \Gamma) = \mathsf{covered}(\mathcal{A}, \Gamma \cap \mathcal{L}(\mathcal{T}_\mathcal{A})) \Rightarrow \\ &\qquad\qquad \forall \mathcal{A}, \Gamma.M_E^\Phi(\mathcal{A}, \Gamma) = M_E^\Phi(\mathcal{A}, \Gamma \cap \mathcal{L}(\mathcal{T}_\mathcal{A}))\end{aligned}$$

10. Imperfect testing does not hold for any $\Phi(\mathcal{A}) = \{\Psi_1, \ldots, \Psi_k\} \subsetneq \mathcal{L}(\mathcal{T}_\mathcal{A})$.

□

Remark 4.15 *The result that precisely* finite additivity *and* imperfect testing *are not valid for elementary coverage criteria in general confirms that our definition is appropriate: if finite additivity were valid, each test goal could be satisfied by exactly one path $\pi \in \mathcal{L}(\mathcal{T}_\mathcal{A})$ only.*

The validity of imperfect testing would contradict the use of coverage criteria as stopping rules: for full coverage, testing would have to continue until all paths have been explored.

Although elementary coverage criteria suffice to model a large class of coverage criteria there exist standard coverage criteria that do not fit this definition. The best known coverage criterion of this kind is modified condition/decision coverage (MC/DC) [RTC92]. MC/DC, by definition, includes statement, condition, and decision coverage. Furthermore the definition of MC/DC given in [RTC92] states:

> ... and each condition in a decision has been shown to affect the decision's outcome independently. A condition is shown to affect a decision's outcome independently by varying just that condition while holding fixed all other possible conditions.

The crucial aspect here is that pairs of test cases are required to demonstrate the effect of varying a single condition, because the condition valuations contradict each other and therefore cannot both occur in a single traversal of the decision. This property of MC/DC has been formally modeled by Vilkomir and Bowen [VB08] using Z notation. Besides MC/DC also coverage criteria for semantic dependence, as defined by Podgurski and Clarke [PC90], require pairs of test cases to cover a single testing target. Pairs of test cases, however, cannot be described using path predicates.

Lemma 4.16 *A pair of test cases cannot be specified using a single path predicate.*

Proof. By Definition 4.4 path predicates are evaluated over a single path. For sets (and therefore also pairs) of paths, existential quantification is introduced (Definition 4.5). Let ϕ be a path predicate and let Γ be a set of paths.

$$\Gamma \models \phi \Leftrightarrow \exists \pi \in \Gamma . \pi \models \phi$$

We show that no interpretation of ϕ can distinguish whether Γ is a singleton set or contains two or more paths:

$$\Gamma = \{\pi, \pi'\} \models \phi \Rightarrow \pi \models \phi \vee \pi' \models \phi$$

W.l.o.g. assume that π satisfies ϕ. Therefore it holds that

$$\Gamma' \models \phi \text{ with } \Gamma' = \{\pi\} \subsetneq \Gamma.$$

□

We conclude that these coverage criteria are not elementary coverage criteria, and in particular MC/DC is not an elementary coverage criterion.

Proposition 4.17 *MC/DC is not an elementary coverage criterion.*
Proof. To achieve coverage, Vilkomir and Bowen [VB08] state that MC/DC requires pairs of test cases for each condition. With Lemma 4.16 it follows that MC/DC therefore requires a coverage criterion constructed as a set of *path set predicates*. Therefore MC/DC matches Definition 4.8 (Coverage Criterion), but not Definition 4.9 (Elementary Coverage Criterion). □

Life would be so much easier if we could just see the source code.

Unknown

Chapter 5

FQL – the FShell Query Language

In this chapter we present our test specification language FQL, the FShell Query Language. We designed FQL as a tool for programmers and test engineers working with ANSI C software. In FQL, the engineer writes declarative specifications of tests, which are subsequently solved by an appropriate back end. The declarative style fully decouples specifications from algorithmic issues of the back end. Hence, in this chapter we solely focus on the syntax and semantics of FQL, and describe how they map to the mathematical model of Chapter 4. The description of algorithmic solving strategies for FQL queries will be given in Chapter 6.

We conclude this chapter with a discussion of the requirements described in Chapter 2 and how FQL addresses these. This includes a list of 24 queries that express the specifications **Q1-24** in FQL.

5.1 FQL Design Overview

Technically, FQL consists of two languages:

(1) The core of FQL are *elementary coverage patterns* (ECPs), i.e., quoted regular expressions whose alphabet are nodes, edges and conditions of a concrete CFA. Referring to low level CFA details, ECPs are *not* intended to be written by human engineers, but rather the formal centerpiece for a precise semantics and implementation.

(2) FQL specifications are very similar to ECPs, but do not refer to CFA details. Instead, they use filter functions to refer to program elements.

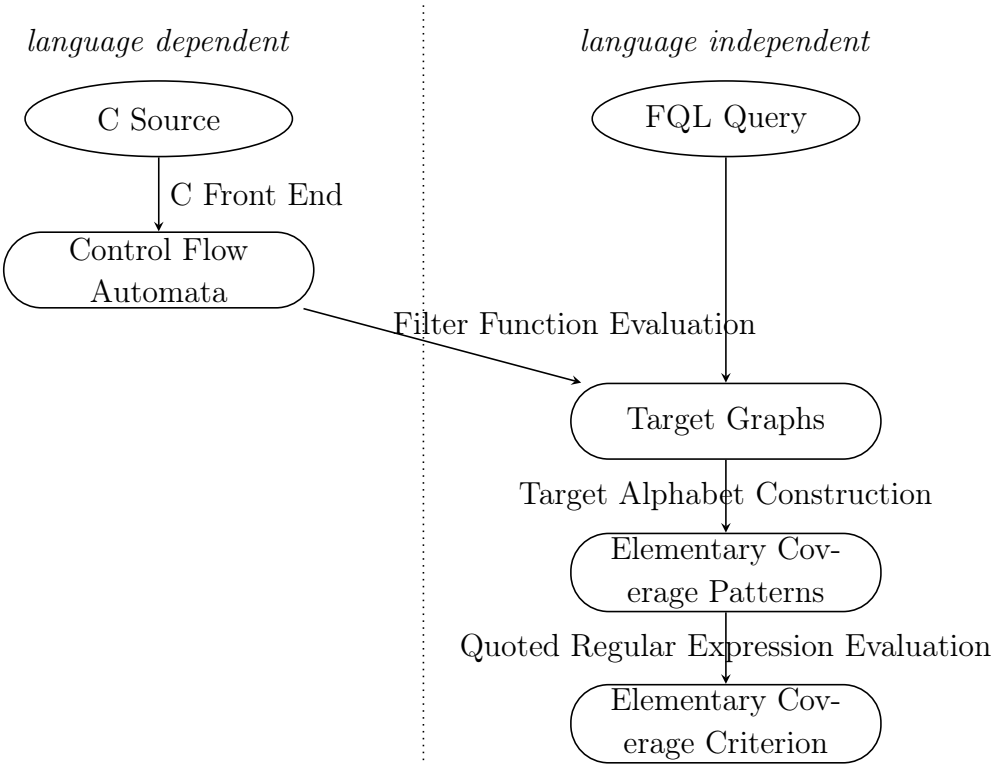


Figure 5.1: FQL language layers

For a given program, an FQL *specification can be easily translated into an ECP by parsing the program and "expanding" the target graphs, computed from filter functions, into regular expressions over the CFA alphabet. Semantically, each* FQL *specification boils down to an elementary coverage criterion.*

The relation of the language layers briefly described above is depicted in Figure 5.1. Given the C source code of a program, a front end constructs CFAs as an intermediate representation of the source code. These are used in the only programming language dependent evaluation step of an FQL query, the filter function evaluation. All further evaluation steps down to ECPs are language independent. In the following sections we describe these layers in a bottom-up fashion, starting with elementary coverage patterns as the core of FQL.

5.2 FQL Elementary Coverage Patterns

Table 5.1 shows the syntax of elementary coverage patterns. The nonterminal symbols P, C, and Φ represent path patterns, coverage specifications, and ECPs, respectively. An elementary coverage pattern `cover` C `passing` P is composed of a coverage specification C and a path pattern P. The alphabets E and L depend on the program under scrutiny: L is a finite set of CFA locations and E is a finite set of CFA edges. The symbols in S are state predicates, e.g., $\{x > 10\}$. By ε we denote the empty word and $\emptyset$ denotes the empty set. We form more complex path patterns over the alphabet symbols using standard regular expression operations. We denote union with '+', concatenation with '.', and Kleene star with '*'.

$$\begin{aligned}
\Phi &::= \texttt{cover}\ C\ \texttt{passing}\ P \\
C &::= C + C \mid C.C \mid \varepsilon \mid \emptyset \mid L \mid E \mid S \mid \texttt{"}P\texttt{"} \\
P &::= P + P \mid P.P \mid \varepsilon \mid \emptyset \mid L \mid E \mid S \mid P^\star
\end{aligned}$$

Table 5.1: Syntax of elementary coverage patterns

A coverage specification is a star-free regular expression over an extended alphabet: In addition to the alphabets L, E and S, we use new symbols introduced using the *quote operator*: Each expression "P", where P is a path pattern, introduces a *single new symbol* "P" in the alphabet of coverage specifications. As informally introduced in Section 3.1.2, the quote operator blocks expansion of the path pattern, which itself is a regular expression. Each quoted path pattern P therefore adds a literal "P" to the alphabet.

We note that the use of regular expressions to describe path patterns is not a limitation of the underlying framework (see Chapter 4), but a deliberate choice to foster usability for the working programmer. Context free properties such as matching lock/unlock calls, however, can therefore currently not be expressed in FQL. Future extensions of FQL may well include more powerful path pattern expressions: the quote operator serves as tokenizer that enables fully distinct treatment of coverage specifications possibly including quoted subexpressions on the one hand and path patterns on the other hand. Therefore we give a separate description of the semantics of path patterns and coverage specifications even for the common operators.

5.2.1 Semantics of Elementary Coverage Patterns

Table 5.2 defines the semantics of path patterns and coverage specifications as formal languages over alphabets of program locations, state predicates, program transitions, and symbols newly introduced by the quote operator. By $\mathcal{L}(P)$ and $\mathcal{L}(C)$ we denote the language of a path pattern P and a coverage specification C, respectively. Except for the newly introduced quote operator, all equations follow standard regular expression semantics. The case of Kleene star $\mathcal{L}(P^\star)$ is only relevant for path patterns, and $\mathcal{L}(\texttt{"}P\texttt{"})$ only appears as part of coverage specifications.

The expression $\texttt{"}P\texttt{"}$ introduces $\texttt{"}P\texttt{"}$ as a new symbol and, thus, $\mathcal{L}(\texttt{"}P\texttt{"})$ results in the singleton set $\{\texttt{"}P\texttt{"}\}$. For example, $\mathcal{L}((\texttt{"}a+b\texttt{"}+\texttt{"}c^\star\texttt{"}).\texttt{"}ac\texttt{"})$ is the set $\{\texttt{"}a+b\texttt{""}ac\texttt{"}, \texttt{"}c^\star\texttt{""}ac\texttt{"}\}$. We discuss the last line of Table 5.2 in the following section.

$\mathcal{L}(P_1 + P_2)$	$= \mathcal{L}(P_1) \cup \mathcal{L}(P_2)$
$\mathcal{L}(C_1 + C_2)$	$= \mathcal{L}(C_1) \cup \mathcal{L}(C_2)$
$\mathcal{L}(P_1.P_2)$	$= \{w_1 w_2 \mid w_1 \in \mathcal{L}(P_1), w_2 \in \mathcal{L}(P_2)\}$
$\mathcal{L}(C_1.C_2)$	$= \{w_1 w_2 \mid w_1 \in \mathcal{L}(C_1), w_2 \in \mathcal{L}(C_2)\}$
$\mathcal{L}(\varepsilon)$	$= \{\varepsilon\}$
$\mathcal{L}(\emptyset)$	$= \emptyset$
$\mathcal{L}(\ell)$	$= \{\ell\}$ where $\ell \in L$
$\mathcal{L}(\varphi)$	$= \{\varphi\}$ where $\varphi \in S$
$\mathcal{L}(e)$	$= \{e\}$ where $e \in E$
$\mathcal{L}(P^\star)$	$= \mathcal{L}(P)^\star$
$\mathcal{L}(\texttt{"}P\texttt{"})$	$= \{\texttt{"}P\texttt{"}\}$
$\mathcal{L}(\texttt{cover}\ C\ \texttt{passing}\ P)$	$= \{w \wedge \texttt{"}P\texttt{"} \mid w \in \mathcal{L}(C)\}$

Table 5.2: Semantics of FQL elementary coverage patterns

5.2.2 Interpretation of Path Patterns as Path Predicates

Given a coverage specification C or path pattern P, we interpret each $w \in \mathcal{L}(C)$ or $w \in \mathcal{L}(P)$ as a path predicate (cf. Definition 4.4). We write $\pi \models w$ iff π satisfies the word w and define the semantics of $\pi \models w$ in Table 5.3 inductively over the structure of w:

$\pi \models \emptyset$	**iff** false
$\pi \models \varepsilon$	**iff** π is the empty sequence $\langle\rangle$
$\pi \models \ell$	**iff** π has the form $\langle s \rangle$ and $\mathsf{pc}(s) = \ell$
$\pi \models \varphi$	**iff** π has the form $\langle s \rangle$ and $s \models \varphi$
$\pi \models e$	**iff** π has the form $\langle ss' \rangle$ and $s' \in \mathsf{post}(e, s)$
$\pi \models \texttt{"}P\texttt{"}$	**iff** there is a $w \in \mathcal{L}(P)$ such that $\pi \models w$
$\pi \models w$	**iff** $\pi \models a \cdot w'$ with $w = aw'$ and $a \in L \cup E \cup S$
$\pi \models w$	**iff** $\pi \models \texttt{"}P\texttt{"} \cdot w'$ with $w = \texttt{"}P\texttt{"}w'$

Table 5.3: Interpretation of path patterns as path predicates

The empty set is unsatisfiable and the empty word ε matches the empty sequence $\langle\rangle$ only. A program counter value ℓ or a state constraint φ matches a singleton state only. A transition e matches a pair $\langle ss' \rangle$ of states, where the semantics of the operation involved in the transition determines whether s' is a concrete successor of s (see Section 4.2).

The coverage specification $\texttt{"}P\texttt{"}$ is satisfied by a path π, iff there is a word $w \in \mathcal{L}(P)$ that is satisfied by π. The case $\pi \models aw$ amounts to predicate concatenation as defined in Section 4.3.

Applying these definitions, an ECP `cover` C `passing` P combines a coverage specification C and a path pattern P to obtain a set of path predicates as defined in the last line of Table 5.2:

$$\mathcal{L}(\texttt{cover}\ C\ \texttt{passing}\ P) = \{w \wedge \texttt{"}P\texttt{"} \mid w \in \mathcal{L}(C)\}.$$

5.3 Target Graphs and CFA Transformers

Target graphs enable the user to directly access natural program entities such as basic blocks, line numbers, decisions etc. without referring to nodes or edges of the CFA. Formally, a target graph is a fragment of a control flow automaton and typically contains those parts of the source code that are relevant for a given testing target.

Definition 5.1 CFA Transformer & Target Graph
A CFA transformer *is a function* $T : \mathsf{CFA} \to \mathsf{CFA}$ *which, on input of a CFA* $\mathcal{A} = \langle L, E, I \rangle$*, computes a* target graph $T[\mathcal{A}] = \langle L', E', I' \rangle$.

The result of applying a CFA transformer T to a CFA $\mathcal{A}$ is denoted by $T[\mathcal{A}]$. A CFA transformer T is either a *filter function* F (cf. Definition 5.2), function composition, or a set-theoretic operation on target graphs. As shown in Table 5.4, FQL has operators that induce these CFA transformers: the operator `COMPOSE` takes as arguments operators T_1 and T_2 to define function composition for any instantiation of T_1 and T_2 as CFA transformers. Analogously '`|`', '`&`', and `SETMINUS` are operators to define set union, set intersection and set subtraction, respectively. Again, these operators take two arguments such that, upon instantiation with CFA transformers T_1 and T_2, a CFA transformer is induced as defined in Table 5.4.

$\texttt{COMPOSE}(T_1, T_2)[\mathcal{A}]$	$= T_1[T_2[\mathcal{A}]]$
$(T_1 \texttt{\|} T_2)[\mathcal{A}]$	$= T_1[\mathcal{A}] \cup T_2[\mathcal{A}]$
$(T_1 \texttt{\&} T_2)[\mathcal{A}]$	$= T_1[\mathcal{A}] \cap T_2[\mathcal{A}]$
$\texttt{SETMINUS}(T_1, T_2)[\mathcal{A}]$	$= T_1[\mathcal{A}] \setminus T_2[\mathcal{A}]$
$\texttt{ID}[\mathcal{A}]$	$= \mathcal{A}$

Table 5.4: Operators inducing CFA transformers

The most important CFA transformers are *filter functions*, which extract a subset of the edges of a CFA. Hence the CFA transformer `ID`, which is the identity function on CFAs as defined in Table 5.4, is also a filter function. We will first formally define filter functions and then discuss them in detail.

Definition 5.2 Filter Functions on CFAs
A filter function *is a CFA transformer* $F : \mathsf{CFA} \to \mathsf{CFA}$ *which computes for every CFA* $\mathcal{A} = \langle L, E, I \rangle$ *a target graph* $F[\mathcal{A}] = \langle L', E', I' \rangle$ *with* $L' \subseteq L$, $E' \subseteq E$, *and* $I' \subseteq L'$, *such that* $E' \subseteq L' \times \mathsf{Lab} \times L'$ *holds.*

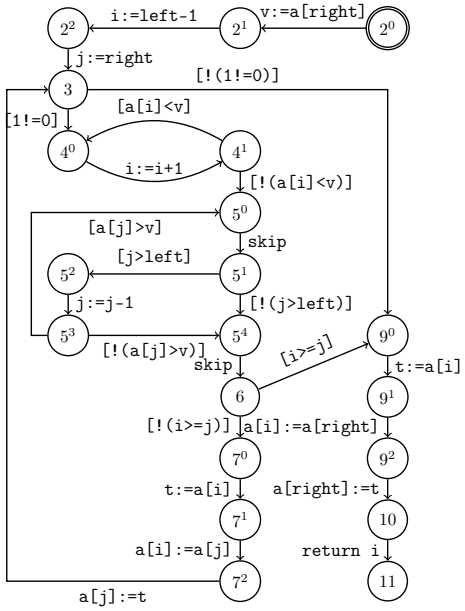
Before defining the various concrete filter functions supported by FQL we exemplify Definition 5.2 on the filter functions `@BASICBLOCKENTRY` and `@CONDITIONGRAPH`. The target graph `@BASICBLOCKENTRY`$[\mathcal{A}]$ contains the edges necessary for basic block coverage on $\mathcal{A}$. The filter function `@CONDITIONGRAPH` extracts the portions of $\mathcal{A}$ that are related to decisions.

```
1 int partition(int a[], int left, int right) {
2   int v = a[right], i = left - 1, j = right, t;
3   for (;;) {
4     while (a[++i] < v) ;
5     while (j > left && a[--j] > v) ;
6     if (i >= j) break;
7     t = a[i]; a[i] = a[j]; a[j] = t;
8   }
9   t = a[i]; a[i] = a[right]; a[right] = t;
10  return i;
11 }
```

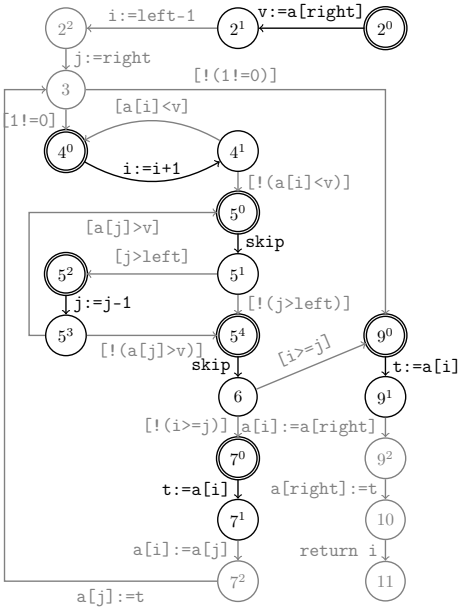
Listing 5.1: Example source code (`sort.c`)

For example, consider Figure 5.2(a), which shows the CFA for the code in Listing 5.1. The target graph `@BASICBLOCKENTRY`$[\mathcal{A}]$ depicted in Figure 5.2(b) (edges not contained in the target graph are grayed out) is obtained by applying the filter function `@BASICBLOCKENTRY` to $\mathcal{A}$. For `@CONDITIONGRAPH` we get the parts of the CFA $\mathcal{A}$ related to decisions in Listing 5.1, see Figure 5.2(c).

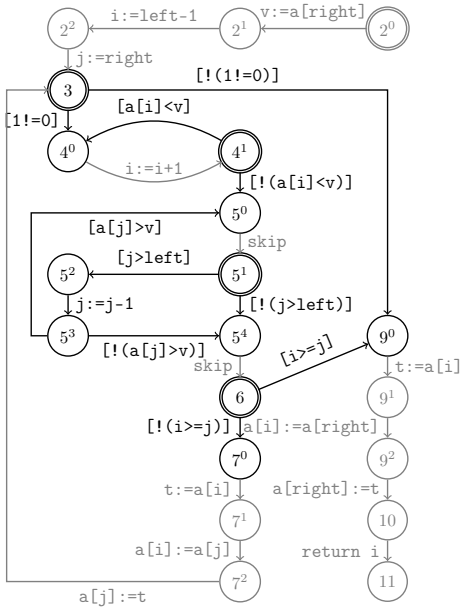
In Definition 5.2, the condition $I' \subseteq L'$ enables a filter function to change the set of initial locations. For instance, `@BASICBLOCKENTRY`$[\mathcal{A}]$, as shown in Figure 5.2(b), sets the initial locations (indicated by double circles) to the start locations of the edges in the target graph. In the definitions of the various filter functions below, we will therefore give the set of initial locations for each filter function. The significance of initial locations will be visible once we define the `PATHS` operator in Section 5.5, because paths only start in initial locations.



(a) Control flow automaton $\mathcal{A}$



(b) Target graph for `@BASICBLOCKENTRY`[$\mathcal{A}$]



(c) Target graph for `@CONDITIONGRAPH`[$\mathcal{A}$]

Figure 5.2: Control flow automaton of `partition` (Listing 5.1) and target graphs

5.4 Filter Functions for ANSI C

Filter functions encapsulate the interface to the programming language. Thus future extensions of FQL may provide both additional filter functions for C programs, if requested by test engineers, and filter functions suitable for different programming languages *without otherwise affecting the design of* FQL . Therefore we use the prefix '`@`' to ensure that filter function names do not collide with other FQL keywords even with future extensions of FQL.

Table 5.5 gives an overview of the filter functions currently supported in FQL. The exact definitions of supported filter functions are specific to ANSI C [Ame99a], hence we first establish according terminology, which is specified in detail below. A detailed specification of filter functions is then given in Section 5.4.2.

`@BASICBLOCKENTRY`	one edge per basic block
`@CONDITIONEDGE`	one edge per (atomic) condition outcome
`@DECISIONEDGE`	one edge per decision outcome (`if`, `for`, `while`, `switch`, `?:`)
`@CONDITIONGRAPH`	all edges contributing to decisions
`@FILE(`a`)`	all edges in file a
`@LINE(`x`)`	all edges in source line x
`@FUNC(`f`)`	all edges in function f
`@STMTTYPE(`*types*`)`	all edges within statements *types*
`@DEF(`t`)`	all assignments to variable t
`@USE(`t`)`	all right hand side uses of variable t
`@CALL(`f`)`	all call sites of f
`@ENTRY(`f`)`	entry edge of f
`@EXIT(`f`)`	all exit edges of f

Table 5.5: Filter functions in FQL – informal overview

The implementation of filter functions, however, is not based on C source code but rather on CFAs. To extract appropriate CFA edges we rely on annotations added to a CFA while parsing the source code. For example, we annotate the first edge of each basic block as "basic block entry edge", bbentry, as described in Section 4.1. Besides such annotations describing code structure we also add basic parsing information such as source file names and line numbers.

5.4.1 ANSI C Specific Terminology

Source Files and Lines. Whenever referring to C source code, we speak of *preprocessed* source text. That is, `#include`, `#define`, etc. have been expanded and replaced as defined by the ANSI C standard. Preprocessing can result in new `#line` directives, which provide information about the original source text file and location, which would otherwise be lost after processing `#include` directives. These `#line` markers are processed by the C front end and file and line number annotations in the CFA are adjusted accordingly, as defined in [Ame99b].

Static CFA. For computing target graphs we require a static CFA, i.e., function calls through function pointers must be resolved by the C front end using static analysis. The same holds true for a sound treatment of `longjmp` and `setjmp`, which must have been resolved using static analysis. As it will not always be possible to statically resolve function pointers to a unique function call and jump targets to a unique location, a sound overapproximation using finite case distinction can be used instead. This could result in target graphs that are supersets of the precise target graph, but no edges will be missed.

Apart from such run-time dependent parts of the CFA, we also have to handle behavior left undefined by the C standard, such as the order of evaluation of function or operand arguments. In building a static CFA the C front end will make arbitrary choices for such implementation-defined steps.

C Basic Blocks. We refer to a *basic block* as a subgraph of the CFA that represents a maximal sequence of source statements which can only be entered at the first of them and exited at the last of them [Muc97]. In case such a basic block only consists of a conditional statement, we introduce a skip statement in order to have a unique edge representing the basic block. Such an additional skip statement occurs in Figure 5.2 on the edge leaving state 5^0.

C Conditions. If a node has more than one successor and as C has no concept of nondeterminism, the node refers to a *condition*, e.g., node 6 refers to `i>=j`. Such expressions are Boolean expressions containing no Boolean operators other than negation. The edges leaving a condition node are called

condition edges which are in this case the edges to 7^0 and 9^0.

Short-Circuit Evaluation and Condition Graphs. In C programs the operators "`&&`" and "`||`" induce *short-circuit evaluation*, i.e., the second argument will only be evaluated if the Boolean outcome cannot be deduced from the first one. Each argument of these operators is itself a condition and we refer to expressions involving short-circuit evaluation as *aggregated Boolean expressions*. Because of short-circuit evaluation, every aggregated Boolean expression induces a non-trivial control flow to abort the evaluation as soon as possible. For example, if the first condition `j>left` in evaluating `(j>left && a[-j]>v)`, occurring in line 5 of Listing 5.2, turns out to be false, then `a[-j]>v` is never evaluated. The control flow automaton induced by a Boolean expression is called the corresponding *condition graph*. For example, the "`while`" statement of line 5 induces the network of nodes 5^0 to 5^4 and includes the conditions `j>left` and `a[-j]>v`. In particular, the condition graph does *not only* consist of the condition edges but also of all other computations which are necessary to evaluate the Boolean expression. For example, the condition graph of `(j>left && a[-j]>v)` includes the transition 5^2 to 5^3 which performs the operation j:=j−1.

Note that aggregated Boolean expressions also occur outside conditional statements – imposing non-trivial control flow in apparently unconditional code, e.g., line 2 of the program in Listing 1.1 does not evaluate `x<5` if `x>2` already evaluated to false.

C Decisions. Following the definition given in DO-178B [RTC92] and affirmed in [Cer02], a *decision* is a "Boolean expression composed of conditions and zero or more Boolean operators." In C code, these are the Boolean expressions controlling a condition statement (an `if`, `switch`, `while`, or `for` statement), every statement involving the conditional operator `?:`. Following the definition given in DO-178B this furthermore includes aggregated Boolean expressions occurring outside conditional statements as discussed above.

Short-circuit evaluation can induce condition graphs that have several outgoing edges for the true or false outcome. For such multiple outgoing edges we add a new state that has only a single outgoing edge labeled with a skip statement. This edge is then used to represent the corresponding outcome of the overall decision. For example in Figure 5.2, the node 5^4 has

been introduced to collect all false outcomes of the while statement in line 5 and the edge from 5^4 to 6 is therefore used to represent the false outcome of this decision (this edge must be inserted as well as a unique basic block entry edge for the if statement following in line 6).

5.4.2 Detailed Specification of Filter Functions

In the following descriptions we will use a function start defined over CFA edges: For a set E of edges, we define the set of start locations $\mathsf{start}(E) = \{\ell \mid (\ell, l, \ell') \in E\}$. We use this function to define the initial locations of the target graphs $I_{F[\mathcal{A}]}$ computed by each filter function F. As discussed in Section 5.5, the initial locations determine the starts of paths in target graphs.

- `@BASICBLOCKENTRY`$[\mathcal{A}]$ consists of all the edges in $\mathcal{A}$ which correspond to the first statement of a basic block. Figure 5.2(b) shows the target graph for `@BASICBLOCKENTRY`$[\mathcal{A}]$, referring to Listing 5.1. Here, $I_{\texttt{@BASICBLOCKENTRY}[\mathcal{A}]} = \mathsf{start}(E_{\texttt{@BASICBLOCKENTRY}[\mathcal{A}]})$, indicated with double circles in Figure 5.2(b).

- `@CONDITIONEDGE`$[\mathcal{A}]$ consists of the edges which correspond to the true and false outcomes of all conditions in $\mathcal{A}$. We define $I_{\texttt{@CONDITIONEDGE}[\mathcal{A}]} = \mathsf{start}(E_{\texttt{@CONDITIONEDGE}[\mathcal{A}]})$.

- `@CONDITIONGRAPH`$[\mathcal{A}]$ consists of all condition graphs induced by the evaluation of a Boolean expression in $\mathcal{A}$. Thus `@CONDITIONGRAPH`$[\mathcal{A}]$ is the superset of `@CONDITIONEDGE`$[\mathcal{A}]$ which contains not only the condition edges but also all the computations interconnecting them. Figure 5.2(c) shows the target graph for `@CONDITIONGRAPH`$[\mathcal{A}]$.

  Each condition graph is a directed acyclic graph with a unique entry node corresponding to the entry locations of the represented decisions, i.e., the locations where the program execution starts to evaluate these (aggregated) Boolean expressions. The set $I_{\texttt{@CONDITIONGRAPH}[\mathcal{A}]}$ consists of the entry locations of all condition graphs.

- `@DECISIONEDGE`$[\mathcal{A}]$ consists of the edges which correspond to a specific outcome of a decision in $\mathcal{A}$ (e.g., for an `if`-statement, there is a true- and a false-edge) and $I_{\texttt{@DECISIONEDGE}[\mathcal{A}]}$ is the set $\mathsf{start}(E_{\texttt{@DECISIONEDGE}[\mathcal{A}]})$.

- `@STMTTYPE(types)`$[\mathcal{A}]$ consists of all those edges which correspond to the execution of all statements of types types, where we allow for types all kinds of statements occurring in C, e.g., `@STMTTYPE(if, switch, for, while, ?: )` selects all computations performed by conditional statements. The set $I_{\texttt{@STMTTYPE(types)}[\mathcal{A}]}$ is $\mathsf{start}(E_{\texttt{@STMTTYPE(types)}[\mathcal{A}]})$.
- `@FILE(file)`$[\mathcal{A}]$ contains all edges of $\mathcal{A}$ annotated with file file and the set of initial locations $I_{\texttt{@FILE(file)}[\mathcal{A}]}$ is the union of initial locations of the CFAs representing the C functions contained in file.
- `@LINE(n)`$[\mathcal{A}]$ contains all edges of $\mathcal{A}$ annotated with line n and the initial locations are defined by $I_{\texttt{@LINE(n)}[\mathcal{A}]} = \mathsf{start}(E_{\texttt{@LINE(n)}[\mathcal{A}]})$.
- `@FUNC(fct)`$[\mathcal{A}]$ contains all edges of $\mathcal{A}$ corresponding to C function fct. The set $I_{\texttt{@FUNC(fct)}[\mathcal{A}]}$ is the singleton set consisting of the initial location of the CFA of fct, if this node is in $\mathcal{A}$.
- `@ENTRY(fct)`$[\mathcal{A}]$ contains all edges of $\mathcal{A}$ that are outgoing edges of the initial location of the CFA of function fct. The initial states are equivalent to those of `@FUNC(fct)`$[\mathcal{A}]$: $I_{\texttt{@ENTRY(fct)}[\mathcal{A}]} = I_{\texttt{@FUNC(fct)}[\mathcal{A}]}$.
- `@EXIT(fct)`$[\mathcal{A}]$ consists of all function return edges of $\mathcal{A}$ annotated with function fct, and edges of $\mathcal{A}$ that have no successor in the CFA corresponding to C function fct. The set $I_{\texttt{@EXIT(fct)}[\mathcal{A}]}$ is $\mathsf{start}(E_{\texttt{@EXIT(fct)}[\mathcal{A}]})$.
- `@CALL(fct)`$[\mathcal{A}]$ contains all function call edges of $\mathcal{A}$ where function fct is called and $I_{\texttt{@CALL(fct)}[\mathcal{A}]}$ is $\mathsf{start}(E_{\texttt{@CALL(fct)}[\mathcal{A}]})$.
- `@DEF(v)`$[\mathcal{A}]$ contains all assignment edges in $\mathcal{A}$ where a symbol v is used as left-hand side. The set $I_{\texttt{@DEF(v)}[\mathcal{A}]}$ is $\mathsf{start}(E_{\texttt{@DEF(v)}[\mathcal{A}]})$.
- `@USE(v)`$[\mathcal{A}]$ consists of all edges of $\mathcal{A}$ where the value of a symbol v is read and $I_{\texttt{@USE(v)}[\mathcal{A}]}$ is $\mathsf{start}(E_{\texttt{@USE(v)}[\mathcal{A}]})$.

