Query-Driven 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. We present a new approach to program testing which enables the programmer to specify test suites in terms of a versatile query language. Our query language subsumes standard coverage criteria ranging from simple basic block coverage all the way to predicate complete coverage and multiple condition coverage, but also facilitates on-the-fly requests for test suites specific to the code structure, to external requirements, or to ad hoc needs arising in program understanding/exploration. The query language is supported by a model checking backend which employs the CBMC framework. Our main algorithmic contribution is a method called *iterative constraint strengthening* which enables us to solve a query for an arbitrary coverage criterion by a single call to the model checker and a novel form of incremental SAT solving: Whenever the SAT solver finds a solution, our algorithm compares this solution against the coverage criterion, and strengthens the clause database with additional clauses which exclude redundant new solutions. We demonstrate the scalability of our approach and its ability to compute compact test suites with experiments involving device drivers, automotive controllers, and open source projects.

1 Introduction

In industrial software engineering, program testing is to remain the pivotal debugging and validation technology. While randomized and directed testing are important to achieve global assurance about software quality, and model-based testing helps to verify the conformance of the program with a high-level specification, there is practical need for a source code oriented white box testing methodology which assists the programmer in the software engineering cycle. Such a methodology should provide seamless support for code-driven testing, i.e., exploration of code under development, and requirement-driven testing for systematic quality assertion.

To address this need, we introduce a query language which combines easy navigation in real life C code with the ability to formulate complex coverage criteria. Providing straightforward queries for standard coverage criteria, our language FQL (FShell query language) aims to strike the right balance between expressiveness and simplicity. In FQL, a *program query* asks for a test suite following the general form

> **in** *prefix* **cover** *goals* **passing** *scope*

where the optional *prefix* directs the query to a specific program part, e.g., a source file or a function, *goals* describes the coverage criterion to be fulfilled by the test suite, and

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

an optional *scope* restricts test cases to pass through certain program paths only. For example, the query

```
> in /bla.c/cmp/ cover @blocks passing @call(err)
```

calls for a test suite which covers all blocks in function `cmp` of file `bla.c`, such that in all test cases, a call to `err` inside `cmp` is performed[1]. Other important and classical coverage criteria such as *predicate coverage*, *condition coverage*, *decision coverage*, *modified condition/decision coverage*, *predicate complete coverage* [1] etc. can also be expressed by natural queries in our language.

The added value of our language lies in the ability to define the query scope and the query goals in a quite flexible manner. Suppose for instance that in Listing 1 we want to cover (1) all calls to `cmp` and (2) all decisions inside `cmp`. This amounts to a coverage criterion which combines basic block coverage for (1) and decision coverage for (2). There are two different possibilities for this combination: either we want to cover the union of all call positions and all decisions, and write the query

```
> in /bla.c/ cover @call(cmp), cmp/@decisions
```

or we want to cover all possible *combinations* of calls to `cmp` and subsequent decisions inside `cmp`, i.e., the Cartesian product of the individual test goals:

```
> in /bla.c/ cover @call(cmp) -> cmp/@decisions
```

To ensure that, for each call site, each of the decisions is reached before the body of `cmp` is left, we write

```
> in /bla.c/ cover @call(cmp) -[@func(cmp)\@exit]> cmp/@decisions
```

Solutions to these queries are test suites. In case of Listing 1, these can be most easily described as sets of triples with input values for the variables x, y, z. For the first query, the singleton test suite $\{(1,0,1)\}$ covers all blocks and decisions in Listing 1; for the second and third case, possible solutions are $\{(1,0,1),(2,0,1),(1,0,2)\}$ and $\{(1,0,0),(0,1,0),(0,0,1)\}$ respectively. Note that the first test suite does not satisfy the second and third queries and the second suite does not satisfy the last query; the third solution, however, satisfies all three queries.

In Section 2 we build the mathematical foundation to formulate testing requirements as queries. In particular, we show how to state a query as a pair $\langle A, Q\rangle$ of automata over predicates. In this pair, the *observation automaton A* corresponds to the *scope* to be explored, and the *test goal automaton Q* specifies the *goals* to be covered. In Section 3, we provide an overview of our query language FQL and show how to translate FQL queries into such a pair $\langle A, Q\rangle$. Thus, the language reduces to a simple mathematical core in which we are able to formulate all relevant coverage criteria in a uniform way.

The second major contribution of this paper is an efficient query engine which integrates our theoretical framework, code instrumentation, bounded model checking, and SAT enumeration into a tool of high efficiency. Our query engine employs and adopts the software model checking framework of Kroening's CBMC [2]. Given a query $\langle A, Q\rangle$, our tool performs the following conceptual steps, cf. Figure 1:

[1] Expressions starting with "@", such as `@blocks`, typically denote sets of program locations, see Section 3.1 for a detailed explanation.

