How did You Specify Your Test Suite ?

Andreas Holzer Michael Tautschnig Helmut Veith
Vienna University of Technology
{holzer, tautschnig, veith}@forsyte.at

Christian Schallhart
Oxford University Computing Laboratory
christian.schallhart@comlab.ox.ac.uk

ABSTRACT

Although testing is central to debugging and software certification, there is no adequate language to specify test suites over source code. Such a language should be simple and concise in daily use, feature a precise semantics, and of course, it has to facilitate suitable engines to compute test suites and assess the coverage achieved by a test suite.

This paper introduces the language FQL designed to fit these purposes. We achieve the necessary expressive power by a natural extension of regular expressions which matches test suites rather than individual executions. To evaluate the language, we show for a list of informal requirements how to express them in FQL. Moreover, we present a test case generation engine for C programs and perform practical experiments with the sample specifications.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*data generators, coverage*

General Terms

Languages, Verification

1. INTRODUCTION

Source code based testing is the most practical and important technique to assure software quality. Testing accompanies the development process from early versions of the implementation all the way to product certification.

In this paper, we describe a novel approach to software testing where the test suites are specified in the language FQL (FShell Query Language). FQL specifications 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 (e.g., "condition coverage"). We have designed FQL as a specification language which is easy to read – it is based on regular expressions – but has an expressive and precise semantics.

ASE'10, September 20–24, 2010, Antwerp, Belgium.

Test specifications in FQL can go well beyond established coverage criteria; in our experience with students, FQL encourages programmers to explore their code more systematically. Fig. 1 contains a list of informal specifications and Table 8 shows how to express them in FQL for C programs. Such specifications can be used in many contexts of which we discuss a few (cf. Sec. 5):

- **Test Case Generation.** FQL enables us to compute test suites *according to user specified coverage criteria*, cf. Sec. 5. 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 FQL test specifications, and generate a covering test suite. When we evaluate the resulting test suite for, e.g., decision coverage, we understand if the requirements contain sufficient detail to guide the implementation.

- **Certification.** We can formulate precise criteria for code certification in FQL and evaluate them on the source code.

  The lack of formal test specifications (even in standards such as DO-178B [14]) has lead to inconsistent tool support. To illustrate the problem, we use the four commercial test tools CoverageMeter [11], CTC++ [12], BullseyeCoverage [8], and Rational Test RealTime (RTRT) [29] to check for condition coverage on the C program shown in Listing 1. 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.

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

  Listing 1: Sample program

  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 an FQL 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 use concise specifications of *missing* test cases as inputs for a heavy-weight tool such as a model checker. An interface with our tool to perform automated coverage completion is part of current work.

- **Systematic Reasoning about Test Specifications.** Finally, we believe that FQL gives us a clean and simple framework to

Scenario 1: Structural Coverage Criteria. The certification of critical software systems often requires coverage criteria such as basic block, condition or decision coverage [27] 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++.

For intensive testing a developer will employ a variant of path coverage [28], 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.

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 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`. The function `unfinished` is allowed to be called outside `sort`—assuming that only the functionality of `unfinished` which is used by `sort` is not testable yet.

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

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 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) [27].

To understand the interaction of two program parts, it is not sufficient to cover the *union* of the test goals induced by each part, but to 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 1: Twenty-four examples of informal test case specifications

study fundamental issues about test specifications such as equivalence and subsumption of specifications, normal forms, distribution of specifications to multiple test servers etc.

Given the practical importance of a test specification language, we were quite surprised that there is very little previous work on this question. We begin by listing the challenges:

(a) **Simplicity and Code Independence.** Simple coverage criteria should be expressed by simple FQL specifications. 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) **Precise Semantics.** FQL specifications should have a simple and unambiguous semantics.

(c) **Expressive Power.** FQL should be based on a small number of orthogonal concepts which allow to express natural coverage criteria including, among others, the examples of Fig. 1.

(d) **Encapsulation of Language Specifics.** Specifications in FQL should be maximally agnostic to the programming language at hand. To this end, FQL should provide a clear and concise binding concept with the underlying programming language.

(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.

In this paper we introduce FQL which is—to the best of our knowledge—the first test specification language which satisfies the requirements (a) to (e). Our previous work [24] focused on algorithmic test case generation, addressing challenge (e). Arguing that test case specification and test case generation have a similar relationship as database query languages and database engines, we introduced the notion of query-driven test case generation and presented a SAT-based test case generation approach. The preliminary specification language used in [24] was a first step towards FQL, but it lacked both an exact semantics and a clean concept.

Organization of this Paper. Sec. 2 provides a gentle introduction into the concepts of FQL. Sections 3 to 4 give a systematic description of the syntax and semantics of FQL. Most of the presentation is language independent, only Sec. 4.2 discusses elements specific to C. Sec. 5 evaluates FQL from four perspectives: (1) We show that the sample specifications of Fig. 1 can be expressed in FQL. (2) We present an improved version of our test case generation tool [24]. (3) Continuing the discussion in this section, we show how FQL can be used in different tool chains. (4) We outline further research around FQL. In Sec. 6 we discuss related work.

2. FQL LANGUAGE CONCEPT

It is natural to specify a *single test case* on a *fixed given program* by a regular expression. For instance, to obtain a test case which goes through line number 17 of the program, one can write a "path pattern" as a regular expression _ $*$.@17._$*$ where _ stands for an arbitrary program command.[1] In writing the above path pattern, we implicitly assume that the alphabet symbols are constraints that a program execution must satisfy. This simple approach has a principal limitation: it only works for a few hand-written test cases on a fixed program.

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. To understand the requirements for the specification language, let us analyze the above verbal specification:

1. The specification requires a *test suite*, i.e., multiple test cases, which together have to achieve coverage.
2. The specification contains a *universal quantifier*, saying that each basic block must be covered by a test case in the test suite.
3. Referring to entities such as "basic blocks" the specification assumes *knowledge about program structure*.
4. 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*.

