Perspectives on White-Box Testing: Coverage, Concurrency, and Concolic Execution

Azadeh Farzan
University of Toronto

Andreas Holzer and Helmut Veith
Vienna University of Technology

***Abstract*—The last years have seen a fruitful exchange of ideas between automated software verification and white-box software testing; the industrial impact of concolic testing for sequential software is the most notable result of this interdisciplinary effort. While concolic testing is very successful at finding bugs, and even achieves verification in the limit, it is often hard to quantify the progress it achieves towards verification. In this paper, we survey two recent projects which aim to remedy this situation: In the FQL project, we devise a test specification language which facilitates precise specification of coverage criteria, and a separation of concerns between test specification and test case generation. In con2colic testing, we develop a concolic testing methodology for concurrent programs where progress is measured in terms of the data flow between program threads.**

I. Introduction

The success of concolic (concrete and symbolic) testing has marked a paradigm shift in white box software testing [1], [2]. Most visibly, Microsoft's internal productivity tool SAGE has set new standards for systematic white box testing of complex software applications on large compute clusters, and has helped the company to catch numerous bugs in software for the mass market [3]. Concolic testing hits a sweet spot between simple semi-syntactic program coverage (e.g. execution of all basic blocks in the program) and full verification: Termination of the concolic test space exploration engine without an assertion violation is tantamount to verification by a model checker, i.e., "*concolic testing is complete in the limit*". While completeness in the limit ensures that important corner cases are not missed, and is thus a highly attractive goal, it comes with two caveats: first, completeness can be stated only relative to the power of the underlying logical reasoning engine over basic program blocks, and second, concolic exploration is usually too expensive to terminate in practice. Thus, concolic testing has three possible outcomes which give different levels of assurance:

(A) *Bug Catching:* If the error location is reached, the tool reports an error, and provides an error trace along with program input that leads to the error.

(B) *Verification:* If concolic testing terminates without reaching the error location, it has proven all syntactic paths to the error location to be semantically impossible (modulo the limitations of the logic solver).

(C) *Partial Coverage:* If concolic testing does *not* terminate within reasonable time and computation resources, we will only know that significant effort was invested into program exploration, but not much is known beyond a quantification of this effort, e.g., the number of paths investigated.

While it is evident that (B) is highly preferable to (C), the dependency on the logical reasoning engine and the expensive concolic exploration make (B) hard to achieve in many cases. Of course, this is not a surprise: concolic testing was conceived for (A) – bug catching – in situations where verification is impossible, and one cannot expect it to provide (B) – verification assurance – through the back door. The challenge thus is to improve (C), as to give the user more feedback and more control over the exploration:

I The user should be able to define the trade-off between the investment of time and computing power and the coverage achieved.

II The exploration algorithm should be able to report progress towards path coverage, and support user control of the exploration process through a specification.

Our work in the last years has addressed this challenge in two orthogonal projects that we discuss in the following sections: In the first project (described in Section II), we have developed FQL, an expressive and precise specification language for coverage criteria for sequential programs. FQL facilitates the rigorous specification of test goals, and enables the user to control the coverage required for individual program parts. Thus, the FQL project is addressing challenge I. By specifying weak and local coverage criteria which can actually be achieved in practice, it indirectly also facilitates II.

978-1-4799-7125-1/15/$31.00 ©2015 IEEE

In the second project (described in Section III), we addressed challenge II directly for concurrent programs. While concolic testing for sequential software is highly developed and quite well understood, the concolic test space exploration strategies for concurrent software are a challenging research topic of current interest. We developed con2colic (concurrent concrete symbolic) testing, an extension of concolic testing to concurrent programs. Con2colic is complete in the limit in the spirit of concolic testing: It combines systematic concolic exploration of program paths with a systematic exploration of *interference scenarios*, i.e., dataflow communication patterns between the threads. The combined two-dimensional exploration makes sure that the complexity of the interference scenarios is gradually increasing; thus, if the con2colic testing tool ConCrest is manually terminated, it will report on the complexity of the thread communication in the scenarios explored so far.

In future work, we plan to bring these projects together. Our vision is to have a version of FQL for concurrent programs which enables to target both specific communication complexity and specific coverage for individual program parts in a combined tool which measures exploration process in a precise manner.

II. Query-Driven Program Testing using FQL

Query-driven program testing is a novel approach to white-box testing. Inspired by the success of databases (where the query language and the database engine are independent of each other), we propose a clear separation of concerns between test specification and test generation:

- A specification language provides an intuitive and computer-readable description of coverage criteria and test goals
- A backend for test-case generation computes the program inputs which achieve the specified coverage.

To describe the coverage criteria, i.e., sets of test goals, in a declarative way, we developed the coverage specification language FQL (FShell Query Language) [4], [5]. Together with the source code of the program under test, an FQL query is given to a test generation backend which then generates a covering test suite (cf. Figure 1). A straightforward possibility to implement test generation for query-based program testing is the use of model checkers which can generate test inputs as witnesses for reachability specifications (or, equivalently, as counterexamples for safety properties) [6], [7]. While this use of model checkers for testing yields a theoretically sound test-generation procedure, it scales poorly for computing complex test suites for large sets of test goals, because each test goal requires an expensive run of the model checker. To mitigate this situation, we developed different test generation backends that enable efficient test generation based on model-checking technology (see Sections II-B and II-C).

A. A Brief Introduction to FQL

The language design of FQL has several layers (cf. Figure 2) but is essentially based on finite automata and regular languages. Below we will discuss each of these layers separately. We begin with a discussion of how test goals and coverage criteria can be formalized as automata and regular expressions. Then, we will discuss how FQL achieves programming language and source code independence in specifications of coverage criteria. Finally, we will provide some example specifications for commonly used coverage criteria.

