How did You Specify Your Test Suite ?

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

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 of 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.

<table>
<thead>
<tr>
<th>Listing 1: Sample program</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo(int x) {</td>
</tr>
<tr>
<td>int a = x &gt; 2 &amp;&amp; x &lt; 5;</td>
</tr>
<tr>
<td>if (a) { 0; } else { 1; }</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

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
Scenarios and Examples of Informal Test Case Specifications

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario 1: Structural Coverage Criteria</td>
<td>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.</td>
</tr>
<tr>
<td>Q1-2 — “Standard Coverage Criteria”</td>
<td>Basic block coverage and condition coverage.</td>
</tr>
<tr>
<td>Q3 — “Alternative Condition Coverage”</td>
<td>Condition coverage as defined by CoverageMeter and CTC++.</td>
</tr>
<tr>
<td>Q4 — “Acyclic Path Coverage”</td>
<td>Cover all acyclic paths through functions main and insert.</td>
</tr>
<tr>
<td>Q5 — “Loop-Bounded Path Coverage”</td>
<td>Cover all paths through main and insert which pass each statement at most twice.</td>
</tr>
<tr>
<td>Scenario 2: Data Flow Coverage Criteria</td>
<td>We give three examples of typical data flow coverage criteria.</td>
</tr>
<tr>
<td>Q6 — “Def Coverage”</td>
<td>Cover all statements defining a variable t.</td>
</tr>
<tr>
<td>Q7 — “Use Coverage”</td>
<td>Cover all statements that use the variable t as right hand side value.</td>
</tr>
<tr>
<td>Q8 — “Def-Use Coverage”</td>
<td>Cover all def-use pairs of variable t.</td>
</tr>
<tr>
<td>Scenario 3: Constraining Test Cases</td>
<td>During development and for code exploration, it is often desired 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.</td>
</tr>
<tr>
<td>Q9 — “Constrained Program Paths”</td>
<td>Basic block coverage with test cases that satisfy the assertion ( j &gt; 0 ) after executing line 2.</td>
</tr>
<tr>
<td>Q10 — “Constrained Calling Context”</td>
<td>Condition coverage in function compare with test cases which call ( \text{compare} ) from inside function sort only.</td>
</tr>
<tr>
<td>Q11 — “Constrained Inputs”</td>
<td>Basic block coverage in function sort with test cases that use a list with 2 to 15 elements.</td>
</tr>
<tr>
<td>Q12 — “Recursion Depth”</td>
<td>Cover function eval with condition coverage and require each test case to perform three recursive calls of eval.</td>
</tr>
<tr>
<td>Scenario 4: Customized Test Goals</td>
<td>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.</td>
</tr>
<tr>
<td>Q13 — “Avoid Unfinished Code”</td>
<td>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.</td>
</tr>
<tr>
<td>Q14 — “Avoid Trivial Cases”</td>
<td>Cover all conditions and avoid trivial test cases, i.e., require that insert is called twice before calling eval.</td>
</tr>
<tr>
<td>Scenario 5: Seamless Transition to Verification</td>
<td>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.</td>
</tr>
<tr>
<td>Q15 — “Restricted Scope of Analysis”</td>
<td>Condition coverage in function partition with test cases that reach line 7 at least once.</td>
</tr>
<tr>
<td>Q16 — “Condition/Decision Coverage”</td>
<td>Condition/decision coverage (the union of condition and decision coverage) [27].</td>
</tr>
</tbody>
</table>

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. |
- Q18-20 — “Cartesian Block Coverage” | Cover all pairs, triples, and quadruples of basic blocks in function partition. |

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: |

- 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. |
- 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. |

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 \( _* \cdot @17 \cdot _* \) 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

\[
\text{"for each basic block in the program there is a test case in the test suite which covers this basic block."}
\]