As described in Section 5.1, FQL technically consists of two languages. We have described elementary coverage patterns, the basic language, in Section 5.2. The language to be used by human engineers, however, should not refer to elements of CFAs, but source code elements instead. This gap is bridged by target graphs and filter functions. Having defined filter functions we are now ready to describe the language used by the test engineer, which we refer to as FQL specifications.

5.5 FQL Specifications

Table 5.6 defines the syntax of FQL specifications. Basic operations like '+' or '.' are the same as in ECPs, but, where ECPs had nodes and edges of a CFA, FQL specifications derive these sets of nodes and edges *from target graphs*. Given a target graph, we can choose to obtain either (i) all nodes, (ii) all edges, or (iii) all paths in the target graph as symbols in coverage specifications or path patterns. In the first and second case, the nodes and edges induce a tractably sized set of symbols. The third case yields an *exponential* number of symbols for an acyclic graph and an unbounded number in general. We thus require the explicit specification of a *bound* that limits the number of recurrences of a node in each path.

$$
\begin{aligned}
\Phi &::= \texttt{cover}\ C\ \texttt{passing}\ P \\
C &::= C\ \texttt{+}\ C \mid C\texttt{.}C \mid \texttt{(}C\texttt{)} \mid N \mid S \mid \texttt{"}P\texttt{"} \\
P &::= P\ \texttt{+}\ P \mid P\texttt{.}P \mid \texttt{(}P\texttt{)} \mid N \mid S \mid P\texttt{*} \\
\\
N &::= \texttt{NODES(}T\texttt{)} \mid \texttt{EDGES(}T\texttt{)} \mid \texttt{PATHS(}T\texttt{,}k\texttt{)} \\
T &::= F \mid \texttt{ID} \mid \texttt{COMPOSE(}T\texttt{,}T\texttt{)} \\
&\quad \mid T\texttt{|}T \mid T\texttt{\&}T \mid \texttt{SETMINUS(}T\texttt{,}T\texttt{)} \\
F &::= \texttt{@BASICBLOCKENTRY} \mid \texttt{@CONDITIONEDGE} \\
&\quad \mid \texttt{@CONDITIONGRAPH} \mid \texttt{@DECISIONEDGE} \mid \texttt{@FILE(}a\texttt{)} \\
&\quad \mid \texttt{@LINE(}x\texttt{)} \mid \texttt{@FUNC(}f\texttt{)} \mid \texttt{@STMTTYPE(}types\texttt{)} \\
&\quad \mid \texttt{@DEF(}t\texttt{)} \mid \texttt{@USE(}t\texttt{)} \mid \texttt{@CALL(}f\texttt{)} \mid \texttt{@ENTRY(}f\texttt{)} \mid \texttt{@EXIT(}f\texttt{)}
\end{aligned}
$$

Table 5.6: Syntax of FQL

More specifically, given a CFA $\mathcal{A}$, a CFA transformer T, and a positive integer k, we apply the operators `NODES(`T`)`, `EDGES(`T`)`, and `PATHS(`T`,`k`)` to obtain *target alphabets* from a target graph $T[\mathcal{A}]$. Given a specification Φ and a CFA $\mathcal{A}$, every operator `NODES(`T`)`, `EDGES(`T`)`, and `PATHS(`T`,`k`)` in Φ expands to a sum $\sum$ (iterated '+') of path patterns which represent the nodes, edges, and k-*bounded paths* in the target graph $T[\mathcal{A}]$, respectively:

$$\begin{aligned}
\texttt{NODES(}T\texttt{)} &\mapsto \sum_{\ell \in L_{T[\mathcal{A}]}} \ell \\
\texttt{EDGES(}T\texttt{)} &\mapsto \sum_{e \in E_{T[\mathcal{A}]}} e \\
\texttt{PATHS(}T\texttt{,}k\texttt{)} &\mapsto \sum_{p \in \mathsf{paths}_k(T[\mathcal{A}])} p
\end{aligned}$$

Intuitively, `NODES(`T`)` is the set of nodes of the target graph $T[\mathcal{A}]$ obtained by applying T to $\mathcal{A}$. Analogously, `EDGES(`T`)` yields the set of edges of the target graph $T[\mathcal{A}]$. For a target graph $T[\mathcal{A}]$, a k-bounded path is a path in $T[\mathcal{A}]$ which starts in one of the initial locations $I_{T[\mathcal{A}]}$ and visits no target graph node $\ell \in L_{T[\mathcal{A}]}$ more than $k > 0$ times. We define:

$$\begin{aligned}
\mathsf{paths}_k(T[\mathcal{A}]) = \{(\ell_0, l_0, \ell_1).(\ell_1, l_1, \ell_2). \ldots .(\ell_n, l_n, \ell_{n+1}) \mid n \geq 0 \wedge \ell_0 \in I_{T[\mathcal{A}]} \\
\wedge\, (\ell_i, l_i, \ell_{i+1}) \in E_{T[\mathcal{A}]} \wedge \ell_i \text{ occurs at most } k \text{ times}\}
\end{aligned}$$

In case a set `NODES(`T`)`, `EDGES(`T`)`, or `PATHS(`T`,`k`)` evaluates to the empty set on $T[\mathcal{A}]$ the corresponding operator expands to the symbol $\emptyset$.

```
1 if ((x > 10
2       && y < 100)
3         ||  (x < y))
4 {  ...  }
5 else
6 {  ...  }
```

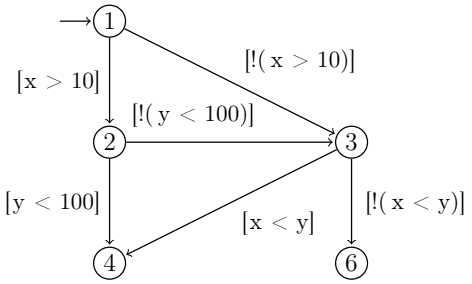


Figure 5.3: Edge- vs. path-coverage

As an example underlining the differences between the three operators consider the target graph shown in Figure 5.3. Let ℓ_i denote the node labeled with i. The CFA $\mathcal{A}$ has the set of nodes $L_{\mathcal{A}} = \{\ell_1, \ell_2, \ell_3, \ell_4, \ell_6\}$ and the operator `NODES(ID)` yields the expression $\ell_1 + \ell_2 + \ell_3 + \ell_4 + \ell_6$. The operator `EDGES(ID)` yields the path pattern $e_{1,2} + e_{1,3} + e_{2,3} + e_{2,4} + e_{3,4} + e_{3,6}$, where $e_{i,j}$ denotes the edge from node ℓ_i to node ℓ_j. With $I_{\mathcal{A}} = \{\ell_1\}$ the operator `PATHS(ID,1)` yields all bounded paths starting in ℓ_1, described by the expression $e_{1,2} + e_{1,3} + e_{1,2}.e_{2,3} + e_{1,2}.e_{2,4} + e_{1,2}.e_{2,3}.e_{3,4} + e_{1,2}.e_{2,3}.e_{3,6} + e_{1,3}.e_{3,4} + e_{1,3}.e_{3,6}$.

Semantics of FQL Specifications

Intuitively, the semantics of an FQL specification Φ is obtained by replacing each occurrence of `NODES`, `EDGES`, and `PATHS` in Φ by the corresponding sum and applying the semantics of Table 5.2. To formalize this step we extend the language of elementary coverage patterns of Table 5.2 by definitions for `NODES`, `EDGES`, and `PATHS` as shown in Table 5.7:

$$\begin{aligned}
&\mathcal{L}(\texttt{NODES(}T[\mathcal{A}]\texttt{)}) &&= \{\ell \mid \ell \in L_{T[\mathcal{A}]}\} \\
&\mathcal{L}(\texttt{EDGES(}T[\mathcal{A}]\texttt{)}) &&= \{e \mid e \in E_{T[\mathcal{A}]}\} \\
&\mathcal{L}(\texttt{PATHS(}T[\mathcal{A}]\texttt{,}k\texttt{)}) &&= \bigcup_{p \in \mathsf{paths}_k(T[\mathcal{A}])} \mathcal{L}(p)
\end{aligned}$$

Table 5.7: Semantics of `NODES`, `EDGES`, `PATHS`

It remains to related an FQL specification Φ to a CFA $\mathcal{A}$ to obtain $T[\mathcal{A}]$ for a CFA transformer T. We proceed according to Table 5.8:

$$\begin{aligned}
&(P_1 \texttt{ + } P_2)[\mathcal{A}] &&= P_1[\mathcal{A}] \texttt{ + } P_2[\mathcal{A}] \\
&(C_1 \texttt{ + } C_2)[\mathcal{A}] &&= C_1[\mathcal{A}] \texttt{ + } C_2[\mathcal{A}] \\
&(P_1 \texttt{ . } P_2)[\mathcal{A}] &&= P_1[\mathcal{A}] \texttt{ . } P_2[\mathcal{A}] \\
&(C_1 \texttt{.} C_2)[\mathcal{A}] &&= C_1[\mathcal{A}] \texttt{.} C_2[\mathcal{A}] \\
&(P)[\mathcal{A}] &&= (P[\mathcal{A}]) \\
&(C)[\mathcal{A}] &&= (C[\mathcal{A}]) \\
&\texttt{NODES(}T\texttt{)}[\mathcal{A}] &&= \texttt{NODES(}T[\mathcal{A}]\texttt{)} \\
&\texttt{EDGES(}T\texttt{)}[\mathcal{A}] &&= \texttt{EDGES(}T[\mathcal{A}]\texttt{)} \\
&\texttt{PATHS(}T\texttt{,}k\texttt{)}[\mathcal{A}] &&= \texttt{PATHS(}T[\mathcal{A}]\texttt{,}k\texttt{)} \\
&S[\mathcal{A}] &&= S \\
&P\texttt{*}[\mathcal{A}] &&= (P[\mathcal{A}])\texttt{*} \\
&\texttt{"}P\texttt{"}[\mathcal{A}] &&= \texttt{"}P[\mathcal{A}]\texttt{"} \\
&(\texttt{cover } C \texttt{ passing } P)[\mathcal{A}] &&= \texttt{cover } C[\mathcal{A}] \texttt{ passing } P[\mathcal{A}]
\end{aligned}$$

Table 5.8: Application of CFA $\mathcal{A}$ to FQL specifications

In this step an FQL specification

$$\Phi = \texttt{cover}\ C\ \texttt{passing}\ P$$

maps a CFA $\mathcal{A}$ to a finite set $\Phi(\mathcal{A})$ of path predicates. We define $\Phi(\mathcal{A})$ by reducing Φ to an ECP:

$$\Phi(\mathcal{A}) = \mathcal{L}((\texttt{cover}\ C\ \texttt{passing}\ P)[\mathcal{A}])$$

Proposition 5.3 *An* FQL *specification* $\Phi = \texttt{cover}\ C\ \texttt{passing}\ P$ *is an elementary coverage criterion (Definition 4.9).*

Proof. We first show that an FQL specification Φ satisfies Definition 4.8 and therefore is a coverage criterion: Φ defines a mapping from CFAs to a set of path patterns, as follows: An FQL specification Φ defines a mapping from CFAs to a set of path predicates by first applying the definitions from Table 5.8, and then applying CFA transformers as described in Table 5.4 and filter functions (Section 5.4.2). We thereby arrive at elementary coverage patterns, which define languages over path patterns (Table 5.2), which we interpret as path predicates (Table 5.3).

It remains to show that this language over path patterns is finite. As a CFA $\mathcal{A}$ by Definition 4.1 has a finite number of locations $L_{\mathcal{A}}$ and therefore also a finite number of edges $E_{\mathcal{A}}$ the expansion in Table 5.7 yields finite languages of nodes, edges, or bounded paths. Filter functions (Definition 5.2) and the CFA transformers defined in Table 5.4 preserve finiteness, hence Table 5.2 yields a finite language for any FQL specification $\Phi = \texttt{cover}\ C\ \texttt{passing}\ P$.

Each word in the resulting finite language is interpreted as a path predicate. The finite set of path predicates is a path set predicate as described in Definition 4.9, hence Φ is a coverage criterion (Definition 4.8). As Φ is a finite set of path predicates, Φ fulfills Definition 4.9 and therefore is an elementary coverage criterion. □

We give an example of the full sequence of evaluation steps from FQL specifications down to sets of path patterns in Section 5.7. For the complete picture, however, we first describe additional syntactic sugar to arrive at the full language available to the human engineer.

5.6 Full FQL Specifications

We first describe the most complete form of FQL queries and then show how this reduces to FQL specifications as described in the preceding section.

Furthermore we present several technically redundant constructions that help to further simplify the use of FQL.

We extend FQL specifications to the form

$$\Phi = \texttt{in}\ T\ \texttt{cover}\ \texttt{\^{}}C\texttt{\$}\ \texttt{passing}\ \texttt{\^{}}P\texttt{\$}$$

The full syntax of FQL is given in Table 5.9. Before presenting the semantics of full FQL specifications in Tables 5.10 and 5.11, we give an intuitive description: the clause `in` T with a CFA transformer T states that, given a CFA $\mathcal{A}$, all filter functions in the `cover` clause are applied to the target graph $T[\mathcal{A}]$. The prefix '`^`' and the suffix '`$`' perform anchoring in analogy to Unix `grep`. If either of those anchors is omitted, `"ID*"` is prepended and appended to `cover` and `passing` clauses. Not only can these anchors be omitted, also the '`in`' and '`passing`' clauses are optional. Furthermore, as `EDGES` is the most common interpretation for target graphs, it is taken as default if neither the operator `NODES`, nor `EDGES`, nor `PATHS` is given.

$$
\begin{aligned}
\Phi &::= \texttt{in}\ T\ \Phi' \mid \Phi' \\
\Phi' &::= \texttt{cover}\ C'\ \texttt{passing}\ P' \mid \texttt{cover}\ C' \\
C' &::= C \mid \texttt{\^{}}C \mid \texttt{\^{}}C\texttt{\$} \mid C\texttt{\$} \\
P' &::= P \mid \texttt{\^{}}P \mid \texttt{\^{}}P\texttt{\$} \mid P\texttt{\$} \\
C &::= C\ \texttt{+}\ C \mid C\texttt{.}C \mid C\texttt{->}C \mid (C) \mid N \mid S \mid \texttt{"}P\texttt{"} \mid C\texttt{==}k \mid C\texttt{<=}k \\
P &::= P\ \texttt{+}\ P \mid P\texttt{.}P \mid P\texttt{->}P \mid (P) \mid N \mid S \mid P\texttt{*} \mid P\texttt{==}k \mid P\texttt{<=}k \mid P\texttt{>=}k \\
N &::= T \mid \texttt{NODES(}T\texttt{)} \mid \texttt{EDGES(}T\texttt{)} \mid \texttt{PATHS(}T\texttt{,}k\texttt{)} \\
T &::= F \mid \texttt{ID} \mid \texttt{COMPOSE(}T\texttt{,}T\texttt{)} \\
&\quad \mid T\texttt{|}T \mid T\texttt{\&}T \mid \texttt{SETMINUS(}T\texttt{,}T\texttt{)} \mid \texttt{NOT(}T\texttt{)} \\
F &::= \texttt{@BASICBLOCKENTRY} \mid \texttt{@CONDITIONEDGE} \\
&\quad \mid \texttt{@CONDITIONGRAPH} \mid \texttt{@DECISIONEDGE} \mid \texttt{@FILE(}a\texttt{)} \\
&\quad \mid \texttt{@LINE(}x\texttt{)} \mid \texttt{@}x \mid \texttt{@FUNC(}f\texttt{)} \mid \texttt{@STMTTYPE(}types\texttt{)} \\
&\quad \mid \texttt{@DEF(}t\texttt{)} \mid \texttt{@USE(}t\texttt{)} \mid \texttt{@CALL(}f\texttt{)} \mid \texttt{@ENTRY(}f\texttt{)} \mid \texttt{@EXIT(}f\texttt{)}
\end{aligned}
$$

Table 5.9: Syntax of full FQL specifications

Tables 5.10 and 5.11 show how the additional constructs and syntactic sugar reduce to FQL specifications as presented in Section 5.5.

$$
\begin{aligned}
&\texttt{in}\ T\ \Phi' && = \Phi' \circledcirc T \\
&\Phi' && = \Phi' \circledcirc \texttt{ID}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{cover}\ C'\ \texttt{passing}\ P' \circledcirc T && = \texttt{cover}\ C' \circledcirc T\ \texttt{passing}\ P' \\
&\texttt{cover}\ C' \circledcirc T && = \texttt{cover}\ C' \circledcirc T\ \texttt{passing}\ \texttt{\^{}ID*\$}
\end{aligned}
$$

$$
C' \circledcirc T = \begin{cases} C \circledcirc T & \text{if } C' \equiv \texttt{\^{}}C\texttt{\$} \\ C \circledcirc T\texttt{."ID*"} & \text{if } C' \equiv \texttt{\^{}}C \\ \texttt{"ID*".}C \circledcirc T & \text{if } C' \equiv C\texttt{\$} \\ \texttt{"ID*".}C \circledcirc T\texttt{."ID*"} & \text{if } C' \equiv C \end{cases}
$$

$$
P' = \begin{cases} P \circledcirc \texttt{ID} & \text{if } P' \equiv \texttt{\^{}}P\texttt{\$} \\ P \circledcirc \texttt{ID."ID*"} & \text{if } P' \equiv \texttt{\^{}}P \\ \texttt{"ID*".}P \circledcirc \texttt{ID} & \text{if } P' \equiv P\texttt{\$} \\ \texttt{"ID*".}P \circledcirc \texttt{ID."ID*"} & \text{if } P' \equiv P \end{cases}
$$

$$
\begin{aligned}
&(C_1\ \texttt{+}\ C_2) \circledcirc T && = C_1 \circledcirc T\ \texttt{+}\ C_2 \circledcirc T \\
&(C_1\texttt{.}C_2) \circledcirc T && = C_1 \circledcirc T\texttt{.}C_2 \circledcirc T \\
&(C_1\texttt{->}C_2) \circledcirc T && = (C_1\texttt{."ID*".}C_2) \circledcirc T \\
&(C) \circledcirc T && = (C \circledcirc T) \\
&\texttt{"}P\texttt{"} \circledcirc T && = \texttt{"}P \circledcirc T\texttt{"}
\end{aligned}
$$

$$
(C\texttt{==}k) \circledcirc T = \begin{cases} \varepsilon & \text{if } k = 0 \\ (\underbrace{C\texttt{.}\ldots\texttt{.}C}_{k \text{ times}}) \circledcirc T \end{cases}
$$

$$
(C\texttt{<=}k) \circledcirc T = \sum_{i=0}^{k} (C\texttt{==}i) \circledcirc T
$$

Table 5.10: Translation of full FQL to FQL specifications

$$
\begin{array}{ll}
(P_1 \texttt{ + } P_2) \circledcirc T & = P_1 \circledcirc T \texttt{ + } P_2 \circledcirc T \\
(P_1 \texttt{.} P_2) \circledcirc T & = P_1 \circledcirc T \texttt{.} P_2 \circledcirc T \\
(P_1 \texttt{->} P_2) \circledcirc T & = (P_1 \texttt{."ID*".} P_2) \circledcirc T \\
(P) \circledcirc T & = (P \circledcirc T) \\
P\texttt{*} \circledcirc T & = (P \circledcirc T)\texttt{*} \\
(P\texttt{==}k) \circledcirc T & = \begin{cases} \varepsilon & \text{if } k = 0 \\ (\underbrace{P \texttt{.} \ldots \texttt{.} P}_{k \text{ times}}) \circledcirc T & \end{cases} \\
(P\texttt{<=}k) \circledcirc T & = \sum_{i=0}^{k} (P\texttt{==}i) \circledcirc T \\
(P\texttt{>=}k) \circledcirc T & = (P\texttt{==}k \texttt{.} P\texttt{*}) \circledcirc T \\
\\
S \circledcirc T & = S \\
\\
N \circledcirc T_1 & = \begin{cases} \texttt{EDGES(COMPOSE(}T_2\texttt{,}T_1\texttt{))} & \text{if } N \equiv T_2 \\ \texttt{NODES(COMPOSE(}T_2\texttt{,}T_1\texttt{))} & \text{if } N \equiv \texttt{NODES(}T_2\texttt{)} \\ \texttt{EDGES(COMPOSE(}T_2\texttt{,}T_1\texttt{))} & \text{if } N \equiv \texttt{EDGES(}T_2\texttt{)} \\ \texttt{PATHS(COMPOSE(}T_2\texttt{,}T_1\texttt{),}k\texttt{)} & \text{if } N \equiv \texttt{PATHS(}T_2\texttt{,}k\texttt{)} \end{cases} \\
\\
\texttt{NOT(}T\texttt{)} & = \texttt{SETMINUS(ID,}T\texttt{)} \\
\\
\texttt{@}x & = \texttt{@LINE(}x\texttt{)}
\end{array}
$$

Table 5.11: Translation of full FQL to FQL specifications (*cont.*)

5.7 Example of FQL Query Evaluation

We show how a specification for condition coverage first describes an elementary coverage criterion and then maps to a coverage predicate given the source code shown in Figure 5.3. As initial specification we use

```
cover @CONDITIONEDGE
```

which constitutes a valid specification for full FQL. We first translate this specification to an FQL specification according to the rules of Tables 5.10 and 5.11:

```
cover @CONDITIONEDGE
cover @CONDITIONEDGE ◎ ID
cover @CONDITIONEDGE ◎ ID passing ^ID*$
cover "ID*".@CONDITIONEDGE ◎ ID."ID*" passing ID*
cover "ID*".COMPOSE(@CONDITIONEDGE,ID)."ID*" passing ID*
```

We finally add the `EDGES` operator to arrive at an FQL specification:

$$\Phi = \texttt{cover "EDGES(ID)*".EDGES(COMPOSE(@CONDITIONEDGE,ID)).}$$
$$\texttt{"EDGES(ID)*" passing EDGES(ID)*}$$

This FQL specification Φ describes the coverage criterion "condition coverage" independently of the program under test. For any given program $\mathcal{A}$, however, we can compute the set of path patterns $\Phi(\mathcal{A})$ that serve as test goals.

As first step we rewrite the above FQL specification as an ECP using sums over each set of edges as described in Section 5.5:

$$\texttt{cover "}\sum_{e \in E_{\texttt{ID}[\mathcal{A}]}} e\texttt{*".}\sum_{e \in E_{\texttt{COMPOSE(@CONDITIONEDGE,ID)}[\mathcal{A}]}} e\texttt{."}\sum_{e \in E_{\texttt{ID}[\mathcal{A}]}} e\texttt{*" passing}\sum_{e \in E_{\texttt{ID}[\mathcal{A}]}} e\texttt{*}$$

With Table 5.4, we simplify $E_{\texttt{ID}[\mathcal{A}]}$ and $E_{\texttt{COMPOSE(@CONDITIONEDGE,ID)}[\mathcal{A}]}$:

$$\texttt{ID}[\mathcal{A}] = \mathcal{A}$$
$$\Rightarrow E_{\texttt{ID}[\mathcal{A}]} = E_{\mathcal{A}}$$

and

$$\begin{aligned}\texttt{COMPOSE(@CONDITIONEDGE,ID)}[\mathcal{A}] &= \texttt{@CONDITIONEDGE}[\texttt{ID}[\mathcal{A}]] \\ &= \texttt{@CONDITIONEDGE}[\mathcal{A}] \\ \Rightarrow E_{\texttt{COMPOSE(@CONDITIONEDGE,ID)}[\mathcal{A}]} &= E_{\texttt{@CONDITIONEDGE}[\mathcal{A}]}\end{aligned}$$

For the example of the program of Figure 5.3 we can determine the target graphs $\mathcal{A}$ and $\texttt{@CONDITIONEDGE}[\mathcal{A}]$. In this case we only describe the sets of edges of these target graphs:

$$\begin{aligned}E_{\mathcal{A}} &= \{e_{1,2}, e_{1,3}, e_{2,3}, e_{2,4}, e_{3,4}, e_{3,6}\} \\ E_{\texttt{@CONDITIONEDGE}[\mathcal{A}]} &= \{e_{1,2}, e_{1,3}, e_{2,3}, e_{2,4}, e_{3,4}, e_{3,6}\}\end{aligned}$$

and therefore

$$\sum_{e \in E_{\mathcal{A}}} = \sum_{e \in E_{\texttt{@CONDITIONEDGE}[\mathcal{A}]}} = e_{1,2} + e_{1,3} + e_{2,3} + e_{2,4} + e_{3,4} + e_{3,6}.$$

Substituting this sum we obtain the elementary coverage pattern

$$\begin{aligned}\texttt{cover}\quad & "(e_{1,2} + \ldots + e_{3,6})^\star".e_{1,2}."(e_{1,2} + \ldots + e_{3,6})^\star" \\ & + "(e_{1,2} + \ldots + e_{3,6})^\star".e_{1,3}."(e_{1,2} + \ldots + e_{3,6})^\star" \\ & + "(e_{1,2} + \ldots + e_{3,6})^\star".e_{2,3}."(e_{1,2} + \ldots + e_{3,6})^\star" \\ & + "(e_{1,2} + \ldots + e_{3,6})^\star".e_{2,4}."(e_{1,2} + \ldots + e_{3,6})^\star" \\ & + "(e_{1,2} + \ldots + e_{3,6})^\star".e_{3,4}."(e_{1,2} + \ldots + e_{3,6})^\star" \\ & + "(e_{1,2} + \ldots + e_{3,6})^\star".e_{3,6}."(e_{1,2} + \ldots + e_{3,6})^\star"\} \\ \texttt{passing}\quad & (e_{1,2} + \ldots + e_{3,6})^\star\end{aligned}$$

Following Table 5.2, this ECP yields the following set $\Phi(\mathcal{A})$ of path patterns:

$$\begin{aligned}\{&"(e_{1,2} + \ldots + e_{3,6})^\star".e_{1,2}."(e_{1,2} + \ldots + e_{3,6})^\star" \wedge "(e_{1,2} + \ldots + e_{3,6})^\star", \\ &"(e_{1,2} + \ldots + e_{3,6})^\star".e_{1,3}."(e_{1,2} + \ldots + e_{3,6})^\star" \wedge "(e_{1,2} + \ldots + e_{3,6})^\star", \\ &"(e_{1,2} + \ldots + e_{3,6})^\star".e_{2,3}."(e_{1,2} + \ldots + e_{3,6})^\star" \wedge "(e_{1,2} + \ldots + e_{3,6})^\star", \\ &"(e_{1,2} + \ldots + e_{3,6})^\star".e_{2,4}."(e_{1,2} + \ldots + e_{3,6})^\star" \wedge "(e_{1,2} + \ldots + e_{3,6})^\star", \\ &"(e_{1,2} + \ldots + e_{3,6})^\star".e_{3,4}."(e_{1,2} + \ldots + e_{3,6})^\star" \wedge "(e_{1,2} + \ldots + e_{3,6})^\star", \\ &"(e_{1,2} + \ldots + e_{3,6})^\star".e_{3,6}."(e_{1,2} + \ldots + e_{3,6})^\star" \wedge "(e_{1,2} + \ldots + e_{3,6})^\star"\}.\end{aligned}$$

By interpreting these path patterns as path predicates (Table 5.3) we arrive at a predicate permissible as an elementary coverage criterion (Definition 4.9) with six test goals.

5.8 Expressive Power and Usability

In the following we show how the test case specifications **Q1-24** of Chapter 2, Figures 2.1–2.3, can be expressed in FQL. We will see that even complex specifications can be written as succinct and natural FQL specifications. Experiments with these specifications will be discussed in Chapter 7.

5.8.1 Scenario 1: Structural Coverage Criteria

[**Q1-2** — *"Standard Coverage Criteria"*] Basic block coverage and condition coverage.

Q1: `cover @BASICBLOCKENTRY`
Q2: `cover @CONDITIONEDGE`

Queries **Q1-2** demonstrate how succinct some of the most common coverage criteria can be expressed in FQL. At the same time we also assign them a formal semantics.

[**Q3** — *"Alternative Condition Coverage"*] Condition coverage as defined by CoverageMeter and CTC++ (see Section 1.2).

Q3: `cover @CONDITIONEDGE & @STMTTYPE(if,switch,for,while,?:)`

According to our observations, the coverage measurement tools CoverageMeter and CTC++ only consider (aggreated) Boolean expressions within the conditional statement of `if`, `switch`, `for`, `while`, and `?:` when computing the goals for "condition coverage". To mimic this coverage criterion, we restrict the target alphabet of condition edges to these statement types by intersecting the respective target graphs.

[**Q4** — *"Acyclic Path Coverage"*] Cover all acyclic paths through functions `main` and `insert`.

[**Q5** — *"Loop-Bounded Path Coverage"*] Cover all paths through `main` and `insert` which pass each statement at most twice.

Q4: `cover PATHS(@FUNC(main) | @FUNC(insert),1)`
Q5: `cover PATHS(@FUNC(main) | @FUNC(insert),2)`

We use the operator `PATHS` to extract bounded paths from a selected target graph. The informal specifications request the union of functions `main` and `insert` as target graph. Acyclic paths correspond to a bound of 1, and the specification for **Q5** implies a bound of 2.

5.8.2 Scenario 2: Data Flow Coverage Criteria

[**Q6** — *"Def Coverage"*] Cover all statements defining a variable `t`.

[**Q7** — *"Use Coverage"*] Cover all statements that use the variable `t` as right hand side value.

Q6: `cover @DEF(t)`
Q7: `cover @USE(t)`

We observe that FQL is not only well suited to express structural code coverage criteria, but also data flow coverage criteria can be expressed in succinct manner. It should be noted, however, that extensions to coverage of *all* definitions, i.e., def-coverage for all variables, require a preceding extraction of all variable names from the code to generate a proper query. Given a tool that parses the source code and lists all variables, a query can be automatically generated as, e.g.,

```
cover @DEF(var1) + @DEF(var2) + @DEF(var3)
```

for extracted variable names `var1`, `var2`, and `var3`.

[**Q8** — *"Def-Use Coverage"*] Cover all def-use pairs of variable `t`.

Q8: `cover @DEF(t)."NOT(@DEF(t))*".@USE(t)`

Query **Q8** describes proper def-use pairs, i.e., the expression `"NOT(@DEF(t))*"` ensures that no further assignment to variable `t` occurs before a read (a use) of variable `t`.

5.8.3 Scenario 3: Constraining Test Cases

[**Q9** — *"Constrained Program Paths"*] Basic block coverage with test cases that satisfy the assertion $j > 0$ after executing line 2.

Q9: `cover @BASICBLOCKENTRY passing ^(@2.{j>0}+NOT(@2))*$`

We combine basic block coverage, as already specified in query **Q1**, with a restriction on the program's state space. The `passing` clause allows to state this constraint independently of the description of structural coverage. The constraint states that any test case must in each step either pass line 2 *and* satisfy the property $j > 0$, or pass some code location other than line 2.

