CSTVA 2016
7th Workshop on
Constraint Solvers in Testing, Verification, and Analysis

Saarbrücken, Germany, July 17, 2016

Affiliated with
The Int'l. Symposium on Software Testing and Analysis (ISSTA 2016).

Preface

TODO

July, 2016

Omer Tripp
Christoph M. Wintersteiger

Table of Contents

Program Committee

Mathieu Acher	University of Rennes I/INRIA
Roberto Bagnara	University of Parma and BUGSENG
Martin Brain	University of Oxford, UK
Stefano Di Alesio	Certus Centre for Software Verification and Validation, Simula Research Laboratory
Julian Dolby	IBM Thomas J. Watson Research Center
Peng Liu	Hong Kong University of Science and Technology
Ruben Martins	University of Texas at Austin
Corina Pasareanu	CMU/NASA Ames Research Center
Markus N. Rabe	University of California, Berkeley
Philipp Ruemmer	Uppsala University, Department of Information Technology
Philippe Suter	IBM Thomas J. Watson Research Center
Omer Tripp	Google
Georg Weissenbacher	Vienna University of Technology
Christoph M. Wintersteiger	Microsoft Research

Design of a Modified Concolic Testing Algorithm with Smaller Constraints

Yavuz Koroglu and Alper Sen

Department of Computer Engineering, Bogazici University, Turkey
yavuz.koroglu@boun.edu.tr
alper.sen@boun.edu.tr

Abstract

Concolic testing is a well-known unit test generation technique. However, bottlenecks such as constraint solving prevents concolic testers to be used in large projects. We propose a modification to a standard concolic tester. Our modification makes more but smaller queries to the constraint solver, i.e. ignores some path conditions. We show that it is possible to reach the same branch coverage as the standard concolic tester while decreasing the burden on the constraint solver. We support our claims by testing several C programs with our method. Experimental results show that our modification improves runtime performance of the standard concolic tester in half of the experiments and results in more than 5x speedup when the unit under test has many infeasible paths.

1 Introduction

Software testing is an important but resource consuming part of software development process. According to a study in 2005, 79% of Microsoft developers write unit tests [11]. Therefore, automated unit testing is an important area of research. Concolic testing is an automated unit testing method first introduced in 2005 [8, 13]. Empirical study on concolic testing shows that scalability is an issue for concolic testing [11]. Therefore making concolic testing more scalable is an open research area.

We propose a modification to the input generation method of concolic testing, called Incremental Partial Path Constraints (IPPC). A standard concolic tester instruments the unit under test (UUT) to collect operations that either affect or depend on symbolic variables during a concrete execution. This sequence of operations is called an *execution trace*. A concolic tester symbolically reexecutes the execution trace to generate *path conditions* that solely depend on the symbolic variables. Then, the concolic tester negates the last path condition that is not negated before. At the final step, constraint solver is called only once with all path conditions to generate a new input vector. As noted in [8], solving more but smaller constraints against one large constraint (i.e. a constraint that contains large number of path conditions) is more efficient. Towards this goal, we modified the input generation step of a standard concolic tester such that we call the constraint solver multiple times, but using only a few path conditions at each invocation. Our experiments show that we can generate inputs that fall into the same equivalence class (i.e. inputs that force the program into generating the same execution trace) using fewer path conditions than a standard concolic tester. Although we pay the cost of more constraint solver calls on

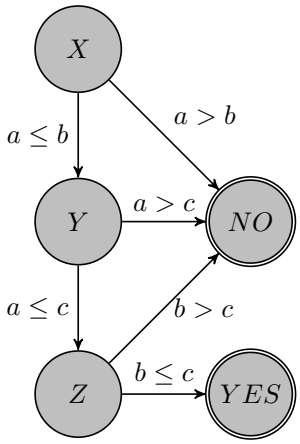


Figure 1: Control Flow Graph of *isSorted(a, b, c)*

average we show that our modification results in more than 5x speedup when UUT has many infeasible paths.

This paper makes two contributions. First, we propose an input generation strategy based on partial path constraints, called IPPC and incorporate our strategy into a standard concolic testing algorithm. Partial path constraints contain a subset of path conditions of an execution trace. Second, we implement our modification using the CREST tool [1], which is a concolic testing framework for C. Our results indicate that IPPC achieves the same branch coverage as the standard concolic tester while decreasing the burden on the constraint solver and leading to speedups.

We illustrate our approach on a small example in the next section. In section 4, we describe our approach in detail. We present our experimental results in section 5. We discuss the related work in section 3 and conclude in section 7.

2 Overview

In this section, we run both the standard concolic tester and IPPC on a small program unit which checks if three numbers are sorted or not, denoted by *isSorted(a, b, c)*. The control-flow graph of the unit under test (UUT) is given in Figure 1.

Figure 2 shows an execution of the standard concolic testing algorithm on *isSorted(a, b, c)*. A standard concolic tester starts from an arbitrary input ($[0,0,0]$ in this case), executes the UUT using the input values. The concolic tester collects the execution trace of this concrete execution and generates the path constraint $\pi_0 = [a \leq b, a \leq c, b \leq c]$. The last condition of π_0 is negated and we get $\pi_1 = [a \leq b, a \leq c, b > c]$. We invoke the constraint solver with π_1, denoted as CS(3), where 3 represents the number of path conditions in π_1. We get $[0,1,0]$ as the new input from the constraint solver. We execute the program with new inputs and collect the path constraint $\pi_2 = [a \leq b, a \leq c, b > c]$. Now, we negate the second condition of π_2 and remove all conditions after the second condition and get $\pi_3 = [a \leq b, a > c]$. We explain the reasons to exclude path conditions that come after the negated condition in Section 4.2. We get $[0,1,-1]$ by invoking CS(2). Concrete execution of the program results in $\pi_4 = [a \leq b, a > c]$. Then, we negate the only remaining path condition and get $\pi_5 = [a > b]$ after removing the last condition. CS(1) generates $[2,1,-1]$. The concolic tester stops generating test inputs after executing the program since the resulting path constraint $\pi_6 = [a > b]$ has no path condition that is not negated before. At the

$i_1 = [0,0,0]$	[initial inputs]
$\pi_0 = [a \leq b, a \leq c, b \leq c]$	[collected]
$\pi_1 = [a \leq b, a \leq c, b > c]$	[generated]
$i_2 = [0,1,0]$	[CS(3)]
$\pi_2 = [a \leq b, a \leq c, b > c]$	[collected]
$\pi_3 = [a \leq b, a > c]$	[generated]
$i_3 = [0,1,-1]$	[CS(2)]
$\pi_4 = [a \leq b, a > c]$	[collected]
$\pi_5 = [a > b]$	[generated]
$i_4 = [2,1,-1]$	[CS(1)]
$\pi_6 = [a > b]$	[collected]
STOP	[Path Exhaustion]

Figure 2: A Sample Execution of Standard Concolic Testing on *isSorted(a, b, c)*

end, we have invoked the constraint solver three times with an average path constraint length of two.

Figure 3 illustrates the execution of our IPPC algorithm on *isSorted(a, b ,c)*. In this case, each time we generate a new path constraint, we do not immediately call the constraint solver. We define the constraint obtained by executing the program a *full path constraint*. We generate a subset of the full path constraint, ϕ^0, which we call a *partial path constraint*. Partial path constraint ϕ^0 is initialized as an array which contains only the negated path condition. Partial path constraint generation phase is called *overapproximation*. We call the constraint solver with ϕ^0. In case the constraint solver does not generate an input which satisfies π, we add the remaining path conditions of π to ϕ^0 and generate larger path constraints such as ϕ^1, ϕ^2 etc. until the constraint solver generates an input that satisfies π. For our example, we only made three calls to the constraint solver with average constraint length of one.

3 Related Work

The idea of incremental constraint solving for concolic testing is as old as concolic testing itself and suggested along with CUTE, one of the first concolic testing tools [13]. However their incremental solving idea should not be confused with the one that we present in this paper. In CUTE, since they negate only one path condition, they keep the variables which are irrelevant to that path condition fixed. Therefore, they solve for variables which are only relevant to the negated path condition and decrease the burden on the constraint solver [13]. In our approach, CREST still does the incremental solving optimization of CUTE. On top of that, we also produce path contraints with fewer path conditions than a standard concolic tester.

Solving only a subset of path conditions instead of the whole path constraint is an approach recently used in finding integer overflows [15]. The motivation is supported by observing that many of the path conditions of an execution trace are irrelevant w.r.t. integer overflows. In our work, we make a more general assumption that some of the path conditions should be irrelevant to the execution trace itself. Experimental results support our motivational assumption.

All concolic testers require a *constraint solver*. Z3 and Yices are commonly used constraint solvers

$i_1 = [0,0,0]$	[initial inputs]
$\pi_0 = [a \leq b, a \leq c, b \leq c]$	[collected]
$\pi_1 = [a \leq b, a \leq c, b > c]$	[generated]
$\phi_1^0 = [b > c]$	[overapproximation]
$i_2 = [0,1,0]$	[CS(1)]
i_2 satisfies π_1	$CONTINUE$
$\pi_2 = [a \leq b, a \leq c, b > c]$	[collected]
$\pi_3 = [a \leq b, a > c]$	[generated]
$\phi_3^0 = [a > c]$	[overapproximation]
$i_3 = [1,1,0]$	[CS(1)]
i_3 satisfies π_3	$CONTINUE$
$\pi_4 = [a \leq b, a > c]$	[collected]
$\pi_5 = [a > b]$	[generated]
$\phi_5^0 = [a > b]$	[overapproximation]
$i_4 = [2,1,0]$	[CS(1)]
i_4 satisfies π_5	$CONTINUE$
$\pi_6 = [a > b]$	[collected]
STOP	[Path Exhaustion]

Figure 3: A Sample Execution of IPPC on *isSorted(a, b, c)*

[6, 3]. Details on the complexity of constraint solving can be found in [18]. Standard constraint solvers are not able to solve all types of constraints [11]. There are constraint solvers which solve nonlinear constraints [9] and constraint solvers which solve constraints involving high-precision rational numbers [16]. Yun et al. has modified CREST to use Z3 constraint solver in order to support nonlinear arithmetic [2]. We could easily implement IPPC modification on top CREST-z3, although plain CREST is sufficient for our experiments.

Concolic testing is not restricted to sequential programs. There are concolic testers which test concurrent software [7]. Therefore, IPPC can also be used to test concurrent units.

Many concolic testers (e.g. CUTE) use bounded depth first search (*bDFS*) instead of bounding the number of iterations as in IPPC and CREST [13, 12]. *bDFS* can be viewed as an attempt to keep the number of path conditions limited. We believe that our approach makes bounding the number of iterations more preferable to bounding the depth of the search as in *bDFS* since experiments show that IPPC already reduces the size of path constraints significantly.

Random branch selection (RND) is an approach introduced in CREST [1]. In random branch selection one negates each path condition of a path constraint with probability 0.5. A variant of this strategy negates only one path condition at random and calls the constraint solver with that path condition.

Control flow directed search (CFG) is a search strategy which guides the search using the static structure of the UUT. CFG assigns weights to edges of the control flow graph of the UUT and calculates distances to each unvisited branch. Then, CFG solves a path constraint which leads to the unvisited branch with the least distance to the current execution trace [1]. We compare our approach with both

CFG and RND.

A recent study on CREST proposes a new search strategy, called DYNASTY, which uses the control flow graph of the UUT to guide the search as in CFG technique [5]. They modify CREST as in our work and increase branch coverage with fewer iterations. Their approach is based on avoiding infeasible paths, whereas our approach decreases the penalty of hitting an infeasible path by using partial path constraints.

4 Incremental Partial Path Constraint (IPPC)

In this section, we describe IPPC in detail.

4.1 Background

We have previously stated that a concolic tester generates path conditions from an execution trace of the UUT. We define the following using this fact:

Definition 1. A *path condition* is a function of symbolic variables to $\{T, F\}$.

Definition 2. An *input vector* is an assignment to all symbolic variables of UUT.

Definition 3. A *full path constraint* is an array whose elements form a set of path conditions.

Definition 4. Any array whose elements form a subset of the elements of the full path constraint is called a *partial path constraint*.

Definition 5. An input vector is said to $satisfy$ a path constraint if and only if the input vector makes all conditions of the path constraint T.

Conjunction of all conditions in a partial path constraint is an overapproximation of conjunction of all conditions in the full path constraint. Therefore the process of generating partial path constraints from full path constraints is called an *overapproximation*.

Definition 6. An algorithm which takes a path constraint and returns an input vector which satisfies that constraint is called a *constraint solver*. If there exists no such input vector, it returns *null*.

We used the Yices constraint solver, which is the default solver in CREST [6, 1].