It will be easy for the reader to confirm that these observations hold true for all test specifications of Sec. 1, with the only exception of observation 4.: Certain test specifications depend on the program under test more than others. The four observations motivate the following definition of coverage criteria (cf. Definition 6):

> *An elementary coverage criterion Φ is a function that maps a program $\mathcal{A}$ to a finite set $\Phi(\mathcal{A})$ of path patterns. A test suite Γ satisfies coverage criterion Φ on program $\mathcal{A}$, if each path pattern 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).*

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 1.-4. discussed above. In the rest of this section, we will discuss the main features of FQL.

- FQL is a natural extension of regular expressions. To cover line 17, we can just write

  `> cover "_*.@17._*"`

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

  `> cover "_*.{x = 0}.@17._*"`

  which requests a test where $x = 0$ holds at line 17.

- Using concatenation and union, but not Kleene star, FQL combines quoted regular expressions into coverage specifications for test suites. This is a **key feature** which we first illustrate on a simple example. When we write

  `> cover "_*.@17._*" + "_*.@32._*"`

  this is tantamount to a list of two path patterns:

  `> cover "_*.@17._*"`

  `> cover "_*.@32._*"`

  Formally, we treat the quoted regular expressions "_$*$.@17._$*$" and "_$*$.@32._$*$" 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. Fig. 2(a). These words 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.

- For program independence and generality, FQL has support to access natural program entities such as basic blocks, files, decisions, etc. For instance, the expression

  `EDGES(@BASICBLOCKENTRY)`

  is equivalent to a regular expression of the form

  $@2 + @5 + @13 + @19 + @25$

  in a short program whose basic blocks start in line numbers 2, 5, 13, 19, and 25. The expression `EDGES(@BASICBLOCKENTRY)` can only be expanded into a regular expression when the test specification is applied, i.e., when the program under test is known. Thus, we can write

  `> cover "_*".EDGES(@BASICBLOCKENTRY)."_*"`

  to achieve basic block coverage. At runtime, this amounts to

  `> cover "_*".(@2 + @5 + @13 + @19 + @25)."_*"`

  which is in turn equivalent to the sequence

  ```
  > cover "_*".@2."_*"
        ⋮
  > cover "_*".@25."_*"
  ```

  of path patterns which, together, specify basic block coverage.

- Expressions such as `@BASICBLOCKENTRY` are used to denote *target graphs*. Target graphs contain parsing information about the program. Mathematically, they are modeled as subgraphs of the program's control flow automaton (a variant of control flow graphs). FQL provides a rich functionality to extract and manipulate target graphs from programs, for instance the operations `&` and `|` for intersection and union of graphs. This feature provides the link to the individual programming language, and is the only language-dependent part of FQL.

  For another example of target graphs, consider

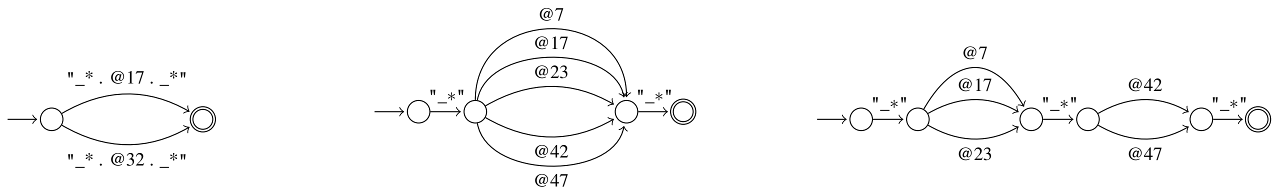
  `PATHS(@FUNC(main),1)`

  which returns all non-cyclic paths through function main, for instance,

  "@1.@2.@3.@5" + "@1.@2.@4.@5" + ...

  In fact, expressions such as @5 which we used above, are shorthands for target graph expressions such as `EDGES(@LINE(5))`.

[1] Similarly, we can write a safety specification $\mathbf{AG}\neg@17$ such that a model checker can compute a counterexample which serves as a test case.



(a) `"_∗.@17._∗" + "_∗.@32._∗"` (b) `(@BASICBLOCKENTRY & @FUNC(foo)) + (@BASICBLOCKENTRY & @FUNC(bar))` (c) `(@BASICBLOCKENTRY & @FUNC(foo)) -> (@BASICBLOCKENTRY & @FUNC(bar))`

Figure 2: Automata resulting from `cover` clauses (lines 7, 17 and 23 are basic blocks entries in `foo`, 42 and 47 are the lines for `bar`)

- 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 "_*".EDGES(@BASICBLOCKENTRY)."_*"
    passing (_.{x ≥ 0})*
  ```

  we request basic block coverage through a test suite where x never becomes negative.

- FQL contains syntactic sugar to simplify test specifications. For instance, `->` stands for `._*.`. Moreover, `_*` is by default added before and after each path pattern.

Let us sum up this introduction to FQL with a comparison of three interesting test specifications:

```
> cover EDGES(@BASICBLOCKENTRY & (@FUNC(foo) | @FUNC(bar)))
> cover EDGES(@BASICBLOCKENTRY & @FUNC(foo)) +
        EDGES(@BASICBLOCKENTRY & @FUNC(bar))
> cover EDGES(@BASICBLOCKENTRY & @FUNC(foo)) ->
        EDGES(@BASICBLOCKENTRY & @FUNC(bar))
```

In the first specification, we require basic block coverage for two functions, `foo` and `bar`. In the second specification, we have the same coverage criterion written in a different way. In the third spec, however, we require a more complex coverage: We want test cases in which all Cartesian combinations of basic blocks in `foo` and `bar` occur in the test suite. To see this, just note that the first two specifications give rise to the $3+2=5$ path patterns of Fig. 2(b), while the third amounts to $3\times 2=6$ path patterns of Fig. 2(c).

In this section, we have explained complex FQL *queries by reduction to simpler intuitive* FQL *queries on concrete programs. To this end, we made didactic simplifications, e.g. we assumed that line numbers can distinguish between basic blocks. In the following sections, we will give a formal and thorough description of* FQL.

3. MATHEMATICAL MODEL

In this section we introduce state-based models for the control flow and the program semantics. Based on these notions, we formalize the notion of coverage criteria.

State-Based Models. Syntactically, we represent programs as control flow automata [20], annotated with parsing information. For example, Fig. 3(a) shows the CFA for the code in Listing 2. Nodes represent program counter values; edges are labeled with operations and annotations, drawn from finite sets Op and An, respectively. An operation $\mathsf{op}\in\mathsf{Op}$ is either a skip statement, assignment, assumption (modeling conditional statements), function call, or function return. Annotations include parsing information such as line numbers or file names, and function names, labels, etc.

DEFINITION 1. *A* control flow automaton *(CFA)* $\mathcal{A}$ *is a tuple* $\langle L,E,I\rangle$*, where* L *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 $\cup$, $\cap$, and $\setminus$ as operations on CFAs:

$$\langle L_1,E_1,I_1\rangle\cup\langle L_2,E_2,I_2\rangle=\langle L_1\cup L_2,E_1\cup E_2,I_1\cup I_2\rangle$$
$$\langle L_1,E_1,I_1\rangle\cap\langle L_2,E_2,I_2\rangle=\langle L_1\cap L_2,E_1\cap E_2,I_1\cap I_2\rangle$$
$$\langle L_1,E_1,I_1\rangle\setminus\langle L_2,E_2,I_2\rangle=\langle L',E',I'\rangle \text{ where}$$

$E'=E_1\setminus E_2$, $L'=\{u,u'\mid(u,l,u')\in E'\}\cup(L_1\setminus L_2)$, and $I'=I_1\cap L'$.

To describe the behavior of a program, we define a transition system as follows:

DEFINITION 2. *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 $\pi=\langle s_0\ldots s_m\rangle$ *such that* $s_0\in\mathcal{I}$ *and* $(s_i,s_{i+1})\in\mathcal{R}$*, for* $0\le i<m$.

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:

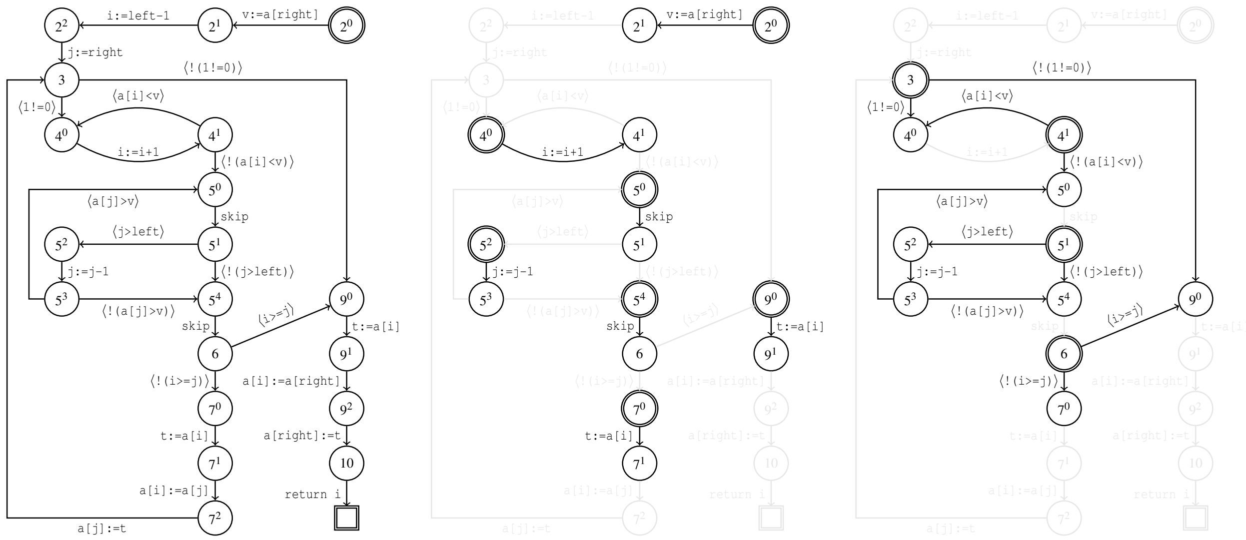
- We consider the operation $\mathsf{op}\in\mathsf{Op}$ as a function $\mathsf{op}:\mathcal{S}\to 2^{\mathcal{S}}$ that takes a program state and determines its successor states.
- By $\mathsf{pc}:\mathcal{S}\to L$ we denote a function that, given a program state s, yields its program location $\mathsf{pc}(s)$.
- 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 3. *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* $\mathcal{I}=\{s\in\mathcal{S}\mid\mathsf{pc}(s)\in I_{\mathcal{A}}\}$.

Predicates & Coverage Criteria. Let $\mathcal{T}=\langle\mathcal{S},\mathcal{R},\mathcal{I}\rangle$ be a transition system. For $\pi=\langle s_0s_1\ldots s_m\rangle$ and $i\le j$ we write $\pi^{i\ldots j}$ to denote the subpath $\langle s_i\ldots s_j\rangle$. With $\langle\rangle$ we denote the empty path. 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 $\Gamma\models\Phi$. 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

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



(a) Control flow automaton $\mathcal{A}$ (b) Target graph for @BASICBLOCKENTRY[$\mathcal{A}$] (c) Target graph for @CONDITIONGRAPH[$\mathcal{A}$]

Figure 3: Control flow automaton of `partition` **(Listing 2) and target graphs**

