A Precise Specification Framework for White Box Program Testing

Andreas Holzer Christian Schallhart Michael Tautschnig Helmut Veith

Formal Methods in Systems Engineering, FB Informatik, TU Darmstadt

holzer@forsyte.de, schallhart@forsyte.de, tautschnig@forsyte.de, veith@forsyte.de

Technical Report TUD-CS-2009-0148

A Precise Specification Framework for White Box Program Testing [⋆]

Andreas Holzer Christian Schallhart Michael Tautschnig Helmut Veith

Formal Methods in Systems Engineering, FB Informatik, TU Darmstadt
holzer@forsyte.de, schallhart@forsyte.de, tautschnig@forsyte.de, veith@forsyte.de

Abstract. Coverage criteria for white box testing naturally fall into two groups: The first group are generic off-the-shelf criteria such as basic block coverage and condition coverage which are applied in the same uniform way to different source code. Although generic criteria are supported by industrial strength tools, simple experiments reveal that the tools disagree on standard notions such as condition coverage. The second group are code specific coverage criteria which are either defined on the fly during program development, or consciously designed to reflect requirements specific to the application and architecture. For the second group, there is a notable lack of tool support. In this paper, we aim to solve the problems of both groups – lack of precision and lack of adequate tool support – in a unified framework: We describe a specification language which designates coverage targets in the source code as building blocks for formal coverage criteria. On the one hand, our language FQL facilitates the precise specification of coverage criteria on the basis of a clean and intuitive semantics; on the other hand, we show how to apply the query-driven program testing paradigm presented at VMCAI 2009 for efficient test case generation with the help of a model checker. Experimental results demonstrate the practical feasibility of our framework.

1 Introduction

In industrial development practice, most testing efforts follow a source code oriented white box approach; in scope, they range widely from quick and local program exploration tasks to certification eligible and coverage achieving testing procedures. As today's agile development processes are typically driven by incremental refactoring and refinement steps, we need testing tools which facilitate the specification and generation of test suites in accordance with the increments of the source code.

In a predecessor paper [1] we have argued that the diversity of requirements calls for a clear separation between the test specification formalism and the test case generation engine. Similar as in databases, the specification formalism needs a simple and intuitive semantics, and should rely on an evolving standard which

[⋆] Supported by DFG grant FORTAS – Formal Timing Analysis Suite for Real Time Programs (VE 455/1-1).

can be extended by new features over time. The test case generation engine on the other hand should be viewed as a backend which can be implemented using different approaches and platforms to achieve optimal performance. Thanks to this database analogy we are speaking of *query-driven* test case generation.

To bootstrap the query-driven approach, we first developed an efficient test case generation backend for C. In [1] we presented two algorithms – iterative constraint strengthening (ICS) and groupwise constraint strengthening (GCS) – which employ bounded model checking and incremental SAT solving [2] for efficient enumeration of complex test cases. Using an informal prototype of the specification language, we demonstrated the good practical performance of our backend. The current paper continues this work with a systematic account of the query language FQL and its query engine FSHELL.

Although natural in retrospect, it came as a surprise to us that the specification language turned out to be the most subtle scientific challenge in this project. The main problem lies in the big variety of natural test strategies some of which rely on the syntax of the program, and some of which relate to program semantics. Having considered multiple approaches we converged to the FQL framework presented in the current paper. We believe that our concepts provide a solid foundation for future development.

Like for a database query language the requirements for FQL combine usability, semantics, and efficiency:

(a) **Precise Semantics.** To the best of our knowledge, there is no commonly established and precise formalism for whitebox testing. This lack of precision in testing undermines the precise meaning of certification standards such as [3].
To illustrate the problem, we use three commercial test tools CoverageMeter [4], CTC++ [5], and BullseyeCoverage [6] to check for condition coverage on the C program shown in Listing 1.

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

Listing 1. Sample program

We compiled the C program using the tool chain of each coverage analysis tool and then ran the programs with two test cases, namely with `x = 1` and with `x = 4`.

In this example, CoverageMeter and CTC++ reported 100% coverage whereas BullseyeCoverage evaluated the test suite to achieve coverage to a degree of 83%. The difference occurs because BullseyeCoverage treats not only the variable `a` in line 3 as condition but also `x>2` and `x<5` in line 2.

(b) **Expressive Power.** In addition to generic criteria such as basic block coverage or condition coverage, FQL should support user defined coverage criteria which refer to program constructs, program regions or paths in a similar manner as programmers reason about and work with programs.

(c) **Encapsulation of Language Specifics.** Most standard coverage criteria can be easily translated between imperative programming languages such as C or ADA. Therefore, 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, and not hardwire programming language specific constructs.

(d) **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 to efficient test case generation algorithms in a natural way.

(e) **Simplicity.** FQL should be usable in day-to-day work by programmers without specific training in formal methods.

The FQL language concept presented in this paper attempts to resolve these requirements. The semantics of FQL is based on a graph presentation of the program called the *control flow automaton* (CFA). A CFA is essentially a control flow graph where the edges are labeled with commands and (in our case) also with other parser annotations. For example, Figure 2(a) shows the CFA of Listing 2, which is part of a quicksort implementation.

FQL coverage criteria are based on the notion of *target graphs*, i.e., those fragments of the source code that are relevant for a given testing target. For instance, a natural target graph for basic block coverage contains one edge for each basic block in the program, see Figure 2(b). FQL enables the programmer to describe target graphs in a simple and flexible formalism. For instance, the query

```
> cover EDGES(@BASICBLOCKENTRY)
```

achieves basic block coverage by requesting a test suite which contains all edges of the above mentioned target graph. Table 1 demonstrates how standard coverage criteria can be expressed in FQL using different target graphs for decisions. We will present these in detail in Section 2.1. FQL also provides many constructs to fine-tune coverage requirements by extracting and combining target graphs. For example, the query

```
> cover EDGES(@BASICBLOCKENTRY)->EDGES(@BASICBLOCKENTRY)
```

requests test cases that pass through each pair of basic blocks (in the sense of a Cartesian product), whenever this is feasible. In many scenarios, this is a powerful approximation of path coverage, which is usually infeasible. The basic

Criterion	FQL Query
Basic Block	`cover EDGES(@BASICBLOCKENTRY)`
Decision	`cover EDGES(@DECISIONEDGE)`
Condition	`cover EDGES(@CONDITIONEDGE)`
Predicate Complete(P)	cover STATES(ID,P)
Multiple Condition	`cover PATHS(@CONDITIONGRAPH,1)`
Path(k)	cover PATHS(ID,k)

Table 1. Common coverage criteria

form of queries can be modified by restricting the program scope and requiring certain code locations to be reached in all test cases. For instance, the query

```
> in @FUNC(partition) cover EDGES(@BASICBLOCKENTRY)
  passing ID*.@5{j > 5}.ID*
```

requests basic block coverage only inside function `partition` with the additional assertion that `j > 5` holds after executing the statement in line 5 of the program.

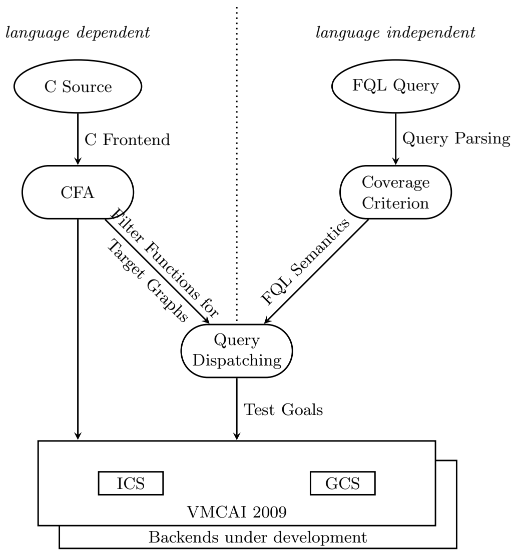


Fig. 1. FQL language layers

In the rest of the paper, we describe how FQL resolves the requirements (a) to (e). FQL achieves criteria (a), (b), and (e) by virtue of the detailed syntax and semantics provided in Sections 2 and 3. The encapsulation of language specifics required in criterion (c) is achieved by our use of target graphs which provide an intermediate layer between language specific and language independent aspects of test case generation, cf. Figure 1. Finally, to achieve criterion (d), we are focusing on C as a concrete programming language. We chose C mainly for two reasons, first, the availability of high quality software model checkers such as CBMC [7], and second, the practical importance of C for embedded and safety-critical software. In Section 4, we describe the query dispatcher, which integrates FQL with the backend of [1], and in Section 5, we substantiate the feasibility of the FQL framework with experimental results on real world C code.

2 Overview of FQL

In this section we give an overview of our query language FQL, the FShell query language. FQL enables the programmer to describe target graphs in a simple flexible formalism. It provides constructs to choose edge- or path-coverage or combinations of these. We illustrate these language features on several examples, before we describe the mathematical framework and the semantics of FQL in detail in Section 3.

2.1 Target Graphs and Filter Functions

A target graph is a fragment of a CFA that precisely describes the edges and paths to be covered by a test suite. For example, a target graph for basic block coverage contains the first edge of each basic block. Figure 2(b) depicts an example of such a target graph obtained from the CFA in Figure 2(a). Edges not contained in the target graph are grayed out. Given a CFA $\mathcal{A}$, we denote this target graph by $\mathsf{basicblockentry}(\mathcal{A})$. $\mathsf{basicblockentry}$ is called a *filter function.* Filter functions extract CFA edges based on annotations added to the CFA $\mathcal{A}$ while parsing the source code. For example, we annotate the first edge of each basic block as "basic block entry edge".

As filter functions map CFAs to (fragments of) CFAs, we first introduce control flow automata more formally. By Op we denote a finite set of operations that correspond to the statements occurring in the source code. These operations are either skip, assignments, assumptions, function calls or function returns. Control flow statements such as `if (i>=j)` in line 9 of Listing 2 are modeled by assumptions: in Figure 2(a) $\langle$`i>=j`$\rangle$ denotes the assumption that `i`$\geq$`j` holds and $\langle$`!(i>=j)`$\rangle$ denotes the assumption of the opposite.

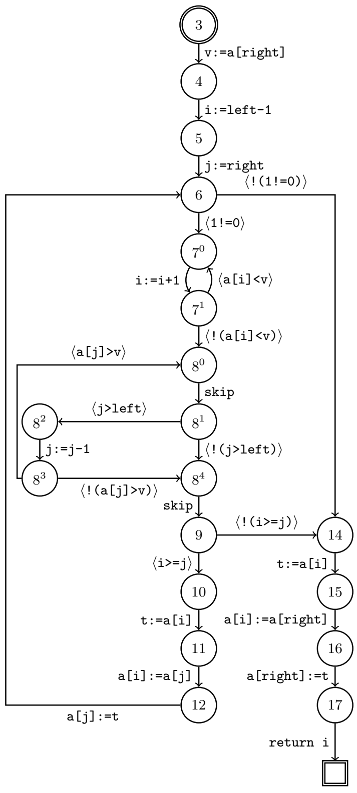
Let An denote the set of possible parsing annotations. We assign a subset of An to each individual transition. These annotations include source file names and line numbers, but also more detailed information such as basic block entries or decisions.