4.2 Algorithm Design of IPPC

We implemented IPPC on top of CREST's standard concolic testing algorithm (denoted by DFS) as shown in Algorithm 1. We initially give $nIterations \leftarrow 0$, $TestInput \leftarrow random\_input$ and $TestInputs \leftarrow \emptyset$. We can see from line 1 that the algorithm is bounded by the maximum number of iterations. At line 2, we check the input vector because constraint solver is assumed to return *null* if an input vector can not be generated, i.e. given path constraint is infeasible. We increment iterations only if the path constraint is satisfiable. At line 4, we execute UUT with *input* and save the full path constraint of the execution trace as π. Lines 8 and 9 ensure that the algorithm performs a depth first search (DFS). We always negate the last unvisited condition of a path constraint. If there exists no path condition that is not negated before, we stop because all paths are executed. At line 11, constraintSolver takes a path constraint and returns a satisfying input vector. We don't terminate the solver, we use it in it's built-in incremental mode. Notice that we exclude the path conditions which come after the negated path condition. Any input vector which satisfies path conditions up to the negated path condition must generate the same execution trace

```
input  : uut (unit under test)
         maxIterations
         testInput
         testInputs
         nIterations
output: testInputs
1  if nIterations < maxIterations then
2      if testInput ≠ null then
3          nIterations ←nIterations +1;
4          π ←execute(uut, testInput);
5          add(testInputs, testInput);
6      end
7      for i ←sizeOf(π)−1 to 0 do
8          if !isNegatedBefore(π[i]) then
9              setNegatedBefore(π[i]);
10             π[i] ← ¬π[i];
11             testInput ← constraintSolver({π[0]...π[i]});
12             return crest(uut, maxIterations, testInput, testInputs, nIterations);
13         end
14     end
15 end
16 return testInputs;
```

Algorithm 1: Concolic Testing Algorithm Implementing DFS with Maximum Iterations (CREST)

up to the occurrence of the negated path condition. Negated path condition will force the program to a different execution trace. So, path conditions coming after the negated condition is not a part of the new execution trace and therefore should be removed. If the unnecessary path conditions are not excluded, they may cause false path constraint infeasibilities. False infeasibilities force the concolic tester not to generate any input vector for the path constraint and therefore cause a decrease in test coverage.

IPPC is exactly the same as Algorithm 1 except we replace line 11 with *input* $\leftarrow$ IPPC_Solve $(UUT, \{\pi[0]...\pi[i]\}, \{\pi[i]\})$. We describe IPPC_Solve in Algorithm 2. We keep the solver running in incremental mode.

Line 1 of Algorithm 2 invokes the constraint solver with ϕ. If the constraint solver is unable to generate an input vector, we conclude that the full path constraint is also *infeasible* and return *null*. We prove the infeasiblity of a full path constraint given the infeasibility of a partial path constraint in Theorem 1. The for loop starting at line 5 checks if the input vector satisfies the full path constraint. If *input* satisfies π, we can return *input*. If not, we *learn* the unsatisfied path condition by adding it to ϕ and generate a new input vector. This process may be continued until $\phi = \pi$ in the worst case where either an input can be generated or π is infeasible given that the constraint solver is sound (if exists, returns a correct test input) and complete (terminates and is correct for all cases). This worst case condition is rare or non existent at least in our experiments.

4.3 Correctness and Completeness

To be able to produce readable proofs, we assume all boolean operations on a path constraint are performed on the conjunction of all conditions of the path constraint.

```
input  : uut (unit under test)
         π (full path constraint)
         φ (partial path constraint)
output: testInput
1  testInput ← constraintSolver(φ);
2  if testInput = null then
3  |  return null;
4  end
5  for i ← 0 to sizeOf(π) − 1 do
6  |  if !sat(testInput, π[i]) then
7  |  |  append(φ, π[i]);
8  |  |  return IPPC_Solve (uut, π, φ);
9  |  end
10 end
11 return testInput;
```

Algorithm 2: IPPC_Solve

Theorem 1. *If a partial path constraint ϕ is unsatisfiable, then its full path constraint π is also unsatisfiable.*

Proof. Since ϕ is unsatisfiable, $\neg\phi$ is valid. Since ϕ is an overapproximation of π, $\pi \rightarrow \phi$ by definition. Therefore by modus tollens, $\neg\pi$ is valid. In other words, π is unsatisfiable. □

We now show the correctness of IPPC_Solve. We change only one line in Algorithm 1. Therefore, to prove the correctness of IPPC, we only need to ensure that for all path constraints, IPPC_Solve generates an input vector that has the same property as the input vector generated by the constraint solver. If such an input vector exists, a constraint solver always generates an input vector that satisfies the full path constraint.

Theorem 2. *(a) IPPC_Solve generates* null *whenever the constraint solver in Algorithm 1 at Line 11 generates* null*. (b) Otherwise IPPC_Solve always generates an input vector that satisfies the full path constraint.*

Proof. Proof of (a) is trivial due to lines 1-4 of Algorithm 2. We can see from lines 5-10, that Algorithm 2 would not stop until the input vector satisfies the full path constraint. Therefore proof of (b) is trivial as well. □

Theorem 3. *IPPC_Solve is correct and eventually terminates, i.e. IPPC_Solve is complete.*

Proof. IPPC_Solve is correct by Theorem 2. The second part of the proof is as follows. At any time, partial path constraint ϕ is either (a) unsatisfiable, or an input vector i is generated. If an input vector i is generated, either (b) i satisfies π, or (c) we add a new path condition of full path constraint to ϕ. If (a), IPPC_Solve returns *null* and terminates. If (b), IPPC_Solve returns i and terminates. If (c) happens $N-1$ times, where N denotes the number of path conditions in π, $\phi \leftrightarrow \pi$ must hold. In other words, we build up the partial path constraint up to the full path constraint. In that case, the constraint solver must either generate *null* or generate a satisfying input vector, therefore the algorithm terminates. □

Table 1: List of Benchmarks

UUT	KLOC	#vars
gcd	0.05	2
bsort	0.05	30
sqrt	0.06	1
prime	0.1	1
factor	0.2	1
replace	0.5	20
ptokens	0.6	40
grep	15	10

Table 2: Experimental Results (Best results are highlighted)

UUT	#Itr	Total CS Calls				Avg Const. Size				Branch Coverage (%)				Time Elapsed (sec)			
		DFS	CFG	RND	IPPC	DFS	CFG	RND	IPPC	DFS	CFG	RND	IPPC	DFS	CFG	RND	IPPC
gcd	1000	5681	866	1105	11004	161.0	4.1	2.9	1.9	100	100	100	100	22.6	5.7	**3.9**	7.5
	5000	25801	4364	5653	64844	188.6	5.7	2.8	1.9	100	100	100	100	119.5	24.8	**21.6**	34.7
	10000	47367	8761	11287	120678	206.3	5.9	2.8	1.9	100	100	100	100	234.5	48.7	**45.7**	125.5
bsort	100	117	53	104	1346	145.7	148.5	88.9	10.3	100	100	100	100	1.1	**0.8**	0.9	1.3
	500	711	284	537	10306	308.9	165.3	115.5	12.7	100	100	100	100	7.6	5.0	**4.8**	9.2
	1000	1462	571	1063	25023	342.5	165.0	132.6	14.3	100	100	100	100	15.5	**8.7**	9.5	21.9
sqrt	1000	1010	980	1064	1006	17.2	2.3	1.8	1.0	94.4	94.4	94.4	94.4	4.3	4.5	4.3	**3.9**
	5000	5001	4894	5372	5006	24.0	2.3	1.8	1.0	94.4	94.4	94.4	94.4	30.1	23.1	22.2	**20.3**
	10000	10010	9820	10732	10014	22.7	2.3	1.8	1.0	94.4	94.4	94.4	94.4	45.9	42.0	42.2	**39.1**
prime	100	1275	92	111	132	103.3	7.5	4.8	1.2	**92.5**	85	62.5	**92.5**	2.3	0.5	0.5	**0.4**
	175	3946	165	212	253	186.6	8.0	4.0	1.2	**92.5**	90	70	**92.5**	10.1	1.0	1.1	**0.8**
	250	4166	236	287	1688	189.6	7.2	4.1	1.8	**92.5**	92.5	82.5	**92.5**	13.5	1.6	1.6	**1.5**
factor	50	2678	80	110	7693	222.1	2.9	4.0	1.9	**94.2**	**94.2**	76.9	**94.2**	8.6	**0.2**	**0.2**	1.5
	75	5734	111	145	25255	249.7	3.4	4.7	1.9	**94.2**	**94.2**	83.2	**94.2**	23.5	**0.3**	**0.3**	4.6
	100	11157	148	243	34218	310.9	2.8	16.0	1.9	**94.2**	**94.2**	89.4	**94.2**	120.3	0.6	**0.5**	6.4
replace	1000	1129	1024	1024	6847	16.6	25.9	31.7	3.8	85.2	**88.7**	85.2	85.2	**4.1**	4.5	**4.1**	5.8
	5000	5623	5193	5167	43695	21.5	28.4	32.4	4.8	88.7	**90.8**	88.7	88.7	**22.0**	23.8	22.4	31.4
	10000	10936	10270	10301	103590	24.5	28.0	33.0	5.6	88.7	**90.8**	90.8	88.7	**43.1**	45.5	44.6	66.9
ptokens	1000	1181	1355	1319	38415	591.0	136.8	102.2	29.8	**85.1**	79.8	79.8	**85.1**	30.0	9.4	**8.6**	18.5
	5000	6394	6976	6739	253829	1949.9	148.7	103.2	37.5	**85.1**	84.4	84.9	**85.1**	212.4	48.4	**47.7**	106.1
	10000	13394	13838	13510	432262	2241.6	163.7	99.7	30.5	**91.5**	89.6	88.9	**91.5**	321.7	106.8	**91.2**	210.8
grep	100	100	N/A	121	436	59.0	N/A	362.0	3.2	16.7	N/A	16.7	16.7	**0.3**	N/A	0.4	0.4
	220	269	N/A	272	1671	178.4	N/A	332.2	4.7	16.7	N/A	16.7	16.7	**0.9**	N/A	**0.9**	1.0
	340	439	N/A	451	3671	324.8	N/A	350.1	8.3	16.7	N/A	16.7	16.7	1.6	N/A	**1.5**	1.9

5 Experimental Study

We implemented IPPC on top of CREST tool. Our implementation of IPPC and experimental results are available online [10]. CREST implements the standard concolic testing algorithm (DFS) and two different improvements, denoted by CFG and RND. We compared IPPC with these three techniques. Techniques we used in experiments are as follows.

1. **DFS:** Standard concolic testing algorithm.
2. **CFG:** Control flow directed testing algorithm.
3. **RND:** Random branch testing heuristic on top of the standard concolic testing algorithm.
4. **IPPC:** Our Incremental Partial Path Constraint algorithm.

Brief explanations of CFG and RND algorithms are given in Section 3.

We carried out the experiments on a virtual Linux guest with 1024MB memory and one CPU hosted by a MacBook Pro with an Intel Core i7 2.9 GHz GPU and 8GB Memory. We collected the following information for each experiment:

1. **Time elapsed:** Time spent to test the UUT.
2. **Total CS Calls:** Total number of constraint solver calls made by the concolic tester.
3. **Avg Const. Size:** Average size of path constraints solved by the constraint solver.
4. **Branch Coverage:** Measurement of total branch coverage of UUT.

Concolic testing involves a degree of randomness due to the randomness of initial inputs. Therefore we performed 10 executions of each experiment and took average values of each measure. All of our experimental results can be found in Table 2.

Table 1 shows the program units we used in our experiments. Column *KLOC* represents the lines of code measured in thousands. Column *#vars* represents the number of symbolic variables.

We implemented five small benchmarks, *gcd*, *bsort*, *sqrt*, *prime* and *factor* to conduct our initial experiments. *gcd* implements the binary greatest common divisor algorithm. We downloaded and modified the bubble sort algorithm *bsort*, which sorts a given array of integers [17]. *sqrt* takes the floor of the square root of a given integer. *prime* decides whether a given integer is prime or not. *factor* is an integer factorization algorithm. *replace* and *grep* are benchmarks that come with CREST framework and used in several research studies on concolic testing [1, 5, 14]. *ptokens* is the printtokens benchmark available at Software Infrastructure Repository (SIR) [4]. *ptokens* tokenizes the given string according to a grammar. For reasons described in Section 6, our implementation of *gcd*, *prime* and *factor* do not contain any bitwise masking or modulo operations. Instead, we decide divisibility via only subtraction and comparison operations. Also, *sqrt* does not use any floating point operations since CREST has no symbolic equivalent of floating point variables.

In our experiments, we used programs in different sizes. None of the given programs are too large in size so they can be tested in reasonable time without getting into scalability issues. In our experimental set, we argue that the structure of UUT is related to the performance of the testers that we use rather than the sizes of the programs. We observed that small programs such as *factor* can have very long runtimes (120 sec). We argue that IPPC will perform well not when the program is small or large, but when the program contains many infeasible paths. Although being small, *prime* and *factor* contain many infeasible paths. Although being large, *grep* and *replace* did not contain many infeasible paths.

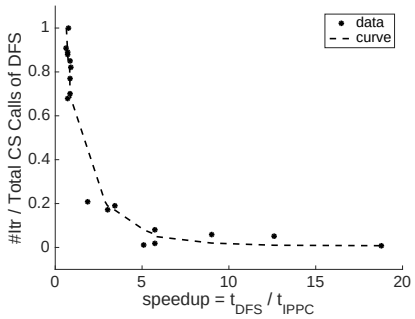
We show all experimental results in Table 2. CFG method is not applicable to *grep* due to a bug in CREST, therefore we used *N/A* to denote the results we could not observe. We used 1000, 5000 and 10000 as the maximum number of iterations (*#Itr*) in experiments. We decreased *#Itr* for *prime*, *factor*, *bsort* and *grep*, since those units have few distinct execution traces. When there are few distinct execution traces and *#Itr* is too high, DFS and IPPC are able to stop before completing *#Itr* iterations, since they check if all execution traces are explored or not. However, CFG and RND have no such stopping condition and iterate *#Itr* times. So, if we kept *#Itr* high for *prime*, *factor*, *bsort* and *grep*, it would unfairly result in bad runtimes for CFG and RND.