```
int partition (int a[], int left, int right) {
  int v = a[right], i = left - 1, j = right, t;
  for (;;) {
    while (a[++i] < v) ;
    while (j > left && a[--j] > v);
    if (i >= j) break;
    t = a[i]; a[i] = a[j]; a[j] = t;
  }
  t = a[i]; a[i] = a[right]; a[right] = t;
  return i;
}
```

Listing 2: Example source code (`sort.c`)

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

DEFINITION 4. *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 5. *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 $\mathcal{T}_{\mathcal{A}}$ iff $\Gamma \models \Phi^{\mathcal{A}}$ holds.*

While our definition of coverage criteria is very general, 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.

DEFINITION 6. *An* elementary coverage criterion Φ *is a coverage criterion defined as follows:*

(i) There is a mapping $\Phi(\mathcal{A}) = \{\Psi_1, \dots, \Psi_k\}$ which maps a CFA $\mathcal{A}$ to a set of test goals $\{\Psi_1, \dots, \Psi_k\}$ *where each Ψ_i is a path predicate.*

(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 $\pi \models \Psi_i$.

MC/DC, for example, is a coverage criterion that is not elementary.

4. SYNTAX AND SEMANTICS OF FQL

We will now describe the language FQL. Semantically, each FQL specification Φ boils down to an elementary coverage criterion. The syntax of FQL follows the ideas of Sec. 2.

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 target graphs such as `@BASICBLOCKENTRY` or `@5` to refer to program elements, cf. Sec. 2. *For a given program, an* FQL *specification can be easily translated into an ECP by parsing the program and "expanding" the target graphs into regular expressions over the CFA alphabet, in a manner similar to (but more complicated than) the didactic examples of Sec. 2.*

4.1 FQL Elementary Coverage Patterns

Table 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 "*".

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.

Table 2 defines the semantics of path patterns and coverage specifications as formal languages over alphabets of program counter locations, state predicates, program transitions, and symbols newly introduced by the quote operator. We use X in places where either P or C may occur and denote by $\mathcal{L}(X)$ the language of a path pattern and a coverage specification, 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 2 in the following paragraph.

Interpretation of Path Patterns as Path Predicates. Given a coverage specification or path pattern X, we interpret each $w \in \mathcal{L}(X)$ as a path predicate. We write $\pi \models w$ iff π satisfies the word w and inductively define the semantics thereof in Table 3. The empty set is unsatisfiable and the empty word ε matches the empty sequence $\langle\rangle$ only. We match individual states with program counter values ℓ and state constraints φ, and pairs of subsequent states with transitions e. The case $\pi \models aw$ amounts to predicate concatenation as defined in Sec. 3. The path pattern $\texttt{"}P\texttt{"}$ is satisfied by a path π, iff there is a word $w \in \mathcal{L}(P)$ that is satisfied by π. Applying these definitions, an ECP combines a coverage specification and a path pattern to obtain a set of path predicates as defined in the last line of Table 2.

4.2 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 7. *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 most important CFA transformers are *filter functions*, which extract a subset of the edges of a CFA.

DEFINITION 8. *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.*

For example, consider the CFA $\mathcal{A}$ depicted in Fig. 3(a): The target graph `@BASICBLOCKENTRY`$[\mathcal{A}]$ depicted in Fig. 3(b) (edges not contained in the target graph are grayed out) is obtained by applying the filter function `@BASICBLOCKENTRY` to $\mathcal{A}$. This target graph 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 in Listing 2, see Fig. 3(c).

In Def. 8 the condition $I' \subseteq L'$ enables a filter function to change the set of initial locations. E.g., `@BASICBLOCKENTRY`$[\mathcal{A}]$, as shown in Fig. 3(b), sets the initial locations (indicated by double circles) to the start locations of the edges in the target graph.

Filter functions encapsulate the interface to the programming language. They extract CFA edges based on annotations added to a CFA while parsing the source code. Table 4 lists the filter functions currently supported in FQL. Their exact definitions are specific to the C programming language, hence we use according terminology.

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

Table 1: Syntax of elementary coverage patterns

$\mathcal{L}(X_1+X_2)$	$= \mathcal{L}(X_1) \cup \mathcal{L}(X_2)$
$\mathcal{L}(X_1.X_2)$	$= \{w_1w_2 \mid w_1 \in \mathcal{L}(X_1), w_2 \in \mathcal{L}(X_2)\}$
$\mathcal{L}(\varepsilon)$	$= \{\varepsilon\}$
$\mathcal{L}(\emptyset)$	$= \emptyset$
$\mathcal{L}(x)$	$= \{x\}$ where $x \in L \cup E \cup S$
$\mathcal{L}(P^\star)$	$= \mathcal{L}(P)^\star$
$\mathcal{L}(\texttt{"}P\texttt{"})$	$= \{\texttt{"}P\texttt{"}\}$