Definition 1 (Control Flow Automaton). *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 the 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 the set of initial locations of $\mathcal{A}$. $\mathcal{L}(\mathcal{A})$ denotes the set of all maximal paths in $\mathcal{A}$, and $\mathcal{L}_B(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A})$ denotes the* set of bounded paths *in $\mathcal{A}$ where no program location in $p \in \mathcal{L}_B(\mathcal{A})$ occurs more than $B > 0$ times.*

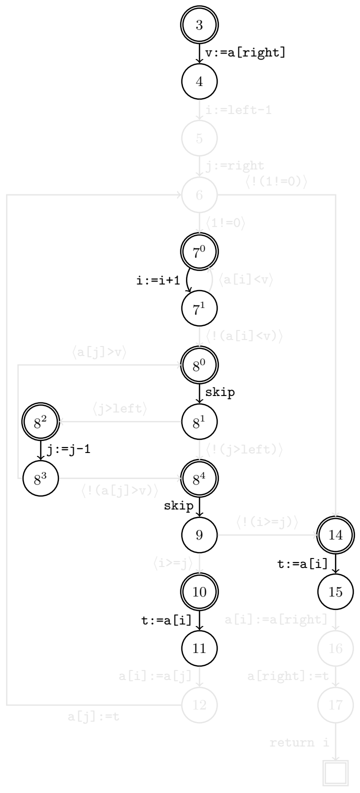
We denote the set of control flow automata with CFA and can now define filter functions.

Definition 2 (Filter Functions on CFAs). *A filter is a function $f : \mathsf{CFA} \rightarrow \mathsf{CFA}$ which, on input of a CFA $\mathcal{A} = \langle L, E, I\rangle$, computes a* target graph *$f(\mathcal{A}) = \langle L', E', I'\rangle$ where $L' \subseteq L$ and $E' \subseteq E$, such that L' does not contain isolated nodes. The set $I' \subseteq L'$ of initial elements is determined by f.*

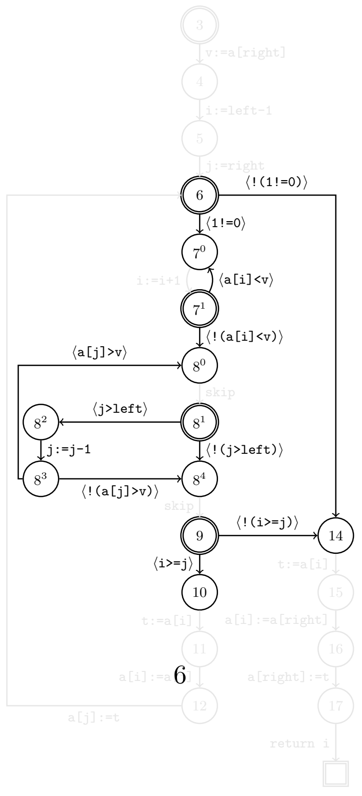
Filter functions encapsulate the interface to the programming language. Hence, to instantiate this concept, we have to fix some terminology partially specific to C.



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



(b) Target graph for basicblockentry($\mathcal{A}$)



(c) Target graph for conditiongraph($\mathcal{A}$)

Fig. 2. Control flow automaton of `partition` (Listing 2) and target graphs

```
 1 int partition(int a[], int left, int right) {
 2   int v, i, j, t;
 3   v = a[right];
 4   i = left - 1;
 5   j = right;
 6   for (;;) {
 7     while (a[++i] < v) ;
 8     while (j > left && a[--j] > v) ;
 9     if (i >= j) break;
10     t = a[i];
11     a[i] = a[j];
12     a[j] = t;
13   }
14   t = a[i];
15   a[i] = a[right];
16   a[right] = t;
17   return i;
18 }
```

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

Basic Blocks We refer to a *basic block* as a subgraph of the CFA that represents a maximal sequence of source statements which can only be entered at the first of them and exited at the last of them [8]. In case such a basic block only consists of a conditional statement, we introduce a skip statement in order to have a unique edge representing the basic block. Such an additional skip statement occurs in Figure 2 on the edges leaving states 8^0.

Conditions If a node has more than one successor, then the node refers to a *condition*, e.g., node 9 referring to i>=j. The edges leaving a condition node are called *condition edges* which are in this case the edges to the nodes 10 and 14. In the C programming language, Boolean expressions are evaluated with *short-circuit semantics,* i.e., aggregated Boolean expressions which involve several conditions are only evaluated until their combined result is determined. For example, if the first condition j>left in evaluating (j>left && a[−−j]>v) occurring in while statement of line 8 turns out to be false, then a[−−j]>v is never evaluated. The control flow automaton induced by a Boolean expression is called the corresponding *condition graph.* For example, the while statement of line 8 induces the network of nodes 8^0 to 8^4 and includes the conditions j>left and a[−−j]>v. In particular, the condition graph does *not only* consist of the condition edges but also of all other computations which are necessary to evaluate the Boolean expression. For example, the condition graph of (j>left && a[−−j]>v) includes the transition 8^2 to 8^3 which performs the operation j:=j−1.

Therefore, every aggregated Boolean expression induces a non-trivial control-flow to abort the evaluation as soon as possible. Note that aggregated Boolean expressions may occur outside conditional statements—imposing non-trivial control-flow in apparently unconditional code, e.g., line 2 of the program in Listing 1 does not evaluate x<5 if x>2 already evaluated to false.

Decisions The Boolean expression controlling a condition statement (in case of C an if, switch, while, or for statement—and every statement involving the conditional operator `?:`) is called a *decision*. As any other Boolean expression, a decision is potentially involving several conditions. If there is more than one edge for one of the possible outcomes of the decision, we let them all pass through a dedicated new state which has only a single outgoing edge labeled with skip statement. This edge is then used to represent the corresponding outcome of the overall decision. For example in Figure 2, the node 8^4 has been introduced to collect all false outcomes of the while statement in line 8 and the edge from 8^4 to 9 is therefore used represent the false outcome of this decision (this edge must be inserted as well as unique basic block entry edge for the if statement following in line 9).

For the conceptual discussion and the subsequent examples, we now list some of the most important filter functions implemented in FSHELL. In these descriptions we will use two additional functions on CFA edges. For a set E of edges, we define the set of start locations $\mathsf{start}(E) = \{\ell \mid (\ell, l, \ell') \in E\}$. Analogously, we define the set of end locations $\mathsf{end}(E) = \{\ell \mid (\ell', l, \ell) \in E\}$.

- $\mathsf{basicblockentry}(\mathcal{A})$ consists of all the edges in $\mathcal{A}$ which correspond to the first statement of a basic block. Figure 2(b) shows the target graph for $\mathsf{basicblockentry}(\mathcal{A})$, referring to Listing 2. Given the set E of edges in $\mathsf{basicblockentry}(\mathcal{A})$, the set of initial locations is $\mathsf{start}(E)$, which are marked with double circles in Figure 2(b).
- $\mathsf{conditionedge}(\mathcal{A})$ consists of all those edges which correspond to a specific outcome of a condition in $\mathcal{A}$. Given the set E of edges in $\mathsf{conditionedge}(\mathcal{A})$, the set of initial locations is $\mathsf{start}(E)$.
- $\mathsf{conditiongraph}(\mathcal{A})$ consists of all condition graphs induced by the evaluation of a Boolean expression in $\mathcal{A}$. Therefore, $\mathsf{conditiongraph}(\mathcal{A})$ is the superset of $\mathsf{conditionedge}(\mathcal{A})$ which contains not only the condition edges but also all the computations interconnecting them.
  The set of initial locations consists of the entry locations of these subgraphs corresponding to the entry locations of the represented decisions—i.e., the locations where the program execution starts to evaluate these Boolean expressions. Figure 2(c) shows the target graph for $\mathsf{conditiongraph}(\mathcal{A})$.
- $\mathsf{decisionedge}(\mathcal{A})$ consists of all those edges which correspond to a specific outcome of a decision in $\mathcal{A}$ (e.g., for an if-statement, there is a true- and a false-edge). Given the set E of edges in $\mathsf{decisionedge}(\mathcal{A})$, the set of initial locations is $\mathsf{start}(E)$.
- $\mathsf{stmttype}[\mathsf{types}](\mathcal{A})$ consists of all those edges which correspond to the execution of all statements of types types, where we allow for types all kinds of statements occurring in C, e.g., $\mathsf{stmttype}[\mathsf{if}, \mathsf{switch}, \mathsf{for}, \mathsf{while}, ?:]$ selects all computations performed by conditional statements.
- $\mathsf{file}[\texttt{file}](\mathcal{A})$ contains all parts of $\mathcal{A}$ induced by the contents of file `file`. The set of initial locations is the set of entry locations of the graphs representing the functions contained in `file`.
- $\mathsf{line}[\texttt{n}](\mathcal{A})$ contains all parts of $\mathcal{A}$ annotated with line `n`. Given the set E of edges in $\mathsf{line}[\texttt{n}](\mathcal{A})$, the set of initial locations is $\mathsf{start}(E)$.

– $\mathsf{func}[\texttt{fct}](\mathcal{A})$ contains all parts of $\mathcal{A}$ corresponding to function `fct`. The set of initial locations is the singleton set containing the entry location of `fct`.
– $\mathsf{id}(\mathcal{A})$ is the identity function, i.e., $\mathsf{id}(\mathcal{A}) = \mathcal{A}$.

For each of the above stated filter functions FQL provides corresponding primitives, e.g., `@BASICBLOCKENTRY` or `@CONDITIONEDGE`. For frequently used expressions we also provide short hands. For example, to select edges for line 8, we use `@8` instead of `@LINE(8)`.

In order to express targets like "cover all conditions in line 8" we add set theoretic operations: `INTERSECT(@CONDITIONEDGE, @8)` applied to Listing 2 results in a target graph with four edges, corresponding to the true/false evaluations of the comparisons `j > left`, and `a[j] > v`. These set theoretic operations amount to corresponding operations on the sets of (initial) nodes and edges of the target graphs. For a more complete list of primitives and operations present in FQL see Section 3.6.

Target graphs and filter functions are essential concepts to achieve the first three design goals stated in Section 1: (a) they explicitly define what test targets are, (b) they allow the flexible combination of their target graphs, and (c) they encapsulate the specifics of a programming language.

2.2 Predicated CFA

In the above example we only considered purely syntactical properties of the program under scrutiny. Assume, however, that we need to reason about specific variable valuations, or a range of variable valuations. For example, we want to test the presented algorithm with $\mathit{left} \leq 1 \wedge \mathit{right} \leq 1$, $\mathit{left} \leq 1 \wedge \mathit{right} > 1$, $\mathit{left} > 1 \wedge \mathit{right} \leq 1$, and $\mathit{left} > 1 \wedge \mathit{right} > 1$.

To achieve this, we add predicates to a CFA: Given a set of n predicates we replace each program location by 2^n *predicated* program locations that represent *all possible* valuations of the predicates at the original CFA location. For the above example, we therefore add the predicates $\mathit{left} \leq 1$ and $\mathit{right} \leq 1$.