1) Specification of Coverage Criteria by Automata: An individual test goal can be naturally specified by a regular expression. A test criterion, however, is a set of test goals, i.e., a set of regular expressions. *Thus, to describe test criteria, we will consider automata whose edges carry regular expressions, i.e., automata whose alphabet contains regular expressions.*

To illustrate this idea, let us begin with the simple coverage criterion *Line-10 coverage*. A test suite that covers a given program with respect to this coverage criterion has to contain a test case which leads to the execution of line 10. We can formalize this coverage criterion in a natural way by describing the required program executions by regular expressions. For example, using the pattern `_*.Line 10._*` we can describe all (terminating) program executions which eventually reach

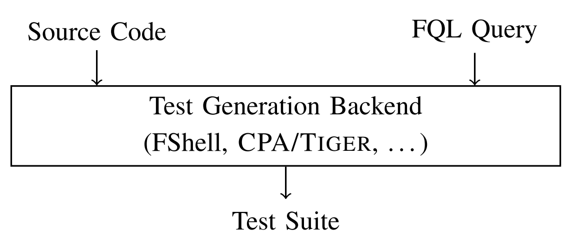

Fig. 1. Query-Driven Program Testing

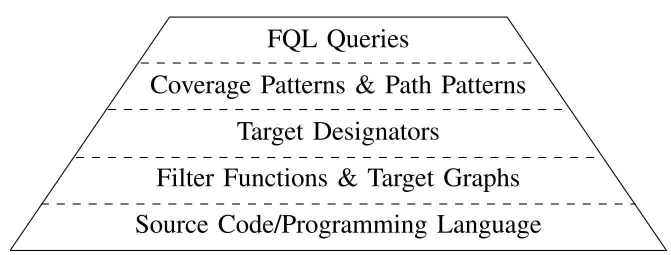

Fig. 2. FQL's Language Levels (Target Graphs and Target Designators are discussed in [4] and omitted in this survey paper.)

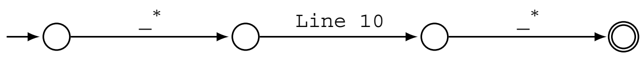

Fig. 3. Finite automaton expressing coverage criterion "Cover Line 10"

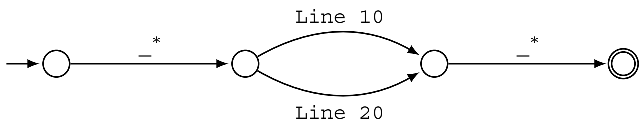

Fig. 4. Finite automaton expressing coverage criterion "Cover Line 10 *and* Line 20"

the code at line 10. Here, the expression `_*` denotes any finite sequence of program statements. Figure 3 shows an automaton which accepts this pattern. Observe that the expression `_*` *is part of the alphabet of the automaton*. That means that the language of the automaton is the singleton set { `_*.Line 10._*` }. Each word (i.e., pattern) in the language of the automaton represents a *test goal* and we require a matching test case for each pattern.

If we want a test suite that covers not only line 10 but also line 20, then we extend the automaton as depicted in Figure 4. The language of that automaton is { `_*.Line 10._*` , `_*.Line 20._*` } and for *each* of the two patterns in that language we require a matching program execution. Depending on the program under test there might be a single program execution covering both test goals or it might require two different program executions.

Our automaton concept allows us to describe coverage criteria in a flexible and simple way:

a) Alternatives: The automaton in Figure 5 again has only a singleton language which consists of the pattern `_*.(Line 10 + Line 20)._*` Thus, the coverage criterion requires only one program execution which covers line 10 *or* line 20; in contrast, the automaton in Figure 4 requires *both* lines as test goals that can be reached by one or two executions.

b) Sequences: Using finite automata labeled with regular expressions we can also encode more complex coverage

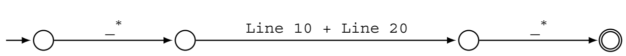

Fig. 5. Finite automaton expressing coverage criterion "Cover Line 10 *or* Line 20"

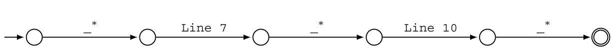

Fig. 6. Finite automaton expressing coverage criterion "Cover Line 7 *followed by* Line 10"

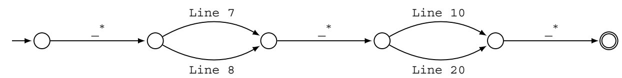

Fig. 7. Finite automaton expressing coverage criterion "*Cartesian product* of lines 7 and 8 with lines 10 and 20"

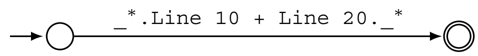

Fig. 8. Alternative automaton expressing coverage criterion "Cover Line-10-*or*-Line-20"

criteria. For instance, the automaton in Figure 6 requires a program execution which first executes line 7 and then line 10. This is expressed by the pattern `_*.Line 7._*.Line 10._*`

c) Cartesian Product: Finally, consider the automaton in Figure 7. The language of this automaton is the set of patterns { `_*.Line 7._*.Line 10._*`, `_*.Line 7._*.Line 20._*`, `_*.Line 8._*.Line 10._*`, `_*.Line 8._*.Line 20._*` }. This coverage criterion corresponds to the Cartesian product of lines 7 and 8 with lines 10 and 20, i.e., 4 test goals.

2) Specification of Coverage Criteria by Quoted Regular Expressions: Above we have presented an automata-based way to express coverage criteria by sets of path patterns where the edges of the automaton carry regular expressions. In FQL, we can express the automaton itself as a regular expression. To distinguish between the automaton whose words are test goals and the regular expressions on the egdes (i.e., in the alphabet), we write the latter in *quotes*. Thus, FQL expressions are regular expressions over quoted regular expressions. In our terminology, we say that an FQL expression is a *coverage pattern* which contains quoted *path patterns.*

For example the coverage pattern `"_*".(Line 10 + Line 20)."_*"` results in the set of path patterns { `_*.Line 10._*`, `_*.Line 20._*` }, i.e. Line-10-*and*-20-coverage in the spirit discussed above (cf. Figure 4). In contrast, the coverage pattern `"_*"."Line 10 + Line 20"."_*"` results in the singleton set { `_*.(Line 10 + Line 20)._*` } and therefore denotes Line-10-*or*-20-coverage as in Figure 5. Observe that the coverage pattern `"_*.(Line 10 + Line 20)._*"` represents Line-10-*or*-20-coverage, too, but the resulting automaton is different (cf. Figure 8). Thus, the same coverage criterion can be expressed in FQL in several ways.