Listing 1. C source code of example `bla.c` with program counters

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

int main(int argc, char* argv[]) {
    int x, y, z, xy, yz, xz;
    ^{8}x = ^{7}input();
    ^{10}y = ^{9}input();
    ^{12}z = ^{11}input();
    ^{14}xy = ^{13}cmp(x, y);
    ^{16}yz = ^{15}cmp(y, z);
    ^{18}xz = ^{17}cmp(x, z);
    if ^{22}(^{19}xy == 1 && ^{20}yz == 1
        && ^{21}xz != 1)
ERROR: ^{23}err();
    ^{24}return 0;
}
```

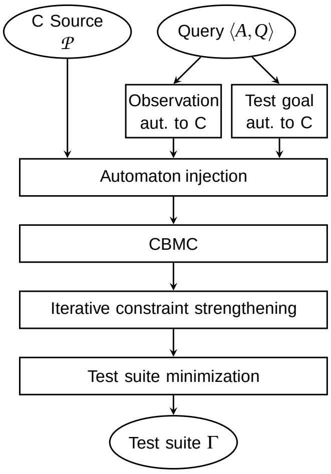

Fig. 1. Query processing

(1) We instrument the source code with monitors derived from the observation automaton A and the test goal automaton Q in such a way that the states of the monitors reflects the automata states.

(2) We use the code base of CBMC to obtain a SAT instance ϕ whose solutions correspond to the program paths π in the scope given by A. The instrumentation of step (1) guarantees that for each solution, we can easily determine which goals of Q are covered.

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

(4) To remove redundant test cases, we perform a test suite minimization. In our current implementation, we only do a simple post-processing; in future work, we plan to implement more aggressive minimization strategies. Note that the ICS enumeration in (3) involves nondeterministic choices which may give leverage to accelerate the algorithm with suitable heuristics.

Iterative Constraint Strengthening. A naive implementation of step (3) above would either use SAT enumeration to compute an enormous number of test cases until the test goals are reached, or it would call the SAT solver for each query goal anew. In *iterative constraint strengthening (ICS)*, we circumvent both problems by modifying the clause database of the SAT solver on-the-fly. Whenever the SAT solver halts to output a solution, we compare the test case obtained from this solution against the test goals. Then we add new clauses to the clause database in such a way that the next solution is guaranteed to satisfy at least one hitherto uncovered test goal. In this way, we exploit incremental SAT solving to quickly enumerate a test suite of high quality: Since we only add new clauses to the clause database, the SAT solver is able to reuse information learned in prior invocations. A similar strategy is used in *groupwise constraint strengthening (GCS)*, a further refinement of ICS. In GCS, we address coverage criteria such as multiple condition coverage or predicate complete coverage which have a nominally exponential number of test goals by partitioning these goals into a small number of groups characterized by a common compound goal.

We show that FSHELL has better practical performance than BLAST's test case generation facility [3]: On comparable hardware, our test suites are computed faster, and contain fewer test cases. Due to the minimization step, our results also improve on those reported in our previous tool paper [4].

Note that our choice of CBMC and bounded model checking as a query solving backend has advantages which come at a price: On the one hand, we achieve excellent performance and have the guarantee that the model-checker respects ANSI-C, which is important for low level code, our primary application area. On the other hand, a bounded model checking approach may be *unable to compute certain test cases* involving paths larger than the constant bound. It is easy to come up with examples where this situation will happen, but it is is detectable by CBMC and accounted for in our implementation; it has neither occurred in the experiments we did for comparison with BLAST, nor in our experiments based on real-life controller code. In future work, we plan to complement the CBMC backend with abstraction-based and randomized test case generation backends.

Related Work. Beyer et al. [3] use the C model checker BLAST [5] for test case generation, focusing on basic block coverage only. BLAST has a two level specification language [6]. On a low level they specify trace properties by observer automata written in a C-like manner. On a high level they relate these automata by reachability queries. In contrast to FSHELL, their language is tailored towards verification. Furthermore, BLAST is based on predicate abstraction whereas CBMC is a SAT-based bounded model checker. As our experiments show, we outperform BLAST regarding test case generation. Lee et al. [7,8] investigate test case generation with model checkers giving coverage criteria in temporal logics. Java PathFinder [9] and SAL2 [10] use model checkers for test case generation, but they do not support C semantics.

2 A Formal Testing Framework

Given a program $\mathcal{P}$, we consider the possibly infinite *transition system* $\mathcal{T} = \langle \mathcal{S}, \mathcal{R}, I \rangle$ induced by $\mathcal{P}$ which consists of the state space $\mathcal{S}$, a transition relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$, and a non-empty set of initial states $I \subseteq \mathcal{S}$.

Definition 1 (State Sequences and Paths). *Given a transition system $\mathcal{T} = \langle \mathcal{S}, \mathcal{R}, I \rangle$, a* state sequence *is a finite and non-empty word $\pi = \langle s_1, \ldots, s_n \rangle \in \mathcal{S}^+$ of states $s_i \in \mathcal{S}$. The sequence π is a* path, *if $\langle s_i, s_{i+1} \rangle \in \mathcal{R}$ holds for all $1 \le i < n$ and if $s_1 \in I$. For a state $s \in \mathcal{S}$, we write $s \in \pi$, iff $s = s_i$ holds for some $1 \le i \le n$, and we denote with $\Pi^{\mathcal{T}} \subseteq \mathcal{S}^+$ the* set of paths *of $\mathcal{T}$.*

We use *state predicates* to describe properties of individual program states and we use *path* and *path set predicates* in the description of individual test goals and coverage criteria.

Definition 2 (State, Path, & Path Set Predicates). *Given a transition system* $\mathcal{T} = \langle \mathcal{S}, \mathcal{R}, I \rangle$*, we define a* state predicate p *as a predicate on the state space* $\mathcal{S}$*, a* path predicate ϕ *as a predicate over the set* $\Pi^{\mathcal{T}}$*, and a* path set predicate Φ *as a predicate over the sets of paths* $2^{\Pi^{\mathcal{T}}}$*. We write* $s \models p$ *iff a state* $s \in \mathcal{S}$ *satisfies* p*,* $\pi \models \phi$ *iff a path* $\pi \in \Pi^{\mathcal{T}}$ *satisfies* ϕ*, and* $\Gamma \models \Phi$ *iff a path set* $\Gamma \subseteq \Pi^{\mathcal{T}}$ *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 $s \models p$, there exists a path $\pi \in \Pi^{\mathcal{T}}$ with $\pi \models \phi$, and there exists a path set $\Gamma \subseteq \Pi^{\mathcal{T}}$ 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 3 (Implicit Existential Quantification). *To evaluate a state predicate* p *over a path* π*, we implicitly interpret* p *to be existentially quantified, i.e.,* $\pi \models p$ *stands for* $\exists s \in \pi . s \models p$*. Analogously, a path predicate* ϕ *is existentially evaluated over a path set* Γ*, i.e.,* $\Gamma \models \phi$ *iff* $\exists \pi \in \Gamma . \pi \models \phi$*.*

