An Introduction to Test Specification in FQL[⋆]

Andreas Holzer[1], Michael Tautschnig[1], Christian Schallhart[2], and Helmut Veith[1]

[1] Vienna University of Technology
Formal Methods in Systems Engineering
{holzer, tautschnig, veith}@forsyte.at
[2] Oxford University Computing Laboratory
christian.schallhart@comlab.ox.ac.uk

Abstract. In a recent series of papers, we introduced a new framework for white-box testing which aims at a separation of concerns between test specifications and test generation engines. We believe that establishing a common language for test criteria will have similar benefits to testing as temporal logic had to model checking and SQL had to databases. The main challenge was to find a specification language which is expressive, simple, and precise. This paper gives an introduction to the test specification language FQL and its tool environment.

1 Introduction

Testing is an integral part of software development. Applications of 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 or functions or enforce a precondition in the execution of a function high up in the call stack.
2. For requirement-based testing, we need test suites which reflect the intended system behavior.
3. For certification, we need test suites that ensure standard coverage criteria (e.g., condition coverage), in connection with industry standards such as DO-178B [1].

In most practical cases the situation is even more complex: For instance, while a system is still under development, we may want to assure condition coverage, but avoid covering certain unimplemented functions. Or we may want to combine basic block coverage in a large piece of code with full path coverage in a small (but critical) function. We possibly want to repeat this procedure for each function. When full path coverage is not achievable, we may want to approximate it by covering all pairs (or triples) of basic blocks in the program that can

⋆ Supported by BMWI grant 20H0804B in the frame of LuFo IV-2 project INTECO and by DFG grant FORTAS – Formal Timing Analysis Suite for Real Time Programs (VE 455/1-1).

be reached. In fact, the moment you start thinking about testing requirements, the wishes quickly exceed the possibilities of existing technology. 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. On the other hand, heuristic automated test case generation techniques such as random testing or directed testing [2,3,4,5,6,7] are very useful for general debugging, but can usually not achieve the coverage goals discussed here.

Approaching testing from a model checking background, we were quite surprised that the literature contains a rich taxonomy and discussions of test coverage criteria, 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. Absence of a formal specification means imprecision: In [8] we showed that commercial tools can be brought to disagree on condition coverage for programs of a few bytes length.

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.

Having a precise language to specify test criteria over a piece of source code opens many interesting research directions and development workflows:

1. Tools for test case generation [9,10,11]
2. Tools for the measurement of coverage [11]
3. Model-based tool chains which translate test specifications from model level to source-code level [12]
4. Hybrid tools which combine underapproximations by testing with overapproximations by model checking [4,6,13]
5. Distributed test case generation guided by precise specifications
6. Hybrid tools which take existing test suites, measure them and add the test cases needed for a certain coverage criterion; this naturally combines with randomized and directed testing, and regression testing

Our challenge therefore 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 the syntax and the semantics of the program under test in one formalism.

The current paper is a pragmatic introduction to FQL. For a more thorough treatment of FQL, we refer the reader to [8].

2 The Language Design Challenge

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."*

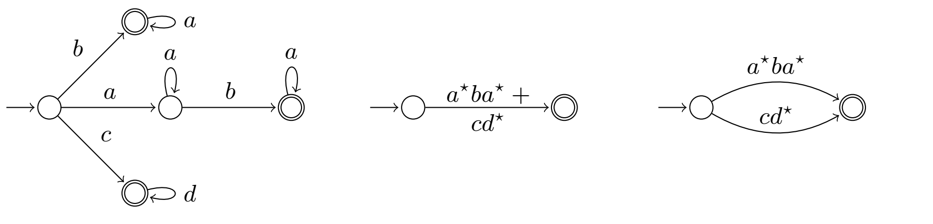
It is clear that basic block coverage can be achieved manually by writing one path pattern for each basic block in the program. The challenge is to find a specification language from which the path patterns can be automatically derived. This language should work not only for simple criteria such as basic 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.

3 Quoted Regular Expressions

Quoted regular expressions are a simple extension of regular expressions which enable us to selectively generate multiple regular expressions from a single one. Thus, quoted regular expressions can also be thought of as "meta-regular expressions". The use of quoted regular expressions will enable us to specify multiple tests in a single expression.