[**Q10** — *"Constrained Calling Context"*] Condition coverage in a function `compare` with test cases which call `compare` from inside function `sort` only.

Q10: `cover @CONDITIONEDGE & @FUNC(compare) passing ^(NOT(@CALL(compare))*.(@CALL(compare) & @FUNC(sort))*)*$`

Instead of constraining the data space we restrict code paths in this query. Note the importance of anchoring using '`^`' and '`$`' in these restrictions: had it been omitted, by default `ID*` would be prepended and appended. As a result, *any* prefix and suffix would be permitted in test cases, as long as some part of the test case matches the restriction. In fact, because of the Kleene star used in these constraints, no match would be required at all (zero repetitions of the constraint).

[**Q11** — *"Constrained Inputs"*] Basic block coverage in function `sort` with test cases that use a list with 2 to 15 elements.

Q11: `cover @ENTRY(sort).{len>=2}.{len<=15}."NOT(@EXIT(sort))*" .@BASICBLOCKENTRY`

In writing query **Q11** we assume that function `sort` takes an argument `len` that describes the length of the input list. We then enforce that function `sort` is not left before matching one of the basic blocks using the expression `"NOT(@EXIT(sort))*"`.

[**Q12** — *"Recursion Depth"*] Cover function `eval` with condition coverage and require each test case to perform three recursive calls of `eval`.

Q12: `in @FUNC(eval) cover @CONDITIONEDGE passing @CALL(eval) .NOT(@EXIT(eval))*.@CALL(eval).NOT(@EXIT(eval))*.@CALL(eval)`

Analogously to query **Q11**, we enforce in query **Q12** that function `eval` is not left – amounting to recursion as we require yet another call of function function `eval` using the target graph `@CALL(eval)`.

[**Q13** — *"Avoid Unfinished Code"*] Cover all calls to `sort` such that `sort` never calls `unfinished`. That function is allowed to be called outside `sort` – assuming that only the functionality of `unfinished` that is used by `sort` is not testable.

[**Q14** — *"Avoid Trivial Cases"*] Cover all conditions and avoid trivial test cases, i.e., require that `insert` is called twice before calling `eval`.

```
Q13: cover @CALL(sort) passing ^(NOT(@FUNC(sort))*
  .(@FUNC(sort) & NOT(@CALL(unfinished)))*.NOT(@FUNC(sort))*)*$
Q14: cover @CONDITIONEDGE passing ^(NOT(@CALL(eval))*
  .@CALL(insert))>=2
```

Queries **Q13-14** showcase further situations where a structural code coverage criterion is combined with restrictions on calling sequences. As described in these examples, such situations both arise in ad hoc testing and debugging where code is incomplete and also in integration testing where trivial test cases must be avoided.

5.8.4 Scenario 4: Customized Test Goals

[**Q15** — *"Restricted Scope of Analysis"*] Condition coverage in a function `partition` with test cases that reach line 7 at least once.

[**Q16** — *"Condition/Decision Coverage"*] Condition/decision coverage (the union of condition and decision coverage) [MSBT04].

[**Q17** — *"Interaction Coverage"*] Cover all possible pairs between conditions in function `sort` and basic blocks in function `eval`, i.e., cover all possible interactions between `sort` and `eval`.

```
Q15: in @FUNC(partition) cover @CONDITIONEDGE passing @7
Q16: cover @CONDITIONEDGE + @DECISIONEDGE
Q17: cover (@CONDITIONEDGE & @FUNC(sort))
  ->(@BASICBLOCKENTRY & @FUNC(eval))
```

In queries **Q15-17** we combine standard coverage criteria in novel ways. The need for such combinations arises in ad hoc testing (**Q15**), test case generation for certification (**Q16** – coverage requirements in higher assurance levels of DO-178B always include the coverage requirements of lower levels), and integration testing. Note the different ways of combining coverage criteria that are required in these situations and hence different FQL operators.

[**Q18-20** — *"Cartesian Block Coverage"*] Cover all pairs, triples, and quadruples of basic blocks in function `partition`.

```
Q18: cover @BASICBLOCKENTRY->@BASICBLOCKENTRY
Q19: cover @BASICBLOCKENTRY->@BASICBLOCKENTRY->@BASICBLOCKENTRY
Q20: cover @BASICBLOCKENTRY->@BASICBLOCKENTRY
  ->@BASICBLOCKENTRY->@BASICBLOCKENTRY
```

In FQL it is easy to construct arbitrary Cartesian products of test goal sets using concatenation ('.'). Most often, however, we are not interested what occurs on a path from one test goal set to another, and hence use an intermittent `ID*`, abbreviated by '`->`'.

5.8.5 Scenario 5: Seamless Transition to Verification

[**Q21** — *"Assertion Coverage"*] Cover all assertions in the source.

[**Q22** — *"Assertion Pair Coverage"*] Cover each pair of assertions with a single test case passing both of them.

```
Q21: cover @STMTTYPE(assert)
Q22: cover @STMTTYPE(assert)->@STMTTYPE(assert)
```

In software verification we are interested in proving all assertions to hold. While proving correctness is almost always impossible using testing, executing all `assert` statements (or, as a refinement thereof, all pairs off `assert` statements) at least provides confidence that the assertions are not trivially violated.

[**Q23** — *"Error Provocation"*] Cover all basic blocks in `eval` without reaching label `init`.

[**Q24** — *"Verification"*] Ask for test cases which enter function `main`, satisfy the precondition, and violate the postcondition.

Q23: `cover (@BASICBLOCKENTRY & @FUNC(eval)) passing ^NOT(@LABEL(init))*$`

Q24: `cover @ENTRY(main) passing @ENTRY(main).{precond()} .NOT(@EXIT(main))*.{!postcond()}.@EXIT(main)`

FQL specifications are well suited to precisely describe certain execution sequences, hence also for specification of paths that are known to be invalid executions. If such a test case exists, then the error can be provoked. Therefore the expected result of such a query would be that no such test case exists. Query **Q23** describes such a code path only, but query **Q24** even uses pre- and postconditions. Again, if such a path exists, it is know to be an error trace.

5.9 Discussion

In this chapter we formally described FQL which is – to the best of our knowledge – the first test specification language which satisfies the requirements (a) to (d) of Chapter 2. Challenge (e) is mainly fulfilled by FShell, which we describe in Chapter 6. We addressed challenges (a)–(d) as follows:

(a) FQL is based on three concepts which enable the user to specify coverage criteria in simple and succinct specifications: (1) To specify single test cases, we use *path patterns* to select program paths. These patterns are given by regular expressions. Future extensions of FQL, however, can also include more powerful path pattern expressions. (2) The alphabet of the path patterns refers to program parts such as basic blocks, line numbers, function calls etc. We use the concept of *filter functions* to extract the target alphabet from an individual program. (3) Most coverage criteria require a test suite to cover a number of individual path patterns which depend on the program under test. For instance, simple basic block coverage yields one path pattern per basic block. We achieve the necessary expressive power by a natural extension of regular expressions – *quoted regular expressions* – which matches test suites rather than individual executions. All three concepts are combined into a concise and easy-to-read formalism with a precise semantics.

(b) Encapsulation of language specifics is achieved by filter functions, which compute *target graphs*. A target graph is a fragment of a control flow

automaton and typically contains those parts of the source code that are relevant for a given testing target. Target graphs provide an intermediate layer between language specific and language independent aspects of test case generation. While the construction of the underlying control flow automata (CFAs) is language dependent, the semantics built atop target graphs and CFAs is language independent.

(c,d) These orthogonal concepts taken together allowed us to build a simple yet expressive declarative language atop the mathematical model of Chapter 4. Section 5.8 demonstrated that all example specifications from Figures 2.1–2.3 can be easily expressed in FQL.

I now suggest that we confine ourselves to the design and implementation of intellectually manageable programs.

Edsger W. Dijkstra

Chapter 6

FShell

In this chapter we describe the implementation of query-driven program testing in FShell. We describe the user interface of FShell, its control commands, and the use of FQL in FShell. The major part of this chapter then explains the steps that FShell internally performs to arrive at a set of test cases. We conclude with a description of test harness generation, which turns these test cases into executable code.

First impressions of the *front end* have been given in Section 3.4. The current *back end* of FShell is based on bounded model checking [BCCZ99], using the code base of CBMC [CKL04]. We chose CBMC as the first back end for query-driven program testing because (1) it supports full C syntax and semantics, (2) bit-precise bounded model checking is conceptually closer to testing than an abstraction/refinement approach, and (3) CBMC is well engineered and offers a very clean design and a stable code base. FShell is, like CBMC, implemented in C++ and accounts for approximately 16k lines of source code. We first summarize the steps that CBMC undertakes to perform program verification and then describe how FShell uses some of CBMC's components to implement query-driven program testing.

6.1 Overview of CBMC's Architecture

Bounded model checkers such as CBMC reduce questions about program paths to Boolean constraints in conjunctive normal form (CNF) which are solved by standard SAT solvers. The back end of FShell employs the functionality of CBMC to obtain SAT instances suitable for test case generation.

Recall that on input of a program annotated with assertions, CBMC outputs a SAT instance the solutions of which describe program paths leading to assertion violations. In order to do so, CBMC performs the following main steps, which are outlined in Figure 6.1:

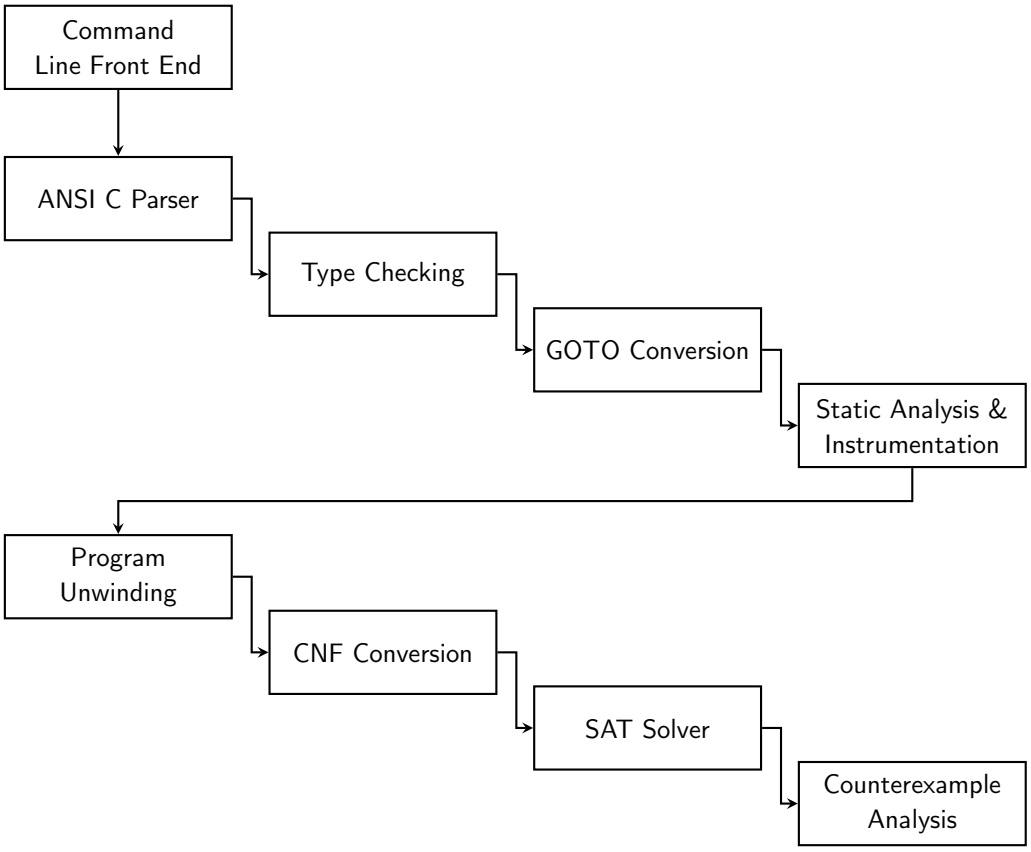


Figure 6.1: CBMC architecture

1. The command line front end processes options given by the user and configures CBMC accordingly. The configuration options mainly concern the system architecture to be assumed, i.e., the bit-width of basic data types and endianness, and operating system specific configuration parameters.

2. The ANSI C parser first calls a C preprocessor (`cl` on Microsoft Windows systems and `cpp` on Unix-like systems) and then reads the preprocessed source files. CBMC therefrom builds a parse tree annotated with source file- and line information.

3. Type checking populates a symbol table by traversing the parse tree, collecting all type names and symbol identifiers, and assigning type information to each symbol that is found. CBMC aborts if any inconsistencies are detected by type checking.

4. CBMC uses "GOTO programs" as intermediate representation. In this language, all non-linear control flow, such as if/switch-statements, loops and jumps, is translated to equivalent *guarded goto* statements. These statements are gotos that include optional guards, such that these guarded gotos also suffice to model if/else branching. The most important classes of statements left at this intermediate level are assignments, gotos, function calls, function returns, declarations, assertions, and assumptions.

   CBMC generates one GOTO program for each C function found in the parse tree. Additionally it adds a new main function that first calls an initialization function for global variables and then calls the original program entry function.

5. As next step, CBMC performs a number of analyses on the GOTO functions. First, function pointers are resolved via a light-weight static analysis that checks for type compatibility between formal parameters of declared functions and the actual parameters at the point of call through a function pointer. All matching targets are combined to a list of conditional calls, where a branch is taken if the actual value of the function pointer matches the address of the target function. Thereby we arrive at a static call graph as discussed in Section 5.4.2. Further analyses and property instrumentation include a pointer analysis with subsequent instrumentation to catch pointer-related errors such as invalid dereferencing by suitable assertions, property instrumentation for assertions on overflows or division by zero, and inlining of small functions.

6. As CBMC implements a variant of bounded model checking it has to pay special attention to loops. Unlike the original bounded model checking algorithm presented in [BCCZ99], CBMC currently does not increase the maximum length of paths as bounded model checking proceeds, and is thus not complete. Instead, CBMC eagerly unwinds loops up to a fixed bound, which can be specified by the user on a per-loop

basis or globally, for all loops. In the course of this unwinding step, CBMC also translates the GOTO functions to static single assignment (SSA) form [AWZ88, RWZ88, CFR+89]. At the end of this process the program is represented as a mathematical equation over renamed program variables in guarded statements. The guards determine whether, given a concrete program execution, an assignment is actually made.

7. The resulting equation is translated into a CNF formula by bit-precise modeling of all expressions plus the Boolean guards (cf. [CKY03]). Here it should be noted that CBMC also supports other decisions procedures as back ends, such as SMT (satisfiability modulo theories) solvers [NOT04, NOT06], in which case an encoding other than CNF is used. For FShell, however, we are only interested in the SAT solver-based back end of CBMC.

8. The CNF formula is passed to the SAT solver, which tries to find a satisfying assignment. Here, such an assignment corresponds to a path violating at least one of the assertions in the program under scrutiny. Conversely, if the formula is unsatisfiable, no assertion can be violated *with the given unwinding bounds*.

9. If a satisfying assignment was found, the bounded model checker has determined a counterexample to the specification given in terms of assertions. To turn the model of the SAT formula into information useful for the user of CBMC, it is translated into a list of assignments. CBMC finds this sequence by consulting the equation of guarded statements: each statement with a guard evaluating to `true` under the computed model constitutes an assignment occurring in the counterexample. The actual values being assigned are also found in the model of the SAT formula.

6.2 Tool Architecture

For a given FQL query, we derive a matching test suite via the following steps:

1. FShell's front end accepts a query and coordinates further processing, in parts using components of CBMC, as outlined in the following steps. Details of the front end are described in Section 6.3.

2. The C front end of CBMC builds GOTO programs from the program under test as described above. We use these GOTO programs as representation of a CFA. We furthermore perform a bounded model checking run to check for failing assertions or insufficient unwinding before starting test case generation (cf. Section 6.3.3).

3. Given a coverage specification, we evaluate all filter expressions over these GOTO programs to obtain the desired target graphs (cf. Section 6.4).

4. The resulting elementary coverage pattern is translated to a pair of *trace automata*, which we introduce in Section 6.5.

5. CBMC builds a Boolean equation describing all states yielding a violation of a given property (assertion) of the input program. It is then able to produce an example of such a violation (counterexample). In test case generation, we use this scheme by adding the property stating that *no* test case matching a given query exists. Any counterexample then describes a path that fulfills the query.

   FShell implements two techniques to enable subsequent enumeration of test cases: the technically straightforward option is instrumentation of trace automata into GOTO functions. The overhead of instrumentation, however, is often prohibitively high (cf. Chapter 7), hence we have implemented a second approach that performs a propositional encoding of trace automata to combine it with CBMC's CNF formula derived from the program under test. Both of these approaches are described in Section 6.6.

6. We efficiently enumerate test cases using guided SAT enumeration as described in Section 6.7. By construction, the resulting test suite satisfies the coverage criterion of the given FQL query.

7. Some of the test cases in the computed test suite, however, may be redundant. We therefore include a minimization step where we again use a SAT solver to compute the minimal subset of the original test suite that still fulfills the coverage criterion (cf. Section 6.8).

8. We print the test suite in terms of a list of required initial values of memory locations (variables) such that these values induce a unique execution path. This procedure is described in Section 6.9.

As a refinement of Figure 1.2, Figures 6.3–6.16 will illustrate these steps in full detail. Figure 6.2 shows the collaboration and interfaces of the involved components. In the following sections we describe all these steps and components in further detail.

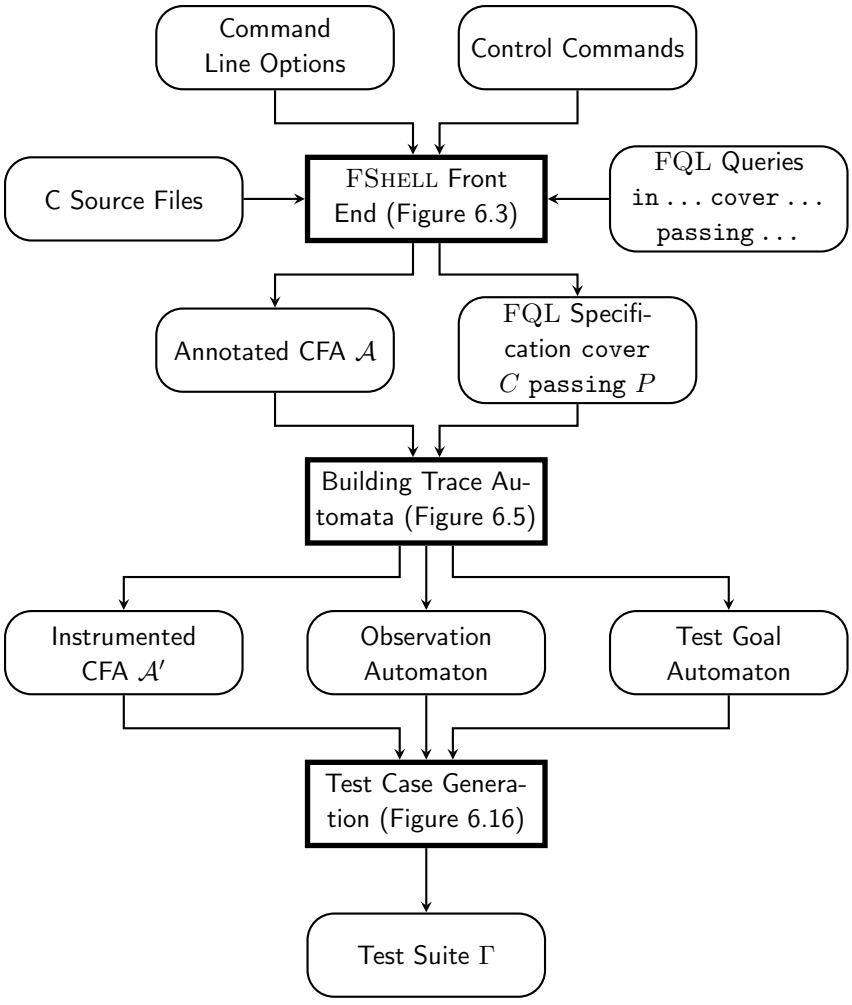


Figure 6.2: FSHELL architecture overview

6.3 Front End and Query Parsing

In the following we will describe the components of Figure 6.3 in detail. From a user's point of view the front end of FSHELL is a shell-like interactive command line interface. A graphical user interface in terms of a plug-in for

the Eclipse IDE [1] is work in progress. Yet also the command line interface aims at good usability by providing standard shell features such as an editing history or macros.

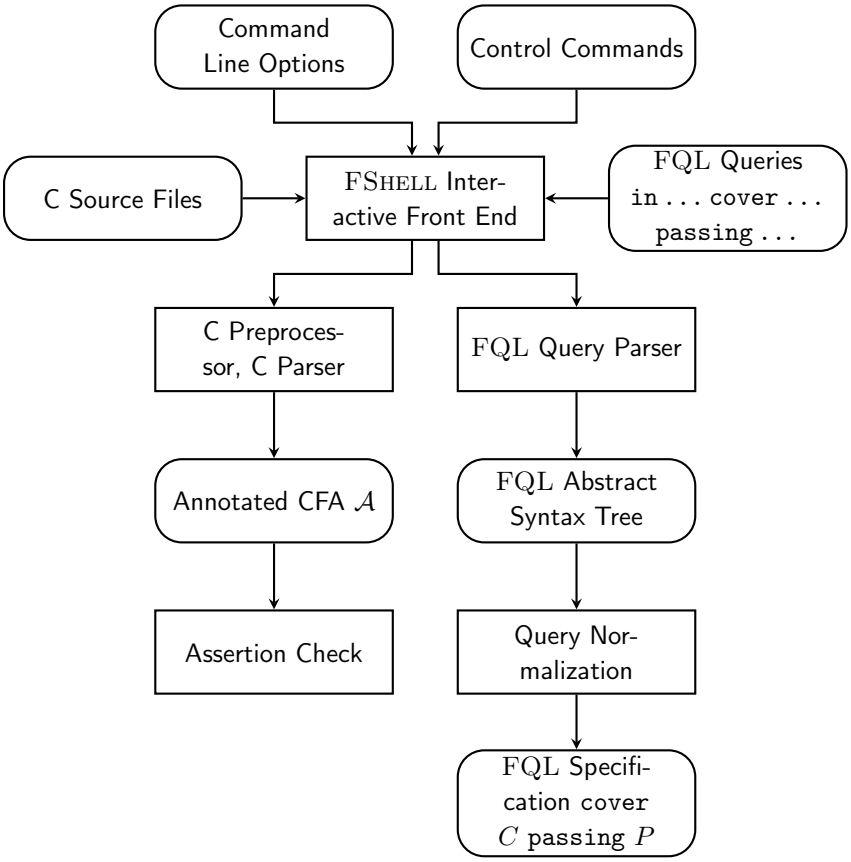


Figure 6.3: FShell front end architecture

6.3.1 Command Line Options

Before entering the interactive shell, however, the user may give a number of command line parameters, which we explain next. The general command syntax to call FShell is

```
fshell [options ...]  [file.c ...]
```

where `[options ...]` are described below and `[file.c ...]` are zero or more source file names.

[1] See `http://www.eclipse.org`

- General options:
  - `--help`, `-h`, `-?`: Display copyright information and the list of command line options.
  - `--version`: Show the current version.
- Platform configuration:
  - `-I` *path*: Add *path* to the C preprocessor's search path for expanding `#include` directives. This option may be given multiple times, which is the case for `-D` *macro* as well:
  - `-D` *macro*: Define the C preprocessor macro *macro*, where *macro* is either only a macro name or of the form *name*=*value*.
  - `--16`, `--32`, `--64`: Set the bit-width of the C type `int` to 16, 32, or 64 bits, respectively.
  - `--LP64`, `--ILP64`, `--LLP64`, `--ILP32`, `--LP32`: Set the bit-widths of int, long, and pointers as defined in [The98].
  - `--little-endian`, `--big-endian`: Set endianness for conversions between words and bytes.
  - `--unsigned-char`: Make `char` type unsigned.
  - `--ppc-macos`, `--i386-macos`, `--i386-linux`, `--i386-win32`, and `--winx64`: Set platform-specific defines, bit-widths, and endianness according to the given processor and operation-system combination.
  - `--no-arch`: Do not enable any platform-specific configuration other than setting the bit-widths to the same configuration the host running FShell uses.
- Options controlling program instrumentation and loop unwinding:
  - `--no-library`: By default CBMC, and therefore FShell, ships an abstracted version of system library functions. This options disables inclusion of such code.
  - `--show-goto-functions`: Display the GOTO functions after instrumentation as described in Section 6.6. This option is primarily useful for debugging purposes.

  - `--no-assumptions`: The programmer can add assumptions to the program under test using the `__CPROVER_assume(`x`)` macro. That is, the property x is enforced to hold true at the program point where the macro was inserted. If `--no-assumptions` is set, assumptions will be ignored.
  - `--enable-assert`: By default, FShell disables user-defined assertions, i.e., they are replaced by skip statements. This option disables such replacement.
  - `--function` f: Use function f as program entry point instead of "`main`".
  - `--depth` k: Perform at most k steps in unwinding the program.
  - `--unwind` k, `--unwindset` L`:`k`,...`, `--show-loop-ids`: Unwind all loops, recursions, and backward gotos at most k times. With `--unwindset` L`:`k`,...` the unwinding bound k is set for loop with id L only, where L can be found using `--show-loop-ids`, which lists all loops with their identifiers.

    After unwinding a loop, an unwinding assertion is added, unless `--no-unwinding-assertions` is given, which is explained below.
  - `--no-unwinding-assertions`, `--partial-loops`: The first option inhibits generation of assertions at the end of a sequence of bodies of an unwound loop. The assertion checks that the loop exit condition is indeed always fulfilled, i.e., the number of unwinding steps was sufficient. With this option, instead, an assumption is added that requires the condition to hold. With the parameter `--partial-loops`, no such assumption is generated. Note that this option implies `--no-unwinding-assertions` and possibly makes the analysis unsound, because each loop is replaced by a fixed number of conditional repetitions of the loop body without any checks whether the loop condition evaluates to false afterwards.

- Options controlling the user interface:
  - `--xml-ui`: Change FShell's output to XML formatted text, which is more suitable for machine processing.

  - `--verbosity` n, `--statistics`: The option `--verbosity` n with $0 \leq n \leq 9$ defines the amount of status information printed while running FShell. $n = 0$ disables all output other than test cases, $n \geq 1$ enables error messages, $n \geq 2$ adds warnings, $n \geq 6$ enables progress information ($n = 6$ is the default), $n \geq 8$ adds statistics, and $n = 9$ yields debugging information. With `--statistics` additional statistics are enabled independently of the verbosity. These statistics include CPU time and memory usage, test suite size, the number of test goals, and further information about the test case generation process which may vary with future release of FShell.
  - `--query-file` f changes to scripted mode where commands are read line by line from file f instead of entering interactive mode.

- Options controlling test case generation strategies:
  - `--use-instrumentation`: Use instrumentation of GOTO functions to represent trace automata, as described in Section 6.6.1. The default is to use the propositional encoding of Section 6.6.2.
  - `--sat-coverage-check`, `--no-internal-coverage-check`: Any computed test case may satisfy more than one test goal. To determine these additional test goals either a SAT solver or a built-in routine can be used, as detailed in Section 6.7.3. Option `--use-instrumentation` implies the use of the SAT solver. Otherwise, using `--sat-coverage-check` the SAT solver is run in addition to the build-in coverage checking function. With the command line option `--no-internal-coverage-check` the built-in routine is disabled even in case of the propositional encoding and the SAT solver will be used exclusively.

- Test suite output:
  - `--show-test-goals`: List the test goals defined by the coverage specification as a list of paths. Each path is described by a list of source code locations. For example, for the session of Section 3.4 we get:

```
Test goal 0 (-2147483647&396&398): <TRUE>...bar.c:2-bar.c:2
  ...(bar.c:2-bar.c:2|bar.c:2-bar.c:3|bar.c:2-bar.c:3|
```

```
    bar.c:2-bar.c:5|bar.c:3|NO_LOCATION-bar.c:6|bar.c:5)
Test goal 1 (-2147483647&397&398): <TRUE>...bar.c:2-bar.c:3
  ...(bar.c:2-bar.c:2|bar.c:2-bar.c:3|bar.c:2-bar.c:3|
    bar.c:2-bar.c:5|bar.c:3|NO_LOCATION-bar.c:6|bar.c:5)
```

In this case we have two test goals, with each of them first passing any statement (`<TRUE>`), then either the `else` or `if` branch (`bar.c:2-bar.c:2` or `bar.c:2-bar.c:3`), and then again any of the statements, now listed as CFA edges.

- `--outfile` f: Write the computed test suite to file f. If the file already exists, the new test suite is appended.
- `--tco-*` options: These options control the format and contents of the test case output, with each of them enabling specific additional information:
  - With `--tco-location`, as was already shown in Section 3.4, the program entry function is printed and for each assignment the C type of the variable and the location of the initialization is shown. This information is required for subsequent test harness generation (cf. Section 6.10).
  - The option `--tco-called-functions` adds another section to the output: for each test case, the sequence of functions called while executing the test case is printed.
  - Analogously, `--tco-assign-globals` causes a printout of the sequence of assignments to global variables.

  The last two options are intended for validation of test results through inspection of the sequence of function calls and assignments to global variables, respectively.

6.3.2 Interactive Shell, Control Commands, and Macros

Once the user launches FShell with options other than `--version`, `--help`, or `--query-file`, FShell enters interactive mode. If source file names were given on the command line, the corresponding files are parsed first. FShell aborts with an error message if parsing fails at this point. Source files are preprocessed using a C preprocessor. For Unix-like systems and for Windows/MinGW this is `cpp`. The user may wish to override this by setting the "`CPP`" environment variable to the desired preprocessor.

In interactive mode FSHELL displays the prompt

```
FShell2>
```

to signal that it is waiting for user input. The input is now either a control command or a macro definition as detailed below, or an FQL query using the syntax described in Section 5.6. FSHELL processes all keywords in a case-insensitive manner. Lines starting with '//' are ignored, which is mainly useful for adding comments in FSHELL query scripts. To save typing for repeated commands or queries, FSHELL's interactive mode features a shell-like history function which stores previous sessions in a file ".fshell2_history". The history is accessed using cursor up- and down keys or Ctrl+r.

Control Commands. The following control commands are supported by FSHELL:

- `HELP`: Display the syntax of all accepted commands as a grammar in Backus-Naur form (BNF).
- `QUIT` or `EXIT`: Leave FSHELL.
- `ADD SOURCECODE '`*f*`'`: Load and parse source file *f*. Equivalent to listing a source file on the command line, apart from the fact that parse errors here are not fatal. They result in an error message only and FSHELL does not abort.

  Optionally, additional C preprocessor defines may be given as `ADD SOURCECODE -D` *macro* `'`*f*`'` where *macro* is only set for loading and parsing *f*.
- `SHOW FILENAMES`: Display the list of loaded source files.
- `SHOW SOURCECODE '`*f*`'`, `SHOW SOURCECODE ALL`: Show the contents of a loaded file *f* or all loaded files, including line numbers.
- `SET ENTRY` *f*: Set program entry to function *f*. Equivalent to the command line option `--function` *f*.
- `SET LIMIT COUNT` *k*: Stop test case generation after at most *k* test cases for any given FQL query.

- `SET NO_ZERO_INIT`: The ANSI C standard mandates that objects with static lifetime are zero-initialized [Ame99c]. If, however, global variables should be treated as inputs for test case generation, e.g., because they are set through direct memory access in embedded systems code, this behavior is undesirable. With this command, zero-initializer are removed.

- `SET ABSTRACT` f: Make function f undefined, only retain its declaration. This command is useful to remove complex functions from the analysis, when their actual behavior can be considered irrelevant for test case generation.

- `SET MULTIPLE_COVERAGE` k: Attempt to perform the test case generation step $k \geq 1$ times for *a single* FQL query to obtain k test suites achieving the same coverage with *distinct* test suites. Later test suites may have fewer test cases and achieve less coverage, if some test goals can not be covered using different test data.