Remark 1. Note that a path π can satisfy a state predicate p *and* its negation $\neg p$, if there exist two states $s, s' \in \pi$ with $s \models p$ and $s' \models \neg p$. Moreover, a state predicate p can also be interpreted over a path set Γ in the natural way, i.e., $\Gamma \models p$ iff $\exists \pi \in \Gamma . \exists s \in \pi . s \models p$.

Program Observations. We use sequences of state predicates (*traces*) to specify program paths. A trace matches a state sequence if each state in the sequence satisfies the corresponding predicate. A trace automaton is an automaton accepting traces; each trace in turn specifies a set of program paths.

Definition 4 (Traces and Trace Automata). *Let* P *be a finite set of state predicates and* $\mathcal{S}$ *be a state space. Then a* trace *is a finite non-empty word* $t = \langle t_1, \ldots, t_n \rangle \in P^+$*. A trace* matches *a state sequence* $\pi = \langle s_1, \ldots, s_n \rangle \in \mathcal{S}^+$ *(denoted with* $\pi \models t$*), iff* $s_i \models t_i$ *for all* $1 \leq i \leq n$*.*

A trace automaton *over* P *is a nondeterministic finite state automaton* A *accepting traces over the alphabet* P*. We write* $\mathcal{L}(A)$ *to denote the set of traces accepted by* A *and* $\mathrm{acc}(A)$ *to denote the set of accepting states of* A*. A trace automaton* A *over* P matches *a state sequence* π *(denoted with* $\pi \models A$*), iff there exists a trace* $t \in \mathcal{L}(A)$ *with* $\pi \models t$*.*

Remark 2. Although we have – for the sake of simplicity – defined trace automata as finite state automata, our framework naturally extends to other types of automata such as push-down automata for which we can construct C monitors, cf. Section 4.1.

We will use traces and trace automata as a natural tool for defining path predicates in the language FQL. In particular, we will employ trace automata for two distinct ends: First, as *observation automata* which *restrict the paths in* $\mathcal{T}$ to those required in a query; and second, as *test goal automata* which specify the individual test goals of a coverage criterion.

Definition 5 (Path Restriction by Observation Automata). *Let* $\mathcal{T}$ *be a transition system and* A *a trace automaton. Then we define the set of paths in* $\mathcal{T}$ *restricted by observation automaton* A *as* $\Pi^{\mathcal{T}}_A = \{\pi \in \Pi^{\mathcal{T}} \mid \pi \models A\}$*.*

Coverage Criteria. In the framework of this paper, we define a *test case* to be a single path in $\Pi_A^{\mathcal{T}}$ and a *test suite* as a subset of $\Pi_A^{\mathcal{T}}$. Correspondingly, a *coverage criterion* imposes a predicate on test suites:

Definition 6 (Test Case & Test Suite). *Let $\mathcal{T}$ be a transition system and let A be an observation automaton for $\mathcal{T}$. Then a* test case *for the set of paths $\Pi_A^{\mathcal{T}}$ is a single path $\pi \in \Pi_A^{\mathcal{T}}$ and a* test suite *Γ is a finite subset $\Gamma \subseteq \Pi_A^{\mathcal{T}}$ of the paths in $\Pi_A^{\mathcal{T}}$.*

Definition 7 (Coverage Criterion). *A* coverage criterion *Φ is a mapping from a transition system $\mathcal{T}$ and an observation automaton A to a path set predicate $\Phi_A^{\mathcal{T}}$ over $2^{\Pi^{\mathcal{T}}}$. We say that $\Gamma \subseteq \Pi_A^{\mathcal{T}}$ satisfies the coverage criterion Φ on $\mathcal{T}$ under the restriction A iff $\Gamma \models \Phi_A^{\mathcal{T}}$ holds.*

While our definition of coverage criteria is very general, most coverage criteria used in practice are based on lists of test goals which need to be satisfied. The test goals themselves are typically either state or path predicates. This prototypical setting is accounted for in the next definition.

Definition 8 ((State) Regular Coverage Criterion and Test Goals). *A* regular coverage criterion *Φ is a coverage criterion constructed in the following way:*

(i) There is a mapping $\Phi(\mathcal{T},A)$ which maps $\mathcal{T}$ and A to a list of test goals *$\Phi(\mathcal{T},A) = \{\Psi_1,\dots,\Psi_k\}$.*
(ii) This mapping induces the coverage criterion $\Phi_A^{\mathcal{T}}$ as follows:

$$\Gamma \models \Phi_A^{\mathcal{T}} \;\; \textit{iff} \;\; \bigwedge_{i=1}^{k} \Pi_A^{\mathcal{T}} \models \Psi_i \Rightarrow \Gamma \models \Psi_i$$

Intuitively, this amounts to the following coverage criterion: "For each test goal which is feasible in $\Pi_A^{\mathcal{T}}$, the test suite Γ must contain a concrete test case."
Φ is a state regular coverage criterion, *if $\Phi(\mathcal{T},A)$ contains only state predicates.*

As an example, consider basic block coverage $\mathsf{BB}_A^{\mathcal{T}}$, which is a state regular coverage criterion: induced by the test goals $\mathsf{BB}(\mathcal{T},A) = \{\mathsf{block}_1,\dots,\mathsf{block}_k\}$. Here k denotes the number of basic blocks in $\mathcal{T}$, and each predicate block_i holds true at the first statement of the i-th basic block in the program.

We will now define test goal automata which are used to specify the test goals needed in regular coverage criteria.

Definition 9 (Test Goal Automaton). *A* test goal automaton *Q is a trace automaton where each accepting state a gives rise to a test goal Ψ_a:*

$$\pi \models \Psi_a \;\; \textit{iff} \;\; \exists t.\pi \models t \textit{ and } Q \textit{ accepts } t \textit{ in state } a$$

Thus, the test goal Ψ_a requires a path matched by a trace which Q accepts in state a. The test goal automaton Q naturally induces a regular coverage criterion $\mathsf{cov}[Q]$ *based on the set* $\mathsf{cov}[Q](\mathcal{T},A) = \{\Psi_a \mid a \in \mathsf{acc}(Q)\}$ *of test goals.*

Note that a single path may match more than one test goal simultaneously: First, each path is matched by a number of different traces, and second, more than one accepting state may be reached through a trace in Q.