$\mathcal{L}(\texttt{cover}\, C\, \texttt{passing}\, P)$	$= \{w \wedge \texttt{"}P\texttt{"} \mid w \in \mathcal{L}(C)\}$

Table 2: Semantics of FQL **elementary coverage patterns**

$\pi \models \emptyset$	**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 $s' \in \mathsf{post}(e, s)$
$\pi \models w$	**iff** $\pi \models a \cdot w'$ with $w = aw'$ and $a \in L \cup E \cup S$ or $\texttt{"}P\texttt{"}$
$\pi \models \texttt{"}P\texttt{"}$	**iff** there is a $w \in \mathcal{L}(P)$ such that $\pi \models w$

Table 3: Interpretation of path patterns as path predicates

Further CFA Transformers. A CFA transformer T is either a filter function F, function composition, a set-theoretic operation on target graphs, or predication $\texttt{PRED}(T, \varphi)$. Applied to a CFA $\mathcal{A}$, $\texttt{PRED}(T, \varphi)$ yields a new CFA that contains for every node $u \in L_{\mathcal{A}}$ two new nodes (u, φ) and $(u, \neg\varphi)$ representing the evaluation of a state predicate φ to true, i.e., (u, φ), and to false, i.e., $(u, \neg\varphi)$. The result of applying T to a CFA $\mathcal{A}$ is denoted by $T[\mathcal{A}]$. See Table 5 for the semantics of all CFA transformers, except filter functions.

4.3 FQL Specifications

Table 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 require the operators `NODES(`T`)`, `EDGES(`T`)`, and `PATHS(`T`, `k`)`. Here, T is a CFA transformer expression and k is a positive integer.

The clause `in` T states that, given a CFA $\mathcal{A}$, all filter functions in the `cover` clause are applied to the target graph $T[\mathcal{A}]$. In practice, this is often used as `in @FUNC(foo) cover EDGES(@CONDITIONEDGE) passing EDGES(ID)*` which is equivalent to the spec `cover EDGES(COMPOSE(@CONDITIONEDGE, @FUNC(foo))) passing EDGES(ID)*`.

`ID`	identity function
`@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 4: Filter functions in FQL

$\text{COMPOSE}(T_1,T_2)[\mathcal{A}] = T_1[T_2[\mathcal{A}]]$
$(T_1|T_2)[\mathcal{A}] = T_1[\mathcal{A}] \cup T_2[\mathcal{A}]$
$(T_1\&T_2)[\mathcal{A}] = T_1[\mathcal{A}] \cap T_2[\mathcal{A}]$
$\text{SETMINUS}(T_1,T_2)[\mathcal{A}] = T_1[\mathcal{A}] \setminus T_2[\mathcal{A}]$
$\text{PRED}(T,\varphi)[\mathcal{A}] = \langle L',E',I'\rangle$ where $\langle L,E,I\rangle = T[\mathcal{A}]$
and $L' = L\times\{\varphi,\neg\varphi\}, I' = I\times\{\varphi,\neg\varphi\}$, and
$E' = \{((u,v),l,(u',v')) \mid v,v' \in \{\varphi,\neg\varphi\},(u,l,u')\in E\}$

Table 5: Semantics of CFA transformers

$\Phi ::= \texttt{in}\ T\ \texttt{cover}\ C\ \texttt{passing}\ P$
$C ::= C+C \mid C.C \mid (C) \mid N \mid S \mid \texttt{"}P\texttt{"}$
$P ::= P+P \mid P.P \mid (P) \mid N \mid S \mid P^\star$

$N ::= \text{NODES}(T) \mid \text{EDGES}(T) \mid \text{PATHS}(T,k)$
$T ::= F \mid \text{PRED}(T,\varphi) \mid \text{COMPOSE}(T,T)$
$\quad \mid T|T \mid T\&T \mid \text{SETMINUS}(T,T)$
$F ::= \text{ID} \mid \text{@BASICBLOCKENTRY} \mid \text{@CONDITIONEDGE}$
$\quad \mid \text{@CONDITIONGRAPH} \mid \text{@DECISIONEDGE} \mid \text{@FILE}(a)$
$\quad \mid \text{@LINE}(x) \mid \text{@FUNC}(f) \mid \text{@STMTTYPE}(types)$

Table 6: Syntax of FQL

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 (iterated "+") of path patterns which represent the nodes, edges, and paths in the target graph $T[\mathcal{A}]$, respectively:

$$\text{NODES}(T) \mapsto \sum_{n\in \mathsf{nodes}(T[\mathcal{A}])} n$$
$$\text{EDGES}(T) \mapsto \sum_{e\in \mathsf{edges}(T[\mathcal{A}])} e$$
$$\text{PATHS}(T,k) \mapsto \sum_{p\in \mathsf{paths}_k(T[\mathcal{A}])} p$$

Intuitively, $\mathsf{nodes}(T[\mathcal{A}])$ is the set of nodes of the target graph $T[\mathcal{A}]$ obtained by applying T to $\mathcal{A}$. The same holds for $\mathsf{edges}(T[\mathcal{A}])$ and $\mathsf{paths}_k(T[\mathcal{A}])$. In case a set $\mathsf{nodes}(T[\mathcal{A}])$, $\mathsf{edges}(T[\mathcal{A}])$, or $\mathsf{paths}_k(T[\mathcal{A}])$ is empty the corresponding operator expands to the symbol $\emptyset$. The semantics of a 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 2.

Formally, we define the functions nodes, edges, and paths_k. For simplicity let us assume the CFA transformer PRED was not applied, then, $\mathsf{nodes}(T[\mathcal{A}]) = L_{T[\mathcal{A}]}$, $\mathsf{edges}(T[\mathcal{A}]) = E_{T[\mathcal{A}]}$, and $\mathsf{paths}_k(T[\mathcal{A}]) = \{p \mid p$ is a k-bounded path in $T[\mathcal{A}]\}$. A k-bounded path in $T[\mathcal{A}]$ is a sequence of edges, starting in $I_{T[\mathcal{A}]}$, in which no target graph node occurs more than k times. In case PRED is applied, the corresponding state predicates have to be inserted into the path patterns at the right place.

As an example consider the target graph shown in Fig. 4. There, $\mathsf{nodes}(\mathcal{A})$ is the set of path patterns $\{\ell_1,\ell_2,\ell_3,\ell_4,\ell_6\}$ and the operator NODES(ID) yields the expression $\ell_1+\ell_2+\ell_3+\ell_4+\ell_6$. Here, ℓ_i denotes the node labeled with i. 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. PATHS(ID, 1) yields the expression $e_{1,2}+e_{1,3}+e_{1,2}e_{2,3}+e_{1,2}e_{2,4}+e_{1,2}e_{2,3}e_{3,4}+e_{1,2}e_{2,3}e_{3,6}+e_{1,3}e_{3,4}+e_{1,3}e_{3,6}$.