Defining Macros. FShell supports C-style macros to simplify the notation of complex or repeated query parts. Macro names are case-sensitive and defined like C macros using the `#define` directive. For instance

```
FShell2> #define st(s) @STMTTYPE(s)
FShell2> cover st(if) + st(assert) + st(while)
```

first defines an abbreviation for the filter function `@STMTTYPE`, which we then use to abbreviate the FQL query `cover @STMTTYPE(if) + @STMTTYPE(assert) + @STMTTYPE(while)`. Macro definitions remain valid for the duration of an FShell session.

6.3.3 Processing FQL Queries

If a user enters text that is neither recognized as a control command nor as a macro definition, FQL query processing is initiated. First macros are expanded and then an FQL query is sent to the FQL parser. The parser generates an abstract syntax tree (AST) for each valid FQL query or aborts with a syntax error if parsing fails.

If parsing the query succeeds a bounded model checking run is performed over the given program, unless this has already been done for some previous

query and no command to change the entry function or zero-initializers has been passed to FShell meanwhile (see Section 6.3.2). This bounded model checking run is necessary to rule out failing assertions, including failing unwinding assertions (cf. option `--no-unwinding-assertions` as described in Section 6.3.1). Any such failing assertion would subsequently cause wrong results in computing test cases: FShell could return paths that do not reach the end of the program, but instead lead to the violated assertion.

Internal Representation of FQL queries. As internal representation of FQL queries we use an irregular heterogeneous AST [Par09]. Implementing an irregular heterogeneous AST requires more initial implementation effort as each node type requires its own class, but permits very efficient traversal of such trees using the visitor pattern: each node type gets its own visitor function and calls of these functions are directly performed through the vtable without any conditional branching. In our implementation of the AST we use sharing of syntactically equivalent subgraphs, which turns the AST into a directed acyclic graph. Using the abstract factory and singleton design patterns [GHJV94] plus reference counting, this yields compact, efficient, and easy-to-use data structures. A similar implementation was used in [GHT09] for representation of regular expressions, where it proved to scale to millions of leaves. Even though we do not expect to handle queries of such size, this design proves useful as we can efficiently copy entire subtrees at the cost of only one additional pointer and create new AST objects at any point in the code without taking care of memory management – all objects will be reclaimed by the singleton factory.

Normalization. The AST generated by the FQL parser may still have an `in` T clause as described in Section 5.6. Normalization performs the steps described in Tables 5.10 and 5.11 to reduce the FQL query to an FQL specification `cover` C `passing` P as described in Section 5.5.

Furthermore simplifications for syntactic equivalences are performed as shown in Table 6.1. After normalization all further processing of FQL queries is safely restricted to FQL specifications of Section 5.5.

$$
\begin{aligned}
P_1 \texttt{ + } P_2 &= P_1 \text{ iff } P_1 \equiv P_2 \\
C_1 \texttt{ + } C_2 &= C_1 \text{ iff } C_1 \equiv C_2 \\
T_1 \texttt{|} T_2 &= T_1 \text{ iff } T_1 \equiv T_2 \\
T_1 \texttt{\&} T_2 &= T_1 \text{ iff } T_1 \equiv T_2
\end{aligned}
$$

Table 6.1: Simplification rules for syntactic equivalences

6.3.4 Running Example

We reconsider the example of Section 3.4 and use that code and query as a running example for the remainder of this chapter. In that section we had applied the query `cover @CONDITIONEDGE` to the source code of Listing 3.3. In Figure 6.4 we show the corresponding CFA $\mathcal{A}$, as it is built by the C front end.

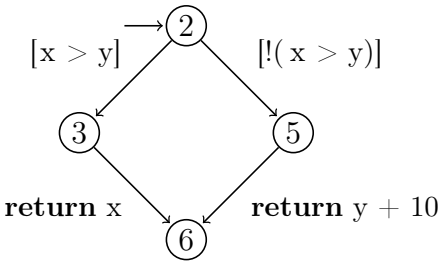


Figure 6.4: CFA $\mathcal{A}$ for Listing 3.3

Processing and normalization, as described in this section, of the query `cover @CONDITIONEDGE` yields the FQL specification

$$\Phi = \texttt{cover "EDGES(ID)*".EDGES(COMPOSE(@CONDITIONEDGE,ID)).}$$
$$\texttt{"EDGES(ID)*" passing EDGES(ID)*}$$

as we described in detail in Section 5.7.

6.4 Computing Target Graphs

Given the preparation of source code and an FQL query as explained in the preceding section we turn the focus to the back end as described in Figures 6.5 and 6.16. At this point we proceed with an annotated CFA $\mathcal{A}$,

which is internally represented as GOTO functions that were generated by CBMC's C front end. The original FQL query has been normalized to an FQL specification $\Phi =$ `cover` C `passing` P. As next step, filter functions and other CFA transformers must be evaluated on $\mathcal{A}$ according to the semantics specified in Section 5.4.2 and Table 5.4, respectively. Furthermore we handle predicates over program variables, which we denoted by the set S in the syntax and semantics described in Chapter 5, in the same step.

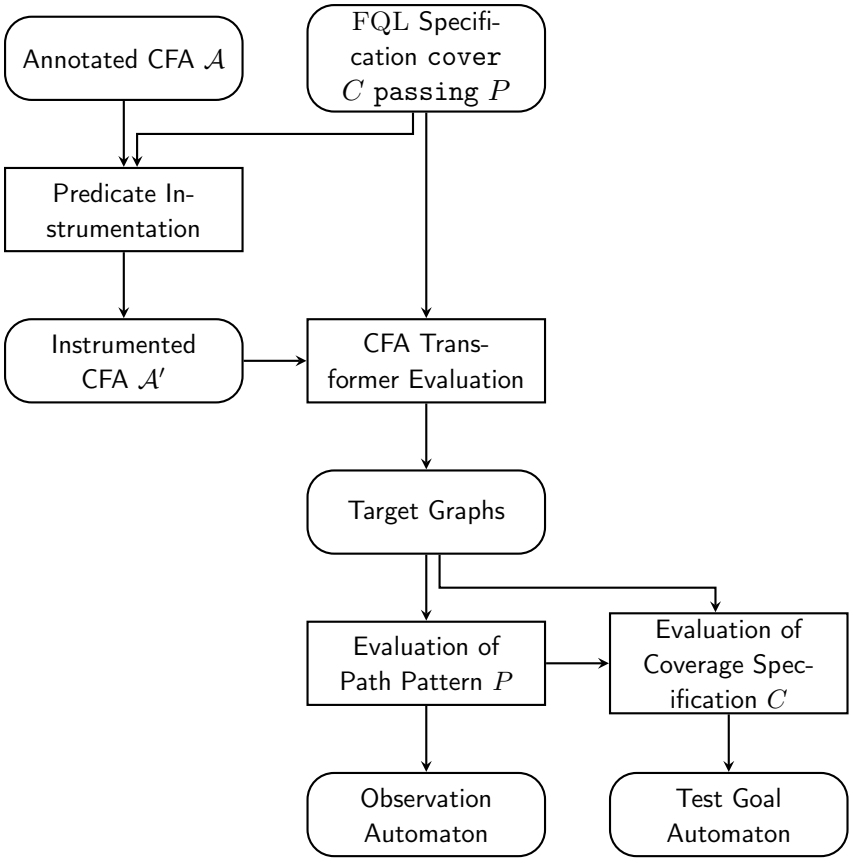


Figure 6.5: FShell back end architecture part I

In FShell we implement this evaluation using a post-order traversal of the query AST, again employing the visitor pattern. The design of our AST, as described above, furthermore facilitates an efficient implementation of caching, such that each expression T is only evaluated once.

We build a mapping $\mathsf{eval} : T \to \mathsf{CFA}$ that maps each filter function or CFA transformer T to a target graph $\mathsf{eval}(T)$. The post-order traversal guarantees that all target graphs of subexpressions have been computed such that CFA

transformers can be applied immediately. The resulting target graph is added to the mapping. If an expression T evaluates to a target graph with an empty set of locations, a warning is emitted:

`warning: Filter expression` T `evaluates to empty target graph`

It should be noted that at the time of writing the implementation of filter evaluation in FShell does not fully conform with the semantics detailed in Section 5.4.2, because annotations have not yet been completely implemented. As a result, the filter functions `@DECISIONEDGE` and `@CONDITIONGRAPH` are currently not supported and `@BASICBLOCKENTRY` possibly yields target graphs containing additional edges.

6.4.1 Example

We return to the running example that we introduced in Section 6.3.4. The FQL specification Φ that we obtained after normalization contains the CFA transformers `ID` and `COMPOSE`, and the filter function `@CONDITIONEDGE`. Cache lookups ensure that the CFA transformer `ID` will only be evaluated once over $\mathcal{A}$ despite occurring four times in this FQL specification. We get the following results for the mapping eval:

$$\begin{aligned}
&\mathsf{eval}(\texttt{ID}) &&= \langle\{\ell_2, \ell_3, \ell_5, \ell_6\},\\
&&&\quad \{e_{2,3}, e_{2,5}, e_{3,6}, e_{5,6}\}, \{\ell_2\}\rangle\\
&\mathsf{eval}(\texttt{@CONDITIONEDGE}) &&= \langle\{\ell_2, \ell_3, \ell_5\}, \{e_{2,3}, e_{2,5}\}, \{\ell_2\}\rangle\\
&\mathsf{eval}(\texttt{COMPOSE(@CONDITIONEDGE,ID)}) &&= \langle\{\ell_2, \ell_3, \ell_5\}, \{e_{2,3}, e_{2,5}\}, \{\ell_2\}\rangle
\end{aligned}$$

6.4.2 Predicates over Program Variables

FShell implements support for predicates via instrumentation of the original CFA. For each unique predicate $\{p\}$ occurring in C or P the corresponding C expression is inserted into the CFA $\mathcal{A}$ using the CFA shown in Figure 6.6. This snippet is inserted by replacing each original node with the new CFA. Each edge of the new fragment is annotated with instrumented to ensure these instrumented edges are not otherwise considered while evaluating filter functions.

This new CFA fragment has distinct successor locations of the initial location ℓ_1, depending on whether p evaluates to true (successor location ℓ_2) or false (ℓ_3) at the location where the fragment is inserted. Thereby the

semantic evaluation of a path pattern $\{p\}$ is shifted to a syntactic match of CFA edges; the semantic aspect is left to the underlying bounded model checking procedure.

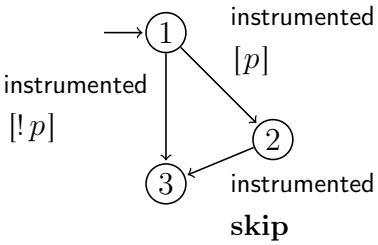


Figure 6.6: CFA snippet for $\{p\}$

In this instrumentation step the variable names occurring in p must be resolved to symbol names that are valid in the scope where the CFA snippet is inserted. If such name resolution fails, e.g., because a variable is only defined in a certain function but not globally, p is replaced by a nondeterministic value. For example, consider a query

```
cover {a > 10}.@CONDITIONEDGE.{x < 5}
```

with two predicates $a > 10$ and $x < 5$. If the query is applied on the code of Listing 3.3 of Section 3.4, where no variable named 'a' occurs in function `foo`, the first predicate will be replaced by a new nondeterministic value nd_1. Therefore two CFA snippets as shown in Figure 6.6 will be inserted for each node, one for the predicate nd_1, and one for the predicate $x < 5$.

Along with instrumentation FShell builds a mapping $\mathsf{pred} : S \to \mathsf{CFA}$ that realizes the translation to a syntactic match as discussed above. Each predicate $p \in S$ maps to the set of edges that result from inserting the edge $\langle \ell_2, \mathbf{skip}, \mathsf{instrumented}, \ell_3 \rangle$ of the CFA shown in Figure 6.6, i.e., $\mathsf{pred}(p) = \{\langle \ell_2, l, \ell_3 \rangle, \langle \ell'_2, l, \ell'_3 \rangle, \ldots\}$.

6.5 Trace Automata

In Chapter 5 we reduced FQL specifications to elementary coverage patterns and then gave a semantics in terms of formal languages. With the preparations of the preceding section, we are set to perform the reduction to an elementary coverage pattern for a given FQL specification Φ. A representa-

tion as a set of words, however, is inefficient and impractical in an operational setting.

Hence we use nondeterministic finite automata: we introduce *trace automata*, which formally model languages over path patterns. In FSHELL, however, we do not use the path predicate semantics of path patterns as FSHELL never explicitly builds the underlying transition system. Instead we will match *program traces* (hence also the name "trace automata") which are the result of unwinding the program as described in Section 6.1, and are formally defined in Section 6.5.2.

Target graphs can yield large sets of locations and edges. An automaton that generates path patterns from locations and edges would thus have an equally large number of transitions. In trace automata we therefore directly use target graphs and have transitions on target graphs.

Definition 6.1 Trace Automaton
A trace automaton $A = \langle Q, \Sigma, I, \Delta, F, G \rangle$ *is a nondeterministic finite automaton with a non-empty finite set of states* $Q = \{q_0, q_1, \ldots, q_n\}$*, where* $\Sigma \subseteq \mathsf{CFA} \cup \{\varepsilon\}$ *is an alphabet of target graphs,* $I \subseteq Q$ *is the set of initial states,* $\Delta \subseteq Q \times \Sigma \times Q$ *is the transition relation,* $F \subseteq Q$ *is the set of final states, and* $G \subset 2^Q \times \{\mathsf{one}, \mathsf{all}, \mathsf{none}\}$ *is a partitioning of* Q *that marks* test goal states.

The language $\mathcal{L}(A)$ *accepted by a trace automaton* A *is the set of path patterns generated by accepting runs of* A*, where a run is described by a sequence of transitions* $q_0 \xrightarrow{\mathcal{A}_0} q_1 \xrightarrow{\mathcal{A}_1} \ldots \xrightarrow{\mathcal{A}_n} q_{n+1}$ *with* $\mathcal{A}_i \in \Sigma$*. The run is accepting, iff* $q_0 \in I$ *and* $q_{n+1} \in F$*. The set of path patterns generated by this run is described by the elementary coverage pattern* $\mathcal{L}(\mathcal{A}_0) \,.\, \mathcal{L}(\mathcal{A}_1) \,.\, \ldots \,.\, \mathcal{L}(\mathcal{A}_n)$*, where* $\mathcal{L}(\mathcal{A}_i)$ *for a CFA* $\mathcal{A}_i = \langle L_{\mathcal{A}_i}, E_{\mathcal{A}_i}, I_{\mathcal{A}_i} \rangle$ *is again an elementary coverage pattern – the sum over all edges and additional locations:*

$$\mathcal{L}(\mathcal{A}_i) = \sum_{e \in E_{\mathcal{A}_i}} e + \sum_{\ell \in L_{\mathcal{A}_i} \wedge \forall \ell' \in L_{\mathcal{A}_i} . (\langle \ell, l, \ell' \rangle \notin E_{\mathcal{A}_i} \wedge \langle \ell', l, \ell \rangle \notin E_{\mathcal{A}_i})} \ell$$

For a query $\Phi =$ `cover` C `passing` P we can now describe translations to a pair of trace automata A_C and A_P for the coverage specification C and the path pattern P, respectively. We call A_C the *test goal automaton* and A_P the *observation automaton*.

The mapping G describing test goal states is relevant for the test goal automaton only. The reason for this mapping lies in the fact that we will

also expand quoted expressions "P" occurring in C to their corresponding trace automata but still need a means to recover the set of test goals from A_C. We use "one" to describe states where all incoming edges of a set of states $Q' \subseteq Q$ only yield *one* test goal, corresponding to a quoted expression in the coverage specification C. With "all" we denote states that result from other coverage specifications, where *each* incoming transition produces one test goal. "none" marks states that are not relevant for determining the set of test goals. We formalize this in the following definition.

Definition 6.2 Test Goals specified by Trace Automata
Each accepting run $q_0 \xrightarrow{\mathcal{A}_0} q_1 \xrightarrow{\mathcal{A}_1} \ldots \xrightarrow{\mathcal{A}_n} q_{n+1}$ *of a trace automaton* $A = \langle Q, \Sigma, I, \Delta, F, G \rangle$ *with* $q_0 \in I$ *and* $q_{n+1} \in F$ *induces a set of path patterns defining test goals if there exists a set of states* $Q' \subseteq Q$ *such that* $(Q', \mathsf{one}) \in G$ *or* $(Q', \mathsf{all}) \in G$ *and* $\{q_0, q_1, \ldots, q_{n+1}\} \cap Q' \neq \emptyset$.

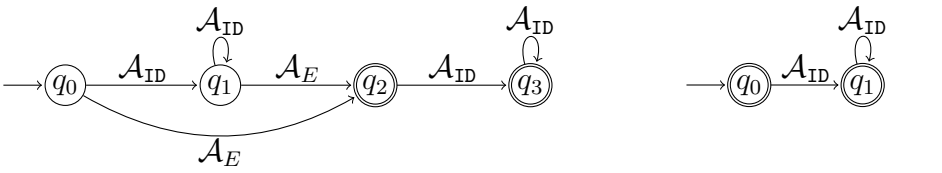
For such a run defining test goals, a transition $q_i \xrightarrow{\mathcal{A}_i} q_{i+1}$ *with* $q_{i+1} \in Q'$ *and* $(Q', \mathsf{all}) \in G$ *yields a* subgoal *for* each path pattern *in* $\mathcal{L}(\mathcal{A}_i)$. *Otherwise, if* $(Q', \mathsf{one}) \in G$ *or* $(Q', \mathsf{all}) \in G$, $\mathcal{L}(\mathcal{A}_i)$ *is* one *subgoal. The set of test goals of such a run is then computed as the Cartesian product of the set of subgoals.*

Each run induces a corresponding sequence of pairs of state sets/markers $\langle (Q_1, m_1), \ldots, (Q_k, m_k) \rangle$ *with* $(Q_j, m_j) \in G$ *and* $q_0 \in Q_1$, $\{q_j, \ldots, q_{j+l}\} \in Q_j$, *and* $q_{n+1} \in Q_k$. *Two accepting runs define the same test goals if they induce the same sequences of state sets and if the runs only differ in states of sets marked "*none*" or "*one*".*

Example. In Figure 6.7 we show the trace automata resulting from the FQL specification of Section 6.3.4. The edges are labeled with target graphs $\mathcal{A}_{\texttt{ID}}$ and $\mathcal{A}_E$ as computed using eval. With the results of Section 6.4.1 and the systematic construction described in the next section we have

$$\begin{aligned} \mathcal{A}_{\texttt{ID}} &= \langle \{\ell_2, \ell_3, \ell_5, \ell_6\}, \{e_{2,3}, e_{2,5}, e_{3,6}, e_{5,6}\}, \emptyset \rangle \\ \mathcal{A}_E &= \langle \{\ell_2, \ell_3, \ell_5\}, \{e_{2,3}, e_{2,5}\}, \emptyset \rangle \end{aligned}$$

The test goal automaton (Figure 6.7(a)) and the observation automaton (Figure 6.7(b)) are precisely specified as follows, which most importantly also



(a) Trace automaton for C (b) Trace automaton for P

Figure 6.7: Trace automata for running example

includes the mapping of test goal states:

$$
\begin{aligned}
A_C = \langle &\{q_0, q_1, q_2, q_3\}, \{\mathcal{A}_{\mathtt{ID}}, \mathcal{A}_E\}, \{q_0\}, \\
&\{(q_0, \mathcal{A}_{\mathtt{ID}}, q_1), (q_1, \mathcal{A}_{\mathtt{ID}}, q_1), (q_0, \mathcal{A}_E, q_2), (q_1, \mathcal{A}_E, q_2), \\
&\ (q_2, \mathcal{A}_{\mathtt{ID}}, q_3), (q_3, \mathcal{A}_{\mathtt{ID}}, q_3)\}, \\
&\{q_2, q_3\}, \{(\{q_0, q_1\}, \mathsf{one}), (\{q_2\}, \mathsf{all}), (\{q_3\}, \mathsf{one})\}\rangle \\
A_P = \langle &\{q_0, q_1\}, \{\mathcal{A}_{\mathtt{ID}}\}, \{q_0\}, \\
&\{(q_0, \mathcal{A}_{\mathtt{ID}}, q_1), (q_1, \mathcal{A}_{\mathtt{ID}}, q_1)\}, \{q_0, q_1\}, \{(\{q_0, q_1\}, \mathsf{none})\}\rangle
\end{aligned}
$$

6.5.1 Construction of Trace Automata

We will now describe the translation of FQL specifications to trace automata. By construction, these translations yield trace automata that describe the same set of path patterns and test goals as their corresponding elementary coverage patterns. In these translations we apply the functions $\mathsf{pred} : S \to \mathsf{CFA}$ and $\mathsf{eval} : T \to \mathsf{CFA}$ of Section 6.4 to obtain target graphs from predicates and CFA transformers, respectively. These target graphs are subgraphs of the CFA $\mathcal{A}' = \langle L', E', I' \rangle$ that is instrumented with predicate information.

Trace Automata for Quoted Regular Expression Operators. In the translation of coverage specifications to trace automata we refrain from using standard textbook translations that involve introducing ε-edges, because subsequent removal of such edges would require extra care for updating the test goal mapping G. For path patterns, however, we will use translations as described in standard textbooks, such as [HU79].

In the following we will use FQL specifications and their corresponding translations to trace automata interchangeably when the distinction is clear

from the context. Hence, when referring to, e.g., the test goal mapping G of the trace automaton resulting from translating a coverage specification C we use G_C as abbreviated notation. In figures we mark the trace automata resulting from preceding translations using sinuous lines.

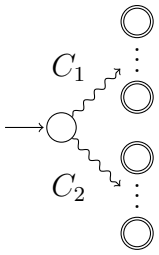


Figure 6.8: Trace automaton for C_1 + C_2

The translation of C_1 + C_2, as shown in Figure 6.8, merges the initial states of the translations of C_1 and C_2. As test goal states we have the union $G = G_{C_1} \cup (G_{C_2} \setminus (I_{C_2}, \mathsf{none}))$.

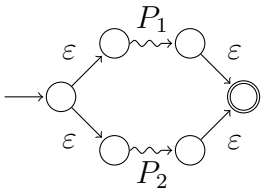


Figure 6.9: Trace automaton for P_1 + P_2

For path patterns P_1, P_2 we use a standard textbook translation shown in Figure 6.9, which introduces ε-edges instead of merging any nodes. As path patterns do not induce any test goals we have $G = \{(Q, \mathsf{none})\}$.

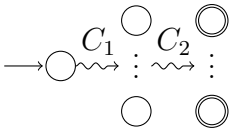


Figure 6.10: Trace automaton for $C_1 \,.\, C_2$

The concatenation of coverage specifications C_1 and C_2 is represented by the trace automaton shown in Figure 6.10. We start with the translation of C_1 to a trace automaton and copy all outgoing transitions of the initial

state of the translation of C_2 to each accepting state of the translation of C_1. The previous initial state I_{C_2} of C_2 is removed and accepting state markers of C_1 are reset. By construction, these operations do not affect the test goal mapping function G for states other than those marked "none". Therefore we have $G = G_{C_1} \cup (G_{C_2} \setminus (I_{C_2}, \mathsf{none}))$.

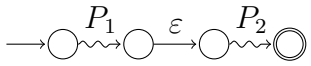


Figure 6.11: Trace automaton for $P_1 \,.\, P_2$

For path patterns we again use a standard textbook translation for concatenation, as we show in Figure 6.11. Note that our constructions guarantee that trace automata of path patterns always have exactly one initial state. The same holds true for the final state, apart from the case of `PATHS(`T`,`k`)`, which will be explained below. We again have $G = \{(Q, \mathsf{none})\}$.

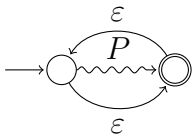


Figure 6.12: Trace automaton for P*

For Kleene star we again use a construction including ε-edges, as shown in Figure 6.12, to ensure we only have one final state. As this case is only relevant for path patterns, we have $G = \{(Q, \mathsf{none})\}$.

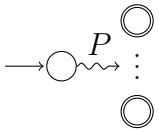
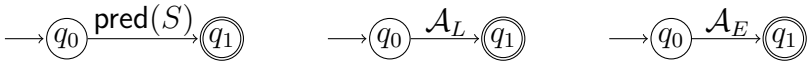


Figure 6.13: Trace automaton for "P"

For a quoted expression we first copy the automaton that we obtained for the path pattern P and transform it to an ε-edge free trace automaton. The resulting trace automaton in general may have fewer states, but possibly has more than one final state, as indicated in Figure 6.13. As the quote operator lifts path patterns to coverage specifications, we reset the test goal states: $G = \{(F, \mathsf{one}), (Q \setminus F, \mathsf{none})\}$.

Trace Automata for Basic Alphabets S and N. In the remaining translations we show trace automata for predicate expressions (alphabet S), and `NODES`, `EDGES` and `PATHS` (alphabet N). We will observe that pred and eval fully encapsulate the programming language specific details – with trace automata we deal with edges and nodes of a CFA only.

Note that these trace automata may become part of both trace automata for path patterns and trace automata for coverage specifications. We therefore eagerly define the test goal states G as necessary for coverage specifications; path patterns, as shown above, override these later on.



(a) Trace automaton for S

(b) Trace automaton for `NODES(`T`)`

(c) Trace automaton for `EDGES(`T`)`

Figure 6.14: Trace automata for basic alphabets other than `PATHS`

As shown in Figure 6.14(a), for any predicate we use the set of edges that pred yields as the target graph of a new transition. Analogously we create new transitions for `NODES(`T`)` and `EDGES(`T`)` as shown in Figures 6.14(b) and 6.14(c), respectively. In these two cases, however, we use eval to obtain the target graphs. We define $\mathcal{A}_L = \langle L_{\mathsf{eval}(T)}, \emptyset, \emptyset \rangle$ for `NODES(`T`)`, i.e., we only use the locations of the target graph obtained by $\mathsf{eval}(T)$. For `EDGES(`T`)` we get $\mathcal{A}_E = \langle L', E_{\mathsf{eval}(T)}, \emptyset \rangle$ where $L' = \{\ell \mid \langle \ell, l, \ell' \rangle \in E_{\mathsf{eval}(T)} \vee \langle \ell', l, \ell \rangle \in E_{\mathsf{eval}(T)}\}$, i.e., we use the edges of the target graph $\mathsf{eval}(T)$ and restrict the locations to those occurring in these edges. For all three cases of Figure 6.14 we define $G = \{(\{q_0\}, \mathsf{none}), (\{q_1\}, \mathsf{all})\}$

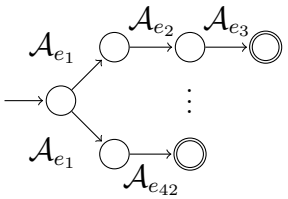


Figure 6.15: Trace automaton for `PATHS(`T`,`k`)`

For `PATHS(`T`,`k`)` we only sketch a possible result in Figure 6.15. The precise result is built by using the construction from C_1 + C_2 (Figure 6.8)

for

$$\texttt{PATHS}(T,k) = \sum_{p \in \mathsf{paths}_k(\mathsf{eval}(T))} p$$

and apply the translation for $C_1 . C_2$ (Figure 6.10) for each path p, which is a sequence of edges. As a result we have one target graph for each such edge, which in Figure 6.15 we denote by $\mathcal{A}_{e_1}$ for an edge e_1. As test goal states we define $G = \{(F, \mathsf{all}), (Q \setminus F, \mathsf{none})\}$.

We initially mark the last states of each such path as accepting states. In case a trace automaton of $\texttt{PATHS}(T,k)$ is used in the translation of a path pattern, additional ε-edges are added at the end of each path leading to a new common final state. Thereby again we arrive at the case of only one final state, which simplifies the construction of trace automata of path patterns as described above.

6.5.2 Program Traces

We want to use trace automata to match program executions. Hence our main interest does not lie in the words or language they generate, but how words are accepted.

These words are given as programs executions, which we describe as sequences of CFA edges. We call these descriptions *program traces*. We distinguish between *abstract traces* and *concrete traces*: abstract traces are linearized and unwound abstractions of CFAs, i.e., an abstract trace contains all possible edge sequences of concrete executions such that any actual program run is described by a subsequence of the abstract trace. A concrete trace induces such a subsequence by all those edges that map to true under a mapping $\iota_{t_\mathcal{A}}$ for an abstract trace $t_\mathcal{A}$.

Definition 6.3 Abstract and Concrete Traces
Bounded unwinding of a CFA $\mathcal{A}$ yields an abstract trace $t_\mathcal{A} = \langle e_0 e_1 \dots e_l \rangle$ *of length $l+1$ with $e_i \in E_\mathcal{A}$. A* concrete trace *is an abstract trace $t_\mathcal{A}$ with an additional mapping $\iota_{t_\mathcal{A}} : E_\mathcal{A} \to \{\mathsf{true}, \mathsf{false}\}$.*

A trace automaton A *accepts* (or matches) a concrete trace if the word induced by the subsequence of edges mapping to true under $\iota_{t_\mathcal{A}}$ is in the language of the path patterns (cf. Table 5.2) generated by A.

Propositional Encoding. CBMC transforms a CFA $\mathcal{A}$ first into an abstract trace $t_{\mathcal{A}}$ and then into a CNF formula, which we denote by $\phi[t_{\mathcal{A}}]$. This CNF formula encompasses an encoding of the mapping $\iota_{t_{\mathcal{A}}}$. A model of the formula therefore describes a concrete trace, and hence a program execution. As explained in Section 6.1, each model corresponds to an execution violating one of the assertions.

CBMC maintains a mapping between $t_{\mathcal{A}}$ and the literals of $\phi[t_{\mathcal{A}}]$. In the constructions that we will use in the following sections this mapping allows us to identify the Boolean variables that express whether an edge or location is reached in a given program execution (a concrete trace with a mapping $\iota_{t_{\mathcal{A}}}$). For an edge e (a location ℓ) we write $\iota_{t_{\mathcal{A}}}(e)$ ($\iota_{t_{\mathcal{A}}}(\ell)$) to express the fact that an edge e (a location ℓ) is reached.

6.6 Integrating Trace Automata

Given an operational description of FQL specifications in terms of automata we need to build an analysis that computes test cases following this specification. Our goal is to map the instrumented CFA $\mathcal{A}$' *and* the trace automata A_C and A_P obtained from the FQL specification Φ to a SAT instance. Using this SAT instance we will perform efficient enumeration of test cases as described in Section 6.7.

We will describe two approaches to obtain such a SAT instance, which adhere to the same interface as indicated by the box in Figure 6.16. The first one, which is described in Section 6.6.1, is based on additional instrumentation of the CFA $\mathcal{A}'$ with code implementing the two trace automata. Here, only test goal states are directly encoded into the CNF formula. In the second approach we encode the entire trace automata into a propositional formula without prior instrumentation. This technique will be described in Section 6.6.2.