We conclude this section with a formal definition of program queries, as introduced in Section 1.

Definition 10 (Program Query & Result). *A* program query $\langle A, Q\rangle$ *consists of an observation automaton A and a test goal automaton Q. A valid* result *to the query* $\langle A, Q\rangle$ *on transition system* $\mathcal{T}$ *is a test suite* $\Gamma \subseteq \Pi_A^{\mathcal{T}}$ *with* $\Gamma \models \mathsf{cov}[Q]_A^{\mathcal{T}}$.

3 Syntax and Semantics of FQL

The FSHELL query language FQL facilitates the specification of test suites over C source code. To decouple the language from the algorithmic details of the query engine, and to provide leeway for different query solving backends, we designed FSHELL as a declarative language. FQL contains three layers which reflect the formal model of Section 2:

(i) state predicates over program variables and the program counter,
(ii) trace automata to express both observation automata and test goal automata, and
(iii) program queries to express coverage criteria.

In the following subsections, we will describe these layers along with examples referring to Listing 1. Due to length restrictions, the presentation of FQL is kept informal; we refer the reader to [11] for more details. Section 4 describes our query solving engine based on bounded model checking.

3.1 State Predicates

We have seen in Section 2 that sets of state predicates are at the center of our formal model. For instance, basic block coverage is induced by the set of test goals $\mathsf{BB}(\mathcal{T}, A) = \{\mathsf{block}_1, \ldots, \mathsf{block}_k\}$. FQL is therefore equipped with *predicate generators* to extract sets of predicates from the C source code, and to create new sets of predicates. For example, the predicate generator `@blocks` yields the set $\{\mathsf{block}_1, \ldots, \mathsf{block}_k\}$ of predicates. Note that each block_i has the form $\mathsf{pc} = \text{const}$ where pc is the program counter. Syntactically, all predicate generators are prefixed with "`@`". Semantically, a predicate generator either yields a set of predicates over the program counter pc, or a set of predicates over the program variables.

Many predicate generators are used to extract sets of predicates from the source code. Examples of such predicate generators include `@file(bla.c)` which captures all program counter values of statements in the source file `bla.c`, `@func(main)` which captures the statements in function `main`, `@line(3)` to capture the statements in line 3, `@call(cmp)` to match all function calls of `cmp`, and `@entry` as well as `@exit` which capture all function entry and exit points respectively. In case of Listing 1 we get, e.g., `@call(cmp)` $= \{\mathsf{pc} = 13, \mathsf{pc} = 15, \mathsf{pc} = 17\}$.

To introduce new predicates not present in the source code, we use the predicate generator `@new-pred(cond)`, where cond is an arbitrary side-effect free C expression. For example, `@new-pred(x <= 7)` generates a singleton set $\{x \leq 7\}$ of state predicates.

For certain coverage criteria such as MC/DC, we also need the predicate generator `@grouped-conditions` which generates a *set of sets*, where each inner set captures the program counter values of the individual predicates which constitute a decision. Returning to Listing 1, we have `@grouped-conditions` $= \{\{\mathsf{pc} = 2\}, \{\mathsf{pc} = 4\}, \{\mathsf{pc} = 19, \mathsf{pc} = 20, \mathsf{pc} = 21\}\}$. To support the succinct formulation of most relevant coverage criteria, FQL contains a rich variety of predicate generators and can be easily extended with further ones without conceptual changes to the language [11].

Operations on Sets of State Predicates. Given two sets A and B of state predicates, FQL provides the following set-theoretic operations:

$$\begin{array}{llll}
\text{(and)} & A\&B \equiv \{a \wedge b \mid a \in A, b \in B\} & A, B \equiv A \cup B & \text{(union)} \\
\text{(or)} & A|B \equiv \{a \vee b \mid a \in A, b \in B\} & A\backslash B \equiv A \setminus B & \text{(difference)} \\
\text{(negation)} & !A \equiv \{\neg a \mid a \in A\} & 2\hat{}A \equiv \{A'|A' \subseteq A\} & \text{(powerset)}
\end{array}$$

We add $\texttt{big-and}(A) \equiv \bigwedge_{a \in A} a$ and $\texttt{big-or}(A) \equiv \bigvee_{a \in A} a$ to describe the conjunction and disjunction of all elements of a set of state predicates. To apply an operation to each element in a set, or to *each set in a set of sets*, we introduce the `set()` operator. Moreover, `union()` forms a single set from a set of sets. Given a set S of sets and an operation $o(s)$ on a set s of state predicates, we define:

$$\texttt{set}(o(s)\ \texttt{:}\ s\ \texttt{in}\ S) \equiv \{o(s) \mid s \in S\} \qquad \texttt{union}(S) \equiv \bigcup_{s \in S} s$$

Operations on Conditions. In describing coverage criteria, *conditions* occurring in the source code play a crucial role. A condition is an atomic expression which is possibly combined with other conditions using &&, ||, and ! to compute the decision involved in executing an **if**, **for**, **while**, **switch** or ?: statement. The generator `@pred-wo-loc()` extracts conditions from source code locations identified by program counter values. In addition, `@predicate()` and `@neg-predicate()` conjoin the extracted conditions with the corresponding predicate over the program counter. For example, let $C = \{\mathsf{pc} = 19, \mathsf{pc} = 20, \mathsf{pc} = 21\}$ be such a set, referring to Listing 1. Then we have

$$\begin{aligned}
\texttt{@pred-wo-loc}(C) &= \{xy = 1, yz = 1, xz \neq 1\} \\
\texttt{@predicate}(C) &= \{\mathsf{pc} = 22 \wedge xy = 1, \mathsf{pc} = 22 \wedge yz = 1, \mathsf{pc} = 22 \wedge xz \neq 1\} \\
\texttt{@neg-predicate}(C) &= \{\mathsf{pc} = 22 \wedge xy \neq 1, \mathsf{pc} = 22 \wedge yz \neq 1, \mathsf{pc} = 22 \wedge xz = 1\}
\end{aligned}$$

Note that $\mathsf{pc} = 22$ refers to the location of the decision inside which the conditions in C occur.

State Regular Coverage Criteria. Besides simple test goals such as `@blocks`, FQL can can also describe more complex coverage criteria. We illustrate this feature on the example of *multiple condition coverage*. Recall that multiple condition coverage requires a test suite to cover—for each decision—all Boolean combinations of all conditions occurring in the respective decision. The test goals are therefore given by