```
if  ((x > 10
      && y < 100)
      ||  (x < y))
{  ...  }
else
{  ...  }
```

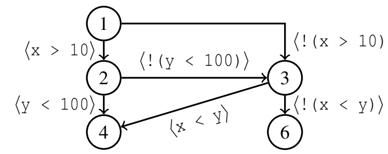


Figure 4: Edge- vs. path-coverage

Syntactic construct	Expanded expression
->	. "ID*".
@k	@LINE(k)
X == 0	ε
X == k	X. ...X (k times)
X <= k	$\Sigma_k^{i=0} X$ == i
P >= k	P == k . $P^\star$
NOT (T)	SETMINUS (ID, T)
_	ID

Table 7: Syntactic sugar

Semantics. An FQL specification

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

maps a CFA $\mathcal{A}$ to a finite set $\Phi(\mathcal{A})$ of path predicates. By C' we denote the coverage specification obtained by first applying the transformer G to $\mathcal{A}$ and then replacing all NODES(T), EDGES(T), and PATHS(T, k) by $\sum_{n\in\mathsf{nodes}(T[G[\mathcal{A}]])} n$, $\sum_{e\in\mathsf{edges}(T[G[\mathcal{A}]])} e$, and $\sum_{p\in\mathsf{paths}(T[G[\mathcal{A}]])} p$, respectively. By P' we denote the path pattern obtained by replacing all occurrences of NODES(T), EDGES(T), and PATHS(T, k) by the corresponding sums (for the passing clause G is not applied). Then, we define $\Phi(\mathcal{A})$ by reducing Φ to an ECP:

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

PROPOSITION 9. *Every* FQL *specification* Φ *satisfies Definition 6 and, therefore, is an elementary coverage criterion.*

Syntactic Sugar. For simpler use, we extend FQL by redundant constructions summarized in Table 7. Further simplifications are:

- If neither the operator NODES, nor EDGES, nor PATHS is given, we use EDGES as default.
- By default, "ID*" is prepended and appended to cover and passing clauses. In analogy to Unix' grep we can avoid this default by writing ("^") at the start or ("$") at the end of an expression.
- Omission of the passing clause is expanded to `in T cover C passing ^ID*$`.
- Omission of the in clause is expanded to `in ID cover C passing P`.

5. EVALUATION

We evaluate FQL in four dimensions: (1) Expessiveness and usability, (2) practical feasibility of test case generation, (3) uses of FQL in the SE tool chain, and (4) potential for further research.

5.1 Expressive Power and Usability

Table 8 shows how the test case specifications **Q1-24** of Fig. 1 can be written in FQL. We see that even complex specifications can be written as succinct and natural FQL specifications. (Experiments with these specs are discussed in the next section.)

We note that inside quotes we can use pattern matching formalisms more powerful than regular expressions with trivial extensions. We can include, e.g., context-free features such as bracket

Q1 `cover @BASICBLOCKENTRY`
Q2 `cover @CONDITIONEDGE`
Q3 `cover @CONDITIONEDGE & @STMTTYPE(if,switch,for,while,?:)`
Q4 `cover PATHS(@FUNC(main) | @FUNC(insert),1)`
Q5 `cover PATHS(@FUNC(main) | @FUNC(insert),2)`
Q6 `cover @DEF(t)`
Q7 `cover @USE(t)`
Q8 `cover @DEF(t)."NOT(@DEF(t))*".@USE(t)`
Q9 `cover @BASICBLOCKENTRY passing ^(@2.{j>0}+NOT(@2))*$`
Q10 `cover @CONDITIONEDGE & @FUNC(compare) passing ^(NOT(@CALL(compare))*. (@CALL(compare) & @FUNC(sort))*)*$`
Q11 `cover @ENTRY(sort).{len>=2}.{len<=15} ."NOT(@EXIT(sort))*".@BASICBLOCKENTRY`
Q12 `in @FUNC(eval) cover @CONDITIONEDGE passing @CALL(eval).NOT(@EXIT(eval))*.@CALL(eval) .NOT(@EXIT(eval))*.@CALL(eval)`
Q13 `cover @CALL(sort) passing ^(NOT(@FUNC(sort))*. (@FUNC(sort) & NOT(@CALL(unfinished)))*. NOT(@FUNC(sort))*)*$`
Q14 `cover @CONDITIONEDGE passing ^(NOT(@CALL(eval))*.@CALL(insert))>=2`
Q15 `in @FUNC(partition) cover @CONDITIONEDGE passing @7`
Q16 `cover @CONDITIONEDGE + @DECISIONEDGE`
Q17 `cover (@CONDITIONEDGE & @FUNC(sort)) ->(@BASICBLOCKENTRY & @FUNC(eval))`
Q18 `cover @BASICBLOCKENTRY->@BASICBLOCKENTRY`
Q19 `cover @BASICBLOCKENTRY ->@BASICBLOCKENTRY->@BASICBLOCKENTRY`
Q20 `cover @BASICBLOCKENTRY->@BASICBLOCKENTRY ->@BASICBLOCKENTRY->@BASICBLOCKENTRY`
Q21 `cover @STMTTYPE(assert)`
Q22 `cover @STMTTYPE(assert)->@STMTTYPE(assert)`
Q23 `cover (@BASICBLOCKENTRY & @FUNC(eval)) passing ^NOT(@LABEL(init))*$`
Q24 `cover @ENTRY(main) passing @ENTRY(main).{precond()}. NOT(@EXIT(main))*.{!postcond()}.@EXIT(main)`

Table 8: Specification examples

matching. (We refrained from doing so in this paper to keep the presentation simple.) Therefore, suitable extensions of FQL can express essentially all elementary coverage criteria. (Note that all elementary coverage criteria are unions of suitable path patterns.)

5.2 Prototype Implementation

Our implementation is based on query-driven program testing [23] augmented with efficient algorithms for SAT enumeration [24]. The implementation currently supports the full range of FQL, except for the CFA transformer PRED. It relies on the source code of CBMC 3.6 [10], a bounded model checker with support for full ANSI C. Currently, we work only with C programs with static CFAs, i.e., there is limited support for function calls by function pointers and no support for longjmp and setjmp. Since we require a fully specified CFA to compute target graphs, we make assumptions about behavior left undefined by the C standard.

Expressiveness. We evaluated the example specifications **Q1-24** shown in Table 8 with our tool. Since most scenarios—for referring to line numbers or function names—make only sense for programs which contain certain tokens, we applied each specification to one of three suitable source files, cf. Table 9. The file list2.c contains the program of Listing 2, and sort1.c and sort2.c contain fragments performing array manipulation.[2] For each spec, we give the number of test goals (#goals), the number of test cases (#tc) determined by the backend, and the number of infeasible test goals (#inf).

The experiments were done on an Intel 2.53 GHz Mac OS X system equipped with 4 GB RAM. With the exception of **Q20** (quadruple basic block coverage), which took 67 seconds, all specs were processed in less than 15 seconds. Each run of the test case generation engine required at most 125 MB of memory.