We show the best algorithms for each benchmark in Table 3 where *All Equal* means all techniques are equally well. The column denoted as *by Runtime First* compares techniques by runtime first, if techniques have similar runtimes, compares branch coverages. The last column compares techniques by coverage first. DFS does not perform well in both columns. In terms of runtimes RND is the best and IPPC is the second. In terms of coverage CFG and IPPC are the best. Hence, IPPC performs well in both categories.

We next compare IPPC with DFS in more detail since IPPC is derived from DFS. We observe from Table 4 that there exists a correlation between Avg Const. Size of DFS / Avg Const. Size of IPPC and speedup of IPPC over DFS. When the gap between the constraint sizes of DFS and IPPC increases, the speedup of IPPC over DFS increases with the slight exception of *grep*. Similarly, Figure 4 shows that whenever the number of constraint solver calls (Total CS Calls) of DFS is close to the maximum number of iterations (#Itr), DFS works faster. If DFS makes many more calls than the number of

Table 3: Best Concolic Testers

UUT	by Runtime First	by Coverage First
gcd	RND	RND
bsort	CFG	CFG
sqrt	IPPC	IPPC&DFS
prime	IPPC	IPPC&DFS
factor	RND	CFG
replace	DFS	CFG
ptokens	RND	IPPC
grep	All Equal	All Equal



UUT	Avg Const. Size Ratio (DFS / IPPC)	Speedup (t_{DFS}/t_{IPPC})
replace	4.4	0.6x
bsort	20.8	0.79x
sqrt	21.3	1.25x
grep	31.8	0.83x
ptokens	48.4	1.7x
gcd	97.5	2.77x
prime	115.6	9.1x
factor	137.3	9.8x

Figure 4 & Table 4: Relationship between Speedup and #Infeasible Constraints (Figure 4) and IPPC Speedup over DFS (Table 4).

iterations, IPPC works faster. Normally, DFS is expected to call constraint solver exactly once for each iteration. DFS makes more than one constraint solver call for an iteration only if the generated path constraint for that iteration is infeasible. In that case, DFS changes the path constraint and calls constraint solver repeatedly until a feasible path is found. We believe *factor* has the largest number of infeasibilities since DFS makes many constraint solver calls for few iterations. Figure 4 shows that the speedup of IPPC over DFS is above 5x when DFS generates four or more infeasible path constraints for each feasible path constraint (i.e. DFS makes more than five constraint solver calls for each iteration, #Itr/Total CS Calls of DFS < 0.2). Therefore, IPPC finds infeasibilities faster for all benchmarks.

In all experiments, the largest path constraint produced by IPPC had a length of 157, whereas the largest path constraints of DFS, CFG and RND had lengths of 2922, 1603 and 1391, respectively. We conclude that we eliminated the need for solving large path constraints to generate test inputs while keeping the runtimes fast and coverage high using IPPC.

We get exactly the same coverage results for both IPPC and DFS. We expect this since both algorithms perform a depth-first search. We argue that the coverage results being similar indicates that IPPC correctly does DFS on the UUT while improving the performance.

IPPC considers constraints that are 10x smaller than other techniques. We believe it is because that IPPC finds infeasibilities early, since most infeasibilities arise from a combination of few conditions in the constraint.

6 Threads to Validity

We assume that the constraint solver is sound, i.e. the constraint solver can find an input vector whenever there exists an input vector which satisfies the path constraint. In general, this assumption is not valid. We know that the constraint solver used in CREST, Yices [6, 11], does not find solutions for nonlinear path conditions (e.g. $x_2 x_1 < 12$). We also know that CREST may not be able to solve conditions involving modulo operation and bitwise masking [11]. All path conditions generated in our experiments are linear, in other words they can be written as $k \bowtie \sum_{i=1}^{N} c_i x_i$ where $\bowtie \in \{<, \leq, =, \neq, >, \geq\}$, $k \in \mathrm{R}$, $\forall i \in \{1, ..., N\}, c_i \in \mathrm{R}$ and N: total number of symbolic variables). All path conditions in our experiments are also free of modulo and bit masking operations. Hence, we safely assume that the constraint solver is sound in our environment. We also do not use any floating point arithmetic.

The UUT can have intermediary variables calculated from symbolic variables and therefore can have path conditions on those intermediary variables. All path conditions that CREST returns are on the initial symbolic variables. Therefore, even if all branch conditions in the code may seem trivial, CREST may fail to generate correct inputs if variables are nonlinear (e.g. multiplication of two symbolic variables).

We assume the UUT to be *sequential* and *deterministic*, i.e. if an input vector i produces an execution trace e, i will always produce e for this UUT. However, for example a process which depends on random numbers could violate this assumption. We carefully chose the experiments so that we never violate these basic assumptions.

We assume the UUT is *terminating*, since if UUT halts, so does the tester as well. The test cases we chose and the test cases in previous work are all terminating.

We report runtimes that we acquired from a virtual environment. We got similar results on an host machine as well.

It is possible that IPPC may learn partial path constraints up to a point that they become full path constraints. So in the worst case, a standard concolic tester is more efficient than IPPC. However, our experimental results show that we require only a small portion of the full path constraint to generate input vectors belonging to the same equivalence class, i.e. input vectors which generate same execution traces when given to UUT.

7 Conclusions and Future Work

In this paper, we designed a constraint solving strategy, called Incremental Partial Path Constraints (IPPC), which eliminates the need for solving large constraints to generate unit tests. We compared our design with other concolic testing algorithms in experiments. We observed that when there are many infeasible path constraints in a program, IPPC has more than 5x speedup over a standard concolic tester. We significantly reduce the number of path conditions required to generate an input vector and show that IPPC dominates other techniques in two of eight cases and has the best coverage levels in three of eight cases. We show that it is possible to reach high branch coverage while decreasing the burden on the constraint solver.

We believe that our constraint solving strategy contains room for improvement and shows promise for future work. The performance of IPPC on nonlinear path constraints (i.e. path constraints that contain at least one nonlinear path condition) or on concurrent software is an important question. Also, it is possible to further improve IPPC by input vector caching, i.e. trying the previously generated input vectors first to avoid calling constraint solver. We believe that our work sufficiently motivates further research on constraint solving strategies involving partial path constraints.

References

[1] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, 2008.

[2] CREST-z3, https://github.com/heechul/crest-z3.

[3] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 2008.

[4] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[5] Yu Dong, Mengxiang Lin, Kai Yu, Yi Zhou, and Yinli Chen. Achieving high branch coverage with fewer paths. In *Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, COMPSACW '11, 2011.

[6] Bruno Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer International Publishing, 2014.

[7] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 2013.

[8] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005.

[9] Pierluigi Nuzzo, Alberto Puggelli, Sanjit A. Seshia, and Alberto Sangiovanni-Vincentelli. CalCS: SMT solving for non-linear convex constraints. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, 2010.

[10] `http://bitbucket.org/yavuzkoroglu/crest-ppc`.

[11] Xiao Qu and Brian Robinson. A case study of concolic testing tools and their limitations. In *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ESEM '11, 2011.

[12] Koushik Sen. *Scalable Automated Methods for Dynamic Program Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, 2006. Available at `http://srl.cs.berkeley.edu/~ksen/doku.php`.

[13] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, 2005.

[14] Hyunmin Seo and Sunghun Kim. How we get there: A context-guided search strategy in concolic testing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, 2014.

[15] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin C. Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, pages 473–486, 2015.

[16] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Păsăreanu. CORAL: Solving complex constraints for symbolic pathfinder. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, 2011.

[17] BubbleSort, `http://www.programmingsimplified.com/c/source-code/c-program-bubble-sort`.

[18] Edward P. K. Tsang. *Foundations of constraint satisfaction.* Computation in cognitive science. Academic Press, 1993.

A CHR-Based Solver for Weak Memory Behaviors*

Allan Blanchard[1,2], Nikolai Kosmatov[1], and Frédéric Loulergue[2]

[1] CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[2] Univ Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France

Abstract

With the wide expansion of multiprocessor architectures, the analysis and reasoning for programs under weak memory models has become an important concern. This work presents an original constraint solver for detecting program behaviors respecting a particular memory model. It is implemented in Prolog using CHR (Constraint Handling Rules). The CHR formalism provides a convenient generic solution for specifying memory models, that benefits from the existing optimized implementations of CHR and can be easily extended to new models. We briefly present the solver design, illustrate the encoding of memory model constraints in CHR and discuss the benefits and limitations of the proposal.

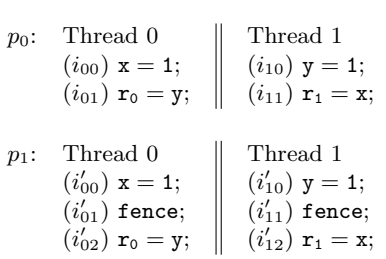
1 Introduction

Concurrent programs are hard to design and implement, especially when running on multiprocessor architectures. Multiprocessors implement weak memory models [2] that allow, for example, instruction reordering or store buffering. Thus multiprocessors exhibit more behaviors than Lamport's *Sequential Consistency* (SC) [9], a theoretical model where the execution of a program corresponds to an interleaving of the different threads.

Recent years have seen many works on formalization of weak memory models, both on hardware and software sides [12, 4, 10, 5, 6, 3, 1]. These formal models give us a way to compute, given a program and a memory model, the set of concrete executions allowed by the model. We can determine for example whether all executions of a program under a certain model are admitted under SC, and therefore, whether we can reason about this program using an interleaving semantics.

We propose a constraint solving based technique that allows to determine all admissible executions of a program under a memory model. This technique relies on Prolog and CHR (Constraint Handling Rules) [8]. A CHR program is a list of rules that act on a store of constraints, by adding or removing constraints in this store. The rules are activated if a combination of constraints in the store match the head of the rule (and satisfy an optional Prolog predicate called the *guard*). In our case, constraints are relations between instructions that are basically related to their order of execution. Rules are used to propagate information and to

*Work partially funded by French ANR (project SOPRANO, grant ANR-14-CE28-0020).




```
1  :- include(sc). % :-include(tso).% for SC or TSO
2
3  program_p0(Vars, [Thread0,Thread1]) :-
4    Vars    = [ x , y ],
5    Thread0 = [ (st,x,1), (ld,y,R0) ],
6    Thread1 = [ (st,y,1), (ld,x,R1) ].
7  program_p1(Vars, [Thread0,Thread1]) :-
8    Vars    = [ x , y ],
9    Thread0 = [ (st,x,1), f(any,any), (ld,y,R0) ],
10   Thread1 = [ (st,y,1), f(any,any), (ld,x,R1) ].
11
12 ?- program_p0(Vars, P), apply_model(Vars, P).
```

Figure 1: (a) Two programs p_0 and p_1, and (b) their implementations with a solving request.

produce, from a given set of relations, more relations (characterizing the instruction ordering) that were previously unknown in the execution.

The contributions of this paper include a novel CHR-based technique for detection of admissible behaviors under memory models and a constraint solver prototype[1] implementing this technique for SC, TSO and PSO models. We discuss the benefits and limitations of this approach, and show how the use of CHR allows us to define our model in a concise and intuitive way, taking advantage of a well-established specification and solving mechanism.

Outline. Section 2 presents weak memory models vs. SC model. Section 3 presents the solver, explains its functionality and optimizations, and points out some soundness and performance aspects. Section 4 discusses its benefits and limitations, and gives future work.

2 Weak Memory Models

A *memory model* describes how a program can interact with memory during its execution. Technical manuals of processors are often vague, and sometimes erroneous. Formal descriptions are necessary, but must be as abstract as possible to ease reasoning about them.

In a parallel context, Lamport [9] proposed *sequential consistency* (SC) where the semantics of the parallel composition of two programs is given by the interleaving of the executions of each program. For example, if we consider that two memory locations `x` and `y` initially contain `0` and that $\mathtt{r_i}$ are processor registers, the program p_0 in Fig. 1a can only give one of the three possible final results: either $\mathtt{r_0} = 0$ and $\mathtt{r_1} = 1$, or $\mathtt{r_0} = 1$ and $\mathtt{r_1} = 1$, or $\mathtt{r_0} = 1$ and $\mathtt{r_1} = 0$. The last result is obtained by the interleaving $(i_{10})(i_{11})(i_{00})(i_{01})$.

However, for the sake of performance, no current multiprocessor provides a sequentially consistent memory model. Models are *relaxed* or *weak* with respect to SC. In most processors, it is also possible to obtain the result $\mathtt{r_0} = 0$ and $\mathtt{r_1} = 0$, not justifiable by an interleaving. To avoid this behavior, one can use *memory barriers* or *fences* that forbid reorderings through them (see program p_1 of Fig. 1a).

There exist operational models for some weak memory models, for example [6]. However this kind of approaches is not really appropriate to capture the various existing memory models of processors and programming languages. Most approaches are based on constraints expressed as partial orders on events. In the case of memory models, the events of interest are related to memory: *read* (*load*) from, and *write* (*store*) to a memory location, as well as memory barriers.