Definition 3 (Predicated CFA). *Given a CFA $\mathcal{A} = \langle L, E, I \rangle$ and predicates $\phi_1, \ldots, \phi_k$ over program variables, the* predicated CFA $\mathcal{A}[\phi_1, \ldots, \phi_k] = \langle L', E', I' \rangle$ *is defined by $L' = L \times \{0,1\}^k$, $I' = I \times \{0,1\}^k$, and*

$$E' = \left\{ ((\ell, \bar{i}), l, (\ell', \bar{i}')) \mid \bar{i}, \bar{i}' \in \{0,1\}^k \wedge (\ell, l, \ell') \in E \right\} .$$

$\mathcal{L}(\mathcal{A}[\phi_1, \ldots, \phi_k])$ and $\mathcal{L}_B(\mathcal{A}[\phi_1, \ldots, \phi_k])$ are defined analogously to the unpredicated case, where $\mathcal{L}_B(\mathcal{A}[\phi_1, \ldots, \phi_k])$ refers to the set of maximal paths in $\mathcal{A}[\phi_1, \ldots, \phi_k]$ where no predicated program location occurs more than $B > 0$ times.

Note, a predicated CFA $\mathcal{A}[\phi_1, \ldots, \phi_k]$ includes 2^{k^2} edges for every edge in $\mathcal{A}$, i.e., $\mathcal{A}[\phi_1, \ldots, \phi_k]$ is derived from structural properties only and does not originate from predicate abstraction where we would skip semantically infeasible edges.

A predicated CFA shares the structural properties of a CFA, in particular, the edges of a predicated CFA are again labeled with pairs of operations and annotations. The filter functions described above can thus be immediately applied to predicated CFAs as well, because adding predicates does not alter operations or annotations. We canonically extend the application of filter functions from CFAs to predicated CFAs. We define the application of a filter function f to a predicated CFA $\mathcal{A}[\phi_1, \ldots, \phi_k]$ as the predication of the target graph $f(\mathcal{A})$, i.e., $f(\mathcal{A}[\phi_1, \ldots, \phi_k]) = f(\mathcal{A})[\phi_1, \ldots, \phi_k]$.

While operations and annotations are not affected by adding predicates, program locations now in addition carry information about predicate valuations: each node $(\ell, \bar{\imath})$ with $\bar{\imath} = (i_1, \ldots, i_k) \in \{0,1\}^k$ asserts that ϕ_j holds at a location ℓ, iff $i_j = 1$.

In FQL, we denote predicates using a C-style notation and delimit predicate expressions by curly braces. For example, the predicates *left* ≤ 1 and *right* ≤ 1 are given by `{left <= 1}`, `{right <= 1}`.

2.3 Path Monitors

In addition to target graphs, which induce sets of test goals, we need a formalism to describe paths. To this end, we introduce *path monitors.* Semantically, a path monitor is a path predicate, i.e., a property which can evaluate to true or false on a given path. We then require that each test case satisfies the predicate given by the monitor. Syntactically, we use regular expressions to describe these path predicates. We call these expressions *path monitor expressions.*

We use filter functions to match (sets of) edges of a CFA. In analogy to regular expressions, FQL provides operators for alternatives ("`+`"), concatenation ("`.`"), and the Kleene star ("`*`"). For example, we use the specification `@3.ID*.@10.ID*.@17` to describe paths which start with the first statement of the procedure in line 3, pass through any edge of the program zero or more times ("`ID*`") but pass line 10 at least once, and reach the **return** statement in line 17. The idiom `.ID*.` occurs frequently. In FQL, therefore also "`->`" may be used instead of `.ID*.`, which shortens the above expression to `@3->@10->@17`.

Because we have the full power of filter functions available, we can also easily express the property that line 10 must *not* be reached with `@3.COMPLEMENT(@10)*.@17`. We can use `COMPLEMENT` to ensure that no test case ever enters some function `unimplemented`: `COMPLEMENT(@CALL(unimplemented))*`. In a similar vein, we can also express API usage rules expressible using regular expressions. For example, the path monitor expression

```
(COMPLEMENT(@CALL(free))+(@CALL(malloc)->@CALL(free)))*
```

requests that if `free` is ever called, it must be preceded by a call to `malloc`.

To loop at least four times through lines lines 6–12, we use a path monitor expression `(@6->@12) >= 4`, which uses the additional bounded repetition operator. Assuming a C program that implements an API for list manipulation, we can ask for test cases that insert into a list no more than ten times and afterwards call a sorting algorithm: `@CALL(insert) <= 10. COMPLEMENT(@CALL(insert))*. @CALL(sort)`

In addition to these syntactic restrictions, pre- and postconditions over program variables can be added. For example, a path monitor expression `{j>5}@6->@8{j==left}` asserts that line 6 is reached with $j > 5$ and $j = \mathit{left}$ must hold after executing the statements in line 8. We can thereby also express invariants: `({j>0}ID)*` requires that $j > 0$ holds before executing any statement. For Listing 2, we can request that `partition` is never called with a null pointer as the first argument: `@CALL(partition){a != NULL}`

Note that there is no technical reason to restrict path monitors to regular languages. Future extensions of FQL may well include, e.g., context free features such as brace matching.

2.4 Simple Coverage Queries

Given a—possibly predicated—CFA, we apply filter functions to obtain a target graph. This subgraph of the CFA then has three natural interpretations to obtain a set of test goals. We can choose to cover either (i) all states, (ii) all edges, or (iii) all paths in the target graph. In the first and second case, the states and edges induce a tractably sized set of test goals. The third case possibly yields an *exponential* number of test goals in an acyclic graph. If the target graph contains cycles, the number of paths would not even be bounded. We therefore require the explicit specification of a *bound* that limits the number of recurrences of a node in each path.

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

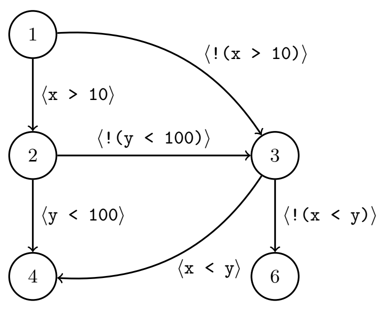


Fig. 3. Edge- vs. path-coverage

Figure 3 illustrates the difference between state-, edge-, and path-coverage. The CFA is acyclic and has five control flow paths. Two test cases such as $(x = 11, y = 99)$ and $(x = 10, y = 10)$ suffice to cover all states occurring in the CFA. To cover all edges, one needs at least three test cases, e.g., picking $(x = 11, y = 100)$ in addition to the two cases used for state coverage. Finally, in case of path coverage, at least five test cases are necessary.

In FQL, we use the keywords `STATES`, `EDGES`, and `PATHS` to choose the interpretation of the target graph. Taking Figure 2 as example, the query

```
> cover STATES(ID)
```

asks to cover all states in Figure 2(a), whereas

```
> cover EDGES(@BASICBLOCKENTRY)
```

asks for a test suite that covers the edges of the target graph shown in Figure 2(b). The query

```
> cover PATHS(@CONDITIONGRAPH, 1)
```

requests a test suite that covers the feasible paths in the target graph depicted in Figure 2(c), where no test goal requires that a CFA node is visited more than once.

An edge- or path-coverage expression may be prefixed or followed by predicates over program variables. This enables the specification of pre- or postconditions that must hold on each coverage target. For example, the query

```
> cover {left>1}PATHS(@CONDITIONGRAPH, 1){j>=left}
```

requests that in all test cases, $left > 1$ holds before entering a path in the condition graph and $j > left$ holds when leaving the condition graph.

While pre- and postconditions require predicates to hold at specific points of program execution, predicated CFAs allow testing with respect to a set of predicates *at all* program locations. Therefore FQL is capable of expressing predicate complete coverage [9] (cf. Table 1). In FQL, we specify predicated CFAs by adding the set of predicates to an edge- or path-coverage expression. For example, to request a test suite that achieves basic block coverage with respect to the predicates $left \leq 1$ and $right \leq 1$, we use the FQL query

```
> cover EDGES(@BASICBLOCKENTRY, {left<=1}, {right<=1})
```

2.5 Coverage Sequences

Edge- and path-coverage specifications already give us a powerful tool to describe test goals. In fact, this suffices to define several generic coverage criteria, as shown in Table 1. In many practical situations, however, we would like to request coverage of several targets by a single program execution. This is best compared to debugging sessions, where the developer sets several breakpoints and seeks executions that touch upon all of these.

In FQL, we therefore provide *coverage sequences*. Coverage sequences concatenate a list of distinct state-, edge-, or path-coverage specifications to a new set of test goals. This new set is made up from the cross product of the subgoals. Reconsider the example from Figure 3. A query

```
> cover EDGES(@1)->EDGES(@3)
```

requests coverage of all edges in line 1 combined with each edge in line 3. Since both lines induce two individual edges each, a matching test suite contains at least four test cases, e.g., $(x = 11, y = 100)$, $(x = 10, y = 10)$, $(x = 100, y = 100)$, and $(x = 10, y = 11)$.

Furthermore, we use path monitors to precisely describe the permissible execution steps between each pair of concatenated coverage specifications. In the above example we used "`->`" to state that any execution between the test goals of the two edge coverage expressions is permissible. If, conversely, executions must be restricted, "`-[ P ]>`" with a path monitor expression `P` is used. As such, "`->`" is only a shorthand for `-[ID*]>`. For a more complex example with a non-trivial path monitor, assume that the function `partition` is called multiple times and that we want to test behavior of `partition` for *each individual* calling context. If we consider condition coverage as sufficient to exercise the entire behavior of a procedure, then we can use the query

```
> cover EDGES(@CALL(partition))
  -[COMPLEMENT(@EXIT(partition))*]>
  EDGES(INTERSECT(@FUNC(partition), @CONDITIONEDGE))
```

where we require to cover *each combination* of a call to partition and a condition edge within partition with a single test case, such that this test case does not leave partition between the selected call and the chosen condition edge. Without the path monitor `COMPLEMENT(@EXIT(partition))*`, the specification would permit a test case which leaves and reenters partition through some other calling context—corrupting the goal of the specification to cover all condition edge for all calling contexts.

2.6 FQL Queries

All of the above examples of `cover`-clauses already pose valid FQL queries. We can thereby define several well known coverage criteria (cf. [10, 9]), as already shown in Table 1. Therein, predicate complete coverage is parametrized by a set P of predicates given as C-expression as in {`j>=left`}. Furthermore, path coverage takes the bound k as limit to the number of occurrences of the same state.

Our definition of condition coverage presumably matches the implementation of BullseyeCoverage, because the filter function `cover EDGES(@CONDITIONEDGE)` captures all conditional edges, even outside decisions. To achieve condition coverage, as defined by CoverageMeter and CTC++, it appears to be sufficient to use

```
> cover EDGES(INTERSECT(@CONDITIONEDGE,
             @STMTTYPE(if,switch,for,while,?:)))
```

i.e., to ignore conditional evaluation outside decisions.

Restricting Program Paths We have already seen the application of path monitors in coverage sequences. To restrict the set of paths permissible in a test suite, however, we add the `passing` clause to an FQL query. For example, to achieve basic block coverage with test cases which satisfy the assertion $j > 5$ before entering the loop in line 6 in Listing 2, we use the query:

```
> cover EDGES(@BASICBLOCKENTRY) passing ID*.@5{j>5}.ID*
```

Default Scope of Analysis FQL provides the prefix `in` to restrict queries to a subset of the program under scrutiny. The following query

```
> in @FUNC(partition) cover EDGES(@CONDITIONEDGE)
  passing ID*.@3->@10->@17.ID*
```

asks for a test suite that achieves condition coverage in function `partition` with paths that must reach line 10 at least once, i.e., that swapping takes place at least once.