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

Fig. 1. Automata resulting from expansion of path patterns and coverage specifications

Syntactically, a quoted regular expression is just a regular expression with quoted subexpressions. We illustrate the effect of quoting on a simple example: A path pattern $a * .b.a * +c.d*$ describes an *infinite* language

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

with a corresponding finite automaton shown in Figure 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

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

and the automaton shown in Figure 1(b). If only the two subexpressions are quoted, i.e., we study "$a * .b.a *$" + "$c.d *$", we obtain two words, cf. Figure 1(c):

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

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 $x + y$.

We can now easily specify test suites by quoted regular expressions. Each word of a quoted regular expression – *a path pattern again describing a formal language* – defines one test goal to be covered. For instance, `"ID* . @4 . ID*" + "ID* . @6 . ID*"` requires that our test suite will contain one test case through line 4, and one (possibly the same one) through line 6.

In the following, we will disallow the use of the Kleene star $*$ *outside* quotes to assure finiteness of the test suite. On the other hand, we note that for path patterns, i.e., *inside* the quotes, we can possibly use pattern matching formalisms that are more powerful than regular expressions. We can allow, e.g., context-free features such as bracket matching.

4 The FQL Language Concept

In the following we will discuss the main features of FQL. We use the C code snippet shown in Listing 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 1. C source code of function `cmp`

4.1 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`.

4.2 Coverage Specifications: Quoted Regular Expressions

Using the regular alternative operator '`+`' we can build a path pattern matching all basic block entries in Listing 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*. Towards this end, we will introduce *coverage specifications*, i.e., quoted regular expressions, which describe a *finite* language *over path patterns*, where each word defines one *test goal*.

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*"
```

We emphasize the finiteness achieved by the omission of the Kleene star outside quotes: An infinite number of test goals would lead test case generation ad absurdum.

To summarize, the notion of coverage specifications/quoted regular expressions allows us to address issues A and B of the above list. In the following section, we will show how to address the remaining issues C and D.

4.3 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 supports 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. [14] 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 2 the CFA for Listing 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.

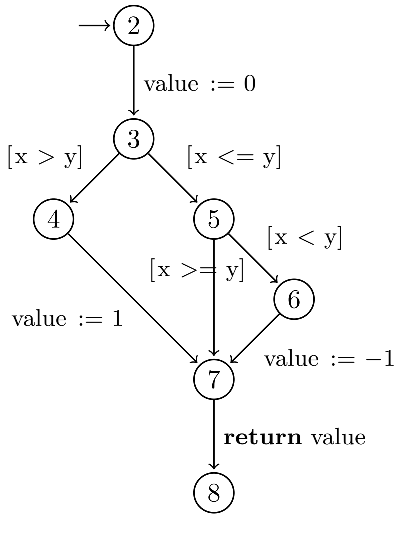


Fig. 2. CFA for Listing 1

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 1 with the CFA of Figure 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`.

4.4 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 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, may 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 1.

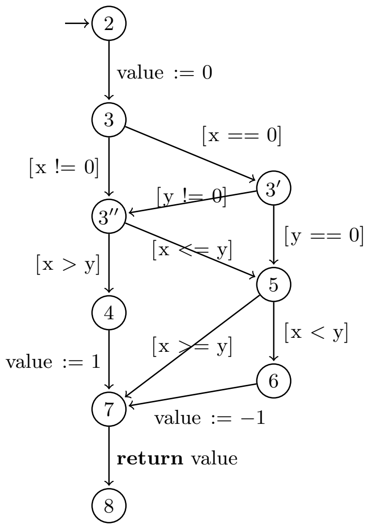
– *Edges.* In FQL, `EDGES(@FUNC(cmp))`, or simply `@FUNC(cmp)`, yields the expression $(2, 3) + (3, 4) + (3, 5) + (4, 7) + (5, 7) + (5, 6) + (6, 7) + (7, 8)$. Hence the coverage specification of a query