6.6.1 Program Instrumentation

We want to use trace automata as a means of specification. CBMC, however, only supports assertions. We therefore instrument the program with a trace automaton such that the resulting program reaches a failing assertion in the course of an execution if, and only if, this program execution – a concrete trace – is matched by the trace automaton. Hence we implement a trace

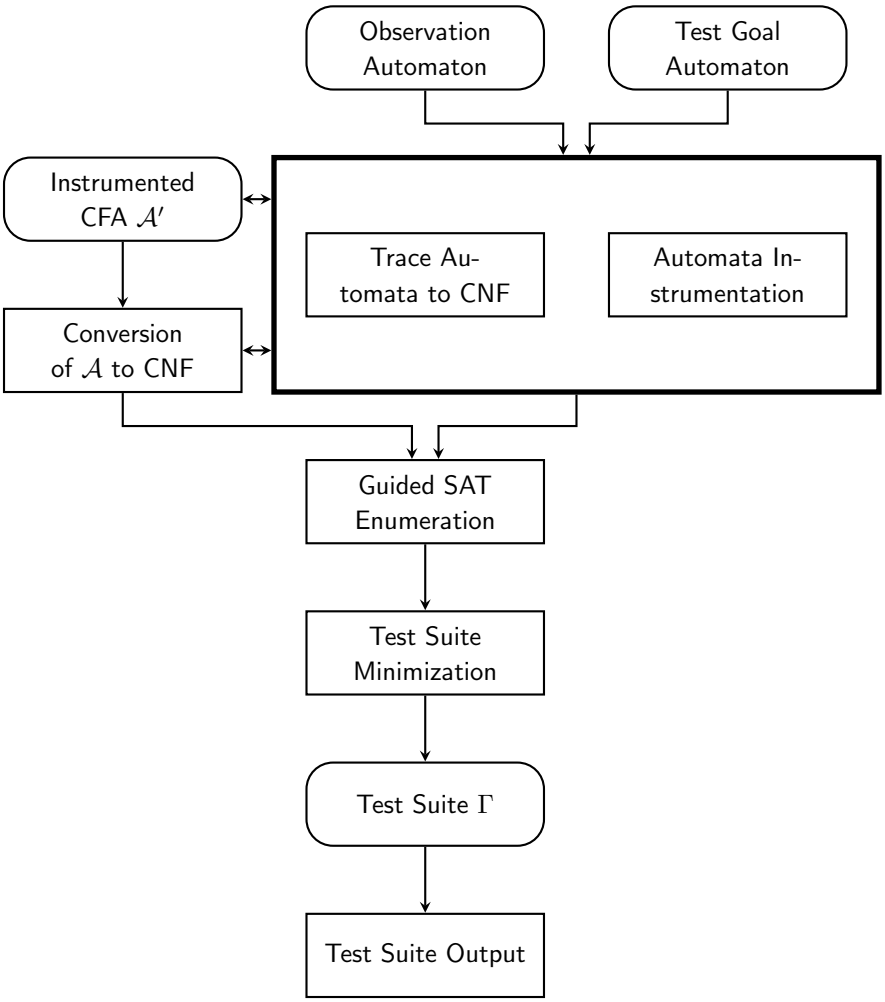


Figure 6.16: FShell back end architecture part II

automaton as a *monitor* of program execution. This monitor is bound to a program via a *logging layer*. The monitor component implements transitions and is called via the logging functions in order to perform such transitions. At the end of the program, i.e., after returning from the program entry function, we add an assertion that requires that the trace automaton has *not* reached any accepting state. As a result, CBMC will try to find counterexamples that violate this assertion, i.e., CBMC computes program executions that reach an accepting state.

Let $A = \langle Q, \Sigma, I, \Delta, F, G \rangle$ be a trace automaton. Program instrumentation proceeds as follows both for the test goal automaton and the observation automaton to build a fully instrumented CFA $\mathcal{A}''$ from $\mathcal{A}'$.

1. Create a global integer variable $state_A$ that stores the current state of the automaton A. The bit-width of this integer variable is set to $\lceil \log_2(|Q|) \rceil$ to use the minimal number of bits encoding the state number.

2. Map each target graph $\mathcal{A}_i \in \Sigma$ to an integer i which we will use as identifier.

3. Walk the locations of the CFA $\mathcal{A}'$ to build up the logging layer. At each location ℓ first check for target graphs $\mathcal{A}_j \in \Sigma$ matching this *location*, i.e., $\ell \in \mathcal{L}(\mathcal{A}_j)$. If there is no such target graph, skip location instrumentation and proceed to edges as described below. Let $\{\mathcal{A}_{i_1}, \ldots, \mathcal{A}_{i_k}\}$ be a set of target graphs matching the current location, i.e., $\ell \in \mathcal{L}(\mathcal{A}_{i_j})$ for all $1 \leq j \leq k$. We may have to match *all of them* and accordingly perform several transitions of the automaton. Thereby we account for the fact that state predicates, which such target graphs are, require repeatedly matching the same program location per the definition of predicate concatenation in Section 4.3. In this case the trace automaton makes several transitions without consuming more of the program execution, while for edges the trace automaton and program execution proceed in a lockstep manner. In order to implement this behavior, we repeatedly nondeterministically log one of these matches using function calls `filter_trans_`A`_`i_j`()` guarded by fresh Boolean variables `nd`j. We insert the GOTO-equivalent of the C code shown in Listing 6.1 r times, where r is the number of target graphs in Σ matching only locations instead of edges. This yields a program that is capable of simulating the above behavior of possibly matching all target graphs.

```
1 {
2   _Bool nd1;
3   _Bool nd2;
4   ...
5   _Bool ndk;
6   if (nd1)
7     filter_trans_A_i1();
8   else if (nd2)
9     filter_trans_A_i2();
10  ...
11  else if (ndk)
12    filter_trans_A_ik();
13 }
```

Listing 6.1: Logging layer

After location matching we check for target graphs $\mathcal{A}_j \in \Sigma \cup \mathsf{eval}(\texttt{ID})$ matching one of the *outgoing edges* e of location ℓ, i.e., $e = \langle \ell, l, \ell' \rangle \in \mathcal{L}(\mathcal{A}_j)$. All edges other than those added by prior instrumentation (marked by the annotation instrumented) will at least be matched by $\mathsf{eval}(\texttt{ID})$. We insert one copy of the code already shown in Listing 6.1 with the modification that the last choice is deterministic, i.e., `else if (ndk)` becomes an "`else`" only. Consequently, if only one matching target graph is found, no nondeterministic branching is used and `filter_trans_`A`_`j is called.

If the location ℓ has two outgoing edges e_1 and e_2, these are necessarily assume-edges of the CFA. In this case the inserted code blocks must be guarded with the appropriate condition. To this end, the guarding condition of each assume-edge is copied. This is safe as the construction performed by CBMC guarantees that these conditions are side-effect free.

4. Add the monitor component. Each function `filter_trans_`A`_`j, as called by the logging part shown in Listing 6.1, computes a fraction of the transition relation of the trace automaton A: depending on the current state q, as stored in $state_A$, `filter_trans_`A`_`j performs one of the transitions $q \xrightarrow{\mathcal{A}_j} q'$ such that $(q, \mathcal{A}_j, q') \in \Delta$.

   We show an example source code of the function `filter_trans_`A`_`j in Listing 6.2. For each state q that has outgoing transitions on $\mathcal{A}_j$

```
1 void filter_trans_A_j() {
2   switch(state_A) {
3     case q: {
4       _Bool nd1;
5       _Bool nd2;
6       if (nd1)
7         state_A = q_a;
8       else if (nd2)
9         state_A = q_b;
10      else
11        state_A = q_c;
12      break;
13    }
15    case q': {
16      _Bool nd1;
17      if (nd1)
18        state_A = q_x;
19      else
20        state_A = q_y;
21      break;
22    }
23    case q'':
24      break;
25    default:
26      assume(0);
27  }
28 }
```

Listing 6.2: Monitor function

we add a nondeterministic choice over its possible successor states. Furthermore we handle the case that `filter_trans_`A_j is called in a state q that has no outgoing transitions on $\mathcal{A}_j$ with `assume(false)`. Such an assumption terminates (symbolic) execution of the path beyond this point, making the final assertion unreachable. Therefore CBMC will not compute such paths. This is formally equivalent to setting $state_A$ to some additional (and hence dead) state q_{-1}, but more efficient.

5. For $F = \{q_{i_1}, \ldots, q_{i_k}\}$ we generate the assertion

   assert (!($state_A == q_{i_1}$ || ... || $state_A == q_{i_k}$));

   which is violated if $state_A$ equals one of the final states of the trace automaton upon reaching the assertion at the end of the program.

The preceding steps guarantee that a test case is an accepting run for the trace automaton A. As we perform this instrumentation both for the test goal automaton and the observation automaton, and we want each test case to be an accepting run for both of the automata we combine the two assertions generated in the last instrumentation step into

assert (!(($state_{A_C} == q_{i_1}$ || ...) && ($state_{A_P} == q_{j_1}$ || ...)));

to assert the combined property.

Example. In Listing 6.3 we present the ANSI C equivalent of the instrumented GOTO functions for the program originally shown in Listing 3.3. We observe a blowup from a six-line program in Listing 3.3 to 61 lines, plus initialization code and the assertion in the `main` function. We study the overhead in the following paragraph.

Overhead of Program Instrumentation. The above instrumentation steps impose a considerable overhead in program size, resulting in a larger abstract trace. We formalize this in the following proposition.

Proposition 6.4 *The number of edges of the abstract trace $t_{\mathcal{A}''}$, denoted by $|t_{\mathcal{A}''}|$, develops polynomially in the number of transitions and the size of the alphabet of the trace trace automata A_C and A_P for any given abstract trace $t_{\mathcal{A}'}$ before instrumentation. We have*

$$|t_{\mathcal{A}''}| \in \mathcal{O}\big((|\Delta_{A_C}| \cdot |\Sigma_{A_C}| + |\Delta_{A_P}| \cdot |\Sigma_{A_P}|) \cdot |t_{\mathcal{A}'}|\big).$$

Proof. In instrumentation as described above we only consider locations and edges of $\mathcal{A}$', i.e., edges not added by a previous automaton instrumentation step. We can therefore consider the overhead of instrumentation for each of A_C and A_P independently. Let $A = \langle Q, \Sigma, I, \Delta, F, G\rangle$ be a trace automaton. For each location of the CFA $\mathcal{A}'$ we insert the logging code at most $|\Sigma|$ times. For outgoing edges we insert one logging block for each edge. As there are at most two such edges we have a bound of $|\Sigma| + 2$, or $\mathcal{O}(|\Sigma|)$. This insertion may occur at each location, resulting in $|t_{\mathcal{A}'}| \cdot \mathcal{O}(|\Sigma|)$ insertions occurring in the abstract trace.

It remains to determine the size of each inserted block. A location or edge may belong to all target graphs of Σ, resulting in $|\Sigma|$ nondeterministic choices in the code of Listing 6.1. Each choice induces a unique function call of the monitor code. The abstract trace has inlined versions of all these functions, amounting to a total of $\mathcal{O}(|\Sigma| + |\Delta|)$ edges for each inserted block. By construction of our trace automata we have $|\Sigma| \leq |\Delta|$.

For the trace automaton A we thus get a bound

$$|t_{\mathcal{A}'}| \cdot \mathcal{O}(|\Sigma|) \cdot \mathcal{O}(|\Delta|) \in \mathcal{O}(|\Delta| \cdot |\Sigma| \cdot |t_{\mathcal{A}'}|).$$

Summing up these bounds for A_C and A_P completes the proof. □

```
1 int state_A_C, state_A_P;

3 void filter_trans_A_C_1() {
4   switch(state_A_C) {
5     case 0: state_A_C = 1;
6       break;
7     case 1: state_A_C = 1;
8       break;
9     case 2: state_A_C = 3;
10      break;
11    case 3: state_A_C = 3;
12      break;
13    default:
14      assume(0);
15  }
16 }
17 void filter_trans_A_C_2() {
18  switch(state_A_C) {
19    case 0: state_A_C = 2;
20      break;
21    case 1: state_A_C = 2;
22      break;
23    default:
24      assume(0);
25  }
26 }
27 void filter_trans_A_P_1() {
28  switch(state_A_P) {
29    case 0: state_A_P = 1;
30      break;
31    case 1: state_A_P = 1;
32      break;
33    default:
34      assume(0);
35  }
36 }

38 int foo(int x, int y) {
39  if (x > y) {
40    _Bool nd1;
41    if (nd1) filter_trans_A_C_1();
42    else filter_trans_A_C_2();
43  } else {
44    _Bool nd1;
45    if (nd1) filter_trans_A_C_1();
46    else filter_trans_A_C_2();
47  }
48  if (x > y)
49    filter_trans_A_P_1();
50  else
51    filter_trans_A_P_1();
52  if (x > y) {
53    filter_trans_A_C_1();
54    filter_trans_A_P_1();
55    return x;
56  } else {
57    filter_trans_A_C_1();
58    filter_trans_A_P_1();
59    return y + 10;
60  }
61 }

64 void main() {
65  state_A_C = 0;
66  state_A_P = 0;
67  int x, y;
68  foo(x,y);
69  assert (!(( state_A_C == 2 ||
70              state_A_C == 3)
71             && (state_A_P == 0 ||
72              state_A_P == 1)));
73 }
```

Listing 6.3: Instrumented version of Listing 3.3

We note that the real overhead may be smaller on some instances as trivially unreachable edges will not occur in the abstract trace computed by CBMC. Yet we do observe the overhead being this large in general. On some randomly picked examples with $|\Sigma_{A_C}| = |\Sigma_{A_P}| = 1$ and $|\Delta_{A_P}| = 2$, varying only Δ_{A_C}, we have:

- $|t_{\mathcal{A}'}| = 29$ (a small program with ten lines of code), but observe $|t_{\mathcal{A}''}| = 787$ for $|\Delta_{A_C}| = 6$; $|t_{\mathcal{A}''}| = 873$ for $|\Delta_{A_C}| = 7$; and $|t_{\mathcal{A}''}| = 1103$ for $|\Delta_{A_C}| = 10$.
- $|t_{\mathcal{A}'}| = 428$ (a part of joplift.c; joplift.c is described in Table 7.1). We find $|t_{\mathcal{A}''}| = 13688$ for $|\Delta_{A_C}| = 6$; $|t_{\mathcal{A}''}| = 15154$ for $|\Delta_{A_C}| = 7$; and $|t_{\mathcal{A}''}| = 19064$ for $|\Delta_{A_C}| = 10$.

For our running example we have $|t_{\mathcal{A}'}| = 14$, $|\Sigma_{A_C}| = 2$, $|\Delta_{A_C}| = 6$, $|\Sigma_{A_P}| = 1$, and $|\Delta_{A_P}| = 1$. We observe a blowup factor of almost 18, to $|t_{\mathcal{A}''}| = 251$.

The overhead for the SAT formula built from the abstract trace, however, is by far not as large – we see factors of 3 or 4, both for the number of variables and the number of clauses, where we had factors of up 44 in the above examples for the overhead in the abstract trace. As a result of the careful design with variables using as few bits as possible, we observe that our additional code permits a very efficient encoding, but the increase for the abstract trace may still be prohibitive. We therefore describe an alternative technique in Section 6.6.2.

Handling Test Goals. Our instrumentation guarantees that any concrete trace computed by CBMC violates the assertion added in the last of the above instrumentation steps, i.e., we reach accepting states of both A_C and A_P within a bounded number of steps. For the test goal automaton, however, we furthermore need to handle test goal states. We do so at the level of the CNF formula.

As described in Section 6.5.2, CBMC translates the fully instrumented CFA $\mathcal{A}''$ to the CNF formula $\phi[t_{\mathcal{A}''}]$, which includes an encoding of the mapping $\iota_{t_{\mathcal{A}''}}$. The test goal states described by G therefore translate to the following:

- For each state $q \in Q'$ with $(Q', \mathsf{all}) \in G$ we first identify all edges $E_q = \{e_1, \ldots, e_k\}$ of $\mathcal{A}''$ where the assignment $state_{A_C} = q$ of Listing 6.2 was inserted. Reaching such an edge amounts to reaching state q. An edge

$e_i \in E_q$ is necessarily reached via a logging call `filter_trans_`A_C`_`j on one of the edges $\{f_{i_1}, \ldots, f_{i_{m_q}}\}$. Each such logging call together with reaching e_i then constitutes one subgoal (cf. Definition 6.2) g_{i_k}, which we construct as follows: the abstract trace $t_{\mathcal{A}''}$ contains multiple copies of each CFA edge as a result of unwinding the CFA and inlining functions. For an edge e we name these $\{e^1, \ldots, e^l\}$. Each edge e_i^j has a corresponding set of logging calls $\{f_{i_1}^j, \ldots f_{i_{m_q}}^j\}$. Subgoals g_{i_k} for $1 \leq k \leq m_q$ are defined by

$$g_{i_k} \equiv \bigvee_{1 \leq j \leq l} \iota_{t_{\mathcal{A}''}}(e_i^j) \wedge \iota_{t_{\mathcal{A}''}}(f_{i_k}^j).$$

We define the set of all goals induced by Q' as a function $\mathsf{goals}(Q')$ with

$$\mathsf{goals}(Q') = \bigcup_{q \in Q'} \left\{ \bigvee_{e_i \in E_q} g_{i_k} \mid 1 \leq k \leq m_q \right\}.$$

- For $q \in Q'$ with $(Q', \mathsf{one}) \in G$ we again collect all edges E_q as described above. In this case, however, we only get a single subgoal and $\mathsf{goals}(Q')$ returns a singleton set with

$$\mathsf{goals}(Q') = \left\{ \bigvee_{q \in Q'} \bigvee_{e_i \in E_q} \bigvee_{1 \leq j \leq l} \iota_{t_{\mathcal{A}''}}(e_i^j) \right\}.$$

We obtain the set of all test goals per Definition 6.2 by inspecting all accepting runs of A_C. The states of an accepting run induce a sequence $\langle (Q_1, m_1), \ldots, (Q_k, m_k) \rangle$ with $(Q_i, m_i) \in G$ where $m_i \in \{\mathsf{one}, \mathsf{all}, \mathsf{none}\}$. For (Q_i, m_i) with $m_i \neq \mathsf{none}$ we use the above function $\mathsf{goals}(Q_i)$ to obtain sets of Boolean variables. For (Q_j, none) we define $\mathsf{goals}(Q_j) = \{\mathsf{true}\}$ and set the set of test goals described by A_C to

$$\Phi(\mathcal{A}) = \{\Psi^1 \wedge \Psi^2 \wedge \ldots \wedge \Psi^k \mid \Psi^i \in \mathsf{goals}(Q_i) \text{ with } 1 \leq i \leq k\}. \tag{6.1}$$

Remark 6.5 *The above construction of test goals via the Cartesian product yields a set of a size being exponential in k.*

For most practical queries k is very small (≤ 5), but still the eager construction of test goals as shown above yields an avoidable overhead: in Section 6.7 we introduce groupwise constraint strengthening to avoid this explicit and eager Cartesian construction.

Example. In Section 6.5.1 we obtained A_C with

$$G = \{(\{q_0, q_1\}, \mathsf{one}), (\{q_2\}, \mathsf{all}), (\{q_3\}, \mathsf{one})\}.$$

The function goals is therefore defined as follows, using line numbers of the code in Listing 6.3 to hint at corresponding edges:

$$\mathsf{goals}(\{q_0, q_1\}) = \left\{ \iota_{t_{\mathcal{A}''}}(e^1_{65}) \vee \bigvee_{1 \le j \le 4} \iota_{t_{\mathcal{A}''}}(e^j_5) \vee \bigvee_{1 \le j \le 4} \iota_{t_{\mathcal{A}''}}(e^j_7) \right\}$$

$$\mathsf{goals}(\{q_2\}) = \left\{ \begin{array}{l} \bigvee_{1 \le j \le 2} \left(\iota_{t_{\mathcal{A}''}}(e^j_{19}) \wedge \iota_{t_{\mathcal{A}''}}(e^j_{42})\right) \vee \bigvee_{1 \le j \le 2} \left(\iota_{t_{\mathcal{A}''}}(e^j_{21}) \wedge \iota_{t_{\mathcal{A}''}}(e^j_{42})\right), \\ \bigvee_{1 \le j \le 2} \left(\iota_{t_{\mathcal{A}''}}(e^j_{19}) \wedge \iota_{t_{\mathcal{A}''}}(e^j_{46})\right) \vee \bigvee_{1 \le j \le 2} \left(\iota_{t_{\mathcal{A}''}}(e^j_{21}) \wedge \iota_{t_{\mathcal{A}''}}(e^j_{46})\right) \end{array} \right\}$$

$$\mathsf{goals}(\{q_3\}) = \left\{ \bigvee_{1 \le j \le 4} \iota_{t_{\mathcal{A}''}}(e^j_9) \vee \bigvee_{1 \le j \le 4} \iota_{t_{\mathcal{A}''}}(e^j_{11}) \right\}$$

The Cartesian product of these sets of subgoals then results in two test goals $\Phi(\mathcal{A}) = \{\Psi_1, \Psi_2\}$.

6.6.2 Propositional Encoding of Trace Automata

We showed in Proposition 6.4 that translation of trace automata through program instrumentation imposes a polynomial overhead. We furthermore noticed that this overhead is indeed observed on abstract traces, resulting in prohibitively large abstract traces. At the same time, however, we found that the SAT instance that CBMC builds from these abstract traces is, whenever such a construction is still feasible, only enlarged by a factor being an order of magnitude smaller.

Thus we propose to replace the instrumentation of trace automata by a direct encoding as a propositional formula. While being more low-level in terms of what objects we manipulate within CBMC's components, and therefore possibly harder to realize and debug, we find two definitive advantages in this step, which are also confirmed by the experiments presented in Section 7.4:

1. The overhead of larger abstract traces is completely avoided.

2. The propositional encoding will even further reduce the overhead in the size of the SAT formula. One of the main reasons here is that nondeterministic choices can be encoded as a Boolean disjunction instead of introducing new variables.

Trace automata, as specified in Definition 6.1, define regular languages over locations and edges. The classical Büchi-Elgot-Trakhtenbrot Theorem [Büc60, Elg61, Tra62] states that regular languages and monadic second order logic (MSO) have the same expressive power and that the translation between these formalisms is effective. In our case we are only concerned with the translation of a nondeterministic finite automaton (NFA) into MSO. Let $A = \langle \{q_0, \ldots, q_k\}, \Sigma, q_0, \Delta, F \rangle$ be an NFA. The property $w \in \mathcal{L}(A)$ translates to $w \models \varphi$ with φ defined as follows:

$$
\begin{aligned}
\varphi &\equiv \exists X_0, \ldots, X_k . \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4 \\
\varphi_1 &\equiv \bigwedge_{0 \leq i \neq j \leq k} \forall x . \neg (X_i(x) \wedge X_j(x)) \\
\varphi_2 &\equiv \forall x . \neg \exists y . S(y, x) \rightarrow X_0(x) \\
\varphi_3 &\equiv \forall x, y . S(x, y) \rightarrow \bigvee_{(i,a,j) \in \Delta} (X_i(x) \wedge P_a(x) \wedge X_j(y)) \\
\varphi_4 &\equiv \forall x . \neg \exists y . S(x, y) \rightarrow \bigvee_{(i,a,j) \in \Delta \wedge j \in F} (X_i(x) \wedge P_a(x))
\end{aligned}
$$

Here, $S(x, y)$ is the successor predicate over natural numbers (including zero). $X_0, \ldots, X_k$ are second order variables with the intended semantics that $X_j(x)$ is true if and only if A is in state q_j after reading position x in word w. Each letter $a \in \Sigma$ induces a first order predicate P_a with $P_a(x)$ evaluating to true if and only if there is an 'a' at position x in word w.

As our intention, however, is to use a SAT solver, we have to come up with a propositional encoding instead of a formula in second order logic. To this end we first observe that the length of words we have to process by a trace automaton $A = \langle Q, \Sigma, I, \Delta, F, G \rangle$ is bounded by $l = |t_{\mathcal{A}'}| \cdot |\Sigma|$: given an abstract trace $t_{\mathcal{A}'}$ we have to process each location at most $|\Sigma|$ times as discussed in Section 6.6.1. From boundedness of word length it follows that the resulting language has at most 2^l words and can be encoded as a propositional formula – at least the naïve encoding exists, where we represent the language as a disjunction over these words.

For a more efficient encoding we follow the above translation according to Büchi-Elgot-Trakhtenbrot and as first step describe an encoding which still

uses first order predicates $P_{\mathcal{A}}$. Given the upper bound on the word length l we arrive at the following formula:

$$\begin{aligned}
\varphi &\equiv \exists x_0^0, \ldots, x_l^1, \ldots, x_0^k, \ldots, x_l^k . \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4 \\
\varphi_1 &\equiv \bigwedge_{0 \leq j \leq l} \bigwedge_{0 \leq m \neq n \leq k} \neg(x_j^m \wedge x_j^n) \\
\varphi_2 &\equiv \bigvee_{q_i \in I} x_0^i \\
\varphi_3 &\equiv \bigwedge_{0 \leq j < l} \bigvee_{(q_m, \mathcal{A}, q_n) \in \Delta} (x_j^m \wedge P_{\mathcal{A}}(j+1) \wedge x_{j+1}^n) \\
\varphi_4 &\equiv \bigvee_{q_j \in F} x_l^j
\end{aligned}$$

We replaced each second order variable X_j, which represented sets of states, by $l+1$ Boolean variables $x_{0\ldots l}^j$ with $X_j(y) \Leftrightarrow x_y^j$. This encoding introduces $(l{+}1){\cdot}(k{+}1)$ existentially quantified Boolean variables. We can, however, also get an encoding that only uses $(l{+}1){\cdot}\lceil \log_2(k{+}1) \rceil$ bits, at the expense of a new first order predicate: we replace $x_i^{0\ldots k}$ by a single bit-vector s_j of $\lceil \log_2(k{+}1) \rceil$ bits and a new first order predicate $Q_j(x)$ that is `true` whenever x is the bit-vector encoding of j. By definition of $Q_j(x)$ the formula $\neg(Q_m(s_j) \wedge Q_n(s_j))$ is a tautology for $m \neq n$. As an additional improvement, φ_1 of the above formula can therefore be omitted.

Implementation. In Algorithm 6.1 we present the algorithm that incrementally builds the formula φ, as described above, for an abstract trace $t_{\mathcal{A}''}$ and a trace automaton $A = \langle Q, \Sigma, I, \Delta, F, G \rangle$. We traverse the abstract trace $t_{\mathcal{A}''}$, which we obtain by first adding the trivial assertion

```
assert (0);
```

at the end of CBMC's `main` function to ensure that there is some assertion. This is similar to the instrumentation of the assertion described in Section 6.6.1, but without specifying any final states of the trace automata. Enforcing program executions that reach final states will be solely handled by the propositional encoding.

We incrementally build the formula φ in ϕ. We initialize ϕ in line 2, which resembles φ_2.

By traversing the abstract trace (lines 3–18) we can evaluate the predicate $P_{\mathcal{A}}$ of φ_3 to translate it into a property using the literals that encode $\iota_{t_{\mathcal{A}''}}$.

input: $t_{\mathcal{A}''}$, $A = \langle Q, \Sigma, I, \Delta, F, G \rangle$

1 $j := 0$
2 $\phi := \bigvee_{q_i \in I} Q_i(s_j)$
3 **foreach** $e = \langle \ell, op, an, \ell' \rangle \in t_{\mathcal{A}''}$ **do**
4 $\quad TG_L := \{\mathcal{A} \mid \mathcal{A} \in \Sigma \wedge \ell \in \mathcal{L}(\mathcal{A})\}$
5 $\quad$ **for** $i := 0$ **to** $i < |TG_L|$ **do**
6 $\quad\quad j := j + 1$
7 $\quad\quad \phi' := \bigvee_{\mathcal{A} \in TG_L \wedge (q, \mathcal{A}, q') \in \Delta} (Q_q(s_{j-1}) \wedge Q_{q'}(s_j))$
8 $\quad\quad \phi := \phi \wedge ((s_{j-1} \leftrightarrow s_j) \vee (\iota_{t_{\mathcal{A}''}}(\ell) \wedge \phi'))$
9 $\quad\quad$ goals := `build_goals`$(t_{\mathcal{A}''},\ A,\ TG_L,\ \ell,\ j)$
10 $\quad TG_E := \{\mathcal{A} \mid \mathcal{A} \in \Sigma \wedge e \in \mathcal{L}(\mathcal{A})\}$
11 $\quad$ **if** $TG_E = \emptyset \wedge \mathsf{instrumented} \notin an$ **then** $\phi := \phi \wedge \neg \iota_{t_{\mathcal{A}''}}(e)$
12 $\quad$ **else**
13 $\quad\quad j := j + 1$
14 $\quad\quad \phi' := \bigvee_{\mathcal{A} \in TG_E \wedge (q, \mathcal{A}, q') \in \Delta} (Q_q(s_{j-1}) \wedge Q_{q'}(s_j))$
15 $\quad\quad$ **if** $\mathsf{instrumented} \in an$ **then** $\phi' := \phi' \wedge (s_{j-1} \leftrightarrow s_j)$
16 $\quad\quad \phi := \phi \wedge (\iota_{t_{\mathcal{A}''}}(e) \rightarrow \phi')$
17 $\quad\quad \phi := \phi \wedge (\iota_{t_{\mathcal{A}''}}(e) \vee (s_{j-1} \leftrightarrow s_j))$
18 $\quad\quad$ goals := `build_goals`$(t_{\mathcal{A}''},\ A,\ TG_E,\ e,\ j)$
19 $\phi := \phi \wedge \bigvee_{q_i \in F} Q_i(s_j)$
20 **return** (ϕ, goals)

Algorithm 6.1: Construction of propositional encoding

We describe this case for edges (lines 10–18); for locations (lines 4–9) the basic approach is the same, but has to be repeated as already described in Section 6.6.1.

Given an edge e in the abstract trace $t_{\mathcal{A}''}$ we collect in TG_E all target graphs $\mathcal{A} \in \Sigma$ that contain e. In accordance with φ_3 we must take one of the possible transitions (line 14), if $\iota_{t_{\mathcal{A}''}}(e)$ evaluates to true (line 16), unless the edge was added by (predicate) instrumentation (line 15). If $\iota_{t_{\mathcal{A}''}}(e)$ evaluates to false, the state bit-vector must remain constant (line 17).

The requirement that we end in an accepting state, φ_4, is added in line 19. The resulting formula ϕ therefore resembles φ as described previously, with $P_{\mathcal{A}}$ being replaced by expressions using $\iota_{t_{\mathcal{A}''}}$.

Handling Test Goals. As part of the traversal we also build the mapping goals, as already described in the preceding section, on the fly. For an incremental construction we call Algorithm 6.2 (function `build_goals()`). We use the same function for both locations ($x = \ell$) and edges ($x = e$).

input: $t_{\mathcal{A}''}$, $A = \langle Q, \Sigma, I, \Delta, F, G \rangle$, TG, x, j

```
1  foreach A ∈ TG do
2      foreach (q, A, q') ∈ Δ do
3          if ∃Q'.q' ∈ Q' ∧ (Q', all) ∈ G then
4              if ∃s.ι_{t_A''}(x) ∧ Q_{q'}(s) ∈ goals(Q') then
5                  goals(Q') := goals(Q') \ {ι_{t_A''}(x) ∧ Q_{q'}(s)}
6                  goals(Q') := goals(Q') ∪ {ι_{t_A''}(x) ∧ (Q_{q'}(s) ∨ Q_{q'}(s_j))}
7              else
8                  goals(Q') := goals(Q') ∪ {ι_{t_A''}(x) ∧ Q_{q'}(s_j)}
9          else if ∃Q'.q' ∈ Q' ∧ (Q', one) ∈ G then
10             g' := false ⋁_{g∈goals(Q')} g
11             goals(Q') := {g' ∨ ι_{t_A''}(x) ∧ Q_{q'}(s_j)}
12 return goals
```

Algorithm 6.2: Function `build_goals()` for incremental construction of goals

Overhead. With this approach, for the same examples that we picked in case of program instrumentation, we now get an increase in the size of the SAT formula of a factor of two only – before we had factors of 3 to 4 for very simple queries and trace automata. Furthermore the growth in the number of variables will be only logarithmic in the number of states of the trace automata.

6.7 Efficient Test Case Enumeration

Given the CNF encoding $\phi[t_{\mathcal{A}''}]$ (including the propositional encoding ϕ of Section 6.6.2) plus the set of test goals goals we want to efficiently compute a set of test cases that satisfies the test goals. For this section we will use the terms "test case" and model of the SAT formula interchangeably. We write $\pi \in \phi[t_{\mathcal{A}''}]$, where π is a path, iff there exists a model of $\phi[t_{\mathcal{A}''}]$ with a concrete trace describing π. In Section 6.9 we describe the translation of models to practically useful test cases. Hence the focus of this section is the use of SAT solving for *efficient* and *guided* enumeration of models – enumeration of arbitrary solutions would be computationally intractable and would yield excessively large test suites. We will therefore introduce guided SAT enumeration in Section 6.7. In guided SAT enumeration we repeatedly solve a SAT instance while incrementally adding clauses to achieve the desired guidance. While technically an arbitrary SAT solving strategy can be used, this process will be most efficient if conflict-driven clause learning (CDCL) is employed. In the following section we will therefore give a short introduction to CDCL SAT solving and describe the interface to the SAT solver that we assume in later sections.