3 FQL Semantics

In the previous section we developed the syntax of FQL which enables us to specify coverage criteria in reference to the (predicated) CFA. For a program represented as CFA, an FQL query describes a coverage criterion as a finite set of test goals. Each test goal requires to either cover a state, an edge, a path or a sequence thereof answering a corresponding state coverage, edge coverage, path coverage, or coverage sequence specification.

In order to be able to speak about the semantics of programs, and, hence, about feasibility of test goals, we introduce transition systems (Section 3.1) and relate CFAs to them (Section 3.2). Then, in Section 3.3, we formalize our notions of test case, test suite and coverage criterion. In Section 3.6, we translate FQL queries to sets of test goals using the concepts defined in Section 3.4.

3.1 Transition Systems

We model the semantics of a program by a transition system:

Definition 4 (Transition System). *A transition system $\mathcal{T}$ is a triple $\langle \mathcal{S}, \mathcal{R}, \mathcal{I} \rangle$ consisting of the state space $\mathcal{S}$, the transition relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$, and the nonempty set of initial states $\mathcal{I} \subseteq \mathcal{S}$. The individual states in $\mathcal{S}$ consist of the program counter and a complete description of all stack and heap contents.*

The feasibility of a test goal can be determined only with respect to the semantics of a program, i.e., the set of all possible program executions. To formalize the notion of program execution, we introduce state sequences and paths:

Definition 5 (State Sequence, Path). *Given a transition system $\mathcal{T} = \langle \mathcal{S}, \mathcal{R}, \mathcal{I} \rangle$, a* state sequence *is a finite word $\pi = \langle s_1, \ldots, s_n \rangle \in \mathcal{S}^*$ of states $s_i \in \mathcal{S}$. We write $s \in \pi = \langle s_1, \ldots, s_n \rangle$, iff $s = s_i$ for some $1 \leq i \leq n$. The sequence π is a* path*, if $s_1 \in \mathcal{I}$ and $\langle s_i, s_{i+1} \rangle \in \mathcal{R}$ holds for all $1 \leq i < n$. We denote with $\mathcal{L}(\mathcal{T}) \subseteq \mathcal{S}^*$ the* set of paths *induced by $\mathcal{T}$.*

We use *state predicates* to describe properties of individual program states and we use *path* and *path set predicates* in our specification of coverage criteria.

Definition 6 (State, Path, & Path Set Predicates). *Given $\mathcal{T} = \langle \mathcal{S}, \mathcal{R}, \mathcal{I} \rangle$, a* state predicate *$p$ is a predicate on the state space $\mathcal{S}$, a* path predicate *ϕ is a predicate over the set $\mathcal{S}^*$, and a* path set predicate *Φ is a predicate over the sets of paths $2^{\mathcal{S}^*}$. We write $s \models p$ iff a state $s \in \mathcal{S}$ satisfies p, $\pi \models \phi$ iff a state sequence $\pi \in \mathcal{S}^*$ satisfies ϕ, and $\Gamma \models \Phi$ iff a path set $\Gamma \subseteq \mathcal{S}^*$ satisfies Φ.*

We call a state predicate p, a path predicate ϕ, or a path set predicate Φ *feasible over $\mathcal{T}$,* iff, respectively, there exists a state $s \in \mathcal{S}$ with $s \models p$, a path $\pi \in \mathcal{L}(\mathcal{T})$ with $\pi \models \phi$, and a path set $\Gamma \subseteq \mathcal{L}(\mathcal{T})$ with $\Gamma \models \Phi$. Frequently, we are looking for a path (path set) which *contains* a state (a path) which satisfies a given state (path) predicate—leading to an *implicit existential quantification:*

Definition 7 (Implicit Existential Quantification). *To evaluate a state predicate p over a path π, we implicitly interpret p to be existentially quantified, i.e., $\pi \models p$ abbreviates $\exists s \in \pi.s \models p$. Analogously, a path predicate ϕ is existentially evaluated over a path set Γ, i.e., $\Gamma \models \phi$ iff $\exists \pi \in \Gamma.\pi \models \phi$.*

Following the definition, we find that a state predicate p can also be interpreted over a path set Γ applying the existential quantification twice, i.e., $\Gamma \models p$ iff $\exists \pi \in \Gamma.\exists s \in \pi.s \models p$. Note, that a path π satisfies a state predicate p *and* its negation $\neg p$, if there exist two states $s, s' \in \pi$ with $s \models p$ and $s' \models \neg p$ (in this case, $s \neq s'$ must hold).

3.2 Transition Systems Induced by CFAs

In this section, we relate CFAs to transition systems by interpreting program locations, and sequences thereof as state and path predicates, respectively. Then, we define the transition system that represents the semantics of the program given by the CFA. We consider a CFA as a special case of a predicated CFA where no predicates are given and only program location information are considered.

We define the set $\mathcal{S}$ of concrete states as the set of evaluations of program counter, stack and heap. Given a predicated CFA $\mathcal{A}[\phi_1, \ldots, \phi_k] = \langle L, E, I\rangle$, we interpret a program location $(\ell, \bar{\imath})$ as a state predicate:

$$s \models (\ell, \bar{\imath}) \Leftrightarrow \mathsf{loc}(s) = \ell \text{ and } \bigwedge_{1\leq j\leq k} s \models \phi_j \Leftrightarrow i_j = 1,$$

where loc maps a state s to its program counter value ℓ. Furthermore, we treat a sequence $p = \langle(\ell_1, \bar{\imath}_1), \ldots, (\ell_n, \bar{\imath}_n)\rangle \in L^*$ as a path predicate:

$$\pi \models p \Leftrightarrow \pi = \langle s_1, \ldots, s_n\rangle \text{ and } \bigwedge_{1\leq j\leq n} s_j \models (l_j, \bar{\imath}_j).$$

Comparing $(\ell, \bar{\imath})$ and $\langle(\ell, \bar{\imath})\rangle$, both interpreted as path predicates, it is important to note that the first variant is interpreted as an implicitly existentially quantified state predicate, whereas the latter is a path predicate by itself (which is therefore not implicitly existentially quantified over paths).

Next, we define the transition system induced by a CFA:

Definition 8 (Induced Transition System). *The transition system* $\mathcal{T}_{\mathcal{A}[\phi_1,\ldots,\phi_k]} = \langle\mathcal{S}, \mathcal{R}, \mathcal{I}\rangle$ induced by $\mathcal{A}[\phi_1, \ldots, \phi_k]$ *is defined by* $\mathcal{I} = \bigcup_{(\ell,\bar{\imath})\in I}\{s \in \mathcal{S} \mid s \models (\ell, \bar{\imath})\}$ *and*

$$\begin{aligned}\mathcal{R} = \{\langle s, s'\rangle \mid \langle s, s'\rangle \models \langle(\ell, \bar{\imath}), (\ell', \bar{\imath}')\rangle \text{ and } s' \in \mathsf{post}(s, \mathsf{op}),\\ ((\ell, \bar{\imath}), (\mathsf{op}, \mathsf{an}), (\ell', \bar{\imath}')) \in E\} \subseteq \mathcal{S} \times \mathcal{S},\end{aligned}$$

where $\mathsf{post}(s, \mathsf{op})$ *denotes the states that result from applying the operation given by* op *to* s.

3.3 Coverage Criteria

We define a *test case* for a transition system $\mathcal{T}$ to be a single path in $\mathcal{L}(\mathcal{T})$ and a *test suite* as a subset of $\mathcal{L}(\mathcal{T})$:

Definition 9 (Test Case and Test Suite). *Let $\mathcal{T}$ be a transition system. Then a* test case *for the set of paths $\mathcal{L}(\mathcal{T})$ 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 $\mathcal{L}(\mathcal{T})$.*

Correspondingly, a *coverage criterion* imposes a predicate on test suites:

Definition 10 (Coverage Criterion). *A* coverage criterion *Φ is a mapping from a CFA $\mathcal{A}$ to a path set predicate $\Phi^{\mathcal{A}}$. We call $\Phi^{\mathcal{A}}$* coverage predicate *and say that $\Gamma \subseteq \mathcal{L}(\mathcal{T})$ satisfies the coverage criterion Φ on $\mathcal{T}$ 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 either state or path predicates. This prototypical setting is accounted for in the next definition.

Definition 11 (Regular Coverage Criterion). *A* regular coverage criterion *Φ is a coverage criterion defined as follows:*

(i) There is a mapping $\Phi(\mathcal{A}) = \{\Psi_1, \ldots, \Psi_k\}$ which maps a CFA $\mathcal{A}$ to a set of test goals *$\{\Psi_1, \ldots, \Psi_k\}$ where each Ψ_i is a path predicate.*
(ii) This mapping induces the coverage predicate $\Phi^{\mathcal{T}_{\mathcal{A}}}$ as follows:

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

Intuitively, the above definition amounts to the following condition on coverage: "For each test goal $\Psi_i \in \Phi(\mathcal{A})$ which is feasible in $\mathcal{L}(\mathcal{T}_{\mathcal{A}})$ (i.e., there exists some path $\pi \in \mathcal{L}(\mathcal{T}_{\mathcal{A}})$ with $\pi \models \Psi_i$), the test suite Γ must contain a concrete test case $\pi' \in \mathcal{L}(\mathcal{T}_{\mathcal{A}})$ with $\pi' \models \Psi_i$."

Since all coverage criteria expressible in FQL are regular, we only deal with regular criteria in the remaining part of the paper.

3.4 Fundamental Coverage Specifications

Below, we describe the semantical foundation for state-, edge-, and path coverage (Section 2.4) and coverage sequences (Section 2.5). Building upon these preliminaries, we will define the semantics of FQL in Section 3.6. In the following, we denote the CFA representing the source code by $\mathcal{A}$.

We start with statecov, edgecov, and pathcov specifications which are building blocks for coverage sequences. All three specifications are parametrized with a filter function and a set of predicates to build target graphs. The states, edges, or paths in the target graph, respectively, induce path predicates that define the test goals:

Definition 12 (State Coverage). *Given a filter function* filter*, and a possibly empty set* $\{\phi_1, \ldots, \phi_k\}$ *of state predicates, we define the* state coverage criterion statecov[filter, $\{\phi_1, \ldots, \phi_k\}$] *as regular coverage criterion with*

$$\text{statecov}[\text{filter}, \{\phi_1, \ldots, \phi_k\}](\mathcal{A}) = \{\langle(\ell, \bar{\imath})\rangle \mid (\ell, \bar{\imath}) \in L\}$$

where $\langle L, E, I\rangle = \text{filter}(\mathcal{A}[\phi_1, \ldots, \phi_k])$*.*

Following the discussion in Section 3.2, each location $(\ell, \bar{\imath})$ of the target graph $\text{filter}(\mathcal{A}[\phi_1, \ldots, \phi_k])$ gives raise to a state predicate and each sequence of locations induces a path predicate. Hence, it appears natural to use the set of locations L as test goals. But when we use a state predicate as building block of a test goal, it would be implicitly existentially quantified—leaving the path to and from $(\ell, \bar{\imath})$ unspecified. However, we want to precisely control how state coverage is embedded into more complex coverage criteria: Therefore, we wrap each relevant location $(\ell, \bar{\imath})$ into a sequence $\langle(\ell, \bar{\imath})\rangle$ over a single location—obtaining a path predicate which matches a state sequence of exactly one state matching the location $(\ell, \bar{\imath})$.