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 \( \Phi \) is a function that maps a program \( A \) to a finite set \( \Phi(A) \) of path patterns. A test suite \( \Gamma \) satisfies coverage criterion \( \Phi \) on program \( A \), if each path pattern in \( \Phi(A) \) is matched by an element of the test suite \( \Gamma \), 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
  \[ > \text{cover } _* \cdot @17 \cdot _* \]
  The quotes indicate that this regular expression is a path pattern for which we request a matching program path. We use the operators \( +, \cdot, \cdot \) for union, Kleene star and concatenation. Note that the regular expressions can contain combinations of conditions and actions, as in
  \[ > \text{cover } _* \cdot \{x = 0\} \cdot @17 \cdot _* \]
  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
  \[ > \text{cover } _* \cdot @17 \cdot _* + _* \cdot @32 \cdot _* \]
  this is tantamount to a list of two path patterns:
  \[ > \text{cover } _* \cdot @17 \cdot _* \]
  \[ > \text{cover } _* \cdot @32 \cdot _* \]
  Formally, we treat the quoted regular expressions \( _* \cdot @17 \cdot _* \) and \( _* \cdot @32 \cdot _* \) as temporary alphabet symbols \( x \) and \( y \) and obtain all words in the resulting regular language \( x + y \) with \( L(x+y) = \{x,y\} \), 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
  \[ \text{EDGES}@\text{BASICBLOCKENTRY} \]
  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 \( \text{EDGES}@\text{BASICBLOCKENTRY} \) 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
  \[ > \text{cover } _* \cdot _* \cdot \text{EDGES}@\text{BASICBLOCKENTRY} \cdot _* \cdot _* \]
  to achieve basic block coverage. At runtime, this amounts to
  \[ > \text{cover } _* \cdot _* \cdot (\{02 + 05 + 013 + 019 + 025\} \cdot _* \cdot _* \]
  which is in turn equivalent to the sequence
  \[ > \text{cover } _* \cdot _* \cdot 02 \cdot _* \cdot _* \]
  \[ \vdots \]
  \[ > \text{cover } _* \cdot _* \cdot 025 \cdot _* \cdot _* \]
  of path patterns which, together, specify basic block coverage.

- Expressions such as \( \text{BASICBLOCKENTRY} \) 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 \( \cdot \) 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
\[ \text{PATHS}@\text{FUNC(main),1} \]
which returns all non-cyclic paths through function main, for instance,
\[ *@1 \cdot 2 \cdot 3 \cdot 5 + @1 \cdot 2 \cdot 4 \cdot 5 + \ldots \]
In fact, expressions such as \( @5 \) which we used above, are short-hand for target graph expressions such as \( \text{EDGES}@\text{LINE(5)} \).

\(^1\)Similarly, we can write a safety specification \( \text{AG} @17 \) such that a model checker can compute a counterexample which serves as a test case.
Let us sum up this introduction to FQL with a comparison of three
however, we require a more complex coverage: We want test cases
set of CFAs
FQL
> cover EDGES(@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) . "_."
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 speci-
fications give rise to the 3 + 2 = 5 path patterns of Fig. 2(b), while
the third amounts to 3 x 2 = 6 path patterns of Fig. 2(c).

In this section, we have explained complex FQL queries by re-
duction 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 for-
malize 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 op ∈ Op is either a skip statement,
assignment, assumption (modeling conditional statements), function call,
or function return. Annotations include parsing information such
as line numbers or file names, and function names, labels, etc.

Definition 1. A control flow automaton (CFA) A is a tuple
(L, E, I), where L is a finite set of program locations, E ⊆ L x Lab x
L is a set of edges that are labeled with pairs of operations and
annotations from Lab = Op x 2^An, and I ⊆ L is a set of initial
locations. We denote the set of CFAs with CFA.