```
union( set(
 union( set( big-and(@predicate(I) & @neg-predicate(D\I)) : I in 2^D ) ) :
D in @grouped-conditions ) )
```

Hierarchical Navigation. In practical queries, the predicate generators `@file(bla.c)`, `@line(3)`, `@func(foo)`, as well as `@entry` and `@exit` occur quite frequently. We therefore allow the following abbreviations which facilitate hierarchical navigation in the source code:

`/bla.c/` = `@file(bla.c)`	`/bla.c/42` = `@file(bla.c) & @line(42)`
`foo/` = `@func(foo)`	`/bla.c/foo/` = `@file(bla.c) & @func(foo)`
`foo/^` = `@entry(foo)`	`foo/$` = `@exit(foo)`
`foo/`*SP* = `@func(foo) &` *SP*	`/bla.c/`*SP* = `@file(bla.c) &` *SP*

In the last line, *SP* is to be replaced by any state predicate expression. Note that FQL also supports macros for frequently used expressions such as complex coverage criteria. Due to space restrictions we do not describe the macro feature in detail.

3.2 Trace Automata

Recall that trace automata are used to define path predicates, and to act as both observation automata and test goal automata. By implicit existential quantification, every state predicate can also be viewed as a path predicate, and it is easy to construct the corresponding automaton. Moreover, a set of state predicates naturally gives rise to an automaton with one accepting state for each state predicate in the set. For example, `@blocks` corresponds to an automaton with $|$`@blocks`$|$ accepting states, one for each basic block. The following list exemplifies the most important automata theoretic operations of FQL which enable the user to manipulate and combine trace automata explicitly: Let A_1, A_2, A_3 be trace automata:

$$A_1\texttt{,}A_2 \equiv A_1 \cup A_2 \qquad \text{(union)}$$

$$A_1\texttt{->}A_2 \equiv A_1 \circ \texttt{true}^* \circ A_2 \qquad \text{(sequencing)}$$

$$A_1\texttt{-[}A_3\texttt{]>}A_2 \equiv A_1 \circ A_3^* \circ A_2 \qquad \text{(restricted sequencing)}$$

Consider for example `main/^->main/$` over Listing 1: the traces of this automaton will match those program executions which pass the exit of `main` (line 19). In contrast, `main/^-[@file(bla.c)\@label(ERROR)]>main/$` requires that between the entry and the exit of `main` only locations other than those labeled “ERROR” (line 18) are seen. Note that each of these operations corresponds to a specific automata theoretic construction. Due to the special role of accepting states in defining test goals, we cannot use the standard automata theoretic minimization techniques, cf. [11].

3.3 Program Queries

We are now ready to define the program queries introduced in Section 1. Let A and B be FQL expressions which can be interpreted as trace automata (i.e., either trace automata, or sets of predicates as explained in the previous section). Then **`cover`** Q **`passing`** A expresses the program query $\langle A, Q \rangle$ with the semantics given in Definition 10.

Recall from Section 1, that FQL queries can also have a prefix. This prefix restricts all state predicates to a certain program part, e.g., a certain file. It is easy to see that the prefix can be moved into A and Q. For example, a query such as