To embed this building block into meaningful coverage criteria, we use coverage sequences, introduced below in Definition 15.

Definition 13 (Edge Coverage). *Given a filter function* filter*, and a possibly empty set* $\{\phi_1, \ldots, \phi_k\}$ *of state predicates, we define the* edge coverage criterion edgecov[filter, $\{\phi_1, \ldots, \phi_k\}$] *as regular coverage criterion with*

$$\text{edgecov}[\text{filter}, \{\phi_1, \ldots, \phi_k\}](\mathcal{A}) = E,$$

where $\langle L, E, I\rangle = \text{filter}(\mathcal{A}[\phi_1, \ldots, \phi_k])$*.*

As for state coverage, we follow Section 3.2 and use sequences of CFA locations as test goals. In this case, the relevant sequences are the edges of the target graph $\text{filter}(\mathcal{A}[\phi_1, \ldots, \phi_k])$.

Definition 14 (Path Coverage). *Given a filter function* filter*, a bound* $B > 0$*, and a possibly empty set* $\{\phi_1, \ldots, \phi_k\}$ *of state predicates, the* path coverage criterion pathcov[filter, B, $\{\phi_1, \ldots, \phi_k\}$] *is defined as regular coverage criterion:*

$$\text{pathcov}[\text{filter}, B, \{\phi_1, \ldots, \phi_k\}](\mathcal{A}) = \mathcal{L}_B(\text{filter}(\mathcal{A}[\phi_1, \ldots, \phi_k]))$$

Therefore, in case of pathcov, for each bounded path p in the target graph $\text{filter}(\mathcal{A}[\phi_1, \ldots, \phi_k])$, a test goal is created.

As an example, we study state-, edge-, and path coverage for the target graph shown in Figure 2. The coverage specification statecov[id, $\emptyset$] yields test goals for each state in Figure 2(a):

$$\text{statecov}\,[\text{id}, \emptyset](\mathcal{A}) = \{\langle 3\rangle, \langle 4\rangle, \langle 5\rangle, \langle 6\rangle, \langle 7^0\rangle, \langle 7^1\rangle, \ldots \langle 17\rangle\}$$

The coverage specification $\mathsf{edgecov}[\mathsf{conditiongraph}, \emptyset]$ yields a set of test goals corresponding to condition coverage. In the following, we omit edge labels for brevity. For Figure 2(c) we have:

$$\begin{aligned}&\mathsf{edgecov}\,[\mathsf{conditiongraph}, \emptyset](\mathcal{A}) = \\ &\quad \big\{\langle 6, 7^0\rangle, \langle 6, 14\rangle, \langle 7^1, 7^0\rangle, \langle 7^1, 8^0\rangle, \langle 8^1, 8^2\rangle, \langle 8^1, 8^4\rangle, \\ &\qquad \langle 8^2, 8^3\rangle, \langle 8^3, 8^0\rangle, \langle 8^3, 8^4\rangle, \langle 9, 10\rangle, \langle 9, 14\rangle\big\}\end{aligned}$$

On the other hand, $\mathsf{pathcov}[\mathsf{conditiongraph}, 1, \emptyset]$ yields the set of test goals for achieving multiple condition coverage. Again looking at Figure 2(c), we get:

$$\begin{aligned}&\mathsf{pathcov}\,[\mathsf{conditiongraph}, 1, \emptyset](\mathcal{A}) = \\ &\quad \big\{\langle 6, 7^0\rangle, \langle 6, 14\rangle, \langle 7^1, 7^0\rangle, \langle 7^1, 8^0\rangle, \langle 8^1, 8^2, 8^3, 8^0\rangle, \\ &\qquad \langle 8^1, 8^2, 8^3, 8^4\rangle, \langle 8^1, 8^4\rangle, \langle 9, 10\rangle, \langle 9, 14\rangle\big\}\end{aligned}$$

3.5 Composite Coverage Specifications

Coverage Sequences We introduce *coverage sequences* to embed the test goals produced by $\mathsf{statecov}$, $\mathsf{edgecov}$, and $\mathsf{pathcov}$ into longer paths.

Definition 15 (Coverage Sequence). *Given n regular coverage criteria $\Phi_1, \ldots, \Phi_n$ and $n+1$ path predicates $\Delta_0, \ldots \Delta_n$, we define the* coverage sequence *$\langle \Delta_0, \Phi_1, \Delta_1, \ldots, \Phi_n, \Delta_n\rangle$ as regular coverage criterion with*

$$\begin{aligned}&\langle \Delta_0, \Phi_1, \Delta_1, \ldots, \Phi_n, \Delta_n\rangle\,(\mathcal{A}) = \\ &\qquad \big\{\psi_{(\phi_1,\ldots,\phi_n)} \mid (\phi_1, \ldots, \phi_n) \in \Phi_1(\mathcal{A}) \times \ldots \times \Phi_n(\mathcal{A})\big\}\end{aligned}$$

where $\pi \models \psi_{(\phi_1,\ldots,\phi_n)}$ holds for a state sequence π iff π can be partitioned as $\pi = \langle \pi'_0, \pi_1, \pi'_1, \ldots, \pi'_{n-1}, \pi_n, \pi'_n\rangle$ such that $\pi'_i \models \Delta_i$ holds for $0 \le i \le n$ and such that $\pi_i \models \phi_i$ holds for $1 \le i \le n$.

In a coverage sequence, the path predicates $\Delta_0, \ldots, \Delta_n$ and individual coverage criteria $\Phi_1, \ldots, \Phi_n$ are interleaved to define for each element of the cross product $\Phi_1(\mathcal{A}) \times \Phi_2(\mathcal{A}) \times \cdots \times \Phi_n(\mathcal{A})$ a test goal. Such a test goal requires a path to first match Δ_0, then to satisfy a test goal from $\Phi_1(\mathcal{A})$, then to match Δ_1, and so forth until the path matches Δ_n.

We use coverage sequences of the form $\langle \mathsf{true}, \Phi, \mathsf{true}\rangle$, where Φ is given in terms of $\mathsf{statecov}$, $\mathsf{edgecov}$, or $\mathsf{pathcov}$, to obtain test goals that achieve coverage of the sets of states, edges, or paths in a natural way: as true matches every state sequences, the paths to and from the elements to be covered are not restricted. Such specifications allow to apply more efficient algorithms to generate covering test suites (see Section 4). For these reasons and to simplify notation, we introduce the following shorthand:

Definition 16 (Simple Coverage Sequences). *Given a regular coverage criterion Φ, we write $\langle\Phi\rangle$ as a short hand for the coverage sequence $\langle \mathsf{true}, \Phi, \mathsf{true}\rangle$. We call such coverage sequences* simple.

We briefly study (simple) coverage sequences on the above example. To obtain a test suite that achieves multiple condition coverage, we specify $\langle \mathsf{pathcov}[\mathsf{conditiongraph}, 1, \emptyset] \rangle$. A test case fulfilling the test goal $\psi_{\langle 8^1, 8^2, 8^3, 8^0 \rangle}$ is then, e.g.,

$$\langle 3, \ldots, 7^0, 7^1, 8^0, 8^1, 8^2, 8^3, 8^0, 8^1, 9, \ldots, 17 \rangle ,$$

which is partitioned as

$$\begin{aligned} \langle 3, \ldots, 7^0, 7^1, 8^0 \rangle &\models \mathsf{true} \\ \langle 8^1, 8^2, 8^3, 8^0 \rangle &\models \langle 8^1, 8^2, 8^3, 8^0 \rangle \\ \langle 8^1, 9, \ldots, 17 \rangle &\models \mathsf{true}. \end{aligned}$$

Further Composite Operators Given regular coverage criteria Φ, Φ_1, and Φ_2, and a state predicate ϕ, we define the composite regular coverage criteria $\Phi_1 \setminus \Phi_2$, $\Phi_1 \cup \Phi_2$, $\Phi_1 \sqcap \Phi_2$, $\mathsf{start{-}in}(\Phi, \phi)$, and $\mathsf{end{-}in}(\Phi, \phi)$. For $\otimes \in \{\setminus, \cup\}$, $\Phi_1 \otimes \Phi_2$ is defined by $(\Phi_1 \otimes \Phi_2)(\mathcal{A}) = \Phi_1(\mathcal{A}) \otimes \Phi_2(\mathcal{A})$. By $(\Phi_1 \sqcap \Phi_2)(\mathcal{A})$ we denote the set of path predicates φ with $\pi \models \varphi$ iff there are $\psi_1 \in \Phi_1(\mathcal{A})$ and $\psi_2 \in \Phi_2(\mathcal{A})$ such that $\pi \models \psi_1$ and $\pi \models \psi_2$. Intuitively, a test goal $\varphi \in (\Phi_1 \sqcap \Phi_2)(\mathcal{A})$, represents the set of paths that results from the intersection of the sets of paths that satisfy a $\psi_1 \in \Phi_1(\mathcal{A})$ and a $\psi_2 \in \Phi_2(\mathcal{A})$, respectively, i.e., $\{\pi \mid \pi \models \varphi\} = \{\pi \mid \pi \models \psi_1\} \cap \{\pi \mid \pi \models \psi_2\}$. For the definition of $\mathsf{start{-}in}(\Phi, \phi)$ and $\mathsf{end{-}in}(\Phi, \phi)$, we use the path predicates $\mathsf{starts{-}in}(\psi, \phi)$, and $\mathsf{ends{-}in}(\psi, \phi)$, where ψ is a path predicate and ϕ is a state predicate. $\pi \models \mathsf{starts{-}in}(\psi, \phi)$ holds for a state sequence π iff π is nonempty, $\pi \models \psi$, and $s \models \phi$ holds for the first state $s \in \pi$. The path predicate $\mathsf{ends{-}in}(\psi, \phi)$ is defined analogously, but, $s \models \phi$ holds for the last state $s \in \pi$. Then, $\mathsf{start{-}in}(\Phi, \phi)(\mathcal{A}) = \{\mathsf{starts{-}in}(\psi, \phi) \mid \psi \in \Phi(\mathcal{A})\}$ and $\mathsf{end{-}in}(\Phi, \phi)(\mathcal{A}) = \{\mathsf{ends{-}in}(\psi, \phi) \mid \psi \in \Phi(\mathcal{A})\}$.

3.6 Semantics of FQL Queries

Table 2 summarizes the semantics of FQL: Tables 2(a)-(d) show the semantics of FQL expressions and Table 2(e) presents the semantics of FQL queries. On the left hand side of each table we show the expression and query, respectively, and on the right hand side we give the translation into our formal framework.