We write L_A, E_A, and I_A to refer to the set of program loca-
tions, the set of edges, and the set of initial locations of a CFA A,
respectively. We define \cup, \cap, and \setminus as operations on CFAs:

(L_1, E_1, I_1) \cup (L_2, E_2, I_2) = (L_1 \cup L_2, E_1 \cup E_2, I_1 \cup I_2)
(L_1, E_1, I_1) \cap (L_2, E_2, I_2) = (L_1 \cap L_2, E_1 \cap E_2, I_1 \cap I_2)
(L_1, E_1, I_1) \setminus (L_2, E_2, I_2) = (L_1 \setminus L_2, E_1 \setminus E_2, I_1 \setminus I_2)

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 I_2.

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

Definition 2. A transition system (S, R, I) consists of a state
space S, a transition relation R ⊆ S x S, and a nonempty set
of initial states I ⊆ S. A state in S consists of a program counter
value and a description of the memory. We denote with c(T) the
set of paths π = \{s_0,...,s_m\} such that s_0 \in I and (s_i, s_{i+1}) \in R, for
0 ≤ i < m.

In order to relate a CFA A = \{L, E, I\} to a corresponding transi-
tion system T = (S, R, I) we fix the following functions:

- We consider the operation op ∈ Op as a function op : S → 2^S
that takes a program state and determines its successor states.
- By pc : S → L we denote a function that, given a program
state s, yields its program location pc(s).
- By post : E x S → 2^S we denote a function that, given a CFA
edge (ℓ, (op, an), ℓ') ∈ E and a program state s, returns the set
\{s' \mid pc(s) = ℓ, pc(s') = ℓ', s' \in op(s)\}.

A CFA A naturally induces a transition system T_A:

Definition 3. Given a CFA A, we define the induced tran-
sition system T_A = (S, R, I) where S contains all possible
program states, R_A = \{(s, s') \mid s \in S \land \exists s = s_0 \in E_A, s_1 \in post(e), \text{and} i = \{s \in S \mid pc(s) = I_A\}.

Predicates & Coverage Criteria. Let T = (S, R, I) be a transition
system. For π = (s_0, s_1,...,s_m) and i ≤ j we write π_{i..j} to denote
the subpath (s_i,...,s_j). With \{\} we denote the empty path. A state
predicate ϕ is a predicate on the state space S, a path predicate ϕ
is a predicate over the set S^*, and a path set predicate ϕ is a pre-
dicate over the set 2^S. We write s \models ϕ iff a state s ∈ S satisfies ϕ,
π \models ϕ iff a path π ∈ S^* satisfies ϕ, and \Gamma \models \Phi iff a path set π \subseteq S^* satisfies Φ. We call a state predicate ϕ, a path predicate ϕ, or a path set predicate ϕ feasible over T, iff, respectively, there exists a
reachable state s ∈ S with s \models ϕ, a path π ∈ L(T) with π \models ϕ, or a path set Γ \subseteq L(T) with Γ \models Φ. We interpret the Boolean
connectives \land, \lor, and \neg on state, path, and path set predicates in the
standard way. For path predicates \phi_1 and \phi_2, we define predicate
concatenation \phi_1 \cdot \phi_2 where \pi \models \phi_1 \cdot \phi_2 holds iff
\(\pi_{0..n} \models \phi_1 \text{ and } \pi_{n+1..|\pi|} \models \phi_2 \text{ for some } 0 \leq n < |\pi|\)
\(\text{or } (\{\} \models \phi_1 \text{ and } \pi \models \phi_2) \text{ or } (\pi \models \phi_1 \text{ and } \{\} \models \phi_2)\)
MC/DC, for example, is a coverage criterion that is not elementary.

Let \( \Gamma \) be a transition system. Then a \( \Phi \)-test case \( \pi \) satisfies coverage criterion \( \Phi \) on \( \Gamma \) iff \( \Gamma \models \Phi^3 \) 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 \( \Phi \) is a coverage criterion defined as follows:

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

(ii) \( \Phi(\mathcal{A}) \) induces the predicate \( \Phi^3 \) such that \( \Gamma \models \Phi^3 \) holds iff for each test goal \( \Psi_i \in \Phi(\mathcal{A}) \) which is feasible over \( \mathcal{T}_\mathcal{A} \), \( \Gamma \) contains a test case \( \pi \in L(\mathcal{T}_\mathcal{A}) \) 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 \( \Phi \) 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 @CONDITIONGRAPH 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 \( \Phi \) represent path patterns, coverage specifications, and ECPs, respectively. An elementary coverage pattern \( \text{cover} P \) 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 \( \varepsilon \) 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 \( \cdot \), and Kleene star with \( \ast \).