```
> 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*"
```



x	y	x > y
true	$*$	true
true	$*$	false
false	true	true
false	true	false
false	false	$*$

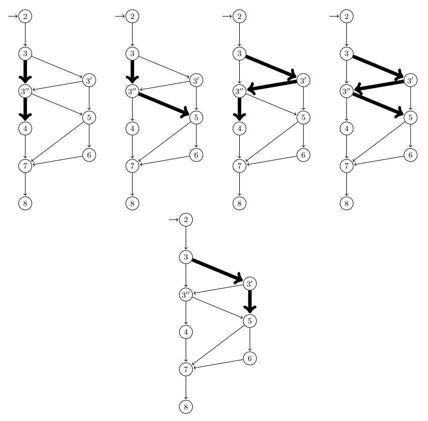


Fig. 3. Multiple condition coverage of `(x || y) && x > y`

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 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. 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. 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. 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.

4.5 Full FQL Specifications

General FQL specifications have the form

in G cover C passing P

where both in G and passing P can be omitted. Table 1 summarizes the full syntax of FQL.

- 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 '$' may 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.

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

Table 1. Syntax of FQL

5 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 function `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 may 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 [15] 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 may 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. Our recent paper [12] is studying the application of FQL to model-based testing systematically.

6 Tool Support

We realized test case generation and coverage measurement engines for C in our tools FShell 2 and FShell 3. FShell 2 offers an interactive (shell-like) frontend where users state their FQL specifications. As backend, FShell 2 uses components of CBMC [16], which enables support for full C syntax and semantics. FShell 2 is available for download in binary form for the most popular platforms at `http://code.forsyte.de/fshell`. In contrast, FShell 3 builds upon the predicate abstraction based software verification framework CPAchecker [17,18]. In addition to test generation, FShell 3 also supports measurement of coverage achieved by an existing test suite.

7 Related Work

Several papers have addressed test specifications from different angles. Beyer et al. [19] use the C model checker BLAST [20] for test case generation, focusing on basic block coverage only. BLAST has a two level specification language [21]. Contrary to FQL, their language is tailored towards verification. Hessel et al. [22,23,24] present the automatic test case generator Uppaal Cover [22] based on Uppaal [25]. They present a specification language, based on parameterized observer automata, for describing coverage criteria. Their method is based on models of a system under test whilst we are able to generate test cases from source code directly. Lee et al. [26,27] investigate test case generation with model checkers giving coverage criteria in temporal logics. Java PathFinder [28] and SAL2 [29] use model checkers for test case generation, but, they do not support C semantics. Geist et al. [30] apply symbolic model checking in test generation for hardware systems.

Most existing formalisms for test specifications focus on the description of test *data*, e.g., TTCN-3 [31] and UML TP [32], but none of them allows to describe structural coverage criteria.

8 Conclusion

We presented a brief overview of our research on query-driven program testing with a focus on the query language FQL. Together with our query solving backends this versatile test case specification language supports automated test input generation for sequential C programs. One of our major next steps will be developing support for concurrent software, both in the backend as well as at language level, following earlier work such as [33] or [34].

References

1. RTCA DO-178B: Software considerations in airborne systems and equipment certification (1992)
2. Bird, D.L., Munoz, C.U.: Automatic generation of random self-checking test cases. IBM Systems Journal **22**(3) (1983) 229–245
3. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Draves, R., van Renesse, R., eds.: OSDI, USENIX Association (2008) 209–224
4. Godefroid, P.: Compositional dynamic test generation. In Hofmann, M., Felleisen, M., eds.: POPL, ACM (2007) 47–54
5. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In Sarkar, V., Hall, M.W., eds.: PLDI, ACM (2005) 213–223
6. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In Young, M., Devanbu, P.T., eds.: SIGSOFT FSE, ACM (2006) 117–127
7. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. In Wermelinger, M., Gall, H., eds.: ESEC/SIGSOFT FSE, ACM (2005) 263–272
8. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: How did you specify your test suite. In Pecheur, C., Andrews, J., Nitto, E.D., eds.: ASE, ACM (2010) 407–416
9. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Fshell: Systematic test case generation for dynamic analysis and measurement. In Gupta, A., Malik, S., eds.: CAV. Volume 5123 of Lecture Notes in Computer Science., Springer (2008) 209–213
10. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In Jones, N.D., Müller-Olm, M., eds.: VMCAI. Volume 5403 of Lecture Notes in Computer Science., Springer (2009) 151–166
11. Beyer, D., Holzer, A., Tautschnig, M., Veith, H.: Directed software verification. (2010) Technical Report.
12. Holzer, A., Januzaj, V., Kugele, S., Langer, B., Schallhart, C., Tautschnig, M., Veith, H.: Seamless testing for models and code. (2010) Technical Report.
13. Godefroid, P., Kinder, J.: Proving memory safety of floating-point computations by combining static and dynamic program analysis. In Tonella, P., Orso, A., eds.: ISSTA, ACM (2010) 1–12
14. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. (2002) 58–70
15. Ball, T.: A theory of predicate-complete test coverage and generation. In de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: FMCO. Volume 3657 of Lecture Notes in Computer Science., Springer (2004) 1–22

16. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ansi-c programs. In Jensen, K., Podelski, A., eds.: TACAS. Volume 2988 of Lecture Notes in Computer Science., Springer (2004) 168–176
17. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: ASE, IEEE (2008) 29–38
18. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. CoRR **abs/0902.0019** (2009)
19. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: ICSE, IEEE Computer Society (2004) 326–335
20. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with blast. In Ball, T., Rajamani, S.K., eds.: SPIN. Volume 2648 of Lecture Notes in Computer Science., Springer (2003) 235–239
21. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The blast query language for software verification. In Giacobazzi, R., ed.: SAS. Volume 3148 of Lecture Notes in Computer Science., Springer (2004) 2–18
22. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using uppaal. In Hierons, R.M., Bowen, J.P., Harman, M., eds.: Formal Methods and Testing. Volume 4949 of Lecture Notes in Computer Science., Springer (2008) 77–117
23. Hessel, A., Pettersson, P.: A global algorithm for model-based test suite generation. Electr. Notes Theor. Comput. Sci. **190**(2) (2007) 47–59
24. Blom, J., Hessel, A., Jonsson, B., Pettersson, P.: Specifying and generating test cases using observer automata. In Grabowski, J., Nielsen, B., eds.: FATES. Volume 3395 of Lecture Notes in Computer Science., Springer (2004) 125–139
25. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. STTT **1**(1-2) (1997) 134–152
26. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: A temporal logic based theory of test coverage and generation. In Katoen, J.P., Stevens, P., eds.: TACAS. Volume 2280 of Lecture Notes in Computer Science., Springer (2002) 327–341
27. Tan, L., Sokolsky, O., Lee, I.: Specification-based testing with linear temporal logic. In Zhang, D., Grégoire, É., DeGroot, D., eds.: IRI, IEEE Systems, Man, and Cybernetics Society (2004) 493–498
28. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with java pathfinder. In Avrunin, G.S., Rothermel, G., eds.: ISSTA, ACM (2004) 97–107
29. Hamon, G., de Moura, L.M., Rushby, J.M.: Generating efficient test sets with a model checker. In: SEFM, IEEE Computer Society (2004) 261–270
30. Geist, D., Farkas, M., Landver, A., Lichtenstein, Y., Ur, S., Wolfsthal, Y.: Coverage-directed test generation using symbolic techniques. In Srivas, M.K., Camilleri, A.J., eds.: FMCAD. Volume 1166 of Lecture Notes in Computer Science., Springer (1996) 143–158
31. Din, G.: Ttcn-3. In Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A., eds.: Model-Based Testing of Reactive Systems. Volume 3472 of Lecture Notes in Computer Science., Springer (2004) 465–496
32. Schieferdecker, I., Dai, Z.R., Grabowski, J., Rennoch, A.: The uml 2.0 testing profile and its relation to ttcn-3. In Hogrefe, D., Wiles, A., eds.: TestCom. Volume 2644 of Lecture Notes in Computer Science., Springer (2003) 79–94
33. Ben-Asher, Y., Eytani, Y., Farchi, E., Ur, S.: Producing scheduling that causes concurrent programs to fail. In Ur, S., Farchi, E., eds.: PADTAD, ACM (2006) 37–40
34. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: IPDPS, IEEE Computer Society (2003) 286