Filter Function Expressions In Table 2(a), a denotes a file name, k a non-negative integer, f a C function identifier, and *types* is a set of C statement types. A filter function expression F given on the left hand side of the table is translated into the filter function $\mathcal{F}[\![F]\!]$ given on the right hand side.

As a prerequisite we define set theoretic operations on filter functions. Given an operation $\otimes \in \{\cup, \cap, \setminus\}$ and filter functions f_1 and f_2, we define the filter function $f_1 \otimes f_2$:

$$(f_1 \otimes f_2)(\langle L, E, I \rangle) = \langle L', E', I' \rangle,$$

where $E' = E_1 \otimes E_2$, $L' = \mathsf{start}(E') \cup \mathsf{end}(E')$, and $I' = I_1 \otimes I_2$ with $\langle L_1, E_1, I_1 \rangle = f_1(\langle L, E, I \rangle)$ and $\langle L_2, E_2, I_2 \rangle = f_2(\langle L, E, I \rangle)$. We define the composition $f_1 \circ f_2$ of two filter functions f_1 and f_2 as $(f_1 \circ f_2)(\mathcal{A}) = f_1(f_2(\mathcal{A}))$.

(a) Semantics of filter function expressions

Filter Function Expr. F	$\mathcal{F}[\![F]\!]$
`ID`	id
`COMPLEMENT(`F_1`)`	$\mathsf{id} \setminus \mathcal{F}[\![F_1]\!]$
`UNION(`F_1`, `F_2`)`	$\mathcal{F}[\![F_1]\!] \cup \mathcal{F}[\![F_2]\!]$
`INTERSECT(`F_1`, `F_2`)`	$\mathcal{F}[\![F_1]\!] \cap \mathcal{F}[\![F_2]\!]$
`SETMINUS(`F_1`, `F_2`)`	$\mathcal{F}[\![F_1]\!] \setminus \mathcal{F}[\![F_2]\!]$
`COMPOSE(`F_1`, `F_2`)`	$\mathcal{F}[\![F_1]\!] \circ \mathcal{F}[\![F_2]\!]$
`@FILE(`a`)`	$\mathsf{file}[a]$
`@LINE(`x`)`	$\mathsf{line}[x]$
`@FUNC(`f`)`	$\mathsf{func}[f]$
`@BASICBLOCKENTRY`	basicblockentry
`@CONDITIONEDGE`	conditionedge
`@DECISIONEDGE`	decisionedge
`@CONDITIONGRAPH`	conditiongraph
`@STMTTYPE[`$types$`]`	$\mathsf{stmttype}[types]$

(b) Semantics of coverage expressions

Coverage Expr. G	$\mathcal{G}[\![G]\!]$
`STATES(`F`, `Φ`)`	$\mathsf{statecov}[\mathcal{F}[\![F]\!], \Phi]$
`EDGES(`F`, `Φ`)`	$\mathsf{edgecov}[\mathcal{F}[\![F]\!], \Phi]$
`PATHS(`F`, `B`, `Φ`)`	$\mathsf{pathcov}[\mathcal{F}[\![F]\!], B, \Phi]$
`SETMINUS(`G_1`, `G_2`)`	$\mathcal{G}[\![G_1]\!] \setminus \mathcal{G}[\![G_2]\!]$
`UNION(`G_1`, `G_2`)`	$\mathcal{G}[\![G_1]\!] \cup \mathcal{G}[\![G_2]\!]$
`INTERSECT(`G_1`, `G_2`)`	$\mathcal{G}[\![G1]\!] \sqcap \mathcal{G}[\![G2]\!]$
$\{\phi\}G_1$	$\mathsf{start{-}in}(\mathcal{G}[\![G_1]\!], \phi)$
$G_1\{\phi\}$	$\mathsf{end{-}in}(\mathcal{G}[\![G_1]\!], \phi)$

(c) Semantics of path monitor expressions

Path Monitor Expr. P	$\mathcal{P}[\![P]\!]$
F	$\mathsf{one{-}of}(\mathcal{F}[\![F]\!])$
$\{\phi\}P_1$	$\mathsf{restrict{-}start}(\mathcal{P}[\![P_1]\!], \phi)$
$P_1\{\phi\}$	$\mathsf{restrict{-}end}(\mathcal{P}[\![P_1]\!], \phi)$
P_1`+`P_2	$\mathcal{P}[\![P_1]\!] \vee \mathcal{P}[\![P_2]\!]$
P_1`.`P_2	$\mathcal{P}[\![P_1]\!] \odot \mathcal{P}[\![P_2]\!]$
P_1`<=`k	$\mathcal{P}[\![P_1]\!]^{\leq k}$
P_1`>=`k	$\mathcal{P}[\![P_1]\!]^{\geq k}$
P_1`*`	$\mathcal{P}[\![P_1 {>}{=} 0]\!]$

(d) Semantics of coverage sequence expressions

Coverage Seq. Expr. C	$\mathcal{C}[\![C]\!]$
G	$\langle \mathsf{true}, \mathcal{G}[\![G]\!], \mathsf{true} \rangle$
$G_0 \ldots$`-[`P_n`]>`G_n	$\langle \mathsf{true}, \mathcal{G}[\![G_0]\!], \ldots,$
	$\mathcal{P}[\![P_n]\!], \mathcal{G}[\![G_n]\!], \mathsf{true} \rangle$

(e) Semantics of FQL queries

FQL *Query Q*	$\mathcal{Q}[\![Q]\!]$
`cover` C	$\mathcal{C}[\![C]\!]$
`in` F `cover` C	$\mathcal{C}[\![F \circledcirc C]\!]$
`cover` C `passing` P	$\mathsf{restrict}(\mathcal{C}[\![C]\!], \mathcal{P}[\![P]\!])$
`in` F `cover` C `passing` P	$\mathsf{restrict}(\mathcal{C}[\![F \circledcirc C]\!], \mathcal{P}[\![F \circledcirc P]\!])$

Table 2. FQL semantics

Coverage Expr. G	$F \circledcirc G$
`STATES(`F'`, `Φ`)`	`STATES(COMPOSE(`F'`, `F`), `Φ`)`
`EDGES(`F'`, `Φ`)`	`EDGES(COMPOSE(`F'`, `F`), `Φ`)`
`PATHS(`F'`, `Φ`)`	`PATHS(COMPOSE(`F'`, `F`), `Φ`)`
`UNION(`G_1`, `G_2`)`	`UNION(`$F \circledcirc G_1$`, `$F \circledcirc G_2$`)`
`INTERSECT(`G_1`, `G_2`)`	`INTERSECT(`$F \circledcirc G_1$`, `$F \circledcirc G_2$`)`
`SETMINUS(`G_1`, `G_2`)`	`SETMINUS(`$F \circledcirc G_1$`, `$F \circledcirc G_2$`)`
$\{\phi\}G_1$	$\{\phi\}(F \circledcirc G_1)$
$G_1\{\phi\}$	$(F \circledcirc G_1)\{\phi\}$
Path Monitor Expr. P	$F \circledcirc P$
F'	`COMPOSE(`F'`, `F`)`
$\{\phi\}P_1$	$\{\phi\}(F \circledcirc P_1)$
$P_1\{\phi\}$	$(F \circledcirc P_1)\{\phi\}$
P_1`+`P_2	$(F \circledcirc P_1)$`+`$(F \circledcirc P_2)$
P_1`.`P_2	$(F \circledcirc P_1)$`.`$(F \circledcirc P_2)$
P_1`<=`k	$(F \circledcirc P_1)$`<=`k
P_1`>=`k	$(F \circledcirc P_1)$`>=`k
P_1`*`	$(F \circledcirc P_1)$`*`
Cov. Seq. Expr. C	$F \circledcirc C$
$G_0 \ldots$`-[`P_n`]>`G_n	$(F \circledcirc G_0) \ldots$`-[`$F \circledcirc P_n$`]>`$(F \circledcirc G_n)$

Table 3. Definition of $F \circledcirc \cdot$

Filter functions are the open interface of FQL to language dependent aspects of source code and FSHELL supports more filter functions than listed in Table 2(a), e.g., `@CALL` or `@EXIT`, which are omitted because of space limitations.

Coverage Expressions Table 2(b) gives the coverage expressions of FQL and their translations to coverage criteria. In addition to state-, edge- and path-coverage expressions, we add expressions for composite coverage criteria as defined in Section 3.5 (excluding coverage sequences). In the table, Φ denotes a possibly empty set of state predicates, and ϕ denotes a state predicate.

Path Monitor Expressions The semantic function $\mathcal{P}[\![.]\!]$, defined in Table 2(c), maps a path monitor expression to a path monitor specification, i.e., a function that maps a CFA to a path predicate. $\mathsf{one{-}of}(\mathsf{filter})$ is defined by $\mathsf{one{-}of}(\mathsf{filter})(\mathcal{A}) = \bigvee_{e \in E} e$ with $\langle L, E, I\rangle = \mathcal{F}[\![F]\!](\mathcal{A})$, i.e., $\mathsf{one{-}of}(\mathsf{filter})(\mathcal{A})$ yields a path predicate that requires a state sequence to match at least one edge in the target graph $\mathcal{F}[\![F]\!](\mathcal{A})$. $\mathsf{restrict{-}start}(p, \phi)$ is a path monitor specification with $\mathsf{restrict{-}start}(p, \phi)(\mathcal{A}) = \mathsf{starts{-}in}(p(\mathcal{A}), \phi)$, and $\mathsf{restrict{-}end}(p, \phi)$ is defined by $\mathsf{restrict{-}end}(p, \phi)(\mathcal{A}) = \mathsf{ends{-}in}(p(\mathcal{A}), \phi)$. Given two path monitor specifications p_1 and p_2, $p_1 \vee p_2$ is defined by $(p_1 \vee p_2)(\mathcal{A}) = p_1(\mathcal{A}) \vee p_2(\mathcal{A})$. The path monitor specification $p_1 \odot p_2$ has the semantics that $\pi \models (p_1 \odot p_2)(\mathcal{A})$ holds for a state sequence π iff π can be partitioned into two state sequences π_1 and π_2 with $\pi_1 \models p_1(\mathcal{A})$ and $\pi_2 \models p_2(\mathcal{A})$. Before defining $p^{\leq k}$, we define p^k: $\pi \models p^k(\mathcal{A})$ holds for a state sequence π iff π can be partitioned into k-many subsequences

π_i such that $\pi_i \models p(\mathcal{A})$ holds for all $1 \leq i \leq k$. Note, $p^0(\mathcal{A})$ requires π to be the empty sequence. Then, $\pi \models p^{\leq k}(\mathcal{A})$ is equivalent to $\exists 0 \leq i \leq k . \pi \models p^i(\mathcal{A})$, and $\pi \models p^{\geq k}(\mathcal{A})$ is equivalent to $\exists i \geq k . \pi \models p^i(\mathcal{A})$.

Coverage Sequence Expressions Table 2(d) shows the translation of coverage sequence expressions to coverage sequences.