Spec	Source	#goals	#tc	#inf	Spec	Source	#goals	#tc	#inf
Q1	list2.c	11	3	0	**Q13**	sort1.c	1	1	0
Q2	list2.c	8	3	0	**Q14**	sort2.c	6	1	1
Q3	list2.c	8	4	0	**Q15**	list2.c	8	1	3
Q4	sort2.c	11	3	4	**Q16**	sort1.c	30	3	0
Q5	sort2.c	20	4	7	**Q17**	sort1.c	12	2	0
Q6	list2.c	2	2	0	**Q18**	list2.c	110	4	42
Q7	list2.c	2	2	0	**Q19**	list2.c	1100	4	829
Q8	list2.c	4	1	2	**Q20**	list2.c	11000	4	10286
Q9	list2.c	11	4	0	**Q21**	sort1.c	2	1	0
Q10	sort1.c	2	2	0	**Q22**	sort1.c	2	1	1
Q11	sort1.c	9	2	1	**Q23**	sort2.c	4	1	0
Q12	sort2.c	2	1	0	**Q24**	sort2.c	2	0	2

Table 9: Experimental results for example specifications

Scalability. To study scalability of our backend to real world embedded systems code, and possibly also software systems, we chose a subset of the specifications and applied them to the following set of programs: **(1)** We picked some tools from the Unix coreutils in Busybox 1.14[3], studied as well in [9], **(2)** we selected kbfiltr.c from the Windows DDK, initially studied in [3], and **(3)** we chose an example use case[4] from [16] 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)** As an example of a complete software package, we analyzed the sources of the SAT solver PicoSAT, version 913[5].

		BB (**Q1**)		CC (**Q2**)		BB^2 (**Q18**)		
Source	SLOC	#goals	#tc	#goals	#tc	#goals	#tc	#inf
coreutils/cat.c	27	16	4	10	4	224	6	39
coreutils/echo.c	161	27	8	20	11	675	93	198
coreutils/nohup.c	33	17	6	12	5	255	13	133
coreutils/seq.c	37	28	7	20	7	728	23	394
coreutils/tee.c	73	21	9	16	8	399	30	127
kbfiltr.c	3507	250	2	196	2	62000	2	61911
pseudo-vfs.c	553	10	3	6	3	80	3	44
matlab.c	3444	30	6	22	6	840	10	441
memman.c	245	53	8	40	8	2756	29	1749
PicoSAT	6592	191	43	153	39	36099	417	26352

Table 10: Summary of experimental results

We summarize our experiments in Table 10. For each source we give the number of lines of code (SLOC)[6]. To compare to previous 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

[2]For source code cf. `http://code.forsyte.de/fshell`
[3]`http://www.busybox.net/`
[4]`http://research.nianet.org/~radu/VFS/`
[5]`http://fmv.jku.at/picosat/`
[6]Measured using David A. Wheeler's SLOCCount tool.

to 10, we compute test suites for 100% coverage of all feasible test goals. In [9] 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 #inf.

All experiments (except for PicoSAT, as discussed below) were performed using at most 350 MB of memory. Each test suite was computed in less than two minutes, except for **Q18** for kbfiltr.c which took four minutes. As PicoSAT has a larger code base, the experiments for basic block coverage and condition coverage took up to ten minutes and required up to 550 MB. For squared basic block coverage, the experiments took approximately 4.5 hours and consumed 2.5 GB of memory.

5.3 FQL in the Tool Chain

To demonstrate practical usefulness of FQL, we describe two ongoing projects with the embedded systems industry.

Measurement-based Execution Time Analysis. Our initial motivation for FQL and the test case generation backend was measurement-based execution time analysis for embedded real-time software. Together with our project partners [33] we are developing a framework to provide early feedback about the distribution of execution times to the developer. In this project, FQL enables us to efficiently compute test suites appropriate for timing analysis.

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) [22]. 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.

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 are currently working on a proper integration into FQL.

5.4 Research Questions about FQL

The language FQL gives rise to a number of interesting questions both about the formalism and efficient evaluation. The following list just mentions a few of them.

- 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 distribute specifications over multiple servers ?
- How can we trace which code changes 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 backends for better efficiency ?
- How can we build efficient predicate abstraction based tools for FQL test case generation ?
- How to obtain feedback about infeasibility of test goals ?
- How can we succinctly describe incomplete coverage ?
- How to capture difficult criteria such as MC/DC ?
- How can we combine FQL with input/output tables and executable specifications ?
- How can we apply FQL to high level models such as UML ?

All these questions can be addressed with the help of FQL.

6. RELATED WORK

Prior to our work, Beyer et al. [3] present a test case generation engine that supports "target predicate coverage", i.e., every program location has to be visited by some test case that enters the location with predicate `p` 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. Like our previous work [24], their work is also mainly addressed at challenge (e). Note that BLAST uses the database analogy in a different way than we do. BLAST uses a query language [4] 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 non-trivial for the working programmer.

Random testing, directed testing and symbolic execution based approaches aim at achieving a high code coverage with respect to standard criteria like basic block or path coverage [5, 9, 17, 18, 19, 31]. These approaches are *not* tailored towards flexible and customized coverage criteria, and are therefore orthogonal to our work. Thus, these approaches, too, are primarily addressing challenge (e). It is an interesting question for future research which FQL specifications can be solved efficiently by directed testing.