Table I contains the grammars for path and coverage patterns. For ease of presentation, we only give simplified versions of FQL's grammar and refer the interested reader to [4], [8]. In Table I, ε denotes the empty word, Σ denotes

TABLE I. (SIMPLIFIED) GRAMMARS FOR PATH PATTERNS P AND COVERAGE PATTERNS C

P	::=	$\varepsilon \mid \Sigma \mid P + P \mid P.P \mid P^* \mid (P)$
C	::=	$\varepsilon \mid \Sigma \mid$ "P" $\mid C + C \mid C.C \mid (C)$
Σ	::=	$\{\varphi\} \mid F$

φ: state predicate, F: filter function.

the alphabet, which we will discuss below, and $P + P$, $P.P$, and P^* denote *union*, *concatenation*, and *Kleene star* as in the case of regular languages. The grammar for coverage patterns differs from the grammar for path patterns in two ways: First, coverage patterns introduce the quotation operation "P", and second, coverage patterns have no Kleene star. *By disallowing the Kleene star, each coverage criterion expressible in FQL is limited to a finite number of test goals.*

Quoted regular expressions were previously introduced in the literature by Afonin et al. [9] under the name *rational sets of regular languages (RSRLs)*. In query-driven program testing, RSRLs have a similar role for test specifications as relational algebra has for databases. In particular, a good understanding of set-theoretic operations is necessary for systematic algorithmic optimization and manipulation of test specifications. We therefore investigated basic properties of RSRLs, especially for the cases that are relevant to FQL in a theoretical paper [10].

3) Programming Language & Source Code Independence: The quoted regular expressions discussed above facilitate the succinct expression of multiple test goals in a single regular expression. We will now introduce language components of FQL which allow us to express the test goals *independently of the program at hand*. This will allow us to express generic coverage criteria such as basic block coverage using simple expressions, and combine them with program specific requirements, e.g., avoiding an unimplemented function. To this end, we will use *filter functions*, i.e., functions which generate regular expressions from the source code.

To illustrate filter functions, let us for simplicity consider a program with just two basic blocks that start in line 10 and 20 respectively. Then, the automaton depicted in Figure 4 or, equivalently, the expression `"_*".(Line 10 + Line 20)."_*"` describe basic block coverage for this particular program.

Instead of explicitly listing lines 10 and 20, we can use the FQL filter expressions `@BASICBLOCKENTRY` and `@ID` to refer to all basic blocks or all program statements respectively. Thus, we can write the above expression in a program independent way as `"@ID*".@BASICBLOCKENTRY."@ID*"`. Given a specific program, the FQL filter expressions are expanded to regular expressions over entities of the program. For our simple example program, `@BASICBLOCKENTRY` is expanded to the regular expression `(Line 10 + Line 20)`.

Expressions like `@BASICBLOCKENTRY` or `@ID` are called *filter functions* because they extract ("filter") syntactic entities from the source code and collect them in a regular expression. In addition to those discussed above, FQL provides plenty of other filter functions. For example, the filter function `@CONDITIONEDGE` yields all evaluations of conditions in the program so that we can describe condition coverage by `"@ID*".@CONDITIONEDGE."@ID*"`. We can also combine filter functions by Boolean connectives: for example, `"@ID*".(@BASICBLOCKENTRY & @FUNC(foo))."@ID*"` describes basic block coverage in function `foo`.

Beyond standard coverage criteria, FQL can also express more program specific coverage criteria: the filter function `@LABEL(L)` refers to a program statement that is annotated with code label `L` and `"@ID*".@LABEL(L1)."@ID*".@LABEL(L2)."@ID*"` requires a test case that first executes the code labeled with `L1` and then executes the code labeled with `L2`. FQL also allows the restriction of the program state using *state predicates*. For example, `@ID*.{x > 10}.@LABEL(L).@ID*` requires that code label `L` is reached at least once when variable `x` is greater than 10.

Syntactically, FQL queries start with the keyword `cover`. For example, the FQL query expressing basic block coverage would be written as `cover "@ID*".@BASICBLOCKENTRY."@ID*"`. By default, FQL assumes `"@ID*"` expressions as prefix and suffix of an FQL query, e.g., basic block coverage can be simply stated by the query `cover @BASICBLOCKENTRY`.

In Tables II-VI, we give example FQL queries to demonstrate FQL's broad applicability for structural coverage criteria (Table II), data-flow coverage criteria (Table III), constraining test cases (Table IV), customized test goals (Table V), and seamless transition to verification (Table VI).

Note that our treatment of FQL syntax and semantics in this survey is simplified and by far not complete; we refer the interested reader for a more complete introduction to FQL to [5] and for a complete reference of FQL to [4], [8].

B. FQL Backend I: Iterative Constraint Strengthening

FShell was the first test case generation tool for FQL, and is based on bounded model checking. FShell uses the concept of *iterative constraint strengthening* [15] to leverage the incremental SAT-solving capabilities of modern SAT-solvers to achieve an efficient test generation approach. Figure 9

TABLE II. SCENARIO 1: STRUCTURAL COVERAGE CRITERIA

The certification of critical software systems often requires coverage criteria such as basic block, condition or decision coverage [11] which refer to entities present in all source code. This results in our first queries:

[*"Standard Coverage Criteria"*] Basic block coverage and condition coverage:

Q1: `cover @BASICBLOCKENTRY`

Q2: `cover @CONDITIONEDGE`

Commercial coverage tools differ on the semantics of condition coverage. Therefore, it is important to be able to express competing criteria and we further state a variant of condition coverage as defined by CoverageMeter [12] and CTC++ [13]:

Q3: `cover @CONDITIONEDGE & @STMTTYPE(if,switch,for,while,?:)`

The &-operator expresses the intersection between two filter functions, i.e., in the above example, only condition evaluations in the `if`, `switch`, `for`, `while`, and `?:` statements are considered for condition coverage.

For intensive testing a developer will employ a variant of path coverage [14], but restrict it to local coverage due to high costs:

[*"Acyclic Path Coverage"*] Cover all acyclic paths through functions `main` and `insert`:

Q4: `cover PATHS(@FUNC(main) | @FUNC(insert), 1)`

[*"Loop-Bounded Path Coverage"*] Cover all paths through `main` and `insert` which pass each statement at most twice:

Q5: `cover PATHS(@FUNC(main) | @FUNC(insert), 2)`

Here, the |-operator denotes the union of two filter functions.