FQL Queries Using the above FQL expressions, we build FQL queries. In the definition of the semantics of FQL queries we use $F \circledcirc E$ which is defined in Table 3 and essentially composes every filter function expression inside a coverage, path monitor, and coverage sequence expression E with the filter function expression F, such that E is applied to the target graph resulting from F. An FQL query then defines a coverage criterion via the semantic mapping $\mathcal{Q}[\![.]\!]$ shown in Table 2(e). Given a regular coverage criterion Φ and a path monitor specification p, $\mathsf{restrict}(\Phi, p)$ denotes a mapping from CFAs to regular coverage criteria where $\mathsf{restrict}(\Phi, p)(\mathcal{A})$ yields a predicate ψ for every path predicate $\varphi \in \Phi(\mathcal{A})$ such that $\pi \models \psi$ holds for a path sequence π iff $\pi \models \varphi$ and $\pi \models p(\mathcal{A})$ holds.

We exemplify these definitions on the following query

```
> in @FILE("sort.c") cover EDGES(@FUNC(partition))
```

which yields the coverage criterion

$$\langle \mathsf{true}, \mathsf{edgecov}[\mathsf{func}[\texttt{partition}] \circ \mathsf{file}[\text{“}\texttt{size.c}\text{”}], \emptyset], \mathsf{true}\rangle .$$

4 Query Dispatching and Optimization

To generate a test suite as solution to an FQL query Q and a program given as CFA $\mathcal{A}$, we analyze the `cover`-clause of Q and choose—depending on its structure—a suitable algorithm to generate the matching test cases. More specifically, simple coverage criteria (recall Definition 16) which use a single state-, edge- or path-coverage expression without heading or tailing path monitors are amenable to specialized and efficient algorithms. Such criteria are of particular importance, since most generic coverage criteria are defined as simple coverage sequences (cf. Table 1).

Our top-level query processing algorithm dispatch is shown in Listing 3. It takes a query Q and a CFA $\mathcal{A}$ to return a test suite Γ which satisfies Q on $\mathcal{A}$. In the first four lines of dispatch, we simplify the query to its base case which consists of a single `cover`-clause without an accompanying `in`- or `passing`-clause: We split the query Q in line 2 into its three constituents `in` I, `cover` C, and `passing` P. Next, the filter I, which describes the scope of the query, is propagated in lines 3 and 4 into C and P by replacing each occurring filter F by `compose`(F, I) (following the rules in Tables 2(e) and 3). Then in line 5, we apply the `passing`-clause P to obtain $\mathcal{L}(\mathcal{A}) = \{\pi \in \mathcal{L}(\mathcal{A}') \mid \pi \models \mathcal{P}[\![P]\!](\mathcal{A}')\}$ where $\mathcal{A}'$ is the original CFA and where the predicate $\mathcal{P}[\![P]\!](\mathcal{A}')$ is defined following the rules of Table 2(c). We implement the latter step by translating P into an automaton which we inject into $\mathcal{A}$ to run in parallel with the original program such that

$\mathcal{A}$ spans only those executions of the original program which also satisfy the `passing`-clause.

```
 1 function dispatch(query $Q$, cfa $\mathcal{A}$) {
 2   extract $I$, $C$, and $P$ from $Q =$ in $I$ cover $C$ passing $P$;
 3   $C := C$[substitute $F$ with COMPOSE$(F, I)$];
 4   $P := P$[substitute $F$ with COMPOSE$(F, I)$];
 5   $\mathcal{A} := \mathcal{P}[\![P]\!](\mathcal{A})$;

 7   if($C =$ STATES$(F)$)      return ICS$(\mathcal{A}$, stategoals$[\mathcal{F}[\![F]\!]](\mathcal{A}))$;
 8   if($C =$ STATES$(F, \Phi)$) return GCS$(\mathcal{A}$, stategoals$[\mathcal{F}[\![F]\!], \Phi](\mathcal{A}))$;
 9   if($C =$ EDGES$(F)$)       return ICS$(\mathcal{A}$, edgegoals$[\mathcal{F}[\![F]\!]](\mathcal{A}))$;
10   if($C =$ EDGES$(F, \Phi)$)  return GCS$(\mathcal{A}$, edgegoals$[\mathcal{F}[\![F]\!], \Phi](\mathcal{A}))$;
11   if($C =$ PATHS$(F, B)$)    return GCS$(\mathcal{A}$, pathgoals$[\mathcal{F}[\![F]\!], B](\mathcal{A}))$;

13   return ICS$(\mathcal{A}, \mathcal{Q}[\![C]\!](\mathcal{A}))$;
14 }
```

Listing 3. Query dispatching

At this point, to answer the original query Q on $\mathcal{A}$, we only have to find a test suite which satisfies the plain `cover`-clause C on the restricted CFA $\mathcal{A}$. Our top-level algorithm distinguishes six cases depending on the structure of C: Aside from the general case handled in line 13, we consider simple state coverage on standard and predicated CFAs in lines 7 and 8, respectively, likewise edge coverage in lines 9 and 10, as well as simple path coverage on standard CFAs in line 11.

We solve these six cases with two algorithms, called *iterative constraint strengthening* (ICS) and *groupwise constraint strengthening* (GCS), introduced in [1]. An invocation to the basic iterative constraint strengthening algorithm $\mathsf{ICS}(\mathcal{A}, G)$ returns a test suite which satisfies all feasible goals in G on $\mathcal{A}$. The groupwise constraint strengthening algorithm GCS is a refinement of ICS and exploits knowledge on mutually exclusive test goals, i.e., test goals which cannot be covered simultaneously by the same test case. Additionally, for GCS to work, all involved test goals must be state predicates. Hence in a call $\mathsf{GCS}(\mathcal{A}, \{G_1, \ldots, G_k\})$, GCS expects the test goals $G = \bigcup_{i=1}^{k} G_i$ to be partitioned into k groups G_i of mutually exclusive test goals [1]: The algorithm requires that for all test goals $p \neq p' \in G_i$ and for all $1 \leq i \leq k$, there exists no state s with $s \models p$ and $s \models p'$.

Both algorithms initially translate the CFA and the test goals into a SAT instance utilizing a bounded model checker [7]. Then each solution to the SAT instance yields a test case which is used in turn to strengthen the constraints obtained so far such that each future solution must cover some hitherto uncovered test goal. This scheme of iteratively strengthening the same constraint database leads naturally to incremental SAT solving [2] as suitable constraint solving back-end.

Note that an FQL `cover`-clause C produces a set of test goals $\mathcal{Q}[\![C]\!](\mathcal{A})$ containing *only* path predicates. In all but the general case in line 13, we avoid to use these test goals directly but derive an alternative and more suitable set of *implementation-level* test goals to drive the search with ICS and GCS more efficiently. Due to space restrictions, we discuss in this paper the handling of edge- and path-coverage only. The two remaining cases for state-coverage in lines 7 and 8 are handled similar to the edge-coverage case.

To handle the two edge-coverage cases (lines 9 and 10) with $C = \texttt{EDGES}(F)$ and $C = \texttt{EDGES}(F, \Phi)$ respectively, we have to specify the coverage of a single program statement as concisely as possible. Thus we would prefer to work—for this purpose—with the classical control flow graph (CFG) which carries the program statements at its nodes (whereas the CFA $\mathcal{A}$ carries them with its edges). Since our approach is based on the CFA representation, we augment the CFA $\mathcal{A}$ with additional *history information* such that we know upon reaching a state in $\mathcal{A}$ the last preceding state, thereby identifying the edge taken last.

Recall for example Figure 3 on page 11: If we have a test goal which requires to cover the transition from state 1 to 3, we cannot directly replace this test goal with the state predicate requiring to reach state 3, since state 3 is reachable through states 1 and 2. Instead, we always store the last preceding state in the history, such that we can check with a state predicate (a) that the path did reach state 3 and (b) that the path visited state 1 directly before.

The only remaining case is simple path coverage (line 11) with $C = \texttt{PATHS}(F, B)$, which we approach in a manner similar to the edge case: To handle paths, the history information does not only describe the last preceding state but is extended to the relevant tailing part of the entire path. Since we allow only a bounded number of occurrences of the same state, all considered paths and therefore all accumulated history information is bounded.

In the remainder of the section, we discuss the three cases corresponding to lines 9 to 11 in detail.

4.1 Simple Edge Coverage

Many common coverage criteria (e.g., basic block, decision, and condition coverage) use coverage specifications of the form $C = \texttt{EDGES}(F)$ which yields $\langle \mathsf{edgecov}[\mathcal{F}[\![F]\!]] \rangle$ as coverage criterion via the rules in Table 2 (omitting the empty set of predicates).

Thus, each individual test goal of such a specification C applied to a CFA $\mathcal{A}$ requires at least one test case to pass through each edge in $\mathsf{edgecov}[\mathcal{F}[\![F]\!]](\mathcal{A})$. Since we want to use state predicates relying on history information, we use the following definition for our implementation level test goals used in conjunction with ICS:

Definition 17 (Edge Goals). *Given a CFA $\mathcal{A}$ and a filter function* filter*,* $\mathsf{edgegoals}[\mathsf{filter}](\mathcal{A})$ *is defined as the set of state predicates*

$$\mathsf{edgegoals}[\mathsf{filter}](\mathcal{A}) = \left\{p_e \mid e \in E_{\mathsf{filter}(\mathcal{A})}\right\}$$

where $E_{\mathsf{filter}(\mathcal{A})}$ is the set of transitions in the filtered CFA $\mathsf{filter}(\mathcal{A})$ *and where $s \models p_e$ holds for a state iff s has been entered through the edge $e = \langle s, s' \rangle$ for some other state s'.*

Using $\mathsf{edgegoals}$, we build a covering test suite for a program $\mathcal{A}$ which satisfies a coverage specification $C = \mathtt{EDGES}(F)$ with $\mathsf{ICS}(\mathcal{A}, \mathsf{edgegoals}[\mathcal{F}[\![F]\!]](\mathcal{A}))$, as invoked in line 9.

4.2 Simple Predicated Edge Coverage

In case of a simple edge coverage $C = \mathtt{EDGES}(F, \{\phi_1, \ldots, \phi_k\})$ on predicated CFAs, we obtain $\langle \mathsf{edgecov}[\mathcal{F}[\![F]\!], \{\phi_1, \ldots, \phi_k\}] \rangle$ as coverage criterion (cf. Table 2).

Analogously to the unpredicated case, we define an alternative set of implementation-level test goals suitable for GCS: We introduce an individual test goal $p_{\bar{e}}$ for each edge $\bar{e} \in E_{\mathsf{filter}(\mathcal{A}[\phi_1,\ldots,\phi_k])}$ in the filtered and *predicated* CFA $\mathsf{filter}(\mathcal{A}[\phi_1, \ldots, \phi_k])$. Since we want to apply GCS in the predicated case, we must group these test goals $p_{\bar{e}}$ into groups of mutually exclusive test goals. To this end, we introduce a group G_e for each edge of the filtered and *unpredicated* CFA $\mathsf{filter}(\mathcal{A})$ and assign $p_{\bar{e}}$ to the group G_e with $\bar{e} \models e$, i.e., $\bar{e}$ has to refine e.