```
> in /bla.c/ cover @line(4),@call(cmp)
   passing @file(bla.c)\@call(not_implemented)
```

which states that both, line 4 and a function call to `cmp` in file `bla.c` must be covered without ever calling `not_implemented()`, is equivalent to

```
> cover /bla.c/4,/bla.c/@call(cmp)
   passing @file(bla.c)\@call(not_implemented)
```

4 Query Processing Algorithms

In this section we describe the query processing algorithms. We first outline how program source code and a query are mapped to a SAT instance, and then detail on iterative and groupwise constraint strengthening in Section 4.2.

4.1 Program Instrumentation and Interfacing with CBMC

Bounded model checkers such as CBMC reduce questions about program paths to Boolean constraints in conjunctive normal form (CNF) which are solved by standard SAT solvers. Our query solving algorithms ICS and GCS employ the functionality of CBMC to obtain SAT instances suitable for test case generation. Recall that on input of a program annotated with assertions, CBMC outputs a SAT instance whose solutions describe program paths leading to assertion violations. To make this functionality useful for test case generation, we first instrument the program with the observation automaton A such that the resulting program reaches a failing assertion in the course of an execution, iff this program execution is matched by A. We therefore implement A as a C function that *monitors* program execution. To this end, the program $\mathcal{P}$ is instrumented to contain a *logging* layer, which reports the matching predicates after each executed step to the monitor. Moreover, we inject the test goal automaton as a second monitor, which only keeps track of the states of the test goal automaton in a distinguished variable, but does not cause assertion violations. Then, using CBMC, the instrumented program is transformed into the CNF-formula $\phi[\pi \in \Pi_A^{\mathcal{T}}]$ which is satisfied by all program executions which reach an accepting state of A within a bounded number of steps. By construction, $\phi[\pi \in \Pi_A^{\mathcal{T}}]$ contains distinguished Boolean variables referring to the state of the query automaton Q; these variables can be used to express the individual test goals. Therefore, a constraint of the form $\phi[\pi \in \Pi_A^{\mathcal{T}}] \wedge \phi[a]$ will satisfy those program executions which (i) respect observation automaton A and (ii) satisfy test goal Ψ_a. *In the rest of this section, we will for simplicity write this constraint as* $\pi \in \Pi_A^{\mathcal{T}} \wedge \pi \models \Psi_a$, *and tacitly assume the translation to CBMC described above.*

4.2 Guided SAT Enumeration

To generate a test suite Γ for a transition system $\mathcal{T}$ matching the query $\langle A,Q\rangle$, i.e., to achieve $\Gamma \models \mathsf{cov}[Q]_A^{\mathcal{T}}$, we introduce *iterative constraint strengthening (ICS).* In ICS, we build a test suite Γ iteratively from a sequence of test suites $\Gamma_0 \subset \Gamma_1 \subset \cdots \subset \Gamma_m$ with $\Gamma_0 = \emptyset$ and $\Gamma_q = \{\pi_1,\ldots,\pi_q\}$ for $1 \leq q \leq m$. In the m-th iteration, we reach a fixpoint when no more new goals can be covered.

Algorithm Overview. In the q-th iteration we build the *path constraint* ICSPC_q (Equation (1)) and obtain the test case π_{q+1} as one of its solutions. Here, ICSPC_q describes those paths in $\Pi_A^{\mathcal{T}}$ which cover a hitherto uncovered test goal. If no such test goal exists any more, ICSPC_q becomes unsatisfiable. Having determined a new test case π_{q+1}, we build ICSPC_{q+1} and continue the procedure with the $(q+1)$-st iteration until we reach an iteration m where ICSPC_m becomes unsatisfiable.

In order to fit the framework of *incremental SAT solving* (cf. [12]), we rewrite ICSPC_q (Equation (2)) in such a way that we are able to describe ICSPC_{q+1} incrementally in terms of ICSPC_q by *only adding* new constraints *without removing or changing* previously added constraints (Equation (3)). Using this incremental formulation of ICSPC_q, we describe iterative constraint strengthening (ICS) based upon an incremental SAT solver in Listing 2. The m paths finally collected by ICS constitute indeed a covering test suite (Theorem 1).

Path Constraints. The initial path constraint ICSPC_0 requires that a path is in $\Pi_A^{\mathcal{T}}$ and covers at least one of the test goals Ψ_a for $a \in \mathsf{acc}(Q)$. Subsequently, in ICSPC_q, we require the path to cover at least one test goal Ψ_a which remained *uncovered* by the test suite Γ_q. Since Γ_{q+1} must cover at least one more test goal than Γ_q, it suffices to *strengthen* the constraint ICSPC_q to obtain ICSPC_{q+1}. Below, we write $\mathsf{uncov}_q = \{a \in \mathsf{acc}(Q) \mid \Gamma_q \not\models \Psi_a\}$ for the set of accepting states which correspond to test goals not covered in Γ_q. Note that $\mathsf{uncov}_0 = \mathsf{acc}(Q)$ since $\Gamma_0 = \emptyset$ covers no test goals at all. Then, for $0 \leq q \leq m$, we search for a solution π_{q+1} to the q-th constraint

$$\mathsf{ICSPC}_q(\pi) := \pi \in \Pi_A^{\mathcal{T}} \wedge \bigvee_{a \in \mathsf{uncov}_q} \pi \models \Psi_a \tag{1}$$

Note that the empty disjunction is equivalent to false, i.e., if $\mathsf{uncov}_q = \emptyset$, then $\mathsf{ICSPC}_q \equiv \mathsf{false}$. Thus, ICSPC_q is satisfied by exactly those paths in $\Pi_A^{\mathcal{T}}$ which satisfy at least one *feasible* test goal still *uncovered* by Γ_q. If no such test goal exists, i.e., if Γ_q *achieves coverage,* then ICSPC_q is unsatisfiable.

Incremental Path Constraints. In incremental SAT solving, we use a single persistent clause database for consecutive solver invocations. When the SAT solver finds a solution, we add new clauses to the clause database, but do not remove any clauses. When the execution of the SAT solver is continued, the learned clauses obtained during earlier invocations remain valid and help to guide the search of the solver. Therefore, we have to construct ICSPC_{q+1} from ICSPC_q by only adding further constraints to the clause database. Observe that $\mathsf{uncov}_{q+1} \subset \mathsf{uncov}_q$ holds for $0 \leq q \leq m-1$. Thus in going from

ICSPC_q to ICSPC_{q+1}, we have to remove all test goals Ψ_a with $a \in \mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1}$ from the disjunction $\bigvee_{a\in\mathsf{uncov}_q} \pi \models \Psi_a$ occurring in Equation (1). To do so, we introduce a new Boolean variable S_a for each accepting state $a \in \mathsf{acc}(Q)$ and write ICSPC_q equisatisfiable as

$$\mathsf{ICSPC}_q(\pi) := \left[\pi \in \Pi_A^{\mathcal{T}} \wedge \bigvee_{a\in\mathsf{acc}(Q)} (S_a \wedge \pi \models \Psi_a)\right] \wedge \bigwedge_{a\notin\mathsf{uncov}_q} \neg S_a \tag{2}$$

Thus ICSPC_q consists of (a) an initial expression, shown above in square brackets, which remains unchanged throughout all iterations, and (b) a conjunction which is expanded from one iteration to the next. Adding $\neg S_a$ to the constraint renders the corresponding disjunct $S_a \wedge \pi \models \Psi_a$ unsatisfiable, and therefore only the disjuncts for $a \in \mathsf{uncov}_q$ remain enabled. Note that for ICSPC_0 we have $\mathtt{true} \equiv \bigwedge_{a\notin\mathsf{uncov}_0} \neg S_a$. Thus, in each iteration step, we use

$$\mathsf{ICSPC}_{q+1}(\pi) := \mathsf{ICSPC}_q(\pi) \wedge \bigwedge_{a\in\mathsf{uncov}_q\setminus\mathsf{uncov}_{q+1}} \neg S_a \tag{3}$$

to obtain ICSPC_{q+1} from ICSPC_q. Since we only add further constraints conjunctively, this approach fits the requirements of incremental SAT solving.

Iterative Constraint Strengthening. In our presentation of the algorithm, we assume a SAT solver which supports the following methods: (a) *Adding constraints* with add(ϕ): The method takes an arbitrary constraint ϕ over variables from arbitrary finite domains. While we use such a general interface to simplify the presentation of our algorithm, our implementation is based upon the SAT instance $\phi[\pi \in \Pi_A^{\mathcal{T}}]$ which we described in Section 4.1. (b) *Checking for satisfiability* with satisfiable(): The method returns true iff there exists a solution to the constraints added to the clause database so far. If a call to satisfiable() returns `true`, a witness is cached. (c) *Obtaining a solution* with solution(): The method returns the last witness cached in a call to satisfiable().

Listing 2. Iterative Constraint Strengthening (ICS)