Constraint-based approach have already been used to model memory models, see for example [11]. In this work, we propose a different approach using the particular case of CHR. Following [3], we use several relations to formalize memory models. The PO relation, for

[1] Available at http://frederic.loulergue.eu/CHR/wmm.tar.gz.

"program-order", simply expresses that an instruction appears before another one in the list of instructions defining a thread. The CO relation, for "coherency-order", is a per location total order of the write actions. The RF relation, for "read-from", determines which write operation assigned the value at a location before this value is read from this location: it associates a unique write to each load. The FR relation, for "from-read", relates each load operation with each of the write operations (if any) that overwrite the value after it was read by the load.

3 The Solver

3.1 Overview

A (candidate) execution can be determined by a total ordering (permutation) of memory stores to each location (CO), and an association of a unique store to each load (RF). (They can conflict PO, i.e. the initial order of instructions in threads.) Relations between instructions are modelled by CHR constraints. To compute admissible executions of a program under a given model, we generate all candidate executions (represented by the aforementioned relations), and filter them on-the-fly using CHR rules defined for the model. The set of candidate executions is easily generated (in our case, by a Prolog program using backtrack) by enumerating all possibilities of these relations. Without additional constraints, this set contains the executions authorized by a very weak generic model [3] allowing lots of intuitively incoherent behaviors (e.g. where, within the same thread, an old value that has been overwritten can still be read from the location after the overwriting operation).

To keep only executions allowed by the target memory model, we define CHR rules to filter out cases where the relations between instructions of a given execution are inadmissible. The principle is to compute the transitive closure of certain ordering relations (depending on the model) that the execution must satisfy according to the model. If the computed transitive closure exhibits a cycle, then, according to the model, some program instruction has to be executed strictly before itself, which is incoherent: the execution is not allowed. If this closure has no cycles, the ordering relations are admissible and the execution is allowed by the model. To efficiently use CHRs, we declare all CHR rules needed by the model before the Prolog program, so that they detect incoherent executions as early as possible during their generation.

Language. An input program is modelled by the list of declared variables and the list of lists of instructions of its threads. The implementation of p_0 and p_1 in Prolog is shown in Fig. 1b, lines 4–6, 8–10. The considered instructions are either a memory operation (load or store) or a fence. A load (resp. a store) is a tuple `(ld, loc, v)` (resp. `(st, loc, v)`) where `loc` is the read (resp. written) location and `v` its value. A fence, written `f(OP1,OP2)`, takes as parameters the types of instructions whose order we want to force. For example, by writing `f(st, ld)`, we state that any store preceding this fence is ordered strictly before any load following the fence. The notation `any` expresses "any type of operation". For example, `f(ld, any)` states that all loads preceding the fence are ordered before any instruction that follows it.

3.2 The Generic Model

A first step is to take the list of instructions of each thread and to enrich them by the thread identifier and instruction index in the list. As we perform this operation, we also create CHR constraints `i/5`, `fence/4` for memory operations and fences of the form `i(n_thread, n_inst, op, loc, v)` and `fence(n_thread, n_inst, t_op1, t_op2)`. In this model we consider PO, CO, RF, FR and barrier relations, defined by CHR constraints

```
1 :- chr_constraint rel/3, trace/3, cycle/2.
2 trace(R,Begin,End) \ trace(R,Begin,End) <=> true.
3 trace(R,Begin,End), rel(R,End,Begin) <=> cycle(R,Begin).
4 trace(R,Begin,End), rel(R,End,Next) ==> inf(Begin,Next) | trace(R,Begin,Next).
5 rel(R,Begin,End) ==> inf(Begin,End) | trace(R,Begin,End).
```

Figure 2: Cycle detection (file `cycle.pl`).

`po/2, co/2, rf/2, fr/2, barrier/2`. We extract `po` relation by reading the list of instructions of each thread and setting `po` for two consecutive memory operations in it. For example, the constraint `po(i(0,0,st,x,1),i(0,1,ld,y,R0))` will be generated to represent the program order of the two instructions i_{00} and i_{01} of thread 0 of p_0 in Fig. 1a. We also define `ipo`, the transitive closure of `po`.

The relations CO and RF are generated by enumerating all possible solutions in a straightforward way. The CO relation is obtained after generating a permutation of all stores to each memory location (with an additional store at the beginning to write *undefined*), and setting the constraints `co(i1,i2)` for any two consecutive stores `(...,i1,i2,...)` in the permutation. The RF relation associates each load to a possible origin store. For the example of execution of Sec. 2, returning $\mathtt{r}_0 = 1$ and $\mathtt{r}_1 = 0$, the constraints are `rf(i(1,0,st,y,1),i(0,1,ld,y,1))` and, as we add a store to *undefined* for initialization, `rf(i(-1,-1,st,x,undefined), i(1,1,ld,x,undefined))`. Notice that the read values (here, Prolog variables `R0,R1`) are unified with the written ones. The `-1` value of `n_thread, n_inst` indicates the initial definition of the program memory state.

The relation FR is computed by two rules: `rf(ST,LD), co(ST, ST2) ==> fr(LD,ST2)` and `fr(LD,ST), co(ST,ST2) ==> fr(LD,ST2)`. The first one means "if there is a RF-relation between `ST` and `LD`, and there is a CO-relation between `ST` and `ST2`, then add a FR-relation between `LD` and `ST2`". The second adds the subsequent overwritings, if any.

Finally, we have some CHR rules that, taking the set of instruction constraints and the set of fence constraints, produce `barrier` constraints that force order on instructions. One example for p_1 in Fig. 1a is `barrier(i(0,0,st,x,1),i(0,2,ld,y,R0))`.

3.3 Cycle Detection

Basically a memory model is defined by the relations between instructions that are allowed by the model. As these relations between two instructions express which one appears before the other, we can transitively produce all chains of dependencies. If a chain is a cycle, the execution exhibits an incoherence, since it means that an instruction happens before itself. The detection of a cycle is done by the CHR code of Fig. 2.

We use three CHR constraints. Constraint `rel(R,Begin,End)` states that instructions `Begin, End` are related by relation `R`. The transitive closure `trace(R,Begin,End)` means that `Begin, End` are transitively related by a trace, that is, a chain of relations `R`, while `cycle(R,Begin)` indicates that a cycle is found from `Begin` to itself.

The first rule (line 2) is a *simpagation* rule, written `A \ B <=> C`. The meaning is "if there exist two constraints `A` and `B`, add `C`, keep `A` and remove `B`". So here, the goal is to remove duplicate traces with the same origin and end, as they will generate the same final traces.

The *simplification* rule at line 3 expresses that, if we have a trace from `Begin` to `End`, and we find a relation between `End` and `Begin`, we have to replace these two constraints by a new one that indicates the existence of a cycle starting from `Begin`.

```
1 :- include(generic_model).
2 :- include(cycle).
3 :- chr_constraint sc/2.
4
5 sc(I0,I1) ==> rel(sc, I0, I1).
6
7 sc(I0,I1), sc(I0,I1) <=> sc(I0,I1).
8 po(I0,I1) ==> sc(I0,I1).
9 co(I0,I1) ==> sc(I0,I1).
10 rf(I0,I1) ==> sc(I0,I1).
11 fr(I0,I1) ==> sc(I0,I1).
```

```
1 :- include(generic_model).
2 :- include(uniproc).
3 :- chr_constraint rfe/2 , ppo/2, tso/2.
4 % po-WR pairs are not preserved by TSO
5 ppo(i(_,_,st,_,_), i(_,_,ld,_,_)) <=> true.
6 ipo(I0,I1) ==> ppo(I0,I1).
7
8 ext(i(T0,_,_,_,_), i(T1,_,_,_,_)):- \+ T0 = T1.
9 rf(I0,I1) ==> ext(I0,I1) | rfe(I0, I1).
10
11 tso(I0,I1) ==> rel(tso, I0, I1).
12 barrier(I0,I1) ==> tso(I0,I1).
13 ppo(I0,I1)     ==> tso(I0,I1).
14 rfe(I0,I1)     ==> tso(I0,I1).
15 co(I0,I1)      ==> tso(I0,I1).
16 fr(I0,I1)      ==> tso(I0,I1).
```

Figure 3: Solver files for memory models (a) SC and (b) TSO

The third rule produces the transitive closure by adding longer traces whenever it finds a relation to continue an existing trace, unless it leads to a cycle since the cycle detection is fired by the previous rule. The order of rules is thus essential for performance. To optimize the search further and to limit the quantity of traces we work on, we consider an arbitrary total order `inf` on instructions, and we add an instruction to a growing trace only if it is strictly greater (w.r.t. `inf`) than the origin of the trace. Indeed, since we compute traces from every instruction at the same time, to detect any cycle it is sufficient to compute traces going only through instructions strictly greater than the trace origin (cf. Sec. 3.5). We ensure this behavior by the CHR syntax `R ==> Guard | N` which says "if we match `R`, add `N` only if `Guard` is true".

The fourth rule selects every known relation between two instructions and adds it as a cycle candidate `trace` (using the same optimization by `inf` as above).

If we wish to keep only allowed executions, we can optimize the constraint filtering even further and add another rule `cycle(_,_) <=> false` that fails whenever a cycle is detected. Thus, inadmissible executions will be rejected as soon as detected.

3.4 Formalization of a Memory Model

To formalize a model we first identify the relations that it preserves and we create a new relation that we name according to the model acronym. Each occurrence of a preserved relation will create a new constraint with this name. The goal is then to produce the transitive closure and to launch the detection of cycles for this constraint.

For example, SC preserves the relations PO, CO, RF and FR and it does not care about barriers (since instructions are necessarily kept in order due to PO). We will name the new relation `sc`. Fig. 3a shows how we model it with CHR. Line 5 launches the computation of the transitive closure. Basically, every time we find a constraint `sc(A,B)` we add a new constraint `rel(sc, A, B)`. The cycle detection is provided by the inclusion of `cycle.pl`, line 2. Cycle rules and `rel` rule are added before any other rule about SC, to ensure the earliest detection of cycles. Lines 7–11 show how to state that `sc` preserves other relations. Every time such a relation is met, we add a new `sc`-relation.

Another interesting case is when the target model is more relaxed, that is, when the model preserves only some relations of the generic model. For example the TSO model will relax two kinds of relations: program order when the instructions are a store followed by a load, and read-from when the two instructions are in the same thread. The reordering propagates

transitively: multiple stores can be reordered after consecutive loads. If the developer wants to restore these relations, they will have to add fences. This model is illustrated in Fig. 3b.

We add a relation named `ppo` ("preserved-program-order"). Every relation `ipo` will generate a relation `ppo` except the ones mentioned before. So we add the rule on line 5, that replaces every constraint `ppo(store, load)` by "true", which means that we just remove it, and a rule (line 6) that will generate `ppo` from `ipo`. We add another relation named `rfe` ("read-from-external"). The associated rule (lines 8–9) builds `rfe`-relations from `rf` by only adding those that concern instructions in different threads. Finally, we create the `tso` relations as for SC. Again, the order of rules here is essential for both soundness and performance (cf. Sec. 3.5).

Most weak memory models respect a coherency between communications (CO, RF, FR) and PO per location. In essence, it restores the uni-processor coherency (e.g. it forbids to read an old value from an overwritten location, that is allowed by the generic model, as mentioned in Sec. 3.1). It is formalized in the included `uniproc` file (not detailed here), which includes and relies on `cycle`, so we do not have to include it again.

The implementation of the PSO model is quite straightforward once TSO is implemented. It consists in weakening the preserved-program-order a bit more by removing all `ipo`-pairs starting with a store instead of only removing store-load pairs. Concretely, we replace the rule line 5 in the TSO model by the following one: `ppo(i(_,_,st,_,_), _) <=> true`.

Applying the solver to programs. The application of the solver for a given model to a program is performed automatically as illustrated in Fig 1b for program p_0 described in Sec. 2. We include the target model, that already includes the generic model with all other necessary definitions. The program is loaded using two variables (variables involved in the program and lists of instructions of threads). Applying the target model solver to the program (line 13) will generate all candidate executions, and the CHR rules of the model will determine if each execution is allowed by the model or not.

In this case the SC model detects 3 admissible executions for both programs p_0 and p_1. The TSO model allows 4 admissible executions for p_0, while for p_1, as we have fences, some relations will be restored and TSO will allow only 3 executions (exactly as SC). TSO allows more executions than SC on p_0 since ST/LD pairs of instructions can be reordered (that can lead to the result $\mathtt{r}_0 = 0$ and $\mathtt{r}_1 = 0$). In p_1, we restore the sequentially consistent behavior of p_0 by adding fences, preventing reordering even in the TSO model.

3.5 Discussion on Soundness and Performance

While using CHR, the order of rules can be essential for both soundness and performance. We emphasize two particular points regarding the order of rules in the proposed solver: cycle detection and computation of preserved-program-order relation `ppo`.

In cycle detection (cf. Fig. 2), we want to ensure that the solver does not miss any traces and avoids to consider equivalent traces when possible. First, as mentioned in Sec. 3.3, the rule at line 2 in Fig. 2 removes equivalent traces that have exactly the same origin and end points. As it is the first rule, we will not waste time by using the following rules for two equivalent traces since we will keep only one of them. Second, trace generation starts simultaneously from all known relations `rel(R,Begin,End)` where instruction `Begin` is less than `End`, while other instructions are added at the end of a trace only if they are greater than its origin (lines 4–5). So, every cycle has a minimal instruction from which we can find this cycle as a trace from an origin through greater instructions followed by a return to the origin, cf. line 3. (Notice that we still need to consider subtraces of the same cycle when they start from other nodes.) Third, the addition of an instruction to a trace (line 4) is done by adding a new constraint, without

removing the "old" ones that can still be necessary to generate other grown traces, so we do not miss paths.