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 \( \text{quote} P \), where \( P \) is a path pattern, introduces a single new symbol \( \text{quote} P \) 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 \( L(\{X\}) \) 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 \( \ell(P^*) \) is only relevant for path patterns, and \( \ell((\star P)) \) only appears as part of coverage specifications. The expression \( \star P \) introduces \( P^* \) as a new symbol and, thus, \( \ell(P^*) \) results in the singleton set \( \{P^*\} \). For example, \( \ell((a+b^*+^{c\star})*.ac^*) \) is the set \( \{a+b^*ac^*, c\star ac^*\} \). 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 \( w \in \ell(X) \) as a path predicate. We write \( \pi \models w \) if \( \pi \) satisfies the word \( w \) and inductively define the semantics thereof in Table 3. The empty set is unsatisfiable and the empty word \( \varepsilon \) matches the empty sequence \( \langle \rangle \) only. We match individual states with program counter values \( \ell \) and state constraints \( \varphi \), and pairs of subsequent states with transitions \( e \).

The case \( \pi \models \ell \varepsilon \) amounts to predicate concatenation as defined in Sec. 3. The path pattern \( \pi P^* \) is satisfied by a path \( \pi \), iff there is a word \( w \in \ell(P) \) that is satisfied by \( \pi \). 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 set of entities such as basic blocks, line numbers, decisions etc. without the C programming language, hence we use according terminology.

#### Definition 7

A CFA transformer is a function \( T : \text{CFA} \rightarrow \text{CFA} \) which, on input of a CFA \( \mathcal{A} = \langle L, E, I \rangle \), computes a target graph \( T[A] = \langle L', E', I' \rangle \).

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 \( F : \text{CFA} \rightarrow \text{CFA} \) which computes for every CFA \( \mathcal{A} = \langle L, E, I \rangle \) a target graph \( F[A] = \langle L', E', I' \rangle \) with \( L' \subseteq L \), \( E' \subseteq E \), and \( I' \subseteq I \), such that \( E' \subseteq L' \times \text{Lab} \times L' \) holds.

For example, consider the CFA \( \mathcal{A} \) depicted in Fig. 3(a): The target graph \( @BASICBLOCKENTRY[A] \) 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., \( @BASICBLOCKENTRY[A] \), 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.

\[
\Phi ::= \text{coverC \: passing \:} P
\]
\[
C ::= C + C | C.C | e | \varnothing | L | E | S | \star P
\]
\[
P ::= P + P | P.P | e | \varnothing | L | E | S | P^*
\]