```
1  func ICS($\Pi_A^{\mathcal{T}}$, $\langle A, Q\rangle$)
2  begin
3    $q$ := 0;  $\Gamma_0$ := $\emptyset$; $\mathsf{uncov}_0$ := $\mathsf{acc}(Q)$;
4    add($\pi \in \Pi_A^{\mathcal{T}} \wedge \bigvee_{a\in\mathsf{acc}(Q)} (S_a \wedge \pi \models \Psi_a)$);
5    while  satisfiable ()  do begin
6      $\pi_{q+1}$ := solution();
7      $\Gamma_{q+1}$ := $\Gamma_q \cup \{\pi_{q+1}\}$; $\mathsf{uncov}_{q+1}$ := $\emptyset$;
8      forall  $a \in \mathsf{uncov}_q$ do
9        if  $\pi_{q+1} \models \Psi_a$ then add($\neg S_a$);
10       else  $\mathsf{uncov}_{q+1}$ := $\mathsf{uncov}_{q+1} \cup \{a\}$;
11     $q$ := $q+1$;
12   end;
13   return $\Gamma_q$;
14 end;
```

The resulting procedure ICS is shown in Listing 2. In line 3 we initialize the iteration counter q, the first test suite Γ_0, and the set of test goals uncov_0 uncovered by Γ_0. Then in line 4, we add the initial expression from Equation (2) and start the search for the first solution in line 5. If a solution is found, it is obtained from the solver, assigned to π_{q+1}, and added to Γ_{q+1}. Then, after initializing uncov_{q+1}, we update the clause database following Equation (3) and fill the set uncov_{q+1} in lines 8 to 10: For each yet uncovered state $a \in \mathsf{uncov}_q$, we check whether π_{q+1} satisfies Ψ_a. If this is the case, $a \in \mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1}$ holds, and thus we add $\neg S_a$ in line 9. Otherwise a remains uncovered by Γ_{q+1} and hence we add a to uncov_{q+1} in line 10. Once no further solution is found in line 5, the accumulated suite Γ_q is returned.

Theorem 1 (Correctness of Iterative Constraint Strengthening). *The test suite* Γ *returned by the algorithm* ICS($\Pi_A^{\mathcal{T}}$,$\langle A, Q\rangle$) in Listing 2 satisfies $\Gamma \models \mathsf{cov}[Q]_A^{\mathcal{T}}$.

Remark 3 (Nondeterminism in Choosing π_{q+1}). Our algorithm leaves the particular choice of π_{q+1} open to the underlying SAT solver (line 6). Potential optimizations could control this choice to minimize the number of test cases necessary to obtain coverage.

Groupwise Constraint Strengthening. Certain regular coverage criteria, such as predicate complete or multiple condition coverage, require an *exponential number of test goals.* For example, recall that multiple condition coverage (Section 3.1) has one test goal for each basic block and *each possible evaluation of all conditions* involved in deciding which edge to choose in leaving the basic block. Hence, the number of test goals is exponential in the number of conditions in each decision. For this reason, the disjunction in ICSPC_0 will be of exponential size—thus rendering iterative constraint strengthening hard for such coverage criteria.

To mitigate this situation, we introduce *groupwise constraint strengthening (GCS)* as an optimization of iterative constraint strengthening. GCS can be combined with ICS and allows to handle all test goals which are state predicates. Let us thus for simplicity assume that all test goals Ψ_a for $a \in \mathsf{acc}(Q)$ are state predicates. To apply GCS, we require the test goals to be partitioned into k distinct *groups* $G_i = \{\Psi_i^1, \ldots, \Psi_i^{k_i}\}$ *of mutually exclusive test goals* for $1 \leq i \leq k$, i.e., we require that there exists no *state* s with $s \models \Psi_i^g$ and $s \models \Psi_i^h$ for all $1 \leq g \neq h \leq k_i$ and $1 \leq i \leq k$.

In the GCS algorithm, we avoid the construction of the initial and very large disjunction $\bigvee_{a \in \mathsf{uncov}_q} \pi \models \Psi_a$, as it appears in ICSPC_q (Equations (1) and (2)): Instead of individual test goals, we use a small number of *compound test goals* comp_i, where each compound test goal represents the goals of the whole group $G_i = \{\Psi_i^1, \ldots, \Psi_i^{k_i}\}$ of individual test goals Ψ_i^j. To represent group G_i, its compound test goal comp_i has to be semantically equivalent (but usually not identical) to $\bigvee_{j=1}^{k_i} \Psi_i^j$. It is important to note however that in many practical cases, comp_i can be *formulated much more succinctly* than $\bigvee_{j=1}^{k_i} \Psi_i^j$. For example, in case of multiple condition coverage, we partition the goals into groups according to the blocks they relate to. Then, $s \models \mathsf{comp}_i$ holds for a state s iff s visits the i-th basic block, i.e., comp_i has the form $\mathsf{pc} = \mathrm{const}$.

Starting with the compound test goal comp_i, we add for each covered test goal Ψ_i^j of group G_i, i.e., for each $\Psi_i^j \in G_i \setminus \mathsf{uncov}_q$, its negation $\neg\Psi_i^j$ to the corresponding compound test goal. This approach yields for each group G_i an *aggregate test goal*

$$\mathsf{aggr}_i^q := \mathsf{comp}_i \wedge \bigwedge_{\Psi_i^j \in G_i \setminus \mathsf{uncov}_q} \neg\Psi_i^j \tag{4}$$

Since we use aggr_i^q to represent the remaining uncovered test goals $G_i \cap \mathsf{uncov}_q$ of the group G_i in iteration q, we will rely on the equivalence

$$\mathsf{aggr}_i^q \equiv \bigvee_{\Psi_i^j \in G_i \cap \mathsf{uncov}_q} \Psi_i^j \tag{5}$$

which follows from the construction and the mutual exclusiveness of the test goals within each group G_i. Written in the form of Equation (5), aggr_i^q does not explicitly refer to any infeasible test goals and only involves *feasible* test goals as subexpressions. This significantly reduces the size of the constructed constraint.