TABLE III. SCENARIO 2: DATA FLOW COVERAGE CRITERIA

We give three examples of typical data flow coverage criteria.

[*"Def Coverage"*] Cover all statements defining a variable `t`:

Q6: `cover @DEF(t)`

[*"Use Coverage"*] Cover all statements that use the variable `t` as right hand side value:

Q7: `cover @USE(t)`

[*"Def-Use Coverage"*] Cover all def-use pairs of variable `t`:

Q8: `cover @DEF(t)."NOT(@DEF(t))*".@USE(t)`

TABLE IV. SCENARIO 3: CONSTRAINING TEST CASES

During development and for code exploration, it is often important to achieve the desired coverage with test cases which, for instance, avoid a call to an unimplemented function. Below we list five examples of this group.

[*"Constrained Program Paths"*] Basic block coverage with test cases that satisfy the assertion $j > 0$ after executing line 10:

Q9:
```
cover @BASICBLOCKENTRY
passing ^(@LINE(10).{j>0}+NOT(@LINE(10)))*$
```

[*"Constrained Calling Context"*] Condition coverage in function `compare` with test cases which call `compare` from inside function `sort` only:

Q10:
```
cover @CONDITIONEDGE & @FUNC(compare)
passing ^(NOT(@CALL(compare))*
        .(@CALL(compare) & @FUNC(sort))*)*$
```

[*"Constrained Inputs"*] Basic block coverage in function `sort` with test cases that use a list with 2 to 15 elements:

Q11:
```
cover @ENTRY(sort).{len>=2}.{len<=15}.
      "NOT(@EXIT(sort))*".@BASICBLOCKENTRY
```

[*"Recursion Depth"*] Cover function `eval` with condition coverage and require each test case to perform three recursive calls of `eval`:

Q12:
```
in @FUNC(eval) cover @CONDITIONEDGE
passing @CALL(eval).NOT(@EXIT(eval))*
        .@CALL(eval).NOT(@EXIT(eval))*.@CALL(eval)
```

[*"Avoid Unfinished Code"*] Cover all calls to `sort` such that `sort` never calls `unfinished`. The function `unfinished` is allowed to be called outside `sort`—assuming that only the functionality of `unfinished` which is used by `sort` is not testable yet:

Q13:
```
cover @CALL(sort)
passing ^(NOT(@FUNC(sort))*.(@FUNC(sort) &
        NOT(@CALL(unfinished)))*.NOT(@FUNC(sort))*)*$
```

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

Q14:
```
cover @CONDITIONEDGE
passing ^(NOT(@CALL(eval))*.@CALL(insert))>=2
```

describes FShell's high-level architecture. FShell extends the bounded model checker CBMC and uses CBMC's functionality to produce a SAT-formula that represents an unrolling of the program under test. The `cover` part of an FQL query gets translated into an *observation automaton* which encodes all test goals, i.e., a set of path patterns. The `passing` part gets translated into a *test goal automaton* that encodes a single path pattern. Both automata are then encoded into the SAT formula that represents the program under test such that a satisfying assignment of the formula represents a program execution that satisfies at least one of the test goals as well as the `passing` clause. From the satisfying assignment corresponding inputs can be derived. Each time test inputs are computed, the SAT-formula is adapted such that, when the SAT-solver is reinvoked, test inputs for a yet uncovered test goal are generated until no further test goal can be covered. In order to achieve efficiency, FShell uses incremental constraint strengthening to exploit the incremental solving capabilities of modern SAT solvers. Each time a satisfying assignment is found, the SAT formula is further constrained such that the next invocation of the SAT solver returns a satisfying assignment that covers a yet uncovered test goal (if such an assignment exists). Since the formula is only constrained, the SAT solver can reuse all previously computed facts about the solution space of the formula. After finishing test generation, FShell performs a test suite minimization to eliminate redundant test cases.

C. FQL Backend II: Multi-goal Reachability Analysis

Our second test case generation backend CPA/TIGER is based on predicate-abstraction-based software model checking instead of bounded model checking. Figure 10 shows the overall structure of this approach. We represent each test goal as a single test-goal automaton and exploit relations between these automata in order to reuse existing reachability information for the analysis of subsequent test goals. Exploiting the sharing of sub-automata in a series of reachability

TABLE V. SCENARIO 4: CUSTOMIZED TEST GOALS

Complementary to the constraints on test *cases* of Scenario 3, we also want to modify the set of test *goals* to be achieved by the test cases.

[*"Restricted Scope of Analysis"*] Condition coverage in function `partition` with test cases that reach line 12 at least once:

Q15:
```
in @FUNC(partition) cover @CONDITIONEDGE passing @LINE(12)
```

[*"Condition/Decision Coverage"*] Condition/decision coverage (the union of condition and decision coverage) [11]:

Q16:
```
cover @CONDITIONEDGE + @DECISIONEDGE
```

To understand the interaction of two program parts, it is not sufficient to cover the *union* of the test goals induced by each part, but to cover their *Cartesian product*:

[*"Interaction Coverage"*] Cover all possible pairs between conditions in function `sort` and basic blocks in function `eval`, i.e., cover all possible interactions between `sort` and `eval`:

Q17:
```
cover (@CONDITIONEDGE & @FUNC(sort))
        ->(@BASICBLOCKENTRY & @FUNC(eval))
```

The operator `->` is shorthand for the expression `."@ID*".`

In a similar spirit, we can also approximate path coverage by covering pairs, triples, etc. of basic blocks:

[*"Cartesian Block Coverage"*] Cover all pairs, triples, and quadruples of basic blocks in function `partition`:

Q18:
```
cover @BASICBLOCKENTRY->@BASICBLOCKENTRY
```

Q19:
```
cover @BASICBLOCKENTRY->@BASICBLOCKENTRY->@BASICBLOCKENTRY
```

Q20:
```
cover @BASICBLOCKENTRY->@BASICBLOCKENTRY->@BASICBLOCKENTRY
        ->@BASICBLOCKENTRY
```

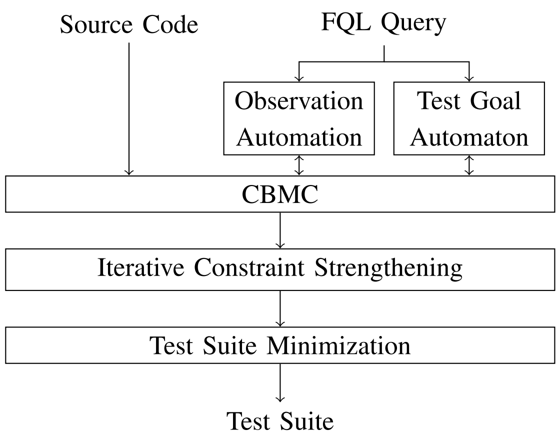

Fig. 9. FShell Architecture [15], [8]

queries, we achieve considerable performance improvements over the standard approach which reinvokes a model checker for each test goal from scratch. The theoretical foundation for the reuse of reachability information is the novel concept of *simulation relation modulo a transition set* which enables the identification of sharable information. Compared to FShell, which requires an a priori fixed coverage criterion, the new test generation backend is a further step towards the analogy of *programs as databases*: similar to databases, we can now *dynamically* query for reachability information in a declarative

TABLE VI. SCENARIO 5: SEAMLESS TRANSITION TO VERIFICATION

When full verification by model checking is not possible, testing can be used to approximate model checking. For instance, we can specify to cover all assertions.

[*"Assertion Coverage"*] Cover all assertions in the source:

Q21:
```
cover @STMTTYPE(assert)
```

[*"Assertion Pair Coverage"*] Cover each pair of assertions with a single test case passing both of them:

Q22:
```
cover @STMTTYPE(assert)->@STMTTYPE(assert)
```

We can finally use test specifications to provoke unintended program behavior, effectively turning a test case into a counterexample. In the following examples, we check the presence of an erroneous calling sequence and the violation of a postcondition:

[*"Error Provocation"*] Cover all basic blocks in `eval` without reaching label `init`:

Q23:
```
cover (@BASICBLOCKENTRY & @FUNC(eval))
passing ^NOT(@LABEL(init))*$
```

[*"Verification"*] Ask for test cases which enter function `main`, satisfy the precondition, and violate the postcondition:

Q24:
```
cover @ENTRY(main)
passing @ENTRY(main)
          .{precond()}.NOT(@EXIT(main))*.{!postcond()}
          .@EXIT(main)
```

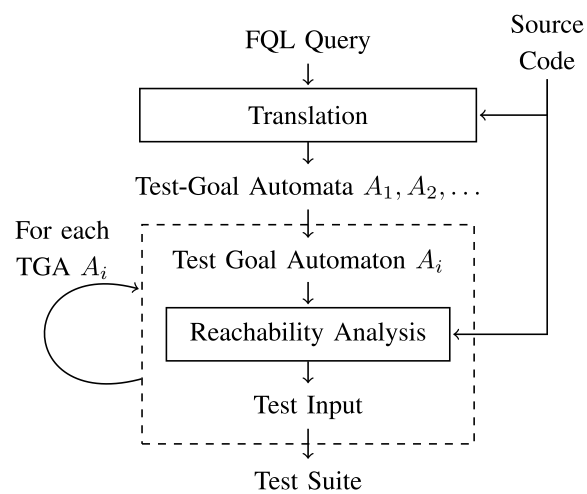

Fig. 10. Multi-goal Reachability Analysis [18]

and efficient way. Providing a dynamic querying mechanism enables us to apply our tools to applications beyond testing, for example, a test generator can then be used to improve the precision of static path-insensitive program analyses in a demand-driven way. We implemented this approach in the test-input generator CPA/Tiger [16]. Recently, CPA/Tiger was extended to support software-product line testing [17].

D. Seamless Testing for Models and Code

Developing the analogy with databases one step further, individual system requirements correspond to views on the behavior of the underlying system — similar to views on data stored in databases. But, in contrast to views on databases, these requirements might be inconsistent with the actual implemented system. *Functional testing* investigates the relationship between specifications and implementations by verifying that each feature described in the specification of a system is properly realized in the implementation. Usually *black-box testing* approaches are used to realize functional testing [19]. This means that one obtains test inputs by only considering the specification of a system without knowing the details of the concrete implementation. This is crucial to identify flaws in the design or missing functionality, which is something that *white-box testing* can not achieve. Both test-input generators FShell and CPA/TIGER are designed to support a white-box testing setting. To make FQL-based test generation also available for black-box testing, we developed a testing methodology [20] that combines black-box and white-box testing and, thereby, is able to infer additional information about the relationship between specification and code. We realized our approach in the context of model-based testing, i.e., parts of the specification are given as models. We show that we can express coverage criteria for models via FQL queries and exploit this to establish a seamless process for testing models and code. Our approach enables us to extend the use of FShell and CPA/TIGER to model-based testing.

Figure 11 describes our approach for model-level test case generation. Currently, we support test generation for UML activity diagrams. Given an activity diagram $\mathcal{M}$, a *generation query* Q_g specifies the coverage that we want to achieve at model level. For example, the query `cover PATHS(@ID, 1)` requires a test suite where all loops in the activity diagram $\mathcal{M}$ are traversed at most once. We translate $\mathcal{M}$ into a C program $\mathcal{M}'$ and the query Q_g into an FQL specification Q'_g. The query Q'_g describes a coverage criterion over the C program $\mathcal{M}'$. The translation is done such that we have a bijective relationship between elements in the model and the generated code. That means that when executing the generated program we can map the execution back to a trace in the model. We use either FShell or CPA/TIGER to generate a test suite that achieves on $\mathcal{M}'$ the coverage required by Q'_g. Then, we execute $\mathcal{M}'$ with the generated test suite and, thereby, obtain a set of executions that we map back to traces in $\mathcal{M}$. These traces constitute the UML-level test suite $\mathsf{S}_{\mathcal{M}}$. After inspecting $\mathsf{S}_{\mathcal{M}}$, the test engineer either releases the suite or adjusts Q_g to enhance the suite until achieving requirements coverage on the model.