### Table 1: Syntax of elementary coverage patterns

<table>
<thead>
<tr>
<th>Filter Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{coverC : passing :} P )</td>
<td>Extracts the portions of the CFA that are covered by the filter function ( P )</td>
</tr>
<tr>
<td>( C + C )</td>
<td>Sum of two coverage patterns</td>
</tr>
<tr>
<td>( C.C )</td>
<td>Composition of two coverage patterns</td>
</tr>
<tr>
<td>( e )</td>
<td>An empty coverage pattern</td>
</tr>
<tr>
<td>( \varnothing )</td>
<td>The empty coverage pattern</td>
</tr>
<tr>
<td>( L )</td>
<td>All edges in line ( L )</td>
</tr>
<tr>
<td>( E )</td>
<td>All edges in edge ( E )</td>
</tr>
<tr>
<td>( S )</td>
<td>All statements in ( S )</td>
</tr>
<tr>
<td>( \star P )</td>
<td>All paths in ( P )</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>( \pi \models )</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varnothing )</td>
<td>iff false</td>
</tr>
<tr>
<td>( \pi \models \pi )</td>
<td>iff ( \pi ) is the empty sequence ( \langle \rangle )</td>
</tr>
<tr>
<td>( \pi \models \ell )</td>
<td>iff ( \pi ) has the form ( \langle s \rangle ) and ( \text{pc}(s) = \ell )</td>
</tr>
<tr>
<td>( \pi \models \varphi )</td>
<td>iff ( \pi ) has the form ( \langle \varphi \rangle ) and ( s \models \varphi )</td>
</tr>
<tr>
<td>( \pi \models e )</td>
<td>iff ( \pi ) has the form ( \langle ss' \rangle ) and ( s' \in \text{pos}(e,s) )</td>
</tr>
<tr>
<td>( \pi \models w )</td>
<td>iff ( \pi \models a \cdot w' ) with ( w = aw' ) and ( a \in L \cup E \cup S ) or ( \pi \models P^* )</td>
</tr>
</tbody>
</table>

### 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 \( \text{pred}(T, \varphi) \). Applied to a CFA \( \mathcal{A} \), \( \text{pred}(T, \varphi) \) yields a new CFA that contains for every node \( u \in \mathcal{A} \) two new nodes \( (u, \varphi) \) and \( (u, -\varphi) \) representing the evaluation of a state predicate \( \varphi \) to true, i.e., \( (u, \varphi) \), and to false, i.e., \( (u, -\varphi) \). The result of applying \( T \) to a CFA \( \mathcal{A} \) is denoted by \( T[A] \). 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 operators like \( \cdot \) and \( \backslash \) are the same as in ECPs, but, where ECPs had nodes and edges of a CFA, FQL specifications require the operators \( \text{nodes}(T) \), \( \text{edges}(T) \), and \( \text{paths}(T, k) \). Here, \( T \) is a CFA transformer expression and \( k \) is a positive integer.

The clause in Table 3 states that, given a CFA \( \mathcal{A} \), all filter functions in the \( \text{cover} \) clause are applied to the target graph \( T[A] \). In practice, this is often used as in \( @\text{FUNC}[\text{foo}] \) \( \text{cover } \langle \text{CONDITIONEDGE} \rangle \) \( \text{passing } \langle \text{EDGES(ID)} \rangle \) * which is equivalent to the spec \( \text{cover } @\text{FUNC}[\text{foo}] \langle \text{CONDITIONEDGE} \rangle \langle \text{EDGES(ID)} \rangle \) *.

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@BASICBLOCKENTRY</td>
<td>identity function</td>
</tr>
<tr>
<td>@CONDITIONEDGE</td>
<td>one edge per basic block</td>
</tr>
<tr>
<td>@DECISIONEDGE</td>
<td>one edge per decision outcome</td>
</tr>
<tr>
<td>@CONDITION_GRAPH</td>
<td>all edges contributing to decisions</td>
</tr>
<tr>
<td>@FILE</td>
<td>all edges in file ( a )</td>
</tr>
<tr>
<td>@LINE</td>
<td>all edges in source line ( x )</td>
</tr>
<tr>
<td>@FUNCTION</td>
<td>all edges in function ( f )</td>
</tr>
<tr>
<td>@STMT_TYPE</td>
<td>all edges within statements ( t )</td>
</tr>
<tr>
<td>@SET</td>
<td>all assignments to variable ( t )</td>
</tr>
<tr>
<td>@USE</td>
<td>all right hand side uses of variable ( t )</td>
</tr>
<tr>
<td>@CALL</td>
<td>all call sites of ( f )</td>
</tr>
<tr>
<td>@ENTRY</td>
<td>entry edge of ( f )</td>
</tr>
<tr>
<td>@EXIT</td>
<td>all exit edges of ( f )</td>
</tr>
</tbody>
</table>
\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Semantics of CF A transformers  \\
\hline
\( \Phi ::= \) in T cover C passing P  \\
\hline
\( C ::= C \lor C \lor \{ C \} \lor \{ N \} \lor | \lor S \lor * \)  \\
\hline
\( P ::= P \times P \times (P) \times | \times S \lor P \lor P \lor \)  \\
\hline
\( N ::= \) NODES(T) \lor EDGES(T) \lor PATHS(T,k)  \\
\hline
\( T ::= F \times PRED(T, \varphi) \lor \) COMPOSE(T,T) \lor \( T | T \lor T \lor \) SETMINUS(T,T)  \\
\hline
\( F ::= \) ID \lor @BASICLOCALENTRY \lor @CONDITIONEDGE  \\
\hline
\hline
\end{tabular}
\caption{Semantics of CF A transformers}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Syntactic construct  \\
\hline
\( \rightarrow \)  \\
\hline
\( \not\in \)  \\
\hline
\hline
\end{tabular}
\caption{Syntactic sugar}
\end{table}