When defining a relation such as preserved-program-order `ppo` (cf. Sec. 3.4), there are two ways to proceed: to define preserved relations or to define removed relations. The first way is always sound (as we only add information), while the second one requires more care. In particular, it is important to place the rule which removes relaxed relations before the rules that generate the constraints for cycle detection (see e.g. lines 5–6, 11, 13 in Fig. 3b). Otherwise, these rules would generate constraints for cycle detection before the relaxed relation is actually removed from the store and could thus forbid allowed executions.

Experiments. We tested our solver for different models against 18 examples[2] provided for the implementation of a dedicated tool Herd [3]. On these small (but representative) examples, our solver returns the same allowed executions as those found by Herd.

Execution time on these examples being too fast, we also compare performances of our solver with Herd on some examples involving a series of (possibly wrong) message passing. Experiments have been performed on an Intel Core i7-4800MQ, 4 cores, 2.7Ghz, 16Go RAM. For example, the following code involves three message passing:

```
1 mp3(V, [T0,T1,T2]) :-
2   V  = [ x, m ] ,
3   T0 = [(st,x,1), (st,m,1), (ld,m,M0),(ld,x,X0)],
4   T1 = [(ld,m,M1),(ld,x,X1),(st,x,2) ,(st,m,2) ],
5   T2 = [(ld,m,M2),(ld,x,X2),(st,x,3) ,(st,m,3) ].
```

In such an example, the typical question is: if we get `M0 = 3`, `M1 = 1` and `M2 = 2`, do we get `X0 = 3`, `X1 = 1` and `X2 = 2`?

As it is composed of multiple writes and reads to the same locations, the combinatorial explosion is very fast. For each model, we indicate the number of allowed executions and the time needed to compute them with our CHR-based solver and with Herd. The timeout is fixed to 1 hour. The number of executions indicated for the generic model corresponds to the number of candidate executions. We present here the results for 3 and 4 message passings (denoted 3MP and 4MP).

Model	Data	3MP	4MP
SC	#exec	678	81 882
	CHR	3.3s	747s
	Herd	5.5s	> 1h
TSO	#exec	800	96 498
	CHR	3.2s	752s
	Herd	4.1s	> 1h
PSO	#exec	2 258	516 030
	CHR	6.4s	2796s
	Herd	3.8s	> 1h
Generic	#exec	147 436	255 000 000
	CHR	3.3s	> 1h
	Herd	1.2s	1405s

In this benchmark, our solver is configured to immediately reject forbidden executions as soon as they are detected, while Herd does not perform such early rejections. When combinatorial explosion becomes really big, our solver computes allowed executions faster than Herd thanks to an early pruning of the search tree.

4 Conclusion and Future Work

We have presented an original CHR-based solver for detection of admissible executions of a given program w.r.t. a given memory model, and illustrated it for SC, TSO and PSO models.

[2] Examples available at http://virginia.cs.ucl.ac.uk/herd/ (record "armed cats").

It is suitable for a rigorous exhaustive analysis of program executions of small programs that becomes intractable for bigger ones due to the combinatorial explosion of their number.

We think that seeing memory models as constraints over executions is well adapted. The design of such a solver is convenient and pragmatic. The generation of basic executions and cycle detection relies on a few optimizations in order to be more efficient and to ensure on-the-fly filtering of constraints. The proposed approach makes the definition of specific models from the generic one very practical and relatively straightforward. In particular, it is not very hard when the model becomes more complicated, as for models like ARM for example. CHR provides an easy way to express constraints about execution of programs, they have also been used for detection of incorrect behaviors in imperative program analysis in [7].

Moreover, the use of a well established mechanism of constraint specification and solving, here Prolog and CHR, brings the benefit of years of optimization and debugging to handle our problem without having to re-develop constraint resolution. We do not claim that our implementation of this problem is most efficient. Dedicated tools like [3] could be faster since they are specialized for this precise problem and can implement a solving engine without being as generic as CHR. But such dedicated tools are harder to develop as they potentially require a new optimized code that has to be carefully developed and debugged.

In future work, we plan to extend the solver to other models, like ARM. It would be interesting to support other kinds of instructions (e.g. binary operations) to handle all kinds of data or address dependencies often used for synchronization in ARM. Another direction would be to experiment on different programs of various sizes to produce precise benchmarks, in order to compare the solver to dedicated tools, as well as to further optimize it.

Acknowledgment. The work of the first author was partially funded by a Ph.D. grant of the French Ministry of Defence. Thanks to the anonymous referees for their helpful comments.

References

[1] Tatsuya Abe and Toshiyuki Maeda. Optimization of a general model checking framework for various memory consistency models. In *PGAS 2014*, 2014.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[3] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Prog. Lang. Syst.*, 2014.

[4] Arvind and Jan-Willem Maessen. Memory model = instruction reordering + store atomicity. In *ISCA 2006*, 2006.

[5] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI 2008*, 2008.

[6] Gérard Boudol and Gustavo Petri. Relaxed memory models: An operational approach. In *POPL 2009*, 2009.

[7] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Program verification using constraint handling rules and array constraint generalizations. In *VPT 2014, co-located with CAV 2014*, pages 3–18, 2014.

[8] Thom Fruhwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 1998.

[9] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program. *IEEE Trans. Comput.*, 1979.

[10] Vijay A. Saraswat, Radha Jagadeesan, Maged M. Michael, and Christoph von Praun. A theory of memory models. In *PPoPP*, pages 161–172. ACM, 2007.

[11] Emina Torlak, Mandana Vaziri, and Julian Dolby. MemSAT: checking axiomatic specifications of memory models. In *PLDI*, pages 341–350, 2010.

[12] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. UMM: An operational memory model specification framework with integrated model checking capability. *Concurr. Comput. : Pract. Exper.*, 17(5-6):465–487, 2005.

A Constraint Solving Problem Towards Unified Combinatorial Interaction Testing

Hanefi Mercan and Cemal Yilmaz

Faculty of Engineering and Natural Sciences,
Sabanci University, Istanbul, Turkey
{hanefimercan,cyilmaz}@sabanciuniv.edu

Abstract

Combinatorial Interaction Testing (CIT) approaches aim to reveal failures caused by the interactions of factors, such as input parameters and configuration options. Our ultimate goal in this line of research is to improve the practicality of CIT approaches. To this end, we have been working on developing what we call *Unified Combinatorial Interaction Testing* (U-CIT), which not only represents most (if not all) combinatorial objects that have been developed so far, but also allows testers to develop their own application-specific combinatorial objects for testing. However, realizing U-CIT in practice requires us to solve an interesting constraint solving problem. In this work we informally define the problem and present a greedy algorithm to solve it. Our gaol is not so much to present a solution, but to introduce the problem, the solution of which (we believe) is of great practical importance.

1 Introduction

Software systems frequently embody a wide spectrum of system variabilities that require testing, such as software and hardware configuration options, user inputs, and thread interleavings. However, exhaustively testing all possible variations in a timely manner (if not impossible at all) is generally far beyond the available resources for testing [15]. For this reason, the testing of modern software systems almost always involve sampling enormous variability spaces and testing representative instances of a system's behavior. In practice, this sampling is commonly performed with techniques collectively referred to as combinatorial interaction testing (or CIT) [15, 10].

CIT approaches work by first defining a model of the system's variability space. This model typically includes a set of factors, each of which takes its value from a particular domain, and a (possibly empty) set of inter-option constraints, each of which invalidates certain factor value combinations, as not all possible combinations may be valid in practice. Based on this model, CIT then generates a sample, meeting a specified *coverage criterion*. That is, the sample contains some specified combinations of factors and their values. For instance, a *t-way covering array*, which is a well-known and frequently-used CIT object, requires that each valid combination of factor values for every combination of t factors, appears at least once in the sample [2]. Here, t is often referred to as the coverage strength.

The basic justification for using CIT is that they can (under certain assumptions) effectively and efficiently exercise all system behaviors caused by the interactions of t or fewer factors. The effectiveness of CIT stems from the coverage properties it provides; e.g., all required t-way combinations of factor values are guaranteed to be covered at least once. The efficiency, on the other hand, stems from the fact that a test case can cover more than one required combination. For example, a configuration composed of n configuration options covers $\binom{n}{t}$ different t-way option setting combinations. That is, testing a single configuration has the potential of testing all $\binom{n}{t}$ combinations. Therefore, carefully generating test cases, such that a full coverage under the given coverage criterion is obtained using a minimum number of test cases can, for example, decrease the cost of testing.

The results of many empirical studies suggest that majority of factor-related failures in practice are caused by the interactions of only a small number of factors. That is, t is small in practice, typically $2 \leq t \leq 6$ with t=2 (i.e., pairwise testing) being the most common case [3, 2, 4, 5]. For a fixed t, as the variability space grows (as the number of factors increases, for example), the size of a CIT object represents an increasingly smaller proportion of the whole space. Thus, very large spaces can efficiently be covered. Consequently, CIT has been successfully used in many application domains, including systematic testing of network protocols [1, 12], input parameters [11], configurations [8, 14], software product lines [7], multi-threaded applications [9], and graphical user interfaces [16].

All these have so far been achieved by having researchers develop specific models for defining variability spaces together with specific coverage criteria to be used for testing, which in turn led to the development of novel CIT objects for every unique testing scenario. However, when the testing scenarios encountered in practice deviate from the ones addressed by researchers, practitioners often have profound difficulties in using these existing CIT objects [15]. As a matter of fact, even small changes in variability spaces and/or coverage criteria, may render existing CIT objects useless. For example, the very first variants of covering arrays supported only pairwise testing (where t=2) of binary factors. That is, when t>2 and/or when factors had varying number of values, these CIT objects were useless. Then, new CIT objects were developed to handle scenarios, in which factors might have different number of values and the covering arrays could be created for $t \geq 2$. But then these new objects suffered in the presence of inter-option constraints. New combinatorial objects were developed to handle system-wide constraints, but they then suffered in the presence of test case-specific inter-option constraints, which led to the development of test case-aware covering arrays [13]. This trend has been going on for decades now. And the bad news is that every time the variability space and/or the coverage criterion changes, a new CIT object needs to be defined, which in turn necessitates the development of specialized construction approaches, algorithms, and tools to compute these objects. Clearly, all these have been greatly hindering the applicability of CIT in practice.

We conjecture that the flexibility, thus the applicability of CIT in practice, would greatly be improved, if there were better tools that allowed practitioners to define their own application-specific variability spaces as well as their own application-specific coverage criteria. That is, rather than we, as researchers, invent new CIT objects for testing and ask practitioners to use them, thus telling them what to test, we would like to enable practitioners to define their own space for testing as well as their own coverage criterion, thus enabling them to invent their own application-specific CIT objects. Our goal as researchers is then to develop powerful tools to efficiently and effectively sample the given space to obtain full coverage under the given criterion. Although such generic tools may not be as efficient as their specialized counterparts, they certainly can provide the flexibility needed in practice. We call this approach *Unified Combinatorial Interaction Testing* (or U-CIT).

In this work we informally introduce U-CIT, which enables practitioners to define their own variability spaces and coverage criteria for testing, and present a unified construction approach to compute specific instances of U-CIT objects.

2 Unified Combinatorial Interaction Testing (U-CIT)

Definition 1. *A U-CIT requirement is an entity that needs be covered. In U-CIT, requirements are expressed as constraints.*

For example, for scenarios, in which standard covering arrays are used for testing highly configurable systems, a U-CIT requirement corresponds to a t-tuple to be covered, where a t-tuple is a set of option-setting pairs for a combination of t distinct configuration options.

Definition 2. *A U-CIT test case is a collection of U-CIT test requirements that can be tested together, i.e., a set of constraints that can be satisfied together.*

In our running scenario, for example, a U-CIT test case corresponds to a system configuration, which is indeed an n-tuple, where n is the number of configuration options.

Definition 3. *A U-CIT space model is a system of constraints that implicitly define the space of all valid U-CIT requirements as well as all valid U-CIT test cases, as not all possible combinations of U-CIT requirements may be valid in practice.*

For our running scenario, a U-CIT space model specifies that 1) every configuration option must have a valid setting in a configuration, 2) a valid U-CIT requirement is a valid t-tuple that does not violate any inter-option constraints, and 3) a valid U-CIT test case is valid configuration that does not violate any inter-option constraints.

Definition 4. *A U-CIT coverage criterion is a criterion that implicitly defines all valid U-CIT requirements that need to be covered.*

For example, the U-CIT coverage criterion for our running scenario states that all valid t-tuples must be covered at least once.

U-CIT takes as input a U-CIT space model and a U-CIT coverage criterion and as output computes a U-CIT object, e.g., a set of valid U-CIT test cases, which achieves a full coverage under the given criterion. Although it is possible to define additional constraints on the emergent properties of the resulting objects, such as the objects must achieve a full coverage with the "minimum" possible testing cost [6], we, for this work, assume one such emergent constraint which aims to minimize the number of test cases required for full coverage.