III. CONCOLIC TESTING OF CONCURRENT PROGRAMS

In our research on test generation for concurrent programs we considered two approaches. The first approach (see Section III-A) takes a concurrent program and translates aspects

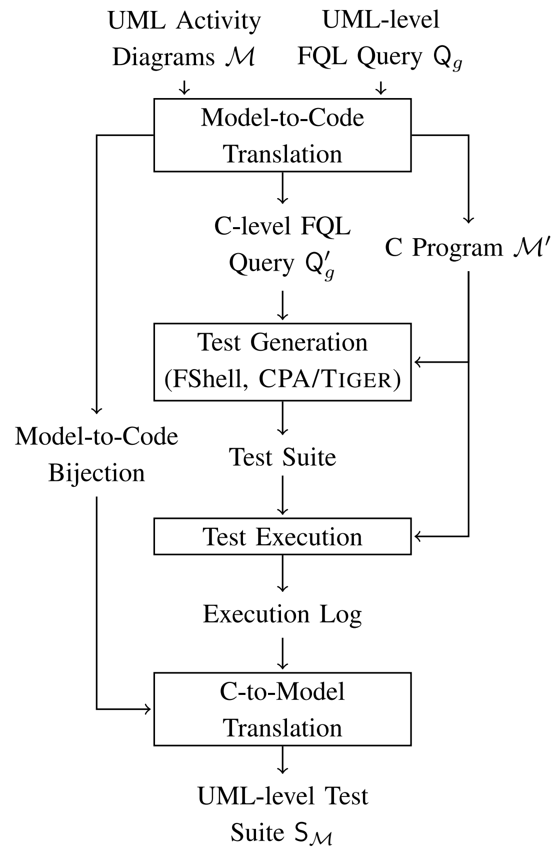

Fig. 11. Generation of Model-level Test Suite [18]

of its behavior into a sequential program. Then, standard test generation approaches for sequential programs, like concolic testing, can be used without any adaptions towards concurrency. In our second approach, we adapted concolic testing towards a concurrency setting (see Section III-B).

A. Bounded-Interference-based Sequentialization

In [21], we proposed an approach that is based on a program transformation technique that takes a concurrent program P as an input and generates a sequential program that simulates a subset of behaviors of P. It is then possible to use an available sequential testing tool to test the resulting sequential program. We introduce a new interleaving selection technique, called *bounded-interference*, which is based on the idea of limiting the degree of *interference* from other threads. An interference occurs when a thread reads a value that is generated by another thread. Our sequentialization technique encodes all interleavings of P with a certain interference degree k in the resulting sequential program $\widehat{P}_k$. All input variables of P are retained as input variables of $\widehat{P}_k$, and new input variables are introduced that encode interleaving choices. It is then possible to use an available sequential testing tool (in our case, PEX) to test the resulting sequential program $\widehat{P}_k$, which through standard systematic input generation for $\widehat{P}_k$, performs both input generation and interleaving exploration for P. The transformation is sound in the sense that any bug discovered by a sequential testing tool in the sequential program is a bug in the original concurrent program yet it lacks completeness.

B. (Con)^{2}colic Testing

To resolve the lack of completeness for bounded programs in [21], we introduced (con)^{2}colic testing [22], an approach that systematically explores both input and interleaving spaces of concurrent programs. (Con)^{2}colic testing can provide meaningful coverage guarantees during and after the testing process. (Con)^{2}colic testing can be viewed as a generalization of sequential concolic (*conc*rete and symb*olic*) testing [1] to concurrent programs that aims to achieve maximal code coverage for the programs. Like [21], (con)^{2}colic testing also exploits interferences among threads: The central objects in (con)^{2}colic testing are *interference scenarios*. An interference scenario represents a set of interferences among threads. Conceptually, interference scenarios describe the prefix of a concurrent program run such that all program runs with the same interference scenario follow the same control flow during execution of that prefix. By systematically enumerating interference scenarios, (con)^{2}colic testing explores the input and scheduling space of a concurrent program to generate tests (i.e., input values and a schedule) that cover a previously uncovered part of the program. We first enumerate all feasible interference scenarios that involve no interference what amounts to enumerating purely thread-local behaviors. After we have completed that, we continue with all feasible interference scenarios that involve one interference, then two interferences, then three, and so forth. Not all interference scenarios that we consider for enumeration are feasible. The interference scenarios involving $k + 1$ interferences are either observed during a concrete program execution or are constructed from an extension of an infeasible interference scenario with an interference from a feasible interference scenario (both having k or less interferences).

Figure 12 shows the four main components of our (con)^{2}colic testing framework: **(1)** A *concolic execution engine* executes the concurrent program according to a given input vector and schedule. The program is instrumented such that, during the execution, all important events are recorded. This information is used to generate further interference scenarios. **(2)** A *path exploration component* decides what *new* scenario to try next, aiming at covering previously uncovered parts of the program. **(3)** A *realizability checker* checks for the realizability of the interference scenario provided by the path

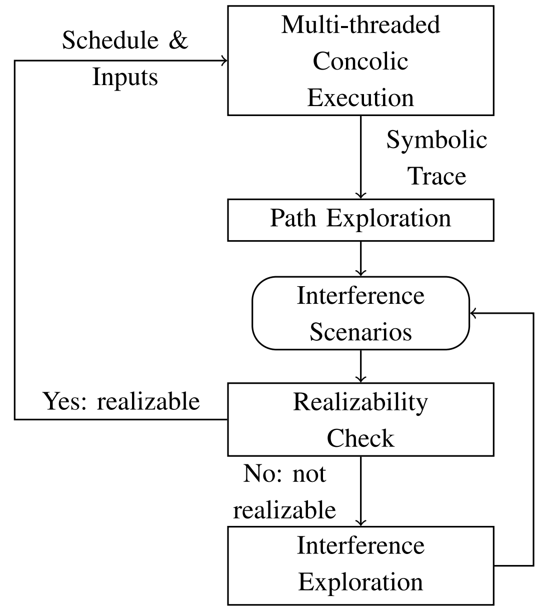

Fig. 12. Overview of (Con)^{2}colic Testing [22].

exploration component. Based on this interference scenario it extracts two constraint systems (one for the input values and one for the schedule) and checks for the satisfiability of them. If both are satisfiable, then the generated input vector and the schedule are used in the next round of concolic execution. **(4)** An *interference exploration component* extends unrealizable interference scenarios by introducing new interferences. (Con)^{2}colic testing can be instantiated with different search strategies to explore the interference scenario space.

