Reducing Test Cases with Causality Partitions

Haijun Wang[1], Xiaohong Guan[1], Qinghua Zheng[1], Ting Liu[1,*], Xiangyang Li[1], Lechen Yu[1], Zijiang Yang[2]
1. MOE Key Lab. for Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an, China
2. Western Michigan University, Kalamazoo MI, U.S.A.
{hjwang, xhguan, tliu, xyli, lcyu}@sei.xjtu.edu.cn; qhzheng@mail.xjtu.edu.cn; zijiang.yang@wmich.edu

***Abstract*—Automatic test case generation using symbolic execution suffers from the problem of *path explosion*: the number of paths to be explored may grow exponentially with the scale of a program. We believe that different paths may exhibit some similar program behaviors, thus it is unnecessary to explore all of the paths to generate test cases. In this paper, a novel model of program causality is proposed to extract all kinds of influences among statements of a program, which consists of four types of program dependencies. Then, a heuristic approach based on program causality is developed to selectively explore program paths in symbolic execution. We have implemented a prototype of our approach within the symbolic execution engine of Java Pathfinder. Experiments on six public benchmarks show that our approach can significantly reduce the number of explored paths and the symbolic execution time so that to improve the performance of test case generation.**

Keywords-Symbolic Execution; Causality Partition; Test Case Generation;

I. INTRODUCTION

Automatic test case generation using symbolic execution [1-3] has recently regained prominence as a technique for systematically exploring the paths of a program. In practice though, symbolic execution suffers from the inherent problem of *path explosion* where the number of paths to be explored can increase exponentially with the scale of a program. While there are techniques for mitigating *path explosion*, e.g. by using compositionality [4], abstraction refinement [5], and parallelization [6], systematically exploring program paths remains a daunting task even for medium-sized programs.

In this paper, we propose a novel test case generation technique that selectively