Having defined aggr_i^q in this way, GCS proceeds like ICS but with Equation (1) replaced by

$$\mathsf{GCSPC}_q(\pi) := \pi \in \Pi_A^{\mathcal{T}} \wedge \bigvee_{i=1}^{k} \pi \models \mathsf{aggr}_i^q \tag{6}$$

Similar to ICS we also adopt GCSPC_q to fit incremental SAT solving: More precisely, we leave the overall constraint (Equation (6)) unchanged and replace aggr_i^q (Equation (4)) by an equisatisfiable and incrementally expandable expression. Thus, we can incrementally strengthen aggr_i^q for each group individually.

The effectiveness of GCS as an optimization of ICS relies on three conditions: (a) The overall number of groups must be small, since we maintain for each group G_i a constraint aggr_i^q. (b) The compound test goal comp_i must be available in a succinct formulation. (c) The fraction of *feasible* test goals Ψ_i^j in each group G_i must be small, since the negation of each feasible test goal is added to aggr_i^q in some iteration q. Conditions (a) and (b) hold for important coverage criteria such as multiple decision or predicates complete coverage. If condition (c) does not hold, then the number of required test cases will be large – but this is inherent in the coverage criterion and not an artefact of GCS.

Remark 4 (Mutual Exclusiveness: State vs. Path Predicates). It is tempting to assume that the mutual exclusiveness defined in terms of states is easily generalized to the level of path predicates. However, this is not the case as mutually exclusive state predicates *do not result* in mutually exclusive path predicates because of their implicit existential quantification, cf. Definition 3 and Remark 1.

5 Experimental Results

In our experiments we investigated test case generation for basic block (BB) and condition coverage (CC). We performed our experiments on a 3.0 GHz AMD64 system with 8 GB RAM. The table below summarizes our results with respect to BLAST. The column “Min” shows the number of test cases removed by our test suite minimization algorithm. Our current implementation of FSHELL is an optimized version of that presented in [4]. It generates fewer test cases, and, after test case generation for basic block coverage, FSHELL minimizes an obtained test suite. The results for BLAST are taken literally from [3], because the version of BLAST performing test case generation is currently unavailable. Beyer et al. performed their experiments on a 3.06 GHz Dell Precision 650 with 4 GB RAM. FSHELL outperforms BLAST, as we achieve coverage with fewer test cases faster. Besides the experiments on the device drivers from BLAST we conducted experiments on an engine controller (`matlab.c`) provided by an industrial collaborator from the automotive industries. It is generated from a MATLAB/Simulink model. Furthermore, we ran our tool on preprocessed sources (`autopilot.i`) generated from

		BLAST (BB)		BB			CC	
Source file	LLOC	#cases	Time[s]	#cases	Time[s]	Min*	#cases	Time[s]
kbfiltr.i	4879	39	300	26	18	6	98	24
floppy.i	6435	111	1500	63	1041	10	175	1259
cdaudio.i	8022	85	1500	71	1240	7	161	1243
parport.i	20698	213	5460	134	1859	21	351	2915
parclass.i	45283	219	2520	156	1324	16	392	2070
matlab.c	2033	-	-	5	30	1	16	31
autopilot.i	3141	-	-	206	894	14	450	1358

source code in PapaBench[2]. The results show that FSHELL scales well when moving from basic block coverage to condition coverage. Experiments concerning more sources and more complex queries can be found in [11].

6 Conclusion

In this paper, we introduced a query language for test case specification together with a query solving backend based on bounded model checking. Our backend is based on two new algorithms which guide the SAT solver to efficiently enumerate a test suite. Our implementation FSHELL demonstrates the effectiveness and versatility of our approach.

References

1. Ball, T.: A theory of predicate-complete test coverage and generation. In: FMCO. (2004) 1–22
2. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: TACAS. (2004) 168–176
3. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating Tests from Counterexamples. In: ICSE. (2004) 326–335
4. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In: CAV. (2008) 209–213
5. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with BLAST. In: SPIN. (2003) 235–239
6. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The Blast Query Language for Software Verification. In: SAS. (2004) 2–18
7. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: A temporal logic based theory of test coverage and generation. In: TACAS. (2002) 327–341
8. Tan, L., Sokolsky, O., Lee, I.: Specification-based testing with linear temporal logic. In: IRI. (2004) 493–498
9. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: ISSTA. (2004) 97–107
10. Hamon, G., de Moura, L.M., Rushby, J.M.: Generating Efficient Test Sets with a Model Checker. In: SEFM. (2004) 261–270
11. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. Technical Report TUD-CS-2008-1013, TU Darmstadt (2008)
12. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: SAT. (2003) 502–518

[2] `http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97`