6.7.1 Overview of CDCL/DPLL SAT Solving

The problem of satisfiability of Boolean formulas was one of the first problems shown to be NP-complete by Cook [Coo71]. As the problem is commonly referred to as "SAT", decision procedures for this problem are called "SAT solvers". The DPLL algorithm [DP60, DLL62] was one of the first sound and complete procedures to solve the SAT problem. DPLL underlies the most successful current solvers, although (incomplete) stochastic local search procedures such as GSAT [SLM92] and WalkSAT [SKC94] were also very popular several years ago.

DPLL-style SAT solvers today employ a number of heuristic improvements over the original procedure, which made the algorithm work efficiently on many practical problems, especially for satisfiable SAT instances. Only recently it has been shown [PD09] that modern CDCL solvers are still capable of producing short refutation proofs also for unsatisfiable instances, given an appropriate choice of heuristics. The main improvements are conflict analysis and clause learning as pioneered by GRASP [SS99] and further improved in CHAFF [ZMMM01], which also added efficiency improvements in Boolean constraint propagation using watched literals [MMZ+01] and dynamic variable reordering. The watched literals scheme was further improved in PicoSAT [Bie08], but recently several solvers moved back to head/tail-lists as earlier proposed in SATO [ZS94], because watched literals are patent covered.

The SAT solver MiniSat [ES03] made a major contribution to wide adoption of SAT solvers. It provided a clean and very compact implementation of the most successful techniques of CDCL SAT solving.

For FShell we employ version 2.2.0 of the MiniSat solver as the decision procedure of CBMC. MiniSat provides the following conceptual application programming interface (function names and argument types do not necessarily match the implementation):

- *Adding constraints* with function `add(`ϕ`)`: The method takes an arbitrary Boolean constraint ϕ (not necessarily in CNF).
- *Checking for satisfiability* with `solve(`ϕ`)`: The method returns `true` iff there exists a solution to the constraints added to the clause database so far *under the additional assumption* ϕ. Assumptions ϕ are not added to the clause database and therefore may vary over a series of calls to `solve()` without making the solver state inconsistent. If a call to `solve()` returns `true`, a witness is cached.
- *Obtaining a model* with `solution()`: The method returns the last witness cached in a call to `solve()`.

6.7.2 Guided SAT Enumeration

To generate a test suite Γ for a CFA $\mathcal{A}$ matching an elementary coverage criterion $\Phi(\mathcal{A})$ we introduce *iterative constraint strengthening (ICS)*. In ICS,

work with the CNF representation $\phi[t_{\mathcal{A}''}]$ to build a test suite Γ iteratively from a sequence of test suites $\Gamma_0 \subset \Gamma_1 \subset \cdots \subset \Gamma_m$ with $\Gamma_0 = \emptyset$ and $\Gamma_q = \{\pi_1, \ldots, \pi_q\}$ for $1 \leq q \leq m$. In the m-th iteration, we reach a fixed point when no more new goals can be covered.

Algorithm Overview. In the q-th iteration we build the *path constraint* ICSPC_q (Equation (6.2)) and obtain the test case π_{q+1} as one of its solutions. Here, ICSPC_q describes those models of $\phi[t_{\mathcal{A}''}]$ which describe paths covering a hitherto uncovered test goal. If no such test goal exists any more, ICSPC_q becomes unsatisfiable. Having determined a new test case π_{q+1}, we build ICSPC_{q+1} and continue the procedure with the $(q+1)$-st iteration until we reach an iteration m where ICSPC_m becomes unsatisfiable.

In order to fit the framework of *incremental SAT solving* (cf. [ES03]), we rewrite ICSPC_q (Equation (6.3)) in such a way that we are able to describe ICSPC_{q+1} incrementally in terms of ICSPC_q by *only adding* new constraints *without removing or changing* previously added constraints (Equation (6.4)). Using this incremental formulation of ICSPC_q, we describe iterative constraint strengthening (ICS) based upon an incremental SAT solver in Algorithm 6.3. The m paths finally collected by ICS constitute indeed a covering test suite (Theorem 6.8).

Path Constraints. The initial path constraint ICSPC_0 requires that a path is a model of $\phi[t_{\mathcal{A}''}]$ and covers at least one of the test goals Ψ for $\Psi \in \Phi(\mathcal{A})$. Subsequently, in ICSPC_q, we require the path to cover at least one test goal Ψ which remained *uncovered* by the test suite Γ_q. Since Γ_{q+1} must cover at least one more test goal than Γ_q, it suffices to *strengthen* the constraint ICSPC_q to obtain ICSPC_{q+1}. Below, we write $\mathsf{uncov}_q = \{\Psi \mid \Psi \in \Phi(\mathcal{A}) \wedge \Gamma_q \not\models \Psi\}$ for the set of test goals not covered in Γ_q. Note that $\mathsf{uncov}_0 = \Phi(\mathcal{A})$ since $\Gamma_0 = \emptyset$ covers no test goals at all. Then, for $0 \leq q \leq m$, we search for a solution π_{q+1} to the q-th constraint

$$\mathsf{ICSPC}_q := \phi[t_{\mathcal{A}''}] \wedge \bigvee_{\Psi \in \mathsf{uncov}_q} \Psi \tag{6.2}$$

Note that the empty disjunction is equivalent to false, i.e., if $\mathsf{uncov}_q = \emptyset$, then $\mathsf{ICSPC}_q \equiv \mathsf{false}$. Thus, ICSPC_q is satisfied by exactly those paths in $\phi[t_{\mathcal{A}''}]$ which satisfy at least one *feasible* test goal still *uncovered* by Γ_q. If no such test goal exists, i.e., if Γ_q *achieves coverage*, then ICSPC_q is unsatisfiable.

Proposition 6.6 ICSPC_q **demands further Coverage**
ICSPC_q *(Equation* (6.2)*) is satisfied by exactly those paths in* $\phi[t_{\mathcal{A}''}]$ *which satisfy at least one feasible test goal taken from* $\Phi(\mathcal{A})$ *being unmatched by* Γ_q*. If no such test goal exists* ICSPC_q *is unsatisfiable.*

Proof. Note that uncov_q is defined as the subset of test goals which are not satisfied by Γ_q. Then Equation (6.2) is satisfied by exactly those paths which are in $\phi[t_{\mathcal{A}''}]$ and which satisfy at least one such test goal in uncov_q. If no such test goal exists, the disjunction in Equation (6.2) becomes empty and ICSPC_q simplifies to false. □

Incremental Path Constraints. In incremental SAT solving, we use a single persistent clause database for consecutive solver invocations. When the SAT solver finds a solution, we add new clauses to the clause database, but do not remove any clauses. When the execution of the SAT solver is continued, the learned clauses obtained during earlier invocations remain valid and help to guide the search of the solver. Therefore, we have to construct ICSPC_{q+1} from ICSPC_q by only adding further constraints to the clause database. Observe that $\mathsf{uncov}_{q+1} \subset \mathsf{uncov}_q$ holds for $0 \leq q \leq m-1$. Thus in going from ICSPC_q to ICSPC_{q+1}, we have to remove all test goals Ψ with $\Psi \in \mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1}$ from the disjunction $\bigvee_{\Psi \in \mathsf{uncov}_q} \Psi$ occurring in Equation (6.2). To do so, we introduce a new Boolean variable S_a for each test goal Ψ_a and write ICSPC_q equisatisfiable as

$$\mathsf{ICSPC}_q := \left[\phi[t_{\mathcal{A}''}] \wedge \bigvee_{\Psi_a \in \Phi(\mathcal{A})} (S_a \wedge \Psi_a) \right] \wedge \bigwedge_{\Psi_a \notin \mathsf{uncov}_q} \neg S_a \tag{6.3}$$

Proposition 6.7 Equations (6.2) **and** (6.3) **are equisatisfiable**
The two formulations of ICSPC_q *in Equation* (6.2) *and in Equation* (6.3) *are equisatisfiable by models involving the* same path π*.*

Proof. Recall the two formulations of ICSPC_q from Equations (6.2) and (6.3), respectively:

$$\phi[t_{\mathcal{A}''}] \wedge \bigvee_{\Psi \in \mathsf{uncov}_q} \Psi \quad \text{and} \quad \left[\phi[t_{\mathcal{A}''}] \wedge \bigvee_{\Psi_a \in \Phi(\mathcal{A})} (S_a \wedge \Psi_a) \right] \wedge \bigwedge_{\Psi_a \notin \mathsf{uncov}_q} \neg S_a$$

($\Rightarrow$) Assume that Equation (6.2) has a model π. Then $\pi \in \phi[t_{\mathcal{A}''}]$ must hold, and there must exist a test goal $\Psi_a \in \mathsf{uncov}_q \subseteq \Phi(\mathcal{A})$ with $\pi \models \Psi_a$. Fix

such an a and assign $S_a = \mathsf{true}$ and $S_{a'} = \mathsf{false}$ for all $a' \neq a$. This expanded assignment satisfies Equation (6.3): $\pi \in \phi[t_{\mathcal{A}''}]$ holds, and for the fixed test goal Ψ_a, the conjunct $S_a \wedge \Psi_a$ holds as well. As we assigned false to all variables $S_{a'}$ for $a' \neq a$, and since $\Psi_a \in \mathsf{uncov}_q$, we find that $\bigwedge_{\Psi_a \notin \mathsf{uncov}_q} \neg S_a$ is satisfied as well – such that the expanded assignment satisfies Equation (6.3) as a whole.

($\Leftarrow$) Assume that Equation (6.3) has a model. Then $\pi \in \phi[t_{\mathcal{A}''}]$ must hold, and there must exist a test goal $\Psi_a \in \Phi(\mathcal{A})$ with $S_a \wedge \Psi_a$. Fix such an a and observe that for this a the variable S_a must be assigned with true. Since $\bigwedge_{\Psi_a \notin \mathsf{uncov}_q} \neg S_a$ must be satisfied as well, $\Psi_a \in \mathsf{uncov}_q$ must hold. Therefore we obtain $\pi \models \Psi_a$ for a $\Psi_a \in \mathsf{uncov}_q$ and consequently the assignment satisfying Equation (6.3) must satisfy Equation (6.2) as well.

Note that in both directions, we did not alter the path π such that both formulations are satisfied by models describing the same set of paths. □

Thus ICSPC_q consists of (a) an initial expression, shown above in square brackets, which remains unchanged throughout all iterations, and (b) a conjunction which is expanded from one iteration to the next. Adding $\neg S_a$ to the constraint renders the corresponding conjunct $S_a \wedge \Psi_a$ unsatisfiable, and therefore only the conjuncts for $\Psi_a \in \mathsf{uncov}_q$ remain enabled. Note that for ICSPC_0 we have $\mathsf{true} \equiv \bigwedge_{\Psi_a \notin \mathsf{uncov}_0} \neg S_a$. Thus, in each iteration step, we use

$$\mathsf{ICSPC}_{q+1} := \mathsf{ICSPC}_q \wedge \bigwedge_{\Psi_a \in \mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1}} \neg S_a \tag{6.4}$$

to obtain ICSPC_{q+1} from ICSPC_q. Since we only add further constraints conjunctively, this approach fits the requirements of incremental SAT solving.

Iterative Constraint Strengthening. The resulting algorithm `ICS` is shown in Algorithm 6.3.

In line 1 we initialize the iteration counter q, the first test suite Γ_0, and the set of test goals uncov_0 uncovered by Γ_0. Then in line 2, we add the initial expression from Equation (6.3) and start the search for the first solution in line 3. If a solution is found, it is obtained from the solver, assigned to π_{q+1}, and added to Γ_{q+1}. Then, after initializing uncov_{q+1}, we update the clause database following Equation (6.4) and fill the set uncov_{q+1} in lines 7 to 9: for each yet uncovered test goal $\Psi_a \in \mathsf{uncov}_q$, we check whether π_{q+1} satisfies Ψ_a (cf. Section 6.7.3). If this is the case, $\Psi_a \in \mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1}$ holds, and thus

input: $\phi[t_{\mathcal{A}''}]$, $\Phi(\mathcal{A})$

1 $q := 0$, $\Gamma_0 := \emptyset$, $\mathsf{uncov}_0 := \Phi(\mathcal{A})$
2 $\texttt{add}(\phi[t_{\mathcal{A}''}] \wedge \bigvee_{\Psi_a \in \Phi(\mathcal{A})} (S_a \wedge \Psi_a))$
3 **while** $\texttt{solve(true)}$ **do**
4 $\quad \pi_{q+1} := \texttt{solution()}$
5 $\quad \Gamma_{q+1} := \Gamma_q \cup \{\pi_{q+1}\}$
6 $\quad \mathsf{uncov}_{q+1} := \emptyset$
7 $\quad$ **foreach** $\Psi_a \in \mathsf{uncov}_q$ **do**
8 $\quad\quad$ **if** $\pi_{q+1} \models \Psi_a$ **then** $\texttt{add}(\neg S_a)$
9 $\quad\quad$ **else** $\mathsf{uncov}_{q+1} := \mathsf{uncov}_{q+1} \cup \{\Psi_a\}$
10 $\quad q := q + 1$
11 **return** Γ_q

Algorithm 6.3: Iterative Constraint Strengthening (ICS)

we add $\neg S_a$ in line 8. Otherwise Ψ_a remains uncovered by Γ_{q+1} and hence we add Ψ_a to uncov_{q+1} in line 9. Once no further solution is found in line 3, the accumulated suite Γ_q is returned.

Theorem 6.8 Correctness of Iterative Constraint Strengthening
The test suite Γ *returned by the algorithm* $\texttt{ICS}(\phi[t_{\mathcal{A}''}], \Phi(\mathcal{A}))$ in Algorithm 6.3 satisfies $\Gamma \subseteq \phi[t_{\mathcal{A}''}] \wedge \Gamma \models \Phi(\mathcal{A})$.

Proposition 6.9 Correctness of Incremental Formulation of ICSPC_q
Equation (6.3) *and the incremental formulation in Equation* (6.4) *are equivalent, where we use*

$$\mathsf{ICSPC}_0 \equiv \left[\phi[t_{\mathcal{A}''}] \wedge \bigvee_{\Psi_a \in \Phi(\mathcal{A})} (S_a \wedge \Psi_a) \right]$$

for the base case.

Proof. $(q = 0)$ First, note that the equivalence holds for the base case $q = 0$ since $\mathsf{uncov}_0 = \Phi(\mathcal{A})$ holds and thus $\bigwedge_{\Psi_a \notin \mathsf{uncov}_q} \neg S_a$ as an empty conjunction simplifies to true, i.e., Equation (6.3) simplifies to ICSPC_0 as stated in the proposition.

$(q \Rightarrow q + 1)$ Considering Equation (6.3), the only difference between ICSPC_q and ICSPC_{q+1} are the different conjuncts in $\bigwedge_{\Psi_a \notin \mathsf{uncov}_q} \neg S_a$. Since

$\mathsf{uncov}_{q+1} \subset \mathsf{uncov}_q$ we find that $\mathsf{uncov}^C_{q+1} = \mathsf{uncov}^C_q \cup (\mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1})$ where the complement refers to $\Phi(\mathcal{A})$ as universe. But these are exactly the conjuncts added with $\bigwedge_{\Psi_a \in \mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1}} \neg S_a$ in Equation (6.4). This proves the equivalence between Equations (6.3) and (6.4) for $q+1$, assuming the equivalence already holds for q – and thus completes the induction. □

Remark 6.10 (Nondeterminism in Choosing π_{q+1}) *The algorithm* `ICS` *leaves the particular choice of π_{q+1} open to the underlying SAT solver (line 4). Potential optimizations could control this choice to minimize the number of test cases necessary to obtain coverage.*

Proof of Theorem 6.8

The constraints added incrementally to the SAT solver up until iteration q amount to Equation (6.4)*:* In line 2, the initial constraint for ICSPC_0 is added to the solver. Then, in line 8, we add the constraint $\neg S_a$ for all $\Psi_a \in \Phi(\mathcal{A})$ with $\pi_{q+1} \models \Psi_a$. Thus all the corresponding test goals Ψ_a are covered *at least* in iteration $(q+1)$ – while they are possibly already covered in some earlier iteration – and therefore we have $\mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1} \subseteq \{\Psi_a \in \Phi(\mathcal{A}) \mid \pi_{q+1} \models \Psi_a\}$. Consequently, in line 8 all conjuncts of $\bigwedge_{\Psi_a \in \mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1}} \neg S_a$ are added to the solver – in addition to some redundant conjuncts $\neg S_a$ for $\Psi_a \notin \mathsf{uncov}_q$ which have been added already in earlier iterations.

ICS solves ICSPC_q in iteration q: Since the collected constraints amount to Equation (6.4), and since Propositions 6.7 and 6.9 together prove that Equation (6.4) is equisatisfiable to Equation (6.2), we find that `ICS` solves in the q-th iteration the constraint ICSPC_q (line 3). Moreover, Propositions 6.7 and 6.9 guarantee that the path π remains unchanged such that the solution obtained in line 4 is a path solving ICSPC_q as defined by Equation (6.2).

ICS is correct if it terminates: By Proposition 6.6, we know that if no solution is found in line 3, then no additional feasible and yet uncovered test goal exists, i.e., the current test suite Γ_q covers all test goals from $\Phi(\mathcal{A})$ which are feasible in $\phi[t_{\mathcal{A}''}]$. Hence, if `ICS` terminates and returns a result in line 11, then the returned test suite satisfies the theorem statement.

ICS terminates: Thus, it remains to show that the loop terminates. If the loop does not terminate in the q-th iteration, then the condition in line 3 is not satisfied, i.e., a path π_{q+1} is found such that π_{q+1} satisfies ICSPC_q. By Proposition 6.6, we know that π_{q+1} must satisfy at least one hitherto uncovered test goal Ψ_a, i.e., $\Psi_a \in \mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1}$. Since at least one such

Ψ_a must exist in every iteration, we obtain $\mathsf{uncov}_{q+1} \subset \mathsf{uncov}_q$ for all $q \geq 0$. Assume that the loop does not terminate. Then ICSPC_q must be satisfiable for all $q \geq 0$. Since uncov_m is empty after $m \leq |\Phi(\mathcal{A})| = k$ iterations, the right conjunct $\bigvee_{\Psi_a \in \mathsf{uncov}_q} \pi \models \Psi_a$ in Equation (6.2) becomes the empty disjunction after $m \leq k$ iterations. But then, ICSPC_m is unsatisfiable – which is a contradiction. □

Groupwise Constraint Strengthening. As stated in Remark 6.5, the number of test goals is *exponential* in the number of test goal groups, as described by the mapping goals. For this reason, the disjunction in ICSPC_0 will be of exponential size – thus rendering iterative constraint strengthening hard for such coverage criteria. But in practice, such criteria are only applicable for two reasons [Bal04]: First, because many of these test goals are, in our terminology, infeasible, and second, because a single test case potentially covers multiple test goals simultaneously.

To mitigate this situation, we introduce *groupwise constraint strengthening (GCS)* as an optimization of iterative constraint strengthening. To apply GCS, we require the test goals to be partitioned into k distinct *groups* $G_i = \{\Psi_1^i, \ldots, \Psi_{k_i}^i\}$ *of mutually exclusive subgoals* for $1 \leq i \leq k$, i.e., we require that there exists no path π with $\pi \models \Psi_g^i \wedge \Psi'$ and $\pi \models \Psi_h^i \wedge \Psi'$ for all $1 \leq g \neq h \leq k_i$ and $1 \leq i \leq k$ with Ψ' being the conjunct of subgoals from each of the other groups G_j. In this setting the set of test goals $\Phi(\mathcal{A})$ can be written as

$$\Phi(\mathcal{A}) = \{\Psi^1 \wedge \Psi^2 \wedge \ldots \wedge \Psi^k \mid \Psi^i \in G_i \text{ with } 1 \leq i \leq k\}$$

which precisely resembles Equation 6.1 of Section 6.6.1 – goals immediately yields such groups.

In the GCS algorithm, we avoid the construction of the initial and large disjunction $\bigvee_{\Psi \in \mathsf{uncov}_q} \Psi$, as it appears in ICSPC_q (Equations (6.2) and (6.3)); in fact we already avoid the eager construction of the conjuncts $\Psi \in \Phi(\mathcal{A})$: we only build the conjunction Ψ when Ψ is known to be feasible. Therefore we do not remove elements from a set uncov_q, but conversely incrementally build a set cov_q of already covered test goals.

Furthermore observe that $\phi[t_{\mathcal{A}''}]$ (including ϕ) by construction already guarantees that any model satisfies at least one test goal. With the observation of mutual exclusiveness it suffices to rule out each test goal that is satisfied.

Therefore GCS proceeds like ICS but with Equation (6.2) replaced by

$$\mathsf{GCSPC}_q := \phi[t_{\mathcal{A}''}] \wedge \bigwedge_{\Psi^1 \wedge \Psi^2 \wedge \ldots \wedge \Psi^k \in \mathsf{cov}_q} \bigvee_{i=1}^{k} \neg \Psi^i \tag{6.5}$$

and with incrementally adding test goals to cov_q. Similar to ICS we also adopt GCSPC_q to fit incremental SAT solving: more precisely, we only start with the constraint $\mathsf{GCSPC}_0 = \phi[t_{\mathcal{A}''}]$ and incrementally add clauses $\bigvee_{i=1}^{k} \neg \Psi^i$ built from cov_q as shown in Equation (6.5).

Written in the form of Equation (6.5), GCSPC_q does not explicitly refer to any infeasible test goals and only involves *feasible* test goals as subexpressions. This significantly reduces the size of the constructed constraint.

The effectiveness of GCS as an optimization of ICS relies on three conditions: (a) mutual exclusiveness of groups must be determined in an efficient way; we have mutual exclusion by construction of trace automata for all coverage specifications other than C_1 + C_2. In the latter case we first eagerly build more complex subgoals, which are then again mutually exclusive. (b) The disjunction of all test goals must be available in a succinct formulation. In our case this is already encoded in $\phi[t_{\mathcal{A}''}]$. (c) The fraction of *feasible* test goals must be small, since the negation of each feasible test goal is added to GCSPC_q in some iteration q. Conditions (a) and (b) hold by means of our construction. If condition (c) does not hold, then mostly also the number of required test cases will be large – but this is inherent in the coverage criterion. In the worst case it is still at most as large as the initial constraint of ICS, ICSPC_0.

6.7.3 Coverage Analysis

As a result of our constructions in Sections 6.6.1 and 6.6.2, each step in guided SAT enumeration initially yields a path covering *only one* additional test goal. Our encoding of trace automata as monitors running in parallel implies that each solution produced by the SAT solver also induces only one run of the test goal automaton. This accepting run of the trace automaton is chosen nondeterministically among several possible accepting runs *for the same program path.*

These alternative accepting runs may yield additional test goals covered by the same test case. In FShell we have implemented two approaches to

discover these additional runs: we can either use the SAT solver by performing repeated runs with additional assumptions, or perform simulation of all possible runs of the test goal automaton. Either of these approaches can be employed for the coverage check $\pi_{q+1} \models \Psi_a$ in line 8 of Algorithm 6.3.

SAT-based Coverage Analysis. In guided SAT enumeration a test case is a model of the SAT formula. We remove all assignments to literals from this model that do not correspond to values of program variables of $\mathcal{A}'$, i.e., all Boolean variables of trace automata processing become undefined. We use the remaining model as initial assumption and start a new SAT solver instance for another run of guided SAT enumeration *with the assumption of the fixed program path*. This process enumerates all test goals satisfied by the same test case. The SAT solver returns "unsatisfiable" (UNSAT) if and only if no further test goals can be satisfied by this test case.

Despite the beauty of using guided SAT enumeration within guided SAT enumeration it incurs a noticeable overhead: this approach requires one call to the SAT solver for each satisfiable test goal. Furthermore a new solver instance must be started for each test case as the UNSAT result of the inner guided SAT enumeration loop only holds true under the assumption of a given test case. Because of this undesirable overhead we implemented another approach for coverage analysis, which does not use a SAT solver:

Simulation of NFA Runs. An abstract program trace and a model computed by the SAT solver yield a concrete program trace. We can therefore simulate the test goal automaton over the concrete program trace to record accepting runs of the test goal automaton that describe test goals (cf. Definition 6.2). We use Algorithm 6.4 to perform this simulation.

In this approach we traverse the concrete program trace in program order. In each step we record the reachable states of the test goal automaton to build up a run tree of the test goal automaton (lines 1–12). We store the subgoals that are known to be satisfied via lookups in the goals mapping (lines 7–11). At the end of the program trace we inspect all accepting states in the list of reached states (line 14). For these states we try to translate each set of subgoals (line 16) into a test goal. The algorithm then returns the set of successfully translated subgoals.

input: $t_{\mathcal{A}''}$, $\iota_{t_{\mathcal{A}''}}$, $\langle Q, \Sigma, I, \Delta, F, G \rangle$, goals

```
1  foreach q ∈ I do
2      subgoals[q] := ∅
3  foreach e ∈ t_{A''} with ι_{t_{A''}}(e) = true do
4      subgoals' := ∅
5      foreach q ∈ subgoals and S ∈ subgoals[q] do
6          foreach q' ∈ Q with ∃A.e ∈ A ∧ (q, A, q') ∈ Δ do
7              if ∃Q'.q' ∈ Q' ∧ (Q', one) ∈ G then
8                  subgoals'[q'] := {subgoals'[q']} ∪ {S ∪ goals(Q')}
9              else if ∃Q'.q' ∈ Q' ∧ (Q', all) ∈ G then
10                 if ∃g ∈ goals(Q').g = ι_{t_{A''}}(e) ∧ (Q_{q'}(s) ∨ ...) then
11                     subgoals'[q'] := {subgoals'[q']} ∪ {S ∪ {g}}
12     subgoals := subgoals'
13 satgoals := ∅
14 foreach q ∈ subgoals with q ∈ F do
15     foreach S ∈ subgoals[q] do
16         if ⋀_{i=1}^{k} ∃Ψ^i ∈ S ∩ G_i then
17             satgoals := satgoals ∪ {⋀_{i=1}^{k} Ψ^i}
18 return satgoals
```

Algorithm 6.4: Simulation of test goal automaton runs

Choosing the Coverage Checking Strategy. Whether FSHELL uses the SAT solver or the built-in simulation for coverage analysis can be controlled by the user as explained in Section 6.3. In our experiments we observed that for common and simple queries the built-in simulation is much faster, but for queries inducing a large amount of nondeterminism in the test goal automaton the SAT-based approach is still preferable.

6.8 Test Suite Minimization

Guided SAT enumeration ensures that each computed test case satisfies at least one additional test goal – but such a test case may also satisfy test goals already fulfilled by previously computed test cases. Thus a test suite may contain redundant test cases, i.e., test cases that only fulfill test goals also fulfilled by at least one other test case. With the help of the coverage analysis described above we can obtain a mapping $\mathsf{covers} : \Gamma \to 2^{\Phi(\mathcal{A})}$ that describes the set of test goals fulfilled by each test case $\pi \in \Gamma$.

Offutt et al. [OPV95] discuss heuristic minimization strategies that traverse a list of test cases forward and/or backwards to eliminate redundant test cases. In FSHELL we implemented an optimal minimization approach that is guaranteed to find a minimal subset of a test suite Γ that still satisfies the same set of test goals. We first show that the decision problem of whether a given test suite can be minimized to a chosen size is NP-complete. Then we present an iterative minimization algorithm that uses incremental SAT solving to arrive at a minimal test suite. Experimental results for this approach are shown in Section 7.5.

Proposition 6.11 Test Suite Minimization is NP-complete [TG05, HO09]
Let Γ be a test suite with a test goal map $\mathsf{covers} : \Gamma \to 2^{\Phi(\mathcal{A})}$. Deciding whether there is a test suite $\Gamma' \subseteq \Gamma$ of size at most k for some integer k is NP-complete.

Proof. The test suite minimization problem is an instance of SET-COVER(U, S, k) with the universe

$$U = \bigcup_{\pi \in \Gamma} \mathsf{covers}(\pi)$$

and a family of subsets $S = \{\mathsf{covers}(\pi) \mid \pi \in \Gamma\}$. SET-COVER is NP-complete [Kar72]. □

Given that the decision problem is NP-complete, we use a series of SAT solver calls to solve the corresponding minimization problem. Using Algorithm 6.5 we determine the minimal solution by strictly monotonically decreasing the size bound k.

input: Γ, $\mathsf{covers} : \Gamma \to 2^{\Phi(\mathcal{A})}$

```
1  disj := ∅
2  foreach π ∈ Γ do
3  |  foreach Ψ ∈ covers(π) do
4  |  |  disj(Ψ) := disj(Ψ) ∨ enc(π)
5  foreach Ψ ∈ disj do add(disj(Ψ))
6  add(enc((∑_{π∈Γ} enc(π)) < K))
7  Γ' := Γ
8  k := |Γ'|
9  while solve(enc(K = k)) do
10 |  Γ'' := ∅
11 |  foreach π ∈ Γ' do
12 |  |  if enc(π) ∈ solution() then Γ'' := Γ'' ∪ {π}
13 |  Γ' := Γ''
14 |  k := |Γ'|
15 return Γ'
```

Algorithm 6.5: Test suite minimization

We first encode the SET-COVER problem by introducing a new Boolean variable for each test case using a mapping function `enc()` (line 4). We encode the minimization constraint as a sum over the new Boolean variables where $(\mathtt{enc}(\pi) = \mathsf{false}) \equiv 0$ and $(\mathtt{enc}(\pi)) = \mathsf{true}) \equiv 1$ with the constraint $< K$ where K is a fresh and sufficiently large bit vector (line 6). We perform iterative minimization in the loop in lines 9–14. We start with $K = k = |\Gamma|$. If the problem is satisfiable, the model of the SAT formula induces a test suite $\Gamma' \subsetneq \Gamma$ and we set $k = |\Gamma'|$. Once the SAT solver returns UNSAT (false), we have shown that Γ' is the smallest possible test suite.

6.9 Computing Test Inputs

In FShell, a test case is internally described by a model π of the program's SAT formula $\phi[t_{\mathcal{A}''}]$. Together with the abstract trace this describes a concrete trace and thus a program execution. To aid the user, however, we need to translate this result into a representation as succinct as possible. As we deal with deterministic programs only, it suffices to list all input to the program, which may include sources of randomness. FShell handles a number of situations that can occur as test input:

1. Arguments to the program entry function.
2. Global variables when `NO_ZERO_INIT` was set, as explained in Section 6.3.
3. Uninitialized local non-static variables.
4. Return values of undefined functions (possible side-effects of these undefined functions are not handled).

We compute the test inputs by stepping through the concrete trace and perform a program analysis for definition-clear paths [RW85, NNH99]. In this analysis we record all uses of variables that occur before a definition of the corresponding variable. Each such uninitialized variable or use of an undefined function for initialization is then taken as test input, where the effective value is found in the model of the SAT formula.

Proposition 6.12 *The definition-clear path analysis as sketched above computes the most succinct representation of a test case.*

Proof. In the programs we analyze all operations contained in abstract traces are deterministic. Hence results of expressions exclusively depend on input or previously computed expressions. If any input is not fixed or a value of an expression is not available, multiple successor states may be possible. Any representation of test cases must therefore at least contain all input values or unavailable expressions. The definition-clear path analysis determines all such inputs, i.e., values that are read before being defined. □

Examples of the resulting output of FShell have been shown in Section 3.4.

6.10 Test Harness Generation

In the preceding section we discussed how we translate paths, as computed in bounded model checking, into a succinct description as a set of test inputs. The purpose of a test harness is to drive executions of the original program under test to follow the previously computed paths – by providing the computed inputs.

Each of the four cases of input, as listed in Section 6.9, requires separate treatment by a test harness, resulting in the following requirements specifications:

1. Arguments to the entry function are handled by a wrapper function. To avoid linking programs caused by, e.g., a duplicate definition of a `main` function, the original program entry function must be renamed. The wrapper then calls the renamed function with the computed inputs.

2. Initialization of global variables is performed in a new initialization function that is called by the wrapper before calling the renamed entry function.

3. Local variables get values assigned at the point of their declaration. This requires modification of the program source code.

4. Undefined functions with computed return values must be defined as new C functions that only return the desired value or a sequence of such values in case of multiple call sites.