What makes a U-CIT approach a unified approach is that requirements to be covered, test cases, and the space from which the test cases will be drawn, are all expressed as constraints. Consequently, the problem of computing a U-CIT object turns into one big, interesting constraint solving problem. Note that we use the term "constraint" in the general sense; any restriction, independent of the logic in which it is specified, is considered to be a constraint. In other words, no matter whether the constraints are specified using Boolean logic, first-order logic, temporal logic, etc., the proposed approach will work as long as an appropriate constraint solver is provided.

Algorithm 1 An algorithm for computing U-CIT objects

Input: A U-CIT space model M, a U-CIT coverage criterion C
Output: A U-CIT object O

▷ Determine all valid U-CIT requirements
$R \leftarrow \{\}$
for each requirement r implied by C **do**
    if $isSatisfiable(r \wedge M)$ **then**
        $R \leftarrow R \cup r$
    end if
end for

▷ Compute a "minimum" number of satisfiable subsets
$S \leftarrow \{\}$
for each $r \in R$ **do**
    $accommodated \leftarrow false$
    for each $R' \in S$ **do**
        if $isSatisfiable(r \wedge M \wedge \bigwedge_{r' \in R'} r')$ **then**
            $R' \leftarrow R' \cup \{r\}$
            $accommodated \leftarrow true$
            break
        end if
    end for
    if not *accommodated* **then**
        $S \leftarrow S \cup \{\{r\}\}$
    end if
end for

▷ Generate the actual test cases
$O \leftarrow \{\}$
for each $R' \in S$ **do**
    $\tau \leftarrow solve(M \wedge \bigwedge_{r' \in R'} r')$
    $O \leftarrow O \cup \tau$
end for
return O

3 Constraint Satisfaction Problem

A U-CIT coverage criterion effectively defines a set of constraints to be satisfied (not necessarily all together, but in groups), each of which represents a U-CIT requirement. Given the requirements to be covered and a U-CIT space model further constraining the variability space from which the U-CIT test cases will be drawn, the constraint satisfaction problem we need to solve is to divide the requirements into a minimum number of non-overlapping sets of requirements, such that within each set, the constraints representing the requirements in the set as well as the model constraints are satisfiable together. In effect, a solution for each set represents a valid U-CIT test case, i.e., a collection of U-CIT requirements that can be tested together. Therefore, the test cases generated for all the sets, represent a U-CIT object achieving full coverage under the given coverage criterion. In particular, by reducing the number of non-overlapping sets, thus the number of test cases, we attempt to reduce testing costs.

4 A Greedy Approach for Computing U-CIT Objects

In this section we present a greedy algorithm (Algorithm 1) to compute U-CIT objects. Given a U-CIT space model M and a coverage criterion C, we first determine all valid U-CIT requirements R (lines 2-7). To this end, we enumerate all the entities to be covered, convert each entity to a constraint r, and then determine whether $r \wedge M$ is satisfiable (line 4). If it is, then r is added in R (line 5). Otherwise, r is invalid.

Once the set of valid requirements R is determined, we divide it into non-overlapping satisfiable subsets S, covering all requirements (lines 9-23). To this end, we start with an empty pool of subsets (line 10). Then, for each requirement r in R, we attempt to accommodate it in an existing subset in the pool (line 14). If such a subset is found, we include r in the subset (line 15). If not, we populate the pool with an initially empty subset and then include r in the newly added subset (line 21). Note that a subset of requirements R' in this context is specified as the logical conjunction of all the requirements included in the set, i.e., $\bigwedge_{r' \in R'} r'$. Consequently, to determine whether a new requirement r can be accommodated in an existing subset R', we solve the respective constraints together with M, i.e., $r \wedge M \wedge \bigwedge_{r' \in R'} r'$ (line 14), if the resulting constraint is satisfiable then we include r in R' (line 15).

After determining the subsets S, to compute the U-CIT object O, we generate a test case by solving the logical conjunction $M \wedge \bigwedge_{r \in R'} r$ for each subset R' (line 28). The set of test cases are then guaranteed to obtain full coverage under the coverage criterion C.

Not that we provide this algorithm as a proof-of-concept algorithm for computing U-CIT objects. That is, in the development of this algorithm, our major concern was correctness, not performance. Consequently, the proposed algorithm suffers from some drawbacks. One issue is that being a greedy algorithm, it may yield locally optimal solutions. Another issue is that the same constraints may end up being solved repeatedly, which may cause scalability issues.

5 An Example: Specifying and Computing Standard Covering Arrays as U-CIT Objects

In this section, we illustrate U-CIT on a hypothetical system by providing details about how our running scenario, in which standard covering arrays are used for configuration testing, can be handled by U-CIT. The example is kept as simple as possible on purpose. In general, the complexity of encodings depends on the complexity of the system under test and/or complexity of the application domain.

Without losing the generality of the proposed approach, the system under test we use in our example has three binary configuration options (o_1, o_2, and o_3), each of which takes the setting of $true$ or $false$, together with two inter-option constraints: $o_2 = true \rightarrow o_3 = true$, invalidating the combination ($o_2 = true$, $o_3 = false$), and $\neg(o_1 = true \wedge o_3 = false)$, invalidating the combination ($o_1 = true$, $o_3 = false$). Furthermore, the system is to be tested using a 2-way covering array, i.e., all valid 2-tuples must be covered at least once.

For this example, the U-CIT space model M can be expressed in Boolean algebra as $(\neg o_2 \vee o_3) \wedge (\neg o_1 \vee o_3)$, where each configuration option is represented by a Boolean variable. In this encoding, each U-CIT requirement simply becomes a Boolean formula representing a 2-tuple. For example, the 2-tuple ($o_1 = false$, $o_2 = true$) is expressed as $(\neg o_1 \wedge o_2)$. To determine whether a U-CIT requirement is valid or not, it is checked whether the respective Boolean formula is satisfiable with the U-CIT space model M. For example, since $(\neg o_1 \wedge o_2) \wedge M$ is

(o_1, o_2)	(o_1, o_3)	(o_2, o_3)
$r_1 : \neg o_1 \wedge \neg o_2$	$r_5 : \neg o_1 \wedge \neg o_3$	$r_8 : \neg o_2 \wedge \neg o_3$
$r_2 : \neg o_1 \wedge o_2$	$r_6 : \neg o_1 \wedge o_3$	$r_9 : \neg o_2 \wedge o_3$
$r_3 : o_1 \wedge \neg o_2$		
$r_4 : o_1 \wedge o_2$	$r_7 : o_1 \wedge o_3$	$r_{10} : o_2 \wedge o_3$

Table 1: All valid U-CIT requirements.

satisfiable, the 2-tuple ($o_1 = false$, $o_2 = true$) is a valid 2-tuple, thus a valid U-CIT requirement. For the same reason, ($o_2 = true$, $o_3 = false$) is not a valid U-CIT requirement.

The first part of Algorithm 1 (lines 2-7), which determines all valid U-CIT requirements, would then generate the 10 U-CIT requirements $r_1, \cdots, r_{10}$ given in Table 1.

	2-way covering array		
$S = \{R'_1, R'_2, R'_3, R'_4\}$	o_1	o_2	o_3
$R'_1 = \{r_1, r_5, r_8\}$	false	false	fase
$R'_2 = \{r_2, r_6\}$	false	true	true
$R'_3 = \{r_3, r_7, r_9\}$	true	false	true
$R'_4 = \{r_4, r_{10}\}$	true	true	true

Table 2: The set of requirements divided into non-overlapping satisfiable subsets of requirements $S = \{R'_1, R'_2, R'_3, R'_4\}$ (left column), and the respective standard 2-way covering array generated (right column).

Next, the set of valid requirements is divided into non-overlapping satisfiable subsets by the second part of Algorithm 2 (lines 10-23). Assuming that the requirements in Table 1 are processed in the order $r_1, \cdots, r_{10}$, the first requirement we process becomes $r_1 : (\neg o_1 \wedge \neg o_2)$. Since the set S is initially empty, we create a new subset $R'_1 = \{r_1\}$ and populate S with R'_1, i.e., $S = \{R'_1\}$. For the second requirement $r_2 : (\neg o_1 \wedge o_2)$, since $r_1 \wedge r_2 \wedge M$, i.e., $(\neg o_1 \wedge \neg o_2) \wedge (\neg o_1 \wedge o_2) \wedge M$, is not satisfiable, r_2 cannot be placed in R'_1. So, we create a new subset $R'_2 = \{r_2\}$ and S becomes $\{R'_1, R'_2\}$. After processing r_4, we would have four subsets R'_1, R'_2, R'_3, and R'_4 in S, containing requirements r_1, r_2, r_3, and r_4, respectively. For requirement $r_5 : (\neg o_1 \wedge \neg o_3)$, as $r_1 \wedge r_5 \wedge M$, i.e., $(\neg o_1 \wedge \neg o_2) \wedge (\neg o_1 \wedge \neg o_3) \wedge M$, is satisfiable r_5 is placed in R'_1 together with r_1. After processing all the requirements, we would have the four subsets of satisfiable U-CIT requirements given in the left column of Table 2.

Finally, in the last part of Algorithm 1 (lines 26-30), for each subset of U-CIT requirements in S, we generate a U-CIT test case, which in this case corresponds to a valid configuration, by solving the requirements in the subset together with the U-CIT space model M. For example, for R'_1 (Table 2), solving $r_1 \wedge r_5 \wedge r_8 \wedge M$ produces the configuration ($o_1 = false, o_2 = false, o_3 = false$). Solving all the subsets would then generate the U-CIT object given in the right column of Table 2, which is indeed a standard 2-way covering array – a set of configurations that covers each valid 2-tuple at least once.

6 Conclusion and Future Work

We believe that U-CIT can greatly improve the flexibility of combinatorial interaction testing in practice. Therefore, efficient and affective approaches for solving the constraint satisfaction

problem we informally introduced in this paper, are of great practical importance. Therefore, we keep on developing languages and model-based tools for defining variability spaces and coverage criteria in a generic manner as well as developing efficient and effective approaches for computing U-CIT objects.

References

[1] Kirk Burroughs, Aridaman Jain, and Robert L. Erickson. Improved quality of protocol testing through techniques of experimental design. In *Communications, 1994. ICC'94, SUPERCOMM/ICC'94, Conference Record,'Serving Humanity Through Communications.'IEEE International Conference on*, pages 745–752. IEEE, 1994.

[2] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23(7):437–444, 1997.

[3] David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE software*, 13(5):83–88, 1996.

[4] Jacek Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. In *Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer*, pages 419–430, 2006.

[5] Siddhartha R. Dalal, Ashish Jain, Nachimuthu Karunanithi, J. M. Leaton, and Christopher M. Lott. Model-based testing of a highly programmable system. In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, pages 174–179. IEEE, 1998.

[6] Gulsen Demiroz and Cemal Yilmaz. Cost-aware combinatorial interaction testing. In *Proceedings of the Internatinoal Conference on Advances in System Testing and Validation Lifecycles*, pages 9–16, 2012.

[7] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 46–55. ACM, 2012.

[8] Rick Kuhn, Yu Lei, and Raghu Kacker. Practical combinatorial testing: Beyond pairwise. *IT Professional*, 10(3):19–23, 2008.

[9] Yu Lei, Richard H Carver, Raghu Kacker, and David Kung. A combinatorial testing strategy for concurrent programs. *Software Testing, Verification and Reliability*, 17(4):207–225, 2007.

[10] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11, 2011.

[11] Patrick J. Schroeder, Pat Faherty, and Bogdan Korel. Generating expected results for automated black-box testing. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 139–148. IEEE, 2002.

[12] Alan W. Williams and Robert L. Probert. A practical strategy for testing pair-wise coverage of network interfaces. In *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*, pages 246–254. IEEE, 1996.

[13] Cemal Yilmaz. Test case-aware combinatorial interaction testing. *Software Engineering, IEEE Transactions on*, 39(5):684–706, 2013.

[14] Cemal Yilmaz, Myra B. Cohen, Adam Porter, et al. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on*, 32(1):20–34, 2006.

[15] Cemal Yilmaz, Sandro Fouche, Myra B. Cohen, Adam Porter, Gulsen Demiroz, and Ugur Koc. Moving forward with combinatorial interaction testing. *Computer*, 47(2):37–45, 2014.

[16] Xun Yuan, Myra B. Cohen, and Atif M. Memon. Gui interaction testing: Incorporating event context. *Software Engineering, IEEE Transactions on*, 37(4):559–574, 2011.

Towards Automated Bounded Model Checking of API Implementations.

Daniel Neville[1,2,3], Andrew Malton[1,4], Martin Brain[2,5], and Daniel Kroening[2,6]

[1] CHACE Centre for High Assurance Computing, Blackberry Ltd., Waterloo, Canada
[2] University of Oxford, Oxford, U.K.
[3] daniel.neville@cs.ox.ac.uk
[4] amalton@blackberry.com
[5] martin.brain@cs.ox.ac.uk
[6] kroening@cs.ox.ac.uk

Abstract

We introduce and demonstrate the viability of a novel technique for verifying that implementations of application program interfaces (APIs) are bug free. Our technique applies a new abstract interpretation to extract an underlying model of API usage, and then uses this to synthesise a set of verifiable program fragments. These fragments are evaluated using CBMC and any potentially spurious property violation is presented to a domain expert user. The user's response is then used to refine the underlying model of the API to eliminate false positives. The refinement-analysis process is repeated iteratively. We demonstrate the viability of the technique by showing how it can find an integer underflow within Google's Brotli, an underflow that has been shown to lead directly to allow remote attackers to execute arbitrary code in CVE 2016-1968.