C. Discussion of Existing Approaches

White-box testing concurrent programs has been a very active area of research in recent years. To alleviate the interleaving explosion problem that is inherent in the analysis of concurrent programs a wide range of *heuristic-based* techniques has been developed. Most of these techniques [23], [24], [25], [26], [27], [28], [29], [30] do not provide meaningful coverage guarantees, i.e., a precise notion of what tests cover. Other such techniques [31], [32], [33], [34], [35] provide coverage guarantees only over the space of interleavings by fixing the input values during the testing process.

Sequentialization techniques [36], [37], [38], [39], [40] translate a concurrent program to a sequential program that has the same behavior (up to a certain context bound), and then perform a *complete static symbolic* exploration of both input and interleaving spaces of the sequential program for the property of interest. Sequentialization of concurrent executions based on linear interfaces [41] bounds the number of context switches that they consider during state-space exploration. In contrast, (con)^{2}colic testing does not put restrictions on the number of context switches that occur when computing an input vector and an interleaving that realize an interference scenario. In fact, an interference scenario with only one interference might require a huge number of context switches due to synchronization. Note that interferences do not refer to locks, i.e., they only refer to read and write accesses of data variables. Locks are handled in a separate constraint system that expresses temporal constraints on the events of an execution. (Con)^{2}colic testing instead puts a bound on the number of interferences. On the other hand, sequentialization based on linear interfaces [41] does not put any restriction on the number of interferences that may happen during an execution that covers a linear interface. Both techniques therefore represent different strategies in exploring the state space of a concurrent program.

Symbolic PathFinder is a test generator that combines symbolic execution, model checking, and constraint solving[1]. It uses on-the-fly partial order reduction to limit the interleavings that have to be considered during state-space exploration. In contrast, (con)^{2}colic testing does not guide its exploration on interleavings but rather enumerates interference scenarios. An interference scenario imposes constraints on the input space and interleavings such that the scenario represents the set of combinations of input vectors and interleavings that trigger executions that cause exactly the same flow of data and control as specified in the interference scenario. For concrete execution, (con)^{2}colic testing then obtains one combination of input vector and interleaving from this set by solving the corresponding constraint systems.

Extensions of concolic testing to concurrent programs have been proposed before. In jCute [42], [43], the program is executed concolically and data-races in the observed execution are identified. Then, either the schedule is fixed and new input values are generated for the same concurrent schedule, or a new schedule is produced by keeping the inputs fixed and simply re-ordering the events involved in a data-race. In contrast to (con)^{2}colic testing, if a timeout occurs, it is impossible to quantify the partial work done as a meaningful coverage measure for the program.

Similar to (con)^{2}colic testing, a recent related work [26], generates tests with the aim of increasing code coverage of concurrent programs. However, it uses an under-approximation of the program (i.e. a set of program runs), as opposed to the actual program. Therefore, it is incomplete. Furthermore, test generation is done by solving a constraint system that encodes the scheduling constraints and the data-flow constraints together while considering the *whole* computation in the runs. However, (con)^{2}colic testing generates separate constraint systems for schedule generation and input generation which are based on only shared variable accesses and synchronization events. This reduces the complexity of the constraint systems drastically and increases scalability.

Some other recent work [44], [45] build a framework based on over- and under-approximations of interferences of the programs to check for safety properties. Like [26], they build a constraint system which includes local computation as well as global computation. Therefore, to reduce scalability issues, they focus only on program slices obtained from program executions.

IV. Tools

Our research on white-box test generation resulted in the three testing tools FShell, CPAtiger, and ConCrest. FShell is available as binary on several platforms[2]. CPAtiger is available online[3] as open source software. To evaluate (con)^{2}colic testing we have implemented the tool ConCrest[4] [22]. It supports multi-threaded C programs and uses a search strategy that targets assertion violations and explores interference scenarios according to the number of interferences in an ascending order.

Acknowledgment

This work was supported by the Canadian NSERC Discovery Grant, the Vienna Science and Technology Fund (WWTF) grant PROSEED, and the Austrian National Research Network projects S11403, S11405, S11402-N23 (RiSE) of the Austrian Science Fund (FWF).

References

[1] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed Automated Random Testing,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’05. New York, NY, USA: ACM, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[2] J. De Halleux and N. Tillmann, “Parameterized Unit Testing with Pex,” in *Proceedings of the 2Nd International Conference on Tests and Proofs*, ser. TAP’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 171–181. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792786.1792801

[3] P. Godefroid, M. Y. Levin, and D. Molnar, “SAGE: Whitebox Fuzzing for Security Testing,” *Commun. ACM*, vol. 55, no. 3, pp. 40–44, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2093548.2093564

[1] http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc, accessed last on 2014-06-05.

[2] http://forsyte.at/software/fshell/

[3] http://forsyte.at/software/cpatiger/

[4] http://forsyte.at/software/concrest/

[4] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "How Did You Specify Your Test Suite," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 407–416. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859084

[5] A. Holzer, M. Tautschnig, C. Schallhart, and H. Veith, "An Introduction to Test Specification in FQL," in *Proceedings of the 6th International Conference on Hardware and Software: Verification and Testing*, ser. HVC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 9–22. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987082.1987087

[6] G. Fraser and F. Wotawa, "Property Relevant Software Testing with Model-checkers," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 6, pp. 1–10, Nov. 2006. [Online]. Available: http://doi.acm.org/10.1145/1218776.1218787

[7] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar, "Generating Tests from Counterexamples," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 326–335. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999437

[8] M. Tautschnig, "Query-Driven Program Testing," Ph.D. dissertation, Vienna University of Technology, 2011.