Formally, we define the functions edges, paths, and \( \text{nodes}(T)[\alpha] \) by the following equations:

\[
\begin{align*}
\text{nodes}(T)[\alpha] & \rightarrow \sum_{n} \text{nodes}(T)[\alpha] \\
\text{edges}(T) & \rightarrow \sum_{e} \text{edges}(T)[\alpha] \\
\text{paths}(T,k) & \rightarrow \sum_{p} \text{paths}(T)[\alpha]
\end{align*}
\]

Intuitively, \( \text{nodes}(T)[\alpha] \) is the set of nodes of the target graph \( T[\alpha] \) obtained by applying \( T \) to \( \alpha \). The same holds for \( \text{edges}(T)[\alpha] \) and \( \text{paths}(T,k)[\alpha] \). In case a set \( \text{nodes}(T)[\alpha] \), \( \text{edges}(T)[\alpha] \), or \( \text{paths}(T,k)[\alpha] \) is empty, the corresponding operator expands to the symbol \( \emptyset \). The semantics of a specification \( \Phi \) is obtained by replacing each occurrence of \( \text{nodes}, \text{edges}, \) and \( \text{paths} \) in \( \Phi \) by the corresponding sum and applying the semantics of Table 2.

Formally, we define the functions nodes, edges, and paths. For simplicity let us assume the CFA transformer \( \text{PRED} \) was not applied, then, \( \text{nodes}(T)[\alpha] = L_T[\alpha] \), \( \text{edges}(T)[\alpha] = E_T[\alpha] \), and \( \text{paths}(T,k)[\alpha] = \{ e \} \) is a k-bounded path in \( T[\alpha] \). A k-bounded path in \( T[\alpha] \) is a sequence of edges, starting in \( L_T[\alpha] \), in which the number of target graph node occurs more than \( k \times \) times. In case \( \text{PRED} \) 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, \( \text{nodes}(\alpha) \) is the set of path patterns \( \{ f_1, f_2, f_3, f_4, f_5 \} \) and the operator \( \text{NODES}(1D) \) yields the expression \( f_1 + f_2 + f_3 + f_4 + f_5 \). Here, \( f_i \) denotes the node labeled with \( i \). The operator \( \text{EDGES}(1D) \) yields the expression \( e_{1,2} + e_{1,3} + e_{1,4} + e_{1,5} + e_{2,3} + e_{2,4} + e_{2,5} + e_{3,4} + e_{3,5} + e_{4,5} \). Here, \( e_{i,j} \) denotes the edge from node \( f_i \) to node \( f_j \). \( \text{PATHS}(1D, 1) \) yields the expression \( e_{1,2} + e_{1,3} + e_{1,4} + e_{1,5} + e_{2,3} + e_{2,4} + e_{2,5} + e_{3,4} + e_{3,5} + e_{4,5} + e_{5,6} \).
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 '/@S,J,0] | NOT(@S,J])'*$
Q10 cover @CONDITIONEDGE & @ FUNC(compare)
   passing 'NOT(@CALL(compare))'.
   @CALL(compare) & NOT(@FUNCTION(sort)))*$)
Q11 cover @ENTRY(sort). [len>=2]. [len<=15].
   NOT(@EXIT(sort))|'@*.@BASICBLOCKENTRY
Q12 in @FUNCTION(eval)
   cover @CONDITIONEDGE
   passing @CALL(eval). NOT(@EXIT(eval))| @CALL(eval)
   .NOT(@EXIT(eval))| @CALL(eval)
Q13 cover @CALL(sort) passing 'NOT(@FUNCTION(sort))'.
   @FUNCTION(sort) & NOT(@FUNCTION(unfinished))).
   NOT(@FUNCTION(sort)))*$
Q14 cover @CONDITIONEDGE passing
   'NOT(@CALL(eval))'. @CALL(insert))=2
Q15 in @FUNCTION(partition)
   cover @CONDITIONEDGE passing @?
Q16 cover @CONDITIONEDGE + @ DECISIONEDGE
Q17 cover @CONDITIONEDGE & @ FUNCTION(sort)
   -> (@BASICBLOCKENTRY & @ FUNCTION(eval))
Q18 cover @BASICBLOCKENTRY->@BASICBLOCKENTRY
Q19 cover @BASICBLOCKENTRY
   -> @BASICBLOCKENTRY->@BASICBLOCKENTRY