1 Introduction

Applying automated formal methods and bug checking to application program interfaces (APIs) is a difficult and challenging process. An arbitrary set of API functions has no obvious entrance point, structure or ordering visible from the raw API code itself, despite the underlying API implementation often requiring a specific ordering of API calls to execute meaningfully. These underlying usage requirements lead to a largely manual approach being preferred by many [10], often by construction of verifiable program fragments [3]. A *fragment* is a sequence of API calls using non-deterministic input that can be verified by a formal methods engine. Because the manual construction of these fragments is time-consuming and hence expensive, our work specifically targets systems where developing a comprehensive test suite is infeasible. Manual processes are also more liable to human error.

There are many complexities to contend with when writing tests. At the highest level, the test author must ensure that the test's sequence of API calls is realistic [12]. Within this work, "realistic" refers to how a typical end-user would use the API for their own development. It is important to focus on verifying realistic code because bugs that are potentially exploitable in a

normal usage pattern of the API are of arguably more interest than those that require unrealistic or pathological uses of the API. This is because there is only a finite amount of verification time available, and realistic calling sequences should be prioritised for verification. After construction of a realistic API calling sequence, the test author must ensure all variables used within the tests are instantiated meaningfully, and data structures such as buffers and pointers initialized correctly. Finally, the author has to ensure the tests are suitable for verification, and constructed in a valid input language for a verification tool.

The size and scope of APIs make them particularly difficult to test and verify, and any corresponding documentation can be complicated due to the need to be precise. Each API call may take a large number of parameters, each of which may be of variable size. Conventional test suites which aim to show an API is *completely* free of unwanted behaviour are somewhat limited due to the huge range of possible inputs. Considering a function with two 64-bit long integers as input. Each long integer can take as many as 2^{64} different values, therefore a test suite would require 2^{128} different tests just to ensure complete coverage of a single function. This is clearly infeasible.

More challengingly, even small APIs have too many permutations to allow tests to be written manually. Consider an API that has 10 API calls available, and the tester wishes to choose six API calls for a single test, in some arbitrary order. There are $10!/(10-6)! = 151,200$ permutations from these requirements, too many to write manually.

Of course, all these difficulties and more apply to any automated method for API verification or API bug checking. It should be noted that, ideally, an API should be bug free even when used outside of the authors' intended usage. Despite these challenges, it should be seen as highly desirable to have an automated method of verifying that a given API is free of common bugs such as division by zero errors, memory access outside array bounds, dereferencing of invalid pointers, memory leaks and arithmetic overflows.

Contributions.

(a) This work contributes a novel refinement-based technique for API verification (Section 3).

(b) Furthermore, given appropriate input, it contributes a method that can extract the underlying usage of the API using a novel abstract interpretation technique (Section 3.2).

(c) This work contributes a devised system then uses the extracted information to synthesise verifiable C program fragments that use the API in a realistic manner (Section 3.3).

(d) Finally, it applies bounded model checking using CBMC to verify the synthesised fragments (Section 3.4). If a false alarm is detected, the counterexample trace and underlying assumptions can be presented to the user for validation (Section 3.5), to allow novel refinement of the derived model (Section 3.6).

2 Background

2.1 Related work

Existing solutions within the sphere of API implementation verification are limited. Whilst there has been investigation into testing and verifying API implementations, these generally require comprehensive tests or verifiable programs to be provided [15]. Should a user have already spent time developing a comprehensive test suite, then any common verification framework is likely to be able to perform a reasonably effective analysis over that test suite. However, as

stated previously, this work targets API implementations where developing a comprehensive test suite is either too difficult, too expensive or simply too large a project.

One previous piece of work has identified the need for gaining an understanding of the calling sequence [1], a key limitation of this approach is that it lacks a refinement feedback loop, meaning that the extracted model cannot be corrected, even if spurious counterexamples are discovered during testing. To expand upon this, our approach introduces an alternative method of extracting information via abstract interpretation, and then places this information in a more general system, one that has the ability to generate generic C programs for analysis by any analysis engine and perhaps most importantly, the ability for the user to refine their model based upon detection of false positives.

Some groups have previously used CBMC [9] for program synthesis. Di Rosa et al. [13] used CBMC to automatically generate test sets that had high branch coverage. This work is effective within the authors' domain, however it does not extend well to APIs due to the rapid state space expansion arising from API calling sequences, which in turn means only a very small subset of possible sequences can be analysed in practice. As stated previously, limited verification time means that any analysed program should be realistic and relevant, something that cannot be ensured with this technique.

Others Samak et al. [14] and Pradel et al. [11] both use test synthesis albeit for concurrent test generation. This could be a natural extension of this work.

CBMC [9] is a Bounded Model Checker for C that has the ability to verify array bounds (including buffer overflows), pointer safety, exceptions and user-specified assertions. It has very similar semantics to C; key differences include calls to declared but undefined functions are allowed, and return a non-deterministic result. Additionally, the use of declared but uninitialized variables as *rvalues* is allowed, and invokes non-determinism.

2.2 Techniques

Abstract interpretation is used for information extraction. It is a theory of sound approximation of a program [5]. In practical terms, it allows over and under-approximate conclusions to be drawn without performing all possible calculations. This balances the values of correctness and computational feasibility.

Realism spectrum. When generating verifiable fragments, it is important to generate a program which is likely meaningful under the API developers intended use case. Whilst it is not *incorrect* to generate unusual fragments like Listing 1, which may indeed turn up errors, it is relatively unlikely that a developer would intentional use the API in such a way that this will occur. It is more useful to investigate errors in *realistic* calling sequences, rather than unrealistic as these are more likely to occur in real world use.

```
api_close(x);
api_close(x);
api_close(x);
```

Listing 1: An unrealistic calling sequence.

The accuracy of a generated program as per its specification can be viewed as a spectrum. One end of the spectrum lies "complete non-determinism", where absolutely no restriction is placed upon the generated program. Anything is acceptable in a purely non-deterministic generation, but this is unlikely to reflect the API developer's intended use case and hence be a poor use of verification time.

On the other end of the spectrum lies "complete determinism", where generated programs *must* match a known calling sequence with no variation. This again is a poor approach as only witnessed input programs can be generated, which naturally cannot extend the set of verifiable programs available.

It is therefore reasonable to conclude that somewhere *within* this spectrum is the ideal place to be. Where specifically depends upon the user's intentions when using this system, although it is likely the most desirable place is somewhere near the deterministic end of the spectrum, where programs generated are similar to previously witnessed input, but sufficiently diverse such that they can exercise many different, yet realistic calling sequences.

3 Our Work

The solution has two parts: Verifiable fragment generation and model checking with refinement.

Firstly, we analyse a small sample of client code of the API. We then construct a model of realistic and typical API use based on this input, and from the model we synthesize a large number of *fragments*. These are written in standard C that can be automatically verified either with a formal verification tool that accepts non-deterministic C as input, for example CBMC.

Secondly, we apply verification to the synthesised fragments and refine the model accordingly. CBMC is used to automatically analyse the fragments according to a CBMC configuration set by the user, this allows appropriate selection of unwinding limits, etc. Should this analysis find potential bugs, an error trace is generated and presented to the user who assesses whether it is a valid and meaningful bug, or spurious. The userâĂŹs response is then used to refine the original fragments generated in Part 1. The user may direct that a derived assumption is to be discarded, modified or alternatively add a new assumption of their own if necessary. This allows flexibility, and far more accurate results than unrefined analysis.

The synthesis engine also allows a user to use an existing test system that only operates on concrete data. Support has been added for concretisation of common types. This permits the generation of fully concrete tests for use outside of formal verification.

3.1 Steps of work

Information extraction Produce an over and under-approximation of the calling sequence and calling context of the API functions given a small sample of client code of the API.

Synthesis Use the approximations to produce a synthesised fragment, which calls the API functions in the appropriate order with the right context. The fragment is exported in C. Instantiations can be non-deterministic or concrete.

Verification Using the fragment(s), attempt to verify the instrumented or developer written assertions in the API implementation using existing BMC techniques.

Validation Should a bug be detected, the counterexample trace, bug report and underlying model are presented to the user for validation.

Refinement Using the user's response, refine the abstraction and continue at **Synthesis** stage.

The key inputs to the algorithm are: a small input of client code of the API, the API header file, the API source code, a user-provided analysis location within the client code and commands for the verification engine.

The key outputs are: a model representing the calling sequence and other extracted data from the client code (to screen or serialised), synthesised verifiable program fragments and any bugs found whilst verifying the synthesised fragments.

3.2 Information Extraction

The first stage of the project is to understand the underlying usage of the API implied by the sample of client code. In particular, there is a focus on extracting likely calling sequences of the underlying API along with appropriate information regarding: the reuse of variables and other symbols between API calls, the use of constants within API calls, and how variables may be used within the body of API calls.

Data is extracted using a selection of abstract interpretation techniques. Abstract interpretation is particularly well-suited as exhaustive execution of any input files would likely be computationally infeasible, it also generates data at all locations in the program, not only the terminal states, although analysis is usually performed by extracting data at the end of the main function. Abstract interpretation allows the rapid static analysis of files, permitting the collection of necessary data in a rapid, user-friendly way.

3.2.1 Pair-wise call extraction via trace analysis

Three areas of interest are maintained to meaningfully determine the calling sequence: prefix, suffix and pair-wise ordering. Prefix refers to the first API call in a given trace, suffix refers to the final API call in a given trace and pair-wise ordering refers to how API calls are called relative to each other, e.g. `open` always immediately precedes `read`.

These are calculated over all traces in the program, with suitable merge operations used. These merge operations are namely intersection and union, and are referred to as the "Required" and "Allowed" sets respectively. In other words, required represents the domain that is valid on *all* traces through a given input and allowed represents the domain that is valid on *any* trace through a given input.

3.2.2 Prefix and suffix

The prefix and suffix represent the possible first and last API calls respectively in a given program.

Consider the program given in Listing 2. There are two paths through the program. Namely (1) `api_open_overwrite` then `api_write`, `api_close` or (2) `api_open_append`, `api_write`, `api_close`.

```
int *x;
if(non_det())
    x = api_open_overwrite("a.txt");
else
    x = api_open_append("a.txt");
api_write("Blank file", x);
api_close(x);
```

Listing 2: A simple API calling sequence.

There are two possibly initial API calls, `api_open_overwrite` or `api_open_append`. This generates an allowed prefix of the union of these two calls, and a required prefix of the intersection of these calls. Correspondingly, there is a single suffix call: `api_close`. Both the allowed and required suffix sets will contain this element.

The operations can be more rigorously defined as a set of operations to be applied during routine abstract interpretation (under a framework with similar semantics to the CProver Abstract Interpretation framework). Each program location retains its own set of information. For this trace analysis, the program is under-approximated to a set of connected basic blocks, each containing only a sequence of API calls, other operations are ignored.

	Allowed (union)	Required (intersection)
Domain	$SA : \mathcal{P}(A)$	$SR : \mathcal{P}(A)$
Merge	$SA_i := \bigcup_{n=1}^{\infty} SA_{i,n}$	$SA_i := \bigcap_{n=1}^{\infty} SA_{i,n}$
Transformer	$SA_o := \alpha$	$SA_o := \alpha$

Table 1: Suffix calculation operations.

	Allowed (union)	Required (intersection)
Domain	$PA : \mathcal{P}(A)$	$SR : \mathcal{P}(A)$
Merge	$PA_i := \bigcup_{n=1}^{\infty} PA_{i,n}$	$PR_i := \bigcap_{n=1}^{\infty} PR_{i,n}$
Transformer	$PA_o := \begin{cases} \alpha \cup PA_i, & \text{if } SA_o \text{ is empty} \\ PA_i, & \text{otherwise} \end{cases}$	$PR_o := \begin{cases} \alpha \cup PR_i, & \text{if } SR_o \text{ is empty} \\ PR_i, & \text{otherwise} \end{cases}$

Table 2: Prefix calculation operations.

For each state calculation, there is incoming data (prior to transform and merge) and outgoing data (after transform and merge). These sets are represented as SA, SR, PA, PR, and outgoing data is represented X_o. When a state has multiple incoming edges, the incoming data for all these edges is presented as $X_{i,n}$. This data is merged to form a single incoming data set (X_i) for calculation. States all initialise as empty.

Whenever an API call is present within an instruction, the transformer is used to calculate the outgoing set, otherwise the outgoing set is a copy of the incoming set.

The suffix domain can be calculated independently of all other information, and is therefore calculated first. The prefix domain relies upon information in the suffix domain.

Given a set of APIs A, with the transformer representing the behaviour when reaching an arbitrary API call α, Tables 3.2.2 and 3.2.2 show the operations performed.

3.2.3 Pair-wise API call ordering

Whilst knowing the first and last element within any arbitrary trace in a program is useful, it cannot be used alone to construct a realistic calling sequence.

To do this, a new domain is introduced, namely the pair-wise ordering domain. The domain is a set of pairs, where a pair ordering (x, y) represents that API call x precedes y on some trace, with no intermediate API call. For example, the allowed set $\{(\texttt{api\_open}, \texttt{api\_write}), (\texttt{api\_write}, \texttt{api\_close})\}$ represents that there is at least one trace where `api_open` precedes `api_write` and at least one trace (possibly the same) where `api_write` precedes `api_close`. Both allowed and required sets are maintained, using union and intersection accordingly to generate their data sets. The transformer uses information from the suffix data structure. Table 3 gives the operations.