[9] S. Afonin and E. Khazova, "Membership and Finiteness Problems for Rational Sets of Regular Languages," *International Journal of Foundations of Computer Science (IJFCS)*, vol. 17, no. 3, pp. 493–506, Jun. 2006.

[10] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "On the Structure and Complexity of Rational Sets of Regular Languages," in *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013, December 12-14, 2013, Guwahati, India*, 2013, pp. 377–388. [Online]. Available: http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2013.377

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

[12] "CoverageMeter 5.0.3," http://www.coveragemeter.com/.

[13] "CTC++ 6.5.3," http://www.verifysoft.com/en.html.

[14] S. C. Ntafos, "A comparison of some structural testing strategies," *IEEE Trans. Softw. Eng.*, vol. 14, no. 6, pp. 868–874, Jun. 1988. [Online]. Available: http://dx.doi.org/10.1109/32.6165

[15] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "Query-Driven Program Testing," in *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 151–166. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-93900-9_15

[16] D. Beyer, A. Holzer, M. Tautschnig, and H. Veith, "Information Reuse for Multi-goal Reachability Analyses," in *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ser. ESOP'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 472–491. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37036-6_26

[17] J. Bürdek, M. Lochau, S. Bauregger, A. Holzer, A. von Rhein, S. Apel, and D. Beyer, "Facilitating Reuse in Multi-Goal Test-Suite Generation for Software Product Lines," in *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-April 18, 2015. Proceedings*, 2015, (accepted).

[18] A. Holzer, "Query-Based Test Case Generation," Ph.D. dissertation, Vienna University of Technology, 2013.

[19] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[20] A. Holzer, V. Januzaj, S. Kugele, B. Langer, C. Schallhart, M. Tautschnig, and H. Veith, "Seamless Testing for Models and Code," in *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*, ser. FASE'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 278–293. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987434.1987461

[21] N. Razavi, A. Farzan, and A. Holzer, "Bounded-Interference Sequentialization for Testing Concurrent Programs," in *Proceedings of the 5th International Conference on Leveraging Applications of Formal Methods, Verification and Validation: Technologies for Mastering Change - Volume Part I*, ser. ISoLA'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 372–387. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34026-0_28

[22] A. Farzan, A. Holzer, N. Razavi, and H. Veith, "Con2Colic Testing," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 37–47. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491453

[23] C. Wang, S. Kundu, M. Ganai, and A. Gupta, "Symbolic Predictive Analysis for Concurrent Programs," in *Proceedings of the 2Nd World Congress on Formal Methods*, ser. FM '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 256–272. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-05089-3_17

[24] F. Sorrentino, A. Farzan, and P. Madhusudan, "PENELOPE: Weaving Threads to Expose Atomicity Violations," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 37–46. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882300

[25] K. Sen and G. Agha, "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools," in *Proceedings of the 18th International Conference on Computer Aided Verification*, ser. CAV'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 419–423. [Online]. Available: http://dx.doi.org/10.1007/11817963_38

[26] N. Razavi, F. Ivani, V. Kahlon, and A. Gupta, "Concurrent Test Generation Using Concolic Multi-trace Analysis," in *Programming Languages and Systems*, ser. Lecture Notes in Computer Science, R. Jhala and A. Igarashi, Eds., vol. 7705. Springer Berlin Heidelberg, 2012, pp. 239–255. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35182-2_17

[27] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps, "ConSeq: Detecting Concurrency Bugs Through Sequential Errors," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 251–264. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950395

[28] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508249

[29] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino, "Predicting

Null-pointer Dereferences in Concurrent Programs," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 47:1–47:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393651

[30] F. Chen, T. F. Serbanuta, and G. Rosu, "jPredictor: A Predictive Runtime Analysis Tool for Java," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 221–230. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368119

[31] M. Musuvathi, S. Qadeer, and T. Ball, "CHESS: A Systematic Testing Tool for Concurrent Software," 2007.

[32] P. Godefroid, "Model Checking for Programming Languages Using VeriSoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '97. New York, NY, USA: ACM, 1997, pp. 174–186. [Online]. Available: http://doi.acm.org/10.1145/263699.263717

[33] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 446–455.

[34] S. Qadeer and J. Rehof, "Context-Bounded Model Checking of Concurrent Software," in *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 93–107. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31980-1_7

[35] M. Emmi, S. Qadeer, and Z. Rakamarić, "Delay-bounded Scheduling," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 411–422. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926432

[36] A. Lal and T. Reps, "Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis," *Form. Methods Syst. Des.*, vol. 35, no. 1, pp. 73–97, Aug. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10703-009-0078-9

[37] S. Torre, P. Madhusudan, and G. Parlato, "Reducing Context-Bounded Concurrent Reachability to Sequential Reachability," in *Proceedings of the 21st International Conference on Computer Aided Verification*, ser. CAV '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 477–492. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02658-4_36

[38] S. Qadeer and D. Wu, "KISS: Keep It Simple and Sequential," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI '04. New York, NY, USA: ACM, 2004, pp. 14–24. [Online]. Available: http://doi.acm.org/10.1145/996841.996845

[39] S. Qadeer, "Poirot: A Concurrency Sleuth," in *Proceedings of the 13th International Conference on Formal Methods and Software Engineering*, ser. ICFEM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=2075089.2075093

[40] Z. Rakamarić, "STORM: Static Unit Checking of Concurrent Programs," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 519–520. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810460

[41] S. La Torre, P. Madhusudan, and G. Parlato, "Model-Checking Parameterized Concurrent Programs Using Linear Interfaces," in *Proceedings of the 22Nd International Conference on Computer Aided Verification*, ser. CAV'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 629–644. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_54

[42] K. Sen, "Scalable Automated Methods for Dynamic Program Analysis," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2006.

[43] K. Sen and G. Agha, "Concolic Testing of Multithreaded Programs and Its Application to Testing Security Protocols," University of Illinois at Urbana Champaign, Tech. Rep. UIUCDCS-R-2006-2676, 2006.

[44] N. Sinha and C. Wang, "On Interference Abstractions," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 423–434. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926433

[45] ——, "Staged Concurrent Program Analysis," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 47–56. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882301