Q20 cover @BASICBLOCKENTRY->@BASICBLOCKENTRY
Q21 cover @STMTTYPE(assert)
Q22 cover @STMTTYPE(assert)->@STMTTYPE(assert)
Q23 cover @BASICBLOCKENTRY & @FUNCTION(eval)
   passing 'NOT(@LABEL(init))'*$
Q24 cover @ENTRY(main)
   passing @ENTRY(main)>({postcond}).
   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-Q24 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. 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.

<table>
<thead>
<tr>
<th>Spec Source</th>
<th>#goals</th>
<th>#tc</th>
<th>#inf</th>
<th>Spec Source</th>
<th>#goals</th>
<th>#tc</th>
<th>#inf</th>
</tr>
</thead>
<tbody>
<tr>
<td>list2.c</td>
<td>11</td>
<td>3</td>
<td>0</td>
<td>sort1.c</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>list2.c</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>sort2.c</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>list2.c</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td>list2.c</td>
<td>8</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>sort2.c</td>
<td>11</td>
<td>3</td>
<td>4</td>
<td>sort2.c</td>
<td>16</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>sort2.c</td>
<td>20</td>
<td>4</td>
<td>7</td>
<td>sort1.c</td>
<td>12</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>list2.c</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>list2.c</td>
<td>110</td>
<td>4</td>
<td>42</td>
</tr>
<tr>
<td>sort2.c</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>list2.c</td>
<td>1100</td>
<td>4</td>
<td>829</td>
</tr>
<tr>
<td>list2.c</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>list2.c</td>
<td>11000</td>
<td>4</td>
<td>1086</td>
</tr>
<tr>
<td>list2.c</td>
<td>11</td>
<td>4</td>
<td>0</td>
<td>sort1.c</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sort1.c</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>sort1.c</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sort1.c</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>sort2.c</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sort2.c</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>sort2.c</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

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 1.14,2 studied as well in [9], (2) we selected kbfiltc from the Windows DDK, initially studied in [3], and (3) we chose an example use case4 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.

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

2For source code cf. http://code.forsyte.de/fshell
3http://www.busybox.net/
4http://research.nianet.org/~radu/VFS/
5http://fmv.jku.at/picosat/
6Measured 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 kbfiltc 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 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 can we 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 \( \text{cover} \{p\} \).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 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 $\text{cover}\{\text{EDGES}(|\text{PRED}(\text{ID}, \phi_1, \ldots, \phi_k)|)\}$ 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) 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.

(d) 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