Most existing formalisms for test specifications focus on the description of test *data*, e.g., TTCN-3 [13] and UML TP [30], but none of them allows to describe structural coverage criteria. Friske et al. [15] have presented coverage specifications using OCL constraints. Although OCL provides the necessary operations to speak about UML models, it may yield hard to read expressions for complex coverage criteria. At the time of publication, no tool support for the framework was reported. Hessel et al. [6] 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 [21]. Briones et al. [7] investigate coverage measures considering the semantics of a specification and weighted fault models to arrive at minimal test suites.

Structural coverage criteria, e.g., basic block coverage, condition coverage, and path coverage are well studied, cf. [26, 28], albeit with different names and a notable lack of precise definitions. Attempts of formalizations using temporal logics [25], automata and graph based approaches [1] or using the Z notation [32] do not consider the specifics of the underlying programming language. *Predicate complete coverage* [2] is an interesting new coverage criterion that subsumes all of the above coverage criteria, except for path coverage. We can express predicate complete coverage by the FQL specification `cover EDGES(PRED(ID,` $\phi_1,\ldots,\phi_k$`))` for a given set of predicates $\phi_1,\ldots,\phi_k$.

7. CONCLUSION

In the introduction of this paper we stated five challenges for the design of a test specification language:

(a,d) **Simplicity, Code Independence and Encapsulation of Language Specifics.** Regular languages as base formalism make FQL easy to read; Table 8 demonstrates that even complex criteria have simple specifications. Our concept of target graphs ensures code independence and the encapsulation of language specifics.

(b) **Precise Semantics.** We have given a formal definition of coverage criteria in Sec. 3 and provided a precise semantics of our language FQL in Sec. 4. Every FQL specification yields an elementary coverage criterion.

(c) **Expressive Power.** We have demonstrated that all informal specifications of Fig. 1 can be expressed in FQL. As argued in Sec. 5.1, essentially all elementary coverage criteria can be expressed by FQL or suitable extensions.

(e) **Tool Suppport for Real World Code.** In Sec. 5.2 we presented experimental results for our test case generation backend. Amongst others, we generated test suites for device drivers, a SAT solver, and embedded systems code.

We consider FQL an open framework to be extended. On the language level, we are currently working on support for path set predicates, which will enable us to specify criteria such as MC/DC.

8. REFERENCES

[1] P. Ammann, J. Offutt, and W. Xu. Coverage criteria for state based specifications. In *FORTEST*, pages 118–156, 2008.

[2] T. Ball. A theory of predicate-complete test coverage and generation. In *FMCO*, pages 1–22, 2004.

[3] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Tests from Counterexamples. In *ICSE*, pages 326–335, 2004.

[4] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The Blast Query Language for Software Verification. In *SAS*, pages 2–18, 2004.

[5] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[6] J. Blom, A. Hessel, B. Jonsson, and P. Pettersson. Specifying and generating test cases using observer automata. In *FATES*, pages 125–139, 2004.

[7] L. B. Briones, E. Brinksma, and M. Stoelinga. A semantic framework for test coverage. In *ATVA*, pages 399–414, 2006.

[8] BullseyeCoverage 7.11.15. `http://www.bullseye.com/`.

[9] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[10] E. M. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, pages 168–176, 2004.

[11] CoverageMeter 5.0.3. `http://www.coveragemeter.com/`.

[12] CTC++ 6.5.3. `http://www.verifysoft.com/en.html`.

[13] G. Din. TTCN-3. In *Model-Based Testing of Reactive Systems*, pages 465–496, 2004.

[14] Software Considerations in Airborne Systems and Equipment Certification (DO-178B). RTCA, 1992.

[15] M. Friske, H. Schlingloff, and S. Weißleder. Composition of model-based test coverage criteria. In *MBEES*, 2008.

[16] A. Galloway, G. Lüttgen, J. T. Mühlberg, and R. Siminiceanu. Model-checking the linux virtual file system. In *VMCAI*, pages 74–88, 2009.

[17] P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[19] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *SIGSOFT FSE*, pages 117–127, 2006.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[21] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing real-time systems using UPPAAL. In *FORTEST*, pages 77–117, 2008.

[22] A. Holzer, V. Januzaj, S. Kugele, C. Schallhart, M. Tautschnig, H. Veith, and B. Langer. Slope testing for activity diagrams and safety critical software. Technical Report TUD-CS-2009-0184, TU Darmstadt, 2009.

[23] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In *CAV*, pages 209–213, 2008.

[24] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. Query-Driven Program Testing. In *VMCAI*, pages 151–166, 2009.

[25] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *TACAS*, pages 327–341, 2002.

[26] J. C. Huang. An approach to program testing. *ACM Comput. Surv.*, 7(3):113–128, 1975.

[27] G. Myers. *The Art of Software Testing*. Wiley, 2004.

[28] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Software Eng.*, 14(6):868–874, 1988.

[29] Rational Test RealTime 7.5. `http://www.ibm.com/software/awdtools/test/realtime/`.

[30] I. Schieferdecker, Z. R. Dai, J. Grabowski, and A. Rennoch. The UML 2.0 testing profile and its relation to TTCN-3. In *TestCom*, pages 79–94, 2003.

[31] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[32] S. A. Vilkomir and J. P. Bowen. From MC/DC to RC/DC: Formalization and analysis of control-flow testing criteria. In *FORTEST*, pages 240–270, 2008.

[33] M. Zolda, S. Bünte, and R. Kirner. Towards Adaptable Control Flow Segmentation for Measurement-Based Execution Time Analysis. In *RTNS*, 2009.