A fully automated solution of these tasks requires parsing the source code and printing the modified program plus possibly tweaking the build environment. As a prototypical instance thereof we provide a solution using Perl scripts: the script `TestEnvGenerator.pl` takes as input the test output of FShell with command line option `--tco-location` being set. The latter is necessary for the test harness generation to have source line numbers and variable type information available. The test harness generator then builds a file "tester.c" that holds the wrapper function and a Makefile "tester.mk" (on Windows systems a batch file "tester.bat" is created instead). The Makefile contains source editing commands. These commands – Perl text replacement commands – are executed on a copy of the original source files. The Makefile furthermore provides the compile and link commands to build the program

using the modified files, and also a cleanup target. The build command honors the `BUILD_FLAGS` variable to set additional compiler options, such as enabling instrumentation for coverage measurement using `gcov` (cf. Section 3.4). The text editing commands insert test inputs for all test cases at once, using arrays of values which are indexed using a global test case counter.

The wrapper code in tester.c sets this counter, using one of two ways: in the basic test harness generator a new `main` function is built in tester.c that takes the test case index from the command line. As an alternative means, also a unit test tool kit can be used for the wrapper function, which may provide additional logging and statistics. We implemented support for CUnit[2], but also other unit test frameworks[3] could be investigated.

[2]See `http://cunit.sourceforge.net`

[3]An overview of test frameworks can be found at `http://www.opensourcetesting.org/unit_c.php`

Applications programming is a race between software engineers, who strive to produce idiot-proof programs, and the Universe which strives to produce bigger idiots.
— So far the Universe is winning.

Rick Cook, Wizardry Compiled

Chapter 7

Evaluation

In this chapter we present an evaluation of query-driven program testing with experimental studies of both FQL and FShell. For a proper evaluation we exercise them in a number of orthogonal ways. We start with a study of the practical applicability of query-driven program testing by describing ongoing and planned projects. Afterwards we turn to FQL in detail and discuss its expressiveness. Following the list of requirements for FQL described in Chapter 2 we further describe the efficiency of query evaluation on a number of complex queries and the applicability of FQL and FShell to real software. We close this chapter with a detailed evaluation of FShell regarding scalability, efficiency of different implementation options, its advantage over competitors, and the effects of test suite minimization.

7.1 Uses of Query-Driven Program Testing

Having a formalism at hand that is usable by the working programmer and that precisely defines the semantics of coverage criteria enables the development of new tools and methods for software engineering in general and for software testing in particular. To demonstrate practical usefulness of query-driven program testing, we first describe two ongoing projects with the embedded systems industry. Afterwards we discuss research directions occurring in the context of FQL and their potential applications.

7.1.1 Measurement-based Execution Time Analysis

Our initial motivation for FQL and the test case generation back end was measurement-based execution time analysis for embedded real-time software. Together with our project partners [BK08, ZK08, ZBK09, BKZT11] we are developing a framework to provide early feedback about the distribution of execution times to the developer. In this project, query-driven program testing enables us to efficiently compute test suites appropriate for timing analysis.

7.1.2 Model/Implementation Consistency Checking

In collaboration with an avionics supplier we are currently developing an automated technique to check consistency of models (UML activity diagrams) and their implementation (C code). We first compute a test suite at model level that, e.g., covers all edges of the model. Each model-level test case then describes a path through the model. We use this model-level test case as path pattern in an FQL `passing` clause and ask for condition coverage at implementation level. The number of test cases computed reflects the relationship between model and implementation and leads to detailed feedback on possibly unintended discrepancies.

7.1.3 Coverage Evaluation

Although there exist tools for coverage analysis these tools are fixed to few specific criteria. In contrast, a coverage analysis tool for FQL enables the user to check for coverage criteria that are specific to a project and where no such tools are available yet. One such example is *focused testing*: For example, does a regression test suite cover a recently changed code part adequately? In case the test suite fails to do so, we want to determine the uncovered parts and derive new specific FQL queries that yield the missing test cases. Here, one can think of an automatic improvement of regression test suites using version control tools, coverage evaluation and our FQL back end in combination.

Coverage evaluation can take place at different levels, e.g., at model level as well as at implementation level of a system under development. Analyzing test suites for these different levels with respect to the same coverage criterion can reveal deficiencies in the validation process. One example of such an

approach is our already mentioned project on checking consistency of model and implementation in an avionics setting.

Finally, coverage analysis enables us to systematically investigate the reasons for the infeasibility of test goals. We can identify uncovered test goals and, then, extract relevant dependencies in the program using techniques like program slicing.

7.1.4 Reasoning on Coverage Criteria

The precise semantics of FQL queries makes it possible to relate coverage criteria to different entities occurring in the software development process. We already discussed coverage evaluation which relates a test suite to a coverage criterion specified in FQL.

Another fundamental question is the relation between different coverage criteria themselves, for example, are two FQL specifications equivalent or does one coverage criterion subsume the other? Such questions arise when comparing coverage criteria originating from requirements to standard structural coverage criteria, like basic block coverage, which are used for certification purposes. Here, the goal is to show that requirements coverage implies structural coverage. Such a reasoning enables us to obtain FQL queries that characterize the difference between two queries. This is not only helpful for test case generation but also for identifying the code behavior that is not yet captured and thus enables an assessment of the requirements and the corresponding validation efforts.

Another question is the investigation of the relation of an FQL specification and the source code under scrutiny. For example, what impact has a code change on the semantics of an FQL specification and what does this tell us about an existing test suite?

The last question we want to address in this section is the relation between coverage criteria expressed as FQL queries and the algorithmic approaches used for test case generation. Since FQL queries are purely declarative, how can we decide what test case generation techniques we should apply to efficiently generate test suites?

7.1.5 Test Case Generation

Our current back end is only a first step towards test case generation for FQL queries. In combination with a coverage analysis tool as discussed above, we will be able to combine different test case generation tools to obtain covering test suites: First, we generate an initial test suite using light-weight techniques, like random testing, then, using our coverage analysis tool, we derive new FQL queries which target at yet uncovered code parts and answer these queries using our back end.

7.1.6 Discussion

Our projects demonstrate the usefulness of FQL's flexible test case specification to practical problems in embedded systems. For avionics software that must conform to highest safety requirements we will, however, need to add support for modified condition/decision coverage. This is beyond the scope of elementary coverage criteria and requires path set predicates as test goals. We consider a proper integration as future work.

7.2 Expressiveness

We use the example specifications from Figures 2.1–2.3 as guidance for our evaluation of sufficient expressiveness of FQL. We already showed in Section 5.8 that in principle we can express all of them in FQL. For further evaluation, and since most scenarios – for referring to line numbers or function names – make only sense for programs which contain certain tokens, we give three sample programs: The file list2.c, shown in Listing 7.1, extends the program of Listing 5.1. The files sort1.c (Listing 7.2) and sort2.c (Listing 7.3) contain fragments performing array manipulation.

We describe the results of evaluation using FShell in Section 7.3.1.

```
1 int partition(int a[], int left, int right) {
2  int v = a[right], i = left - 1, j = right, t;
3  for (;;) {
4   while(a[++i] < v) ;
5   while(j > left && a[--j] > v) ;
6   if(i >= j) break;
7   t = a[i]; a[i] = a[j]; a[j] = t;
```

```
 8 }
 9 t = a[i]; a[i] = a[right]; a[right] = t;
10 return i;
11}

13int main(int argc,char * argv[]) {
14 int A[3];
15 partition(A, 0, 2);
16}
```

Listing 7.1: Source code of list2.c

```
 1#include <stdio.h>

 3int compare(int a,int b) {
 4 if(a <= b) return 1;
 5 return 0;
 6}

 8int unfinished();

10void sort(int * a,int len) {
11 int i, t;
12 for(i=1; i<len; ++i) {
13  if(compare(a[i-1],a[i])) continue;
14  unfinished();
15  t=a[i];
16  a[i]=a[i-1];
17  a[i-1]=t;
18 }
19 return;
20}

22void eval(int * a,int len) {
23 int i;
24 for(i=0; i < 3; ++i)
25  printf("a[%d]=%d\n", i, a[i]);
26 return;
27}

29int main(int argc,char * argv[]) {
30 int i;
31 int A[3];
32 int verify;

34 for(i=0;i < 3;++i)
35  sort(A, 3);

37 if(verify) {
38  assert(compare(A[0],A[1]));
39  assert(compare(A[1],A[2]));
40 }

42 eval(A, 3);

44 return 0;
45}
```

Listing 7.2: Source code of sort1.c

```
1 #include <stdio.h>

3 int rand_init;

5 void insert(int * a,int pos) {
6  int rand;
7  a[pos] = rand%rand_init;
8  return;
9 }

11 void eval(int* a,int  first ,int  last){
12  if( first  > last) return;
13  printf("a[%d]=%d\n",first,a[first ]);
14  eval(a+1, first +1, last );
15  return;
16 }

18 int precond() {
19  return (rand_init == 7);
20 }

22 int postcond() {
23  return (rand_init == 7);
24 }

26 int main(int argc,char * argv[]) {
27  int i;
28  int A[3];
29  int max;

31  if(max > 0)
32 init :  rand_init = 7;

34  for(i=0; i<max; ++i)
35    insert(A, i);

37  eval(A, 0, max);

39  return 0;
40 }
```

Listing 7.3: Source code of sort2.c

7.3 Experimental Evaluation

In the following sections we describe experiments using FShell. We apply it on a range of source codes that we consider possible real-world scenarios for future use of FQL with FShell. We first give an overview of the source code we use and then describe the execution environment used in all subsequent experiments.

For each program we describe origin and size, plus necessary options: As FShell builds upon a bounded model checker, in several cases the number of loop unwindings to be performed must be fixed beforehand. In Table 7.1 we list all main source file names, the number of lines of code (SLOC)[1], and unwinding options.

The nature of our sources is as follows:

[1]Measured using David A. Wheeler's SLOCCount tool.

Source	SLOC	Unwinding Options
list2.c	15	`-unwind 3`
sort1.c	36	`-unwind 4`
sort2.c	30	`-unwind 4 -no-unwinding-assertions`
cdaudio	16279	
floppy	15813	`-unwind 10 -no-unwinding-assertions`
kbfiltr	10287	
parport	45753	`-unwind 1 -no-unwinding-assertions`
pseudo-vfs.c	553	
matlab.c	3444	
memman.c	377	`-unwind 5 -no-unwinding-assertions`
adpcm.c	504	`-unwind 100 -partial-loops`
nsichneu.c	2361	
statemate.c	1053	`-unwind 3`
autopilot	5945	`-unwind 20 -partial-loops`
fly_by_wire	4609	`-unwind 10 -partial-loops`
cat.c	27	`-unwind 10 -no-unwinding-assertions`
echo.c	161	`-unwind 3 -no-unwinding-assertions`
nohup.c	33	`-unwind 10 -no-unwinding-assertions`
seq.c	37	`-unwind 5 -no-unwinding-assertions`
tee.c	73	`-unwind 5 -no-unwinding-assertions`
PicoSAT	6646	`-unwind 3 -no-unwinding-assertions`
joplift.c	1184	

Table 7.1: Source code and unwinding options used in our experiments

- *list2.c, sort1.c, sort2.c:* The sources, as shown in the preceding section, were manually crafted for experimenting with complex queries.
- *cdaudio, floppy, kbfiltr, parport:* Preprocessed device drivers from the Windows Driver Development Kit used in [BCH+04b] for generating test cases with BLAST. The source code was downloaded from `http://www.sosy-lab.org/~dbeyer/blast_mc/`. We had to remove inline assembly as CBMC's C front end does not yet support VC++ inline assembly.
- *pseudo-vfs.c:* A simplified version of the Linux virtual file system layer, available at `http://research.nianet.org/~radu/VFS/`. In [GLMS09] this was used to apply model checking tools to the Linux virtual file system layer.
- *matlab.c:* An engine controller provided by an industrial collaborator from the automotive industries. It is generated from a MATLAB/Simulink model.
- *memman.c:* As another example of industrial code we examined a dynamic memory manager for airborne software systems.
- *adpcm.c, nsichneu.c, statemate.c:* These files were taken from the Mälardalen WCET Benchmark suite, available at `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`. adpcm.c implements functionality for adaptive pulse code modulation, nsichneu.c simulates a Petri net, and statemate.c was automatically generated by STARC (STAtechart Real-time Code generator).
- *autopilot, fly_by_wire:* Source code from PapaBench, a real-time embedded benchmark built from software controlling an unmanned aerial vehicle. We used version 0.4, downloaded from `http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97`.
- *cat.c, echo.c, nohup.c, seq.c, tee.c:* We picked some tools from the Unix coreutils in Busybox 1.14, which can be downloaded from `http://www.busybox.net/`. These tools were also studied in [CDE08].
- *PicoSAT:* As an example of a complete software package, we analyzed the sources of the SAT solver PicoSAT. We used version 913 from `http://fmv.jku.at/picosat/`.

- *joplift.c:* An elevator control software initially used for benchmarking a Java-optimized processor (JOP) in [PS07]. A student translated the code manually from Java to C and, for use with CREST [BS08], instrumented, and merged into a single file using CIL [NMRW02].

Experimental Setup. All experiments described in this chapter were performed on a Debian/Lenny 64bit Linux system running on an Intel Xeon series E5345 CPU running at 2.33 GHz. The system is equipped with 16 GB of main memory, but most of our experiments only require a fraction of that. We describe execution times and memory usage in detail with each of our experiments.

We used version 1.2 of FShell 2, as available from `http://code.forsyte.de/fshell`. Newer versions of FShell will appear on this web page (superseding version 1.2), which shall not invalidate any of the experimental results.

7.3.1 Efficient Evaluation of Complex Queries

We evaluated the example specifications **Q1-24** shown in Section 5.8 with our tool. In order to do so, we applied each specification to one of the three suitable source files, which were shown in Listings 7.1–7.3.

In Table 7.2 we summarize the results of evaluating **Q1-24** from Section 5.8 with FShell. For each FQL specification we first list the source file it was evaluated on. Then we give the number of test goals (#goals), the number of test cases (#cases) determined by the back end, and the number of infeasible test goals (#infeas).

All specifications were processed in less than one second (see column "Time"). Each run of the test case generation engine required no more than 60 MB of memory (column "Mem"). We observe that FShell scales well even for **Q20** with more than 14,000 test goals.

In addition to this broad range of complex queries on small program fragments we experimented with non-trivial queries on larger code bases. We applied a series of queries to autopilot from the PapaBench suite (see also description above). We used the following templates for queries **C1-6** on autopilot that enforce a certain function call stack and focus coverage criteria to single functions:

Query	Source	#goals	#cases	#infeas	Time [s]	Mem [MB]
Q1	list2.c	11	3	0	0.27	53
Q2	list2.c	8	3	0	0.28	53
Q3	list2.c	8	3	0	0.29	53
Q4	sort2.c	6	2	2	0.12	53
Q5	sort2.c	9	3	3	0.11	53
Q6	list2.c	2	1	0	0.16	53
Q7	list2.c	2	1	0	0.17	53
Q8	list2.c	4	1	2	0.18	53
Q9	list2.c	11	3	0	0.27	57
Q10	sort1.c	2	1	0	0.13	52
Q11	sort1.c	8	1	1	0.40	58
Q12	sort2.c	2	1	0	0.10	53
Q13	sort1.c	1	1	0	0.11	52
Q14	sort2.c	6	1	1	0.10	52
Q15	list2.c	8	1	3	0.30	54
Q16	sort1.c	29	2	0	0.12	52
Q17	sort1.c	12	1	0	0.10	52
Q18	list2.c	121	4	53	0.85	58
Q19	list2.c	1331	4	1060	0.74	56
Q20	list2.c	14641	4	13927	0.87	59
Q21	sort1.c	2	1	0	0.08	52
Q22	sort1.c	4	1	3	0.10	52
Q23	sort2.c	3	1	0	0.09	52
Q24	sort2.c	2	0	2	0.11	53

Table 7.2: Experimental results for example specifications

C1-3
```
cover @CONDITIONEDGE & @FUNC(fn)
passing @ENTRY(periodic_task) -> @ENTRY(fn)
```

C1-3
```
cover (@CONDITIONEDGE & @FUNC(periodic_task))
      -> (@CONDITIONEDGE & @FUNC(fn))
passing @ENTRY(periodic_task) -> @ENTRY(fn)
```

We instantiate these templates by substituting for fn the names of functions `course_run`, `climb_control_task`, and `altitude_control_task`. We thus query for a test suite satisfying condition coverage in function fn (**C1-3**), and combined condition coverage of function `periodic_task` and function fn in queries **C4-6**. For all test cases we require that they first pass through the entry of function `periodic_task` and thereafter reach the entry of function fn.

The results are summarized in Table 7.3. We conclude that FShell nicely handles non-trivial queries over larger code bases – autopilot has 5945 lines of code. Note, however, that execution times range from 4.8 seconds to 26 seconds, with a comparatively small variance in memory usage only. A closer look at each of the experiments showed that in all cases approximately 2 seconds were spent in preparation of the SAT formula (unwinding and query augmentation), and the rest of the time is spent in solving the SAT formula. This states a prime example of query cost varying with the exact choice of test goals: the cost of computing appropriate inputs for condition coverage in fn varies with the choice of fn. Given that our options for fn are functions occurring on distinct code paths, this was to be expected.

Query	Function fn	#goals	#cases	#infeas	Time[s]	Mem[MB]
C1	`course_run`	3	1	1	4.77	280
C2	`climb_control_task`	3	1	1	22.43	275
C3	`altitude_control_task`	3	1	1	14.71	283
C4	`course_run`	57	1	21	12.81	305
C5	`climb_control_task`	57	1	21	14.34	302
C6	`altitude_control_task`	57	1	21	25.65	304

Table 7.3: Experimental results for complex queries

7.3.2 Applicability to Real-World Software

To study applicability of our back end to real-world embedded systems code, and possibly also software systems, we chose a set of program-independent queries and applied them to the following set of programs: (1) we picked some tools from the Unix coreutils in Busybox, studied as well in [CDE08], (2) we selected kbfiltr from the Windows DDK, initially studied in [BCH+04b], and (3) we chose an example use case from [GLMS09] where model checking tools were applied to the Linux virtual file system layer. In addition to these well studied examples we applied our framework on two industrial case studies. (4) We performed test case generation for an engine controller code generated from a MATLAB/Simulink model (matlab.c). (5) We examined a dynamic memory manager for airborne software systems (memman.c). (6) For a more extensive study of embedded systems code we added three examples from the Mälardalen WCET Benchmark suite. (7) As an example of a complete software package, we analyzed the sources of the SAT solver PicoSAT.

We summarize our experiments in Table 7.4. To compare to previous

	BB (**Q1**)		CC (**Q2**)		BB^2 (**Q18**)		
Source	#goals	#cases	#goals	#cases	#goals	#cases	#infeas
cat.c	15	3	10	3	225	3	164
echo.c	23	3	18	3	529	3	502
nohup.c	19	4	14	4	361	10	212
seq.c	33	5	26	4	1089	15	602
tee.c	17	2	14	2	289	2	242
kbfiltr	239	53	201	54	57121	164	48492
pseudo-vfs.c	9	3	6	3	81	3	53
matlab.c	42	4	35	4	1764	10	1393
memman.c	55	2	42	3	3025	4	2230
adpcm.c	130	1	72	1	16900	1	6354
nsichneu.c	508	1	506	1	258064	1	193039
statemate.c	277	1	247	1	76729	1	75677
PicoSAT	396	29	294	28	156816	401	79442

Table 7.4: Summary of real-world experiments

work, we first established basic block coverage (specification **Q1**). We give the number of test goals and the number of test cases that were necessary to cover these test goals. Given loop bounds of 3 to 100 (cf. Table 7.1), we compute test suites for 100% coverage of all feasible test goals. In [CDE08] in many cases coverage of more than 90% is achieved, but the feasibility of the remaining test goals is not investigated.

Furthermore, we achieved condition coverage with spec **Q2** and "squared" basic block coverage with spec **Q18** for all benchmarks. In case of **Q18**, many of the resulting test goals are expectedly infeasible. We include these numbers in the column "#infeas".

As shown in full detail in Table 7.9, columns titled "SAT Encoding", all experiments (except for Mälardalen benchmarks and PicoSAT, as discussed below) were performed using at most 350 MB of memory. Each test suite was computed in less than 22 seconds. As Mälardalen benchmarks and PicoSAT have a larger code base involving a series of loops, the experiments for basic block coverage and condition coverage took up to 54 seconds and required up to 733 MB. For squared basic block coverage, the experiments took approximately 55 minutes and consumed 6.9 GB of memory.

7.3.3 Comparing to other Test Case Generation Approaches

We first compare FShell with random testing and directed testing – two well-established techniques for test case generation, and afterwards compare with BLAST, which is conceptually closest to FShell. We strictly focus on performance in this comparison, leaving aside other added value of query-driven program testing and usability aspects such as FShell effectively printing test suites or not requiring any instrumentation.

For the comparison with random testing and directed testing we chose an elevator control software (joplift.c as listed above). Even though this is a program of moderate size only, as a real-world control software it contains a number bit-operations and its control has to distinguish a series of cases. This results in 124 possible branches, out of which 123 are feasible. The remaining one cannot be taken as surrounding guards fix the value of the corresponding condition. As coverage metric we chose condition coverage (**Q2**), which means we have 123 feasible test goals.

To perform random testing we inserted calls to `rand()` to initialize in-

put parameters randomly. For improved coverage results we added specific knowledge about the code and mapped the randomly chosen values into the interval $[0, 99]$.

Directed testing was done using CREST [BS08], which implements concolic testing as pioneered by CUTE [SMA05] and DART [GKS05]. CREST supports several traversal strategies, but for our experiments depth-first search performed best.

Our results are summarized in Table 7.5. For both random testing and directed testing the number of iterations had to be fixed, as shown in column "Iterations". For random testing this also determines the number of test cases being generated. A uniform random number generator would eventually yield full coverage as the number of iterations is increased. As expected, random testing is extremely fast; here, the time of one second, as needed for one million iterations, even includes repeated execution of the main control loop of joplift.c for coverage measurement. Coverage, however, remains low. Despite one million test cases being computed, not even 50% of the branches are taken. In directed testing, using CREST, the iteration count only sets an upper limit on the number of test cases being computed. Once no more branches can be covered, test case generation stops. As the decision procedures employed by CREST are incomplete for C programs, an uncovered branch does not necessarily mean infeasibility. As shown in Table 7.5, we observe such behavior in our experiment. CREST stops after covering 78 of 123 branches. Most likely this is due to bit-operations being employed by joplift.c. For CREST, the fastest test case generation approach was found to be a depth-first search strategy. This resulted in an execution time of approximately 13 seconds.

Tool	Iterations	#cases	Time [s]	Coverage
Random	10	10	0.01	16.4%
Random	1000000	1000000	1.03	42.1%
CREST	10	10	0.31	43.9%
CREST	5000	527	13.22	63.4%
FShell	–	36	1.30	100.0%

Table 7.5: Comparison of FShell with random testing and directed testing to achieve condition coverage

Although FShell is not as fast as random testing, it was the only approach to achieve full coverage in this experiment. In order to do so, only 36 test cases are necessary. Of course it must be stated that the chosen benchmark is favorable for FShell: it includes few loops and is of moderate code size, requires bit-precise modeling, and contains complex conditions. An iterative approach like directed testing will likely scale much better to large programs with complex loops.

As the concepts implemented in FShell are related to both directed testing and model checking, we chose BLAST as further benchmarking reference. To the best of our knowledge, the test case generation engine built into BLAST is the only tool coming close to the capabilities of FShell in terms of controllability of test case generation. Still, it should be noted that BLAST is a full fledged model checker and is not as optimized towards testing as FShell is. A detailed discussion of BLAST and its query language was given in Section 1.6.

Their set of benchmarks, presented in [BCH+04b], is well documented and all source files are publicly available. To achieve equivalent test goals, we generated test suites with basic block coverage. The results are summarized in Table 7.6. The results for BLAST are taken literally from [BCH+04b], because the version of BLAST performing test case generation is currently unavailable. As Beyer et al. performed their experiments on a 3.06 GHz Dell Precision 650 with 4 GB RAM, comparison of execution times should be taken with a grain of salt – our system has a lower clock rate but likely comes with a higher memory bandwidth, larger caches, etc.

	BLAST		FShell (**Q1**)		
Source	#cases	Time [s]	#cases	Time[s]	Speedup
cdaudio	85	1500	76	24.03	62.4
floppy	111	1500	69	33.24	45.1
kbfiltr	39	300	53	3.76	79.8
parport	213	5460	133	117.16	46.6

Table 7.6: Results on device drivers – comparison with BLAST

Compared to the execution times reported in [BCH+04b] we observe speedups of 45 to nearly 80 times. The generated test suites are mostly

smaller, with the exception of kbfiltr, where FShell computes 53 test cases and BLAST found a covering test suite with 39 test cases.

	FShell (**Q18**)		
Source	#goals	#cases	Time[s]
cdaudio	147456	249	118.36
floppy	195364	232	178.02
kbfiltr	57121	158	19.77
parport	1067089	349	447.33

Table 7.7: Results on device drivers – **Q18**

To study scalability on these device drivers, we additionally computed "squared" basic block coverage (**Q18**). As shown in Table 7.7, we find covering test suites for possibly more than one million test goals (parport with 1,067,089 test goals) in less than eight minutes. Our experiments with **Q18** required up to 3.4 GB of memory, whereas basic block coverage (**Q1**) was doable within 1.4 GB of memory.

As discussed in the next section, however, we had to refrain from testing a file parclass of the Windows DDK suite, which was successfully studied in [BCH+04b].

7.3.4 Scalability

To discuss scalability of query-driven program testing we must distinguish complexity inherent in queries on the one hand and the cost of evaluating queries over complex programs using FShell on the other hand. The first problem is intrinsically tied to query-driven program testing. Dealing with complex programs, however, is almost exclusively an issue of the back end. The only inevitable aspect is evaluation of filter functions to target graphs, the complexity of which is program dependent.

The feasibility of handling complex queries was already shown in Section 7.3.1. Furthermore, experiments resulting in large sets of test goals were described in Table 7.7: for parport FShell easily handled more than one million test goals. In Table 7.2 we also showed that going from simple basic block coverage (**Q1**) over "squared" basic block coverage to quadruples of basic blocks scales well.

The scalability of FShell as a back end in query-driven program testing is tied to the source code under scrutiny. As FShell employs bounded model checking the main limitations are large code bases and complex loops. In our experiments, as described above, we hence focused on programs of moderate size, with few loops. One of the programs that failed testing using FShell is parclass from the Windows DDK, which was previously studied using BLAST in [BCH+04b]. This device driver consists of 100853 SLOC and has several loops. We aborted all attempts of unwinding using FShell 2, version 1.2, after 12 hours. As we already reported successful testing using earlier versions of FShell [HSTV08], however, this specific case deserves further investigation.

Finally we also studied the net effect of using incremental test case generation vs. naïve iterative invocations of the underlying model checker. Note that in the latter case coverage constraints are not even considered, i.e., in this setup all calls likely yield the same test case.

Source	**Q2**	**Q18**
kbfiltr	19.8	12.5
PicoSAT	8.7	1.9

Table 7.8: Speedup by incremental test case generation

In Table 7.8, we present the speedup achieved by generating covering test suites in a single test job. We exemplified these comparisons for condition coverage (**Q2**) and "squared" basic block coverage (**Q18**). To estimate the speedup, for the naïve approach we multiplied the cost of computing a single test case with the number of test cases computed by FShell in incremental test case generation. Most notably, this simplified model misses the effort needed for computing the coverage of one test case, which is the main cost in test case generation for PicoSAT/**Q18**. Therefore, in this case, we find the estimated speedup to be only as low as a factor of 1.9. For kbfiltr we observe a speedup that is an order of magnitude larger.

7.4 Comparison of Instrumentation vs. Native SAT Encoding

As described in Chapter 6, FShell supports two modes of solving FQL queries. The technically straightforward approach is the use of instrumentation, where automata corresponding to FQL queries are added as additional code to the program. This instrumented program is then translated into a SAT formula. While more easily implemented, it hinders efficiency in a number of ways. First, programs with a large number of statements yield correspondingly many instrumentation points, causing a blowup of the program. Second, non-trivial automata, as derived from queries such as "squared" basic block coverage, result in complex interactions of the instrumentations added to the program. At the time of unwinding the instrumented program this yields guards in the resulting equation that have to consider all these complex interactions.

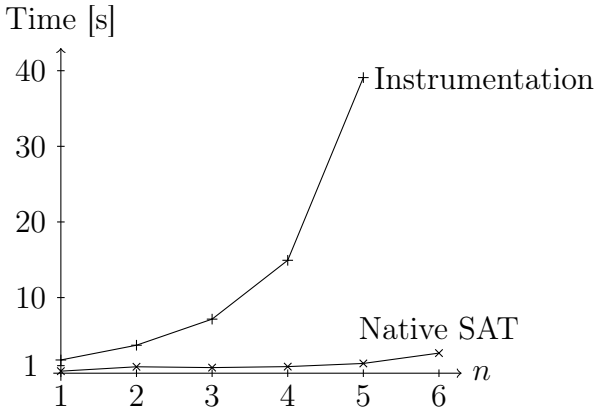
To avoid such problems, we further implemented the direct encoding of automata into the SAT formula of the unmodified program. This avoids problems with blowup in the unwinding step, at the expense of a more complex SAT formula. The experiments described in this section show that this additional cost is clearly acceptable as (1) scalability with complex queries is improved and (2) we gain speedups of factors of more than 700.

We study the scalability for complex queries on the example of multiple basic block coverage, i.e., queries **Q1**, **Q18-20**, and new queries for five- and sixfold combination of basic blocks (abbreviated as BB^5 and BB^6, respectively). We apply these queries to list2.c from Listing 7.1. Figure 7.1 shows the results of this comparison. We observe that even for BB^6, which yields 1,771,561 test goals, the SAT-based approach scales well in terms of execution time (2.66 seconds), whereas the use of instrumentation causes execution time to rise to nearly 4 minutes. Notice that list2.c is only 15 lines in size.

To investigate scalability of the SAT-based encoding in comparison to instrumentation on real-world code we used the examples described in Section 7.3.2. The results are shown in Table 7.9. Through the use of native SAT encoding we observe speedups over the instrumentation-based approach up to a factor of 707.29. For benchmarks with a large number of statements resulting from unwinding the program, such as adpcm.c and PicoSAT, we notice a comparatively small speedup (factors of 4 and 34) for "squared"

	Instrumentation		SAT Encoding	
Query	Time [s]	Mem [MB]	Time [s]	Mem [MB]
Q1	1.75	81	0.27	53
Q18	3.70	92	0.85	58
Q19	7.15	105	0.74	56
Q20	14.93	119	0.87	59
BB^5	39.09	151	1.29	83
BB^6	224.31	451	2.66	339

(a) Execution times and memory usage for **Q1**, **Q18-20**, BB^5, and BB^6



(b) Execution times for n-fold Cartesian combination of basic blocks (BB^6 omitted for instrumentation-based approach)

Figure 7.1: Comparison of instrumentation-based approach vs. native SAT encoding

basic block coverage. This is mainly due to coverage analysis employed in the SAT-based setting becoming costly. We are investigating optimizations using AVL DAGs as proposed by Myers [Mye84] and persistent data structures [DSST89] to remedy this.