3.2.4 Variable and Constant extraction

A common design pattern is to use a constant value to dictate a mode operator for an API. Consider `fopen` in Listing 3. The second input is a mode operator that dictates access mode for a given file [8]. It is unlikely that an experienced developer would use `fopen` in an arbitrary

	Allowed (union)	Required (intersection)
Domain	$OA : \mathcal{P}(A \times A)$	$OR : \mathcal{P}(A \times A)$
Merge	$OA_i := \bigcup_{n=1}^{\infty} OA_{i,n}$	$OR_i := \bigcap_{n=1}^{\infty} OR_{i,n}$
Transformer	$OA_o := OA_i \cup \bigcup_{a \in SA_i} \{(a, \alpha)\}$	$OA_o := OA_i \cap \bigcup_{a \in SR_i} \{(a, \alpha)\}$

Table 3: Pair-wise calculation operations.

manner, so using a completely non-deterministic character for *mode* is likely to be a poor choice. Instead, it should be initialised non-deterministically to one valid input within its realistic values set.

After constant propagation has been applied, the two techniques are looking over and looking into.

```
fopen (filename, mode);
```

Listing 3: `fopen`

The first technique, *looking over* is straight-forward: analysis is performed over the input files and identifying where a constant has been used as an argument in an API call. Given a function α with x arguments, a table is maintained for all arguments to track constants. This technique is also applied effectively to variables. This is vital as the same variable is frequently used in multiple API calls in a single calling sequence.

The second technique, *looking into* is more complex. Whilst examining the input data is interesting, and will no doubt generate realistic results, this is a limited approach. There may be other methods built into the code by the API developer that the API user is unaware of. This means it is unlikely that the API user will intentionally use these in an initial input file. It is therefore interesting to examine the range of possible meaningful values for an argument by looking forward, into the API code itself.

This is accomplished by performing analysis within the API bodies. Each API call is instantiated with non-deterministic arguments and constant propagation is applied. Whenever an input argument is compared against a constant value, these values can be extracted (with appropriate semantics) to generate a set of values that direct control flow. This technique results in data being extracted that allows Synthesis in different areas of the Realism Spectrum as detailed in Section 2.2.

3.3 Synthesis

The information extracted from the input data is then used to synthesise further tests for bounded model checking.

3.3.1 Procedure

Program synthesis has been implemented in CProver's Goto language framework. This language is particularly amenable to synthesis, thanks partly to its reduced instruction set and clear semantic structure. It has one additional instruction not included in C, `assume`. This instruction directs the verification engine not to analyse paths where an `assume` constraint evaluates to false.

Process.

1. Configuration initialisation and RNG seeding.
2. The *main* function is added to the synthesised program (SP.)

3. Known API prototypes are added to the SP.
4. Known symbols that will be necessary are added to the SP.
5. A walk [See: The Walk] is performed over the information extracted from the tests, specifically a walk is performed over the calling sequence information. This adds API calls to the SP. When each API call is added, the `assume` database is checked to see whether to insert an `assume` statement prior to API call.
6. The SP is exported as ANSI-C code to file.

The Walk.

1. An appropriate API function is chosen.
   (a) For the first call: A function that is required (or at least allowed, configuration dependent) to appear first.
   (b) For the last call: A function that is required (or at least allowed, configuration dependent) to appear last.
   (c) For any other call: A function which is required to appear following the previous call (or at least allowed, configuration dependent) or another function dictated by a configuration strategy such as randomisation.
2. The API's arguments types are extracted and evaluated, including return type.
3. Symbols are selected with appropriate instantiation to support the API call [See Symbol Selection.]
4. Instantiated pointers are pointed to appropriate data structures [with necessary recursive instantiation].

Symbol Selection. Given an arbitrary API call with n arguments and a return value k, it is essential to ensure appropriate symbols are used to ensure data is passed through the API calling sequence in a realistic method.

Symbol selection supports several different techniques. It is heavily configuration dependent to ensure good quality results.

Given API call α at position k

(a) If a suitable constant (string, numeric, etc.) is used at $\alpha : k$, add this to the consideration list.

(b) If a suitable variable is used at $\alpha : k$ that has been used previously in an API call, and there is suggestion of such a pattern in the input data: add this to the consideration list.

(c) If there is an appropriate constant or symbol available: select according to some strategy.

(d) Otherwise: Declare and instantiate a new variable in a suitably non-deterministic manner (unless full concretisation is enabled in configuration.).

Instantiation. Instantiation is non-trivial, especially for non-primitives. Consideration must be made as to how to initialise pointers, `structs` and other data structures. Because CBMC will be used to verify the synthesised programs, it is not necessary to assign a concrete value to all declared variables. For example, given the code in Listing 4, CBMC is perfectly capable of handling the inherent non-determinism.

```
int x;
int y;
int z = api_do(x, y);
```

Listing 4: A simple, valid calling sequence

However, for more complex structures such as pointers, failing to instantiate will result in trivial errors occurring, such as null pointer exceptions. This is demonstrated in Listing 5. On line 3, a deference takes place, but CBMC cannot assume that the pointers are instantiated correctly, therefore any instrumented pointer assertions will return `false`. To handle this, all pointers must be instantiated (or intentionally left null, as per configuration) before use.

Whenever a new pointer symbol is created for the purpose of being used in an API, a corresponding symbol must be created and instantiated for this new pointer symbol to point. This is demonstrated for integers in Listing 6. This will ensure both variables are suitable instantiated prior to call. This technique is also used for pointers to large objects such as buffers.

This strategy can be applied recursively for non-primitives. Where a necessary `struct` or similar is declared and then each internal primitive is initialised with the above strategy. A challenge arises when a `struct` contains a pointer to itself, as this can lead to an infinite initialisation loop, this behaviour is handled by modifying the Synthesis Engine's configuration.

```
int *x;
int *y;
int z = api_do(*x, *y);
```

Listing 5: Example of use of non-initialised pointers

3.4 Verification

Once the fragments have been generated, the user's desired testing or verification procedure is followed. The intended verification engine for this project is CProver's CBMC. The generated fragments are compiled into CProver's intermediate reduced instruction-set language (GOTO), and bug checks are instrumented into the program in the form of assertions. CBMC then attempts verification on each program under the configuration provided by the user.

For each program, if an assertion is violated, then a counterexample trace is generated for validation. If no assertion is violated then CBMC will proceed to the next program.

```
int *x;
int x_pointed_to_1;
x = &x_pointed_to_1;
int *y;
int y_pointed_to_1;
y = &y_pointed_to_1;
int z = api_do(*x, *y)
```

Listing 6: Initialisation of pointers.

3.5 Validation

If a counterexample *is* found within a fragment, it is presented to the user. The user can then manually inspect the calling sequence that lead to the bug.

If the user confirms the bug is genuine, the violated property is identified and the corresponding line of raw source code logged accordingly. The user can then review this later to ensure bugs in the underlying are fixed. Once the user believes they have removed the bug, they can re-run the same synthesised fragment to give a better indication of whether the bug is fixed. The user can then choose to continue verification or stop to fix any bugs found.

If however, the bug is not genuine, there may be a miscalculation in the underlying API calling model. Therefore, the user may wish to *refine* the model.

3.6 Refinement

To refine the model, the user is provided with the property that was violated, a counterexample that led to the bug and the underlying assumptions that were generated in the Information Extract stage of the process.

The user can at this point decide on the most appropriate action. They can ignore the error, and continue execution. They can abandon execution, and find an alternative input, or most preferably, they can refine the underlying model. The model's constraints can be added to, modified, or deleted. This process can be repeated until the user is satisfied they have removed the offending incorrect rules. After the user has refined the model, the process returns to Step 2: Synthesis. New fragments are generated using the refined model, and analysis continues.

The refinement approach borrows from the underlying principles behind counterexample guided abstraction refinement (CEGAR) [4], a specific similarity occurs when a failing proof is found, both CEGAR and this approach result in analysis of the cause of failure, specifically the counterexample trace, refining this model allows a more precise verifiable program fragment to be synthesised, and hence more effective analysis.

4 Results

To show the viability of this novel refinement-based technique a test candidate was selected from Github. Brotli is a generic-purpose lossless compression algorithm used for data compression [2]. Both an encoder and decoder are available. Specifically, the decoder was chosen for analysis.

The decoder is attractive for this type of analysis for several reasons. Firstly, Brotli's decoder is written in pure C with few external dependencies. This means its build process can easily be modified to use CProver's compiler. Thirdly, there was at least one known bug in Brotli (CVE-2016-1968 [6]) suggesting that effective verification techniques had not been applied to Brotli, this was eventually fixed by the developers in commit 37a320d [7]. Finally, the code was large and complex enough to perform meaningful and non-trivial evaluation. Brotli's decoder has 14,000 effective lines of C code, of which 2,000 are contained in C header files. Analysis was performed using an Ubuntu 64-bit virtual machine running in Oracle VM VirtualBox with 8 GB of RAM and with non-dedicated access to two cores of an Intel 4810MQ at 2.80 GHz.

Once Brotli had been configured to compile under CBMC, the code was analysed and a realistic test was written using components from the Brotli API; manual creation was necessary due to the limited test data within Brotli. This test was executed concretely, but produced no bugs or assertion violations. This sample test was then used as input for the Data Extraction Engine and Synthesis Engine, which created multiple fragments that could be used under a verification engine. These were then analysed using CBMC's signed overflow and bound checks. The CVE vulnerable property was placed under scrutiny. CBMC found a possible arithmetic underflow for the property directly associated with the exploit. The violated assertion directly corresponds to the patch added by the authors to mitigate this bug.

The performance of information extraction was rapid, even within a virtual machine environment, Step One: Information Extraction was complete within 0.4 s. 1,000 verifiable Brotli fragments could be synthesised from this data in 6s. The model was refined once to modify an instantiation.

5 Conclusions and Future Work

We have shown in this work that API analysis, despite its challenges, can be confronted using a mix of formal verification, data mining techniques and program synthesis. We have shown that underlying API usage data can be extracted rapidly, and used to synthesise verifiable fragments.

We have demonstrated a method to allow refinement of the fragments after verification to improve the model, and explained how the user can interact to improve the quality of their results quickly, without being required to manually author numerous tests themselves. Finally, we have shown that it can be applied effectively to real-world code.

To expand upon this, we have devised a new abstract interpretation technique for under-approximate information extraction using regular expressions which is being implemented, as well as the introduction of other domains (e.g. interval domain) to better represent some APIs. We are also intent on developing far richer results to show the effectiveness of the technique. Additionally, we desire the introduction of a formal grammar for synthesis. Finally, to encourage use of the technique, a user interface will be created to connect the currently distinct components and it is hoped this work will become part of the open-source CProver suite of tools.

References

[1] Mithun Puthige Acharya. *Mining API Specifications from Source Code for Improving Software Reliability*. PhD thesis, North Carolina State University, 2009.

[2] J. Alakuijala and Z. Szabadka. Brotli compressed data format. http://www.ietf.org/id/draft-alakuijala-brotli-09.txt, April 2016. [Online; accessed 17-April-2016].

[3] Michael Churchman. API testing: Why it matters, and how to do it. https://blog.udemy.com/api-testing/, April 2014. [Online; accessed 17-April-2016].

[4] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings*, chapter Counterexample-Guided Abstraction Refinement, pages 154–169. Springer, Berlin, Heidelberg, 2000.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.

[6] CVE. CVE-2016-1968. Available from MITRE, CVE-ID CVE-2016-1968., January 20 2016.

[7] Google. Brotli commit 37a320d. https://github.com/google/brotli/commit/37a320dd81db8d546cd24a45b4c61d87b45dcade, 2016.

[8] The IEEE and The Open Group. The open group base specifications issue 7: fopen. http://pubs.opengroup.org/onlinepubs/9699919799/functions/fopen.html, 2013. [Online; accessed 17-April-2016].

[9] Daniel Kroening and Michael Tautschnig. CBMC C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413, Berlin, Heidelberg, 2014. Springer.

[10] Josh Poley. Best practices: API testing. https://msdn.microsoft.com/en-us/library/cc300143.aspx, February 2008. [Online; accessed 17-April-2016].

[11] Michael Pradel and Thomas R. Gross. Fully automatic and precise detection of thread safety violations. *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation - PLDI '12*, page 10, 2012.

[12] Asha K. R. and Shwetha D. J. API testing: Picking the right strategy. In *Pacific Northwest Software Quality Conference 2015*, pages 261–270, 2015.

[13] Emanuele Di Rosa, Enrico Giunchiglia, Massimo Narizzano, Gabriele Palma, and Alessandra Puddu. Automatic generation of high quality test sets via CBMC. In Markus Aderhold, Serge Autexier, and Heiko Mantel, editors, *VERIFY-2010. 6th International Verification Workshop*, volume 3 of *EPiC Series in Computing*, pages 65–78. EasyChair, 2012.

[14] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. Synthesizing racy tests. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*, pages 175–185, 2015.

[15] Charles P. Shelton, Philip Koopman, and Kobey Devale. Robustness testing of the Microsoft Win32 API. In *2000 International Conference on Dependable Systems and Networks (DSN 2000)*, pages 261–270. IEEE, 2000.

Author Index