Definition 18 (Predicated Edgegoals). *Given a CFA $\mathcal{A}$, a filter function* filter*, and a set of predicates $\{\phi_1, \ldots, \phi_k\}$, we define* $\mathsf{edgegoals}[\mathsf{filter}, \phi_1, \ldots, \phi_k](\mathcal{A})$ *as the set of groups*

$$\mathsf{edgegoals}[\mathsf{filter}, \{\phi_1, \ldots, \phi_k\}](\mathcal{A}) = \left\{G_e \mid e \in E_{\mathsf{filter}(\mathcal{A})}\right\}$$

each containing the state predicates

$$G_e = \left\{p_{\bar{e}} \mid \bar{e} \models e \text{ and } \bar{e} \in E_{\mathsf{filter}(\mathcal{A}[\phi_1,\ldots,\phi_k])}\right\}$$

where $E_{\mathsf{filter}(\mathcal{A}[\phi_1,\ldots,\phi_k])}$ is the set of transitions in the filtered and predicated CFA $\mathsf{filter}(\mathcal{A}[\phi_1, \ldots, \phi_k])$ *and where $s \models p_{\bar{e}}$ holds for a state s iff s has been entered through the edge $\bar{e}$.*

Observe that two different goals $p_{\bar{e}} \neq p_{\bar{e}'} \in G_e$ are mutually exclusive: While $\bar{e}$ and $\bar{e}'$ refine the same edge e in the CFA, they each require different evaluations of the predicates $\{\phi_1, \ldots, \phi_k\}$ such that no path can possibly satisfy $\bar{e}$ and $\bar{e}'$ simultaneously (not withstanding the application of GCS, the same path can pass through $\bar{e}$ and $\bar{e}'$ at *different times*).

Then, to solve $C = \mathtt{EDGES}(F, \{\phi_1, \ldots, \phi_k\})$ on a CFA $\mathcal{A}$, we use $\mathsf{GCS}(\mathcal{A}, \mathsf{edgegoals}[\mathcal{F}[\![F]\!], \{\phi_1, \ldots, \phi_k\}](\mathcal{A}))$ to construct a matching test suite (line 10).

4.3 Simple Path Coverage

A coverage specification $C = \texttt{PATHS}(F, B)$ yields the coverage criterion $\langle \mathsf{pathcov}[\mathcal{F}[\![F]\!], B, \emptyset]\rangle$ following the rules of Table 2 (dropping the empty set of predicates).

Analogously to edge coverage and the predicates p_e for each edge e, we introduce for each state sequence π to be covered a corresponding test goal p_π. To use GCS, we need to partition these goals into groups of mutually exclusive test goals: We put two test goals p_π and $p_{\pi'}$ into the same group G_e if π and π' end in the same transition e. Thus, all test goals $p_\pi \neq p_{\pi'} \in G_e$ are mutually exclusive—since a state is either reached through π or π' but not through both sequences at the same time.

Definition 19 (Path Goals). *Given a CFA $\mathcal{A}$ and a filter function* filter*, we define* $\mathsf{pathgoals}[\mathsf{filter}, B](\mathcal{A})$ *as the set of groups*

$$\mathsf{pathgoals}[\mathsf{filter}, B](\mathcal{A}) = \{G_e \mid e \in E_{\mathsf{filter}(\mathcal{A})}\}$$

with the state predicates

$$G_e = \{p_\pi \mid \pi \in \mathcal{L}_B(\mathsf{filter}(\mathcal{A})) \text{ and } \pi \text{ ends in } e\}$$

where $s \models p_\pi$ holds for a state s iff s has been entered through a path which followed the state sequence π immediately before entering s.

Hence, to solve a coverage specification $C = \texttt{PATHS}(F, B)$, we use $\mathsf{GCS}(\mathcal{A}, \mathsf{pathgoals}[\mathcal{F}[\![F]\!], B](\mathcal{A}))$, as shown in line 11.

5 Implementation and Experiments

We presented a first implementation of query-driven program testing in [11]. It has since been augmented with efficient algorithms for enumeration [1]. In the technical report version of the VMCAI paper [12] we also demonstrated the *algorithmic* feasibility of complex queries on several benchmarks, using a prototypical early version of FQL.

Our implementation, FShell, uses the code base of CBMC 3.2 [7], a bounded model checker with support for full ANSI C semantics. FShell features an interactive, shell-like, interface to state FQL queries and control commands. Furthermore, support for macros is provided to ease the use of complex queries. An integration with the Eclipse IDE is part of our ongoing work.

Currently, FShell can only be used on C programs with static CFAs, i.e., there is limited support for function calls by function pointers and no support for longjmp and setjmp. Behavior left undefined by the C standard is fixed in an arbitrary manner, since our approach requires a fully specified CFA to formulate test targets in terms of target graphs.

To show the viability of our approach we selected a set of C programs of different origin. We first picked some tools from the Unix coreutils implementation in Busybox 1.14[1], which has also be studied in [13]. Furthermore we generated test data for kbfiltr.c from Microsoft Windows DDK, experimental results for which had first been presented in [14]. In [15] model checking tools were applied to a manually cleaned up version of the Linux virtual file system layer. The provided code[2] includes an example use case, which was included in our analysis.

In addition to these well studied examples we applied our framework on two industrial case studies. First, we performed test case generation for an engine controller code generated from a MATLAB/Simulink model (matlab.c). Second, we examined a dynamic memory manager for airborne software systems (memman.c).

		Basic Block		Condition		BB $\times$ BB		
Source	LLOC	#goals	#tc	#goals	#tc	#goals	#tc	#inf
coreutils/cat.c	14	12	5	10	5	144	12	25
coreutils/echo.c	72	26	8	20	11	626	30	92
coreutils/nohup.c	20	12	5	12	5	144	15	24
coreutils/seq.c	21	26	7	20	9	626	25	116
coreutils/tee.c	45	20	2	15	4	362	9	74
kbfiltr.c	1764	225	61	204	64	49731	278	35773
pseudo-vfs.c	359	6	3	6	3	36	3	16
matlab.c	2098	37	6	34	6	1297	10	780
memman.c	127	52	3	42	4	2704	8	1270

Table 4. Summary of experimental results

The summary of our experiments is presented in Table 4. For each source we analyzed, we give the number of logical lines (LLOC, determined by counting the number of ";" occurring in the code). To compare to previous work, we first established basic block coverage for each of the programs. We give the number of test goals (#goals) and the number of test cases (#tc) that were necessary to cover these test goals. Given a loop bound, we compute test suites for 100% coverage of all feasible test goals. In [13] in many cases coverage of more than 90% is achieved, but there is no information about infeasibility of the remaining test goals.

Furthermore, we established condition coverage for each of the benchmarks and computed test suites for the query

```
> cover EDGES(@BASICBLOCKENTRY)->EDGES(@BASICBLOCKENTRY)
```

as discussed in Section 1 and abbreviated as BB $\times$ BB in Table 4. Naturally, many of the resulting test goals will be infeasible. We included these numbers in the column #inf.

[1] `http://www.busybox.net/`

[2] `http://research.nianet.org/~radu/VFS/`

The evaluation was performed on an Intel 2.53 GHz system equipped with 4 GB RAM. Apart from BB $\times$ BB for kbfiltr.c, which took 71 seconds, each analysis finished within at most 30 seconds and used no more than 450 MB of memory.

6 Related Work

Most existing formalisms for test specifications focus on the description of test data. Well established approaches like TTCN-3 [16] and UML TP [17, 18] may be applied both in white- and blackbox settings. None of them, however, allows to describe structural coverage criteria.

As our focus rests on structural code coverage, we only study related work on whitebox testing. In this context, generic coverage criteria, e.g., basic block coverage, condition coverage, and path coverage are well studied, cf. [19, 10], albeit with different names and with a notable lack of precise definitions. Attempts of formalizations using temporal logics [20], automata and graph based approaches [21] or using the Z notation [22] do not consider the specifics of the underlying programming language, which we encapsulate in filter functions.

Apart from the missing link to the programming language, these mathematical frameworks are not easily accessible by the working programmer. Such an easy-to-use query language has been built into the BLAST model checker [23], which was also used to generate basic block covering test suites for C code [14]. The BLAST query language [24, 25] is tailored towards verification and can describe test goals only by reachability properties. It does does not exhibit the necessary support for specification of complex test suites. Implementation-wise, the work of BLAST is closest related as we also use a model checker as backend. BLAST, however, is based on predicate abstraction, whereas CBMC implements SAT-based bounded model checking.

7 Conclusion

We have presented the syntax and semantics of FQL, a precise and expressive query language for the specification of formal coverage criteria. Using the concept of target graphs, we showed how to precisely specify coverage targets and how to combine them to powerful coverage criteria — generic as well as code-specific ones. Furthermore, by performing experiments on real world code using our tool FShell we exemplified the practical strength of the theoretical concepts underlying FQL.

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. Concerning the query solving backend, we are working on an alternative engine based on predicate abstraction and configurable program analysis [26].

References

1. Holzer, A., Tautschnig, M., Schallhart, C., Veith, H.: Query-Driven Program Testing. In: VMCAI. (2009) 151–166
2. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: SAT. (2003) 502–518
3. : Software Considerations in Airborne Systems and Equipment Certification (DO-178B). Radio Technical Commission for Aeronautics (1992)
4. : CoverageMeter 5.0.3. `http://www.coveragemeter.com/` (2009)
5. : CTC++ 6.5.3. `http://www.verifysoft.com/en.html` (2009)
6. : BullseyeCoverage 7.11.15. `http://www.bullseye.com/` (2009)
7. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: TACAS. (2004) 168–176
8. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
9. Ball, T.: A theory of predicate-complete test coverage and generation. In: FMCO. (2004) 1–22
10. Ntafos, S.C.: A comparison of some structural testing strategies. IEEE Trans. Software Eng. **14**(6) (1988) 868–874
11. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In: CAV. (2008) 209–213
12. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. Technical Report TUD-CS-2008-1013, Technische Universität Darmstadt (2008)
13. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. (2008) 209–224
14. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating Tests from Counterexamples. In: ICSE. (2004) 326–335
15. Galloway, A., Lüttgen, G., Mühlberg, J.T., Siminiceanu, R.: Model-checking the linux virtual file system. In: VMCAI. (2009) 74–88
16. Din, G.: TTCN-3. In: Model-Based Testing of Reactive Systems. (2004) 465–496
17. Dai, Z.R., Grabowski, J., Neukirchen, H., Pals, H.: From design to test with UML: Applied to a roaming algorithm for bluetooth devices. In: TestCom. (2004) 33–49
18. Schieferdecker, I., Dai, Z.R., Grabowski, J., Rennoch, A.: The UML 2.0 testing profile and its relation to TTCN-3. In: TestCom. (2003) 79–94
19. Huang, J.C.: An approach to program testing. ACM Comput. Surv. **7**(3) (1975) 113–128
20. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: A temporal logic based theory of test coverage and generation. In: TACAS. (2002) 327–341
21. Ammann, P., Offutt, J., Xu, W.: Coverage criteria for state based specifications. In: FORTEST. (2008) 118–156
22. Vilkomir, S.A., Bowen, J.P.: From MC/DC to RC/DC: Formalization and analysis of control-flow testing criteria. In: FORTEST. (2008) 240–270
23. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with BLAST. In: SPIN. (2003) 235–239
24. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The Blast Query Language for Software Verification. In: SAS. (2004) 2–18
25. Beyer, D., Noack, A., Lewerentz, C.: Simple and Efficient Relational Querying of Software Structures. In: WCRE. (2003) 216–225
26. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: ASE. (2008) 29–38