		Instrumentation		SAT Encoding		
Source	Query	Time [s]	Mem [MB]	Time [s]	Mem [MB]	Speedup
cat.c	**Q1**	3.88	127	0.27	88	14.37
cat.c	**Q2**	3.10	124	0.27	88	11.48
cat.c	**Q18**	7.61	144	0.30	90	25.37
echo.c	**Q1**	7.45	151	0.36	92	20.69
echo.c	**Q2**	7.11	150	0.36	92	19.75
echo.c	**Q18**	13.26	173	0.40	95	33.15
nohup.c	**Q1**	0.56	89	0.12	82	4.67
nohup.c	**Q2**	0.55	88	0.13	82	4.23
nohup.c	**Q18**	1.35	93	0.20	82	6.75
seq.c	**Q1**	6.15	131	0.31	84	19.84
seq.c	**Q2**	6.10	129	0.35	84	17.43
seq.c	**Q18**	20.84	152	1.10	88	18.95
tee.c	**Q1**	9.37	156	0.41	98	22.85
tee.c	**Q2**	9.05	156	0.38	98	23.82
tee.c	**Q18**	16.84	179	0.52	102	32.38
kbfiltr	**Q1**	195.42	348	3.77	158	51.84
kbfiltr	**Q2**	171.25	341	3.48	158	54.97
kbfiltr	**Q18**	1174.74	542	21.37	342	54.97
pseudo-vfs.c	**Q1**	0.19	57	0.02	47	9.50
pseudo-vfs.c	**Q2**	0.17	57	0.01	47	17.00
pseudo-vfs.c	**Q18**	0.24	59	0.02	48	12.00
matlab.c	**Q1**	105.81	139	0.67	93	157.93
matlab.c	**Q2**	98.97	134	0.54	94	183.28
matlab.c	**Q18**	109.70	147	1.35	100	81.26
memman.c	**Q1**	18.70	183	1.97	101	9.49
memman.c	**Q2**	15.45	179	1.94	101	7.96
memman.c	**Q18**	44.35	214	2.63	105	16.86

Source	Query	Instrumentation Time [s]	Instrumentation Mem [MB]	SAT Encoding Time [s]	SAT Encoding Mem [MB]	Speedup
adpcm.c	**Q1**	5347.08	1277	7.56	374	707.29
adpcm.c	**Q2**	5001.25	1245	7.43	373	673.12
adpcm.c	**Q18**	13258.60	1848	3112.60	1253	4.26
nsichneu.c	**Q1**	1862.84	903	8.02	321	232.27
nsichneu.c	**Q2**	1850.56	903	5.60	321	330.46
nsichneu.c	**Q18**	11706.10	1220	62.50	434	187.30
statemate.c	**Q1**	252.46	338	0.77	91	327.87
statemate.c	**Q2**	232.31	329	0.72	91	322.65
statemate.c	**Q18**	473.45	461	1.67	110	283.50
PicoSAT	**Q1**	9276.24	2112	52.86	733	175.49
PicoSAT	**Q2**	8184.68	2057	53.14	705	154.02
PicoSAT	**Q18**	112856.00	5205	3260.04	6865	34.62

Table 7.9: Execution times and memory usage for real-world code experiments with instrumentation-based approach vs. native SAT encoding

7.5 Minimization of Test Suites

In practical testing it is often crucial to have test suites as small as possible, for a number of reasons: (1) execution of each test case must be observed to spot wrong behavior; (2) in certified processes each test case must be documented; (3) execution of tests takes time and blocks resources. Automatic test case generation must hence ensure that the produced test suites are as small as possible. FShell therefore implements a-posteriori minimization of test suites that computes the smallest covering test suite from a given set of test cases.

As this step requires solving an optimization problem, it can be costly. We therefore describe the overhead in terms of memory and time incurred by the minimization step. In Table 7.10 we list for a selected subset of our benchmarks the test cases removed by minimization ("#removed"), the resulting number of test cases ("#cases"), the time the minimization step took, and the additional memory required to perform this step. We see that mini-

Source	Query	#removed	#cases	Time [s]	Mem [MB]
parport	**Q1**	10	133	0.04	0
parport	**Q18**	174	349	4.90	46
parport	**Q2**	15	133	0.04	0
cat.c	**Q1**	1	3	0.00	0
cat.c	**Q2**	0	3	0.00	0
cat.c	**Q18**	2	3	0.00	0
seq.c	**Q1**	2	5	0.00	0
seq.c	**Q2**	6	4	0.00	0
seq.c	**Q18**	12	15	0.02	0
matlab.c	**Q1**	2	4	0.00	0
matlab.c	**Q2**	1	4	0.00	0
matlab.c	**Q18**	0	10	0.01	0
PicoSAT	**Q1**	23	29	0.04	0
PicoSAT	**Q2**	14	28	0.02	0
PicoSAT	**Q18**	226	401	59.12	3780
joplift.c	**Q2**	13	36	0.02	0

Table 7.10: Results and overhead of test suite minimization

mization results in test suites 40% the size of the originally computed suite (**Q2** on seq.c). Even though there is not always potential for minimization, even reducing already small test suites works well. For large test suites, as in case of parport of PicoSAT, the minimization steps becomes more costly. Still we get reductions to two thirds of the original size and hence believe that this additional effort is well spent.

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

Edsger W. Dijkstra

Chapter 8

Conclusions

In this dissertation we presented a new approach to program testing: query-driven program testing follows the design principles of database systems, where we view the source code as a database of program executions, and the user formulates queries over the program. The query language, FQL, is designed as a declarative specification language for test suites. Its declarative nature shields the user from implementation details of the query solving back end.

In Chapter 2 we stated five challenges for the design of a test specification language:

(a) **Simplicity and Code Independence.** Regular languages as base formalism make FQL easy to read; Section 5.8 demonstrates that even complex criteria have simple specifications. Our use of control flow automata and the concept of target graphs ensure code independence.

(b) **Encapsulation of Language Specifics.** We obtain target graphs from control flow automata using filter functions (Section 5.4.2), which uniquely encapsulate the programming language specific aspects of the software under test and coverage criteria defined thereupon.

(c) **Precise Semantics.** We have given a formal definition of coverage criteria in Chapter 4 and provided a precise semantics of our language FQL in Chapter 5. Every FQL specification yields an elementary coverage criterion.

(d) **Expressive Power.** We have demonstrated that all informal specifications of Figures 2.1–2.3 can be expressed in FQL. Yet we designed

FQL as an open framework to be extended as new requirements arise.

(e) **Tool Support for Real World Code.** In Chapter 7 we presented experimental results for our test case generation back end FShell, which we presented in Chapter 6. Amongst others, we generated test suites for device drivers, a SAT solver, and embedded systems code.

The query solving back end accepts an FQL specification and C source code as input. As output it then computes a test suite matching the elementary coverage criterion described by the FQL specification. Thus our tool FShell treats a C program as a database which is queried by the user. FShell implements a query solving back end based on bounded model checking. Our back end uses two new algorithms which guide the SAT solver to efficiently enumerate test suites.

Our implementation FShell demonstrates the effectiveness and versatility of query-driven program testing. FShell provides an interactive shell-like interface to state FQL queries. It builds upon CBMC to translate a given C program into a SAT formula. To solve a query, FShell additionally encodes automata representing the path patterns derived from the query.

The new algorithms, iterative and groupwise constraint strengthening, enable us to efficiently generate test suites for a query given in our specification language FQL. Starting with an initially empty test suite, we incrementally add new test cases that satisfy yet uncovered test goals until we reach a fixed point. Then every feasible test goal is covered by the resulting test suite. Our experimental results confirm the practical feasibility and relative efficiency of our approach.

Future Work

Despite the contributions already made in this work, there remain a number of research questions to be addressed and technical improvements to be implemented. We will first list a set of open questions for FQL, and then discuss open issues around FShell.

The language FQL gives rise to a number of interesting questions, both about the formalism and its efficient evaluation.

- How to check equivalence and subsumption of specifications? How can we approximate a specification by a simpler one with a larger test suite;

where is a good trade-off? How can we rewrite a specification into a normal form for which test cases can be found more easily?

- How can we trace code changes that compromise the meaning of a test specification? How can we reuse existing test suites after code changes? When can we reuse existing test suites for new specifications?
- Which specifications are amenable to directed testing? How can we combine incomplete light-weight testing with FQL back ends for better efficiency? How can we build efficient predicate abstraction based tools for FQL test case generation? How can we distribute specifications over multiple servers and solve them in parallel?
- How to obtain feedback about an infeasible test goal? How can we succinctly describe a set of test goals that is not yet covered by a given test suite?
- How to capture criteria such as MC/DC that go beyond elementary coverage criteria? How can we combine FQL with input/output tables and executable specifications? How can we apply FQL to high level models such as UML? How can we extend FQL to support testing of concurrent software?

Concerning FShell, we also plan to add a number of improvements in future versions.

- **Usability.** We plan to provide a plug-in for the Eclipse IDE. Furthermore, the output of the command line tool shall be improved according to feedback from industry and users in academia.
- **Concurrent Software.** One of our major next steps will be developing support for concurrent software, both in the back end as well as at language level, following earlier work such as [BAEFU06] or [FNU03]. We expect improved support for concurrent C programs in CBMC, which will also bring support to testing concurrent software to FShell. For proper testing of such software, however, we will also need to extend FQL, as said above.
- **Performance Improvements.** Extensive studies of FQL and its properties will also foster more efficient processing of certain queries in FShell.

Bibliography

[A-M07] *Proceedings of the 3rd Workshop on Advances in Model Based Testing, A-MOST 2007, co-located with the ISSTA 2007 International Symposium on Software Testing and Analysis, London, United Kingdom, July 9-12*. ACM, 2007.

[Ame99a] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1999.

[Ame99b] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C*, chapter 6.10.4. In ANSI C [Ame99a], 1999.

[Ame99c] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C*, chapter 6.7.8. In ANSI C [Ame99a], 1999.

[AOX08] Paul Ammann, Jeff Offutt, and Wuzhi Xu. Coverage criteria for state based specifications. In Hierons et al. [HBH08], pages 118–156.

[AR04] George S. Avrunin and Gregg Rothermel, editors. *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*. ACM, 2004.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *POPL*, pages 1–11, 1988.

[BAEFU06] Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. Producing scheduling that causes concurrent programs to fail. In Ur and Farchi [UF06], pages 37–40.

[Bak57] Charles L. Baker. Review of D. D. McCracken, digital computer programming. *Math. Comput.*, 11(60):298–305, 1957.

[Bal04] Thomas Ball. A theory of predicate-complete test coverage and generation. In de Boer et al. [dBBGdR05], pages 1–22.

[BBS06] Laura Brandán Briones, Ed Brinksma, and Mariëlle Stoelinga. A semantic framework for test coverage. In Graf and Zhang [GZ06], pages 399–414.

[BBvB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Cleaveland [Cle99], pages 193–207.

[BCH+04a] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The Blast query language for software verification. In Giacobazzi [Gia04], pages 2–18.

[BCH+04b] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In ICSE 2004 [ICS04], pages 326–335.

[BGN+03] Michael Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Towards a tool environment for model-based testing with AsmL. In Petrenko and Ulrich [PU04], pages 252–266.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[BHJP04] Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson. Specifying and generating test cases using observer automata. In Grabowski and Nielsen [GN05], pages 125–139.

[BHM+09] Angelo Brillout, Nannan He, Michele Mazzucchi, Daniel Kroening, Mitra Purandare, Philipp Rümmer, and Georg Weissenbacher. Mutation-based test case generation for Simulink models. In de Boer et al. [dBBHL10], pages 208–227.

[Bie08] Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[BK08] Sven Bünte and Raimund Kirner. The acquaintance of hardware timing effects: A sine qua non to validate temporal requirements in embedded real time systems. In *Junior Scientist Conference*, November 2008.

[BKZT11] Sven Bünte, Raimund Kirner, Michael Zolda, and Michael Tautschnig. Improving the confidence in measurement-based timing analysis. In ISORC 2011 [ISO11]. To appear.

[BM83] David L. Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[BM93] W. G. Bently and E. F. Miller. Ct coverage âĂŤ initial results. *Software Quality Journal*, 2:29–47, 1993. 10.1007/BF00417425.

[BNvGV07] Koen Bertels, Walid A. Najjar, Arjan J. van Genderen, and Stamatis Vassiliadis, editors. *FPL 2007, International Conference on Field Programmable Logic and Applications, Amsterdam, The Netherlands, 27-29 August 2007*. IEEE, 2007.

[BS08] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In Inverardi et al. [IIV08], pages 443–446.

[Büc60] J. Richard Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[Bul] BullseyeCoverage 7.11.15. `http://www.bullseye.com/`.

[BV05] Franz Baader and Andrei Voronkov, editors. *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004, Montevideo, Uruguay, March 14-18, 2005, Proceedings*, volume 3452 of *Lecture Notes in Computer Science*. Springer, 2005.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[CCGR99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model verifier. In Halbwachs and Peled [HP99], pages 495–499.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Draves and van Renesse [DvR08], pages 209–224.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Kozen [Koz82], pages 52–71.

[Cer02] Certification Authorities Software Team. What is a "decision" in application of modified condition/decision coverage (MC/DC) and decision coverage (DC)? CAST-10, 2002.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of com-

puting static single assignment form. In *POPL*, pages 25–35, 1989.

[Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, 4(3):178–187, 1978.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Jensen and Podelski [JP04], pages 168–176.

[CKY03] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In DAC 2003 [DAC03], pages 368–371.

[Cle99] Rance Cleaveland, editor. *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*. Springer, 1999.

[CMe] CoverageMeter 5.0.3. `http://www.coveragemeter.com/`.

[COM01] *25th International Computer Software and Applications Conference (COMPSAC 2001), Invigorating Software Development, 8-12 October 2001, Chicago, IL, USA*. IEEE Computer Society, 2001.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In STOC 1971 [STO71], pages 151–158.

[CPRZ89] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. Software Eng.*, 15(11):1318–1332, 1989.

[CTC] CTC++ 6.5.3. `http://www.verifysoft.com/en.html`.

[CTF01] Philippe Chevalley and Pascale Thévenod-Fosse. Automated generation of statistical test cases from UML state diagrams. In COMPSAC 2001 [COM01], pages 205–214.

[DAC01] *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. ACM, 2001.

[DAC03] *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*. ACM, 2003.

[DAT05] *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany*. IEEE Computer Society, 2005.

[dBBGdR05] Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors. *Formal Methods for Components and Objects, Third International Symposium, FMCO 2004, Leiden, The Netherlands, November 2 - 5, 2004, Revised Lectures*, volume 3657 of *Lecture Notes in Computer Science*. Springer, 2005.

[dBBHL10] Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors. *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, volume 6286 of *Lecture Notes in Computer Science*. Springer, 2010.

[DCM82] Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors. *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, volume 137 of *Lecture Notes in Computer Science*. Springer, 1982.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[Din04] George Din. TTCN-3. In Broy et al. [BJK+05], pages 465–496.

[DJK+99] Siddhartha R. Dalal, Ashish Jain, Nachimuthu Karunanithi, J. M. Leaton, Christopher M. Lott, Gardner C. Patton, and Bruce M. Horowitz. Model-based testing in practice. In *ICSE*, pages 285–294, 1999.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DNSVT07] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: a systematic review. In *WEASELTech*, pages 31–36, 2007.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

[DvR08] Richard Draves and Robbert van Renesse, editors. *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association, 2008.

[ECB01] *8th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2001), 17-20 April 2001, Washington, DC, USA*. IEEE Computer Society, 2001.

[EJ05] Michael D. Ernst and Thomas P. Jensen, editors. *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05, Lisbon, Portugal, September 5-6, 2005*. ACM, 2005.

[Elg61] Calvin C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98:21–51, 1961.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Giunchiglia and Tacchella [GT04], pages 502–518.

[FNU03] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In IPDPS 2003 [IPD03], page 286.

[FSWl08] Mario Friske, Bernd-Holger Schlingloff, and Stephan Weißleder. Composition of model-based test coverage criteria. In Giese et al. [GHNS08], pages 87–94.

[FW88] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Software Eng.*, 14(10):1483–1498, 1988.

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

[FW06] Gordon Fraser and Franz Wotawa. Property relevant software testing with model-checkers. *ACM SIGSOFT Software Engineering Notes*, 31(6):1–10, 2006.

[FWA09] Gordon Fraser, Franz Wotawa, and Paul Ammann. Testing with model checkers: a survey. *Softw. Test., Verif. Reliab.*, 19(3):215–261, 2009.

[Gen09] Ian P. Gent, editor. *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*. Springer, 2009.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Eng.*, 1(2):156–173, 1975.

[GH99] Angelo Gargantini and Constance L. Heitmeyer. Using model checking to generate tests from requirements specifications. In Nierstrasz and Lemoine [NL99], pages 146–162.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994.

[GHK+06] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. Synergy: a new algorithm for property checking. In Young and Devanbu [YD06], pages 117–127.

[GHNS08] Holger Giese, Michaela Huhn, Ulrich Nickel, and Bernhard Schätz, editors. *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV, Schloss Dagstuhl, Germany, 7.-9. April 2008, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, volume 2008-2 of *Informatik-Bericht*. TU Braunschweig, Institut für Software Systems Engineering, 2008.

[GHT09] Hermann Gruber, Markus Holzer, and Michael Tautschnig. Short regular expressions from finite automata: Empirical results. In Maneth [Man09], pages 188–197.

[Gia04] Roberto Giacobazzi, editor. *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*. Springer, 2004.

[Gie08] Holger Giese, editor. *Models in Software Engineering, Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*, volume 5002 of *Lecture Notes in Computer Science*. Springer, 2008.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Sarkar and Hall [SH05], pages 213–223.

[GLM08] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In NDSS 2008 [NDS08].

[GLMS09] Andy Galloway, Gerald Lüttgen, Jan Tobias Mühlberg, and Radu Siminiceanu. Model-checking the Linux virtual file system. In Jones and Müller-Olm [JMO09], pages 74–88.

[GM08] Aarti Gupta and Sharad Malik, editors. *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*. Springer, 2008.

[GN05] Jens Grabowski and Brian Nielsen, editors. *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, volume 3395 of *Lecture Notes in Computer Science*. Springer, 2005.

[God07] Patrice Godefroid. Compositional dynamic test generation. In Hofmann and Felleisen [HF07], pages 47–54.

[Gou83] John S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Trans. Software Eng.*, 9(6):686–709, 1983.

[GT04] Enrico Giunchiglia and Armando Tacchella, editors. *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*. Springer, 2004.

[GZ06] Susanne Graf and Wenhui Zhang, editors. *Automated Technology for Verification and Analysis, 4th International Symposium, ATVA 2006, Beijing, China, October 23-26, 2006*, volume 4218 of *Lecture Notes in Computer Science*. Springer, 2006.

[HAS01] *6th IEEE International Symposium on High-Assurance Systems Engineering (HASE 2001), Special Topic: Impact of Networking, 24-26 October 2001, Albuquerque, NM, USA, Proceedings*. IEEE Computer Society, 2001.

[HBH08] Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors. *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*. Springer, 2008.

[HdMR04] Grégoire Hamon, Leonardo Mendonça de Moura, and John M. Rushby. Generating efficient test sets with a model checker. In SEFM 2004 [SEF04], pages 261–270.

[Her76] P. M. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3):92–96, 1976.

[Het88] Bill Hetzel. *The complete guide to software testing.* QED Information Sciences, Inc., Wellesley, MA, USA, 2 edition, 1988.

[HF07] Martin Hofmann and Matthias Felleisen, editors. *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007.* ACM, 2007.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[HLM+08] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In Hierons et al. [HBH08], pages 77–117.

[HLSU02] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In Katoen and Stevens [KS02], pages 327–341.

[HO09] Hwa-You Hsu and Alessandro Orso. Mints: A general framework and tool for supporting test-suite minimization. In ICSE 2009 [ICS09], pages 419–429.

[Hol92] John H. Holland. *Adaptation in natural and artificial systems.* MIT Press, Cambridge, MA, USA, 1992.

[Hor02] R. Nigel Horspool, editor. *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*. Springer, 2002.

[How75] William E. Howden. Methodology for the generation of program test data. *IEEE Trans. Computers*, 24(5):554–560, 1975.

[HP99] Nicolas Halbwachs and Doron Peled, editors. *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*. Springer, 1999.

[HRV+03] Mats Per Erik Heimdahl, Sanjai Rayadurgam, Willem Visser, George Devaraj, and Jimin Gao. Auto-generating test sequences using model checkers: A case study. In Petrenko and Ulrich [PU04], pages 42–59.

[HS91] Mary Jean Harrold and Mary Lou Soffa. Selecting and using data for integration testing. *IEEE Softw.*, 8:58–65, March 1991.

[HSTV08] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. FShell: Systematic test case generation for dynamic analysis and measurement. In Gupta and Malik [GM08], pages 209–213.

[HSTV09] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. Query-driven program testing. In Jones and Müller-Olm [JMO09], pages 151–166.

[HSTV10] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. How did you specify your test suite ? In Pecheur et al. [PAN10], pages 407–416.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Hua75] J. C. Huang. An approach to program testing. *ACM Comput. Surv.*, 7(3):113–128, 1975.

[HW03] Dieter Hogrefe and Anthony Wiles, editors. *Testing of Communicating Systems, 15th IFIP International Conference, TestCom 2003, Sophia Antipolis, France, May 26-28, 2003, Proceedings*, volume 2644 of *Lecture Notes in Computer Science*. Springer, 2003.

[ICS04] *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*. IEEE Computer Society, 2004.

[ICS09] *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009.

[IIV08] Paola Inverardi, Andrew Ireland, and Willem Visser, editors. *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*. IEEE, 2008.

[Int98] International Electrotechnical Commission. Functional safety of electrical / electronic / programmable electronic safety-related systems. IEC standard 61508, 1998.

[Int03] International Organization for Standardization. *ISO/IEC 13485:2003: Medical devices – Quality management systems – Requirements for regulatory purposes*. International Organization for Standardization, Geneva, Switzerland, 2003.

[Int06] International Organization for Standardization. *IEC 62304:-2006: Medical device software - Software life cycle processes*. International Organization for Standardization, Geneva, Switzerland, 2006.

[IPD03] *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*. IEEE Computer Society, 2003.

[ISO11] *2011 IEEE International Symposium on Object/Component/-Service-Oriented Real-Time Distributed Computing, ISORC 2011, Newport Beach, CA, USA, 28-31 March 2011*. IEEE Computer Society, 2011.

[JMO09] Neil D. Jones and Markus Müller-Olm, editors. *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*. Springer, 2009.

[JP04] Kurt Jensen and Andreas Podelski, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*. Springer, 2004.

[JSE96] B. F. Jones, H. H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, USA, 1972.

[KGTB07] Nicolas Kicillof, Wolfgang Grieskamp, Nikolai Tillmann, and Víctor A. Braberman. Achieving both model and code coverage with automated gray-box testing. In A-MOST 2007 [A-M07], pages 1–11.

[Koz82] Dexter Kozen, editor. *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer, 1982.

[KS02] Joost-Pieter Katoen and Perdita Stevens, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*. Springer, 2002.

[KV08] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In Gupta and Malik [GM08], pages 423–427.

[LK83] Janusz W. Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Trans. Software Eng.*, 9(3):347–354, 1983.

[Man09] Sebastian Maneth, editor. *Implementation and Application of Automata, 14th International Conference, CIAA 2009, Sydney, Australia, July 14-17, 2009. Proceedings*, volume 5642 of *Lecture Notes in Computer Science*. Springer, 2009.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[MM63] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, 1963.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In DAC 2001 [DAC01], pages 530–535.

[MSBT04] Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. *The Art of Software Testing*. Wiley, 2 edition, June 2004.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[Mye84] Eugene W. Myers. Efficient applicative data types. In *POPL*, pages 66–75, 1984.

[NDS08] *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008.

[NL99] Oscar Nierstrasz and Michel Lemoine, editors. *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, volume 1687 of *Lecture Notes in Computer Science*. Springer, 1999.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In Horspool [Hor02], pages 213–228.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[NOT04] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract dpll and abstract dpll modulo theories. In Baader and Voronkov [BV05], pages 36–50.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(). *J. ACM*, 53(6):937–977, 2006.

[NS09] Ashalatha Nayak and Debasis Samanta. Model-based test cases synthesis using UML interaction diagrams. *ACM SIGSOFT Software Engineering Notes*, 34(2):1–10, 2009.

[Nta88] Simeon C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Software Eng.*, 14(6):868–874, 1988.

[OPV95] A. Jefferson Offutt, Jie Pan, and Jeffrey M. Voas. Procedures for reducing the size of coverage-based test sets. In *Proceedings of Twelfth International Conference on Testing Computer Software*, pages 111–123, 1995.

[Pai75] Michael R. Paige. Program graphs, an algebra, and their implication for programming. *IEEE Trans. Software Eng.*, 1(3):286–291, 1975.

[PAN10] Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors. *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. ACM, 2010.

[Par09] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 1st edition, 2009.

[PC90] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Software Eng.*, 16(9):965–979, 1990.

[PD09] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning sat solvers with restarts. In Gent [Gen09], pages 654–668.

[PHP99] Roy P. Pargas, Mary Jean Harrold, and Robert Peck. Test-data generation using genetic algorithms. *Softw. Test., Verif. Reliab.*, 9(4):263–282, 1999.

[PP06] Lori L. Pollock and Mauro Pezzè, editors. *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*. ACM, 2006.

[PS07] Christof Pitter and Martin Schoeberl. Time predictable CPU and DMA shared memory access. In Bertels et al. [BNvGV07], pages 317–322.

[PU04] Alexandre Petrenko and Andreas Ulrich, editors. *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003*, volume 2931 of *Lecture Notes in Computer Science*. Springer, 2004.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Dezani-Ciancaglini and Montanari [DCM82], pages 337–351.

[RH01a] Sanjai Rayadurgam and Mats Per Erik Heimdahl. Coverage based test-case generation using model checkers. In ECBS 2001 [ECB01], pages 83–.

[RH01b] Sanjai Rayadurgam and Mats Per Erik Heimdahl. Test-sequence generation from formal requirement models. In HASE 2001 [HAS01], pages 23–31.

[RTC92] Software considerations in airborne systems and equipment certification. RTCA/DO-178B, 1992.

[RTR] Rational Test RealTime 7.5. `http://www.ibm.com/software/awdtools/test/realtime/`.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Eng.*, 11(4):367–375, 1985.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *POPL*, pages 12–27, 1988.

[SDGR03] Ina Schieferdecker, Zhen Ru Dai, Jens Grabowski, and Axel Rennoch. The UML 2.0 testing profile and its relation to TTCN-3. In Hogrefe and Wiles [HW03], pages 79–94.

[SEF04] *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*. IEEE Computer Society, 2004.

[SH05] Vivek Sarkar and Mary W. Hall, editors. *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. ACM, 2005.

[SKC94] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *AAAI*, pages 337–343, 1994.

[SLM92] Bart Selman, Hector J. Levesque, and David G. Mitchell. A new method for solving hard satisfiability problems. In *AAAI*, pages 440–446, 1992.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Wermelinger and Gall [WG05], pages 263–272.

[SS99] João P. Marques Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[STO71] *Conference Record of Third Annual ACM Symposium on Theory of Computing, 1971, Shaker Heights, Ohio, USA*. ACM, 1971.

[TB03] Jan Tretmans and Ed Brinksma. TorX: Automated model-based tesing. In *First European Conference on Model-Driven Software Engineering*, pages 31–43, 2003.

[TG05] Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In Ernst and Jensen [EJ05], pages 35–42.

[The98] The Open Group. *Data Size Neutrality and 64-bit Support*. IEEE, 1998.

[Tra62] Boris A. Trakhtenbrot. Finite automata and monadic second order logic (russian). *Siberian Math. J*, 3:103–131, 1962. (English translation in *Amer. Math. Soc. Transl.* **59**, 1966, 23–55).

[TS05] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In Wermelinger and Gall [WG05], pages 253–262.

[TS06] Nikolai Tillmann and Wolfram Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38–47, 2006.

[TSL04] Li Tan, Oleg Sokolsky, and Insup Lee. Specification-based testing with linear temporal logic. In Zhang et al. [ZGD04], pages 493–498.

[UF06] Shmuel Ur and Eitan Farchi, editors. *Proceedings of the 4th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), PADTAD 2006, Portland, Maine, USA, July 17, 2006*. ACM, 2006.

[UL06] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[UY93] Hasan Ural and Bo Yang. Modeling software for accurate data flow representation. In *ICSE*, pages 277–286, 1993.

[VB08] Sergiy A. Vilkomir and Jonathan P. Bowen. From MC/DC to RC/DC: Formalization and analysis of control-flow testing criteria. In Hierons et al. [HBH08], pages 240–270.

[VCG+08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with Spec Explorer. In Hierons et al. [HBH08], pages 39–76.

[VPK04] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with java PathFinder. In Avrunin and Rothermel [AR04], pages 97–107.

[Wey86] Elaine J. Weyuker. Axiomatizing software test data adequacy. *IEEE Trans. Software Eng.*, 12(12):1128–1138, 1986.

[WG05] Michel Wermelinger and Harald Gall, editors. *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005.* ACM, 2005.

[WHH80] Martin R. Woodward, David Hedley, and Michael A. Hennell. Experience with path analysis and testing of programs. *IEEE Trans. Software Eng.*, 6(3):278–286, 1980.

[WRHM06] Michael W. Whalen, Ajitha Rajan, Mats Per Erik Heimdahl, and Steven P. Miller. Coverage metrics for requirements-based testing. In Pollock and Pezzè [PP06], pages 25–36.

[WRKP05] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter P. Puschner. Automatic timing model generation by CFG partitioning and model checking. In DATE 2005 [DAT05], pages 606–611.

[WS07] Stephan Weißleder and Bernd-Holger Schlingloff. Deriving input partitions from uml models for automatic test generation. In Giese [Gie08], pages 151–163.

[YD06] Michal Young and Premkumar T. Devanbu, editors. *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2005, Portland, Oregon, USA, November 5-11, 2006*. ACM, 2006.

[ZBK09] Michael Zolda, Sven Bünte, and Raimund Kirner. Towards adaptable control flow segmentation for measurement-based execution time analysis. In Laurent George, Maryline Chetto, and Mikael Sjodin, editors, *17th International Conference on Real-Time and Network Systems*, pages 35–44, Paris, France, October 2009.

[ZGD04] Du Zhang, Éric Grégoire, and Doug DeGroot, editors. *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, IRI - 2004, November 8-10, 2004, Las Vegas Hilton, Las Vegas, NV, USA*. IEEE Systems, Man, and Cybernetics Society, 2004.

[ZH93] Hong Zhu and P. A. V. Hall. Test data adequacy measurement. *Softw. Eng. J.*, 8:21–29, January 1993.

[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.

[ZK08] Michael Zolda and Raimund Kirner. Divide and measure: CFG segmentation for the measurement-based analysis of resource consumption. Technical Report 65/2008, Technische Universität Wien, Institut für Technische Informatik, 2008.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

[ZS94] Hantao Zhang and Mark E. Stickel. Implementing the davis-putnam algorithm by tries. Technical report, The University of Iowa, 1994.

Curriculum Vitae

Michael Tautschnig

Girardigasse 4/25
1060 Wien
Austria
michael.tautschnig@gmail.com
www.forsyte.at/~tautschnig

BIRTH 28th February 1983 in Rum, Austria
CITIZENSHIP Austrian

EDUCATION

Technische Universität München Munich, Germany
10/2002 – 4/2006
Diploma in computer science (major) and mathematics (minor) with distinction. Thesis title: Development of a tool to solve mixed logical/linear constraint problems. Thesis supervisor: Prof. Manfred Broy.

BRG Adolf-Pichler-Platz Innsbruck, Austria
9/1993 – 6/2001
Secondary school finished with distinction.

PROFESSIONAL ACTIVITIES

Program Committee
CSR 2010.

Conference Referee
STVR; STACS 2011; DATE 2011; HVC 2010; RTSS 2010; FMCAD 2010; ICTAC 2010; CAV 2010; CSR 2010; FMCAD 2009; QEST 2009; CAV 2009; ISoLA 2008; LATA 2007; ICALP 2007; CSR 2007.

Oxford University ComLab *Research assistant*
Oxford, United Kingdom **since 1/2011**
Research assistant in the group of Dr. Daniel Kroening.

Vienna University of Technology *Research assistant*
Vienna, Austria **1/2010 – 12/2010**
Ph.D. student in the group of Prof. Helmut Veith. Key researcher in BMWI grant 20H0804B in the frame of LuFo IV-2 project INTECO in collaboration with Diehl Aerospace.

Simon Fraser University *Internship*
Vancouver, Canada **11/2008 – 12/2008**
Research collaboration with Prof. Dirk Beyer.

Technische Universität Darmstadt *Research assistant*
Darmstadt, Germany **3/2008 – 12/2009**
Ph.D. student in the group of Prof. Helmut Veith. Key researcher in DFG grant FORTAS – Formal Timing Analysis Suite for Real Time Programs (VE 455/1-1); key researcher in INTECO in collaboration with Diehl Aerospace.

Technische Universität München *Research assistant*
Munich, Germany **5/2006 – 2/2008**
Ph.D. student in the group of Prof. Helmut Veith. Key researcher in research collaboration with BMW; key researcher in DFG grant FORTAS.

To me vi is zen.
To use vi is to practice zen.
Every command is a koan.
Profound to the user, unintelligible to the uninitiated.
You discover truth every time you use it.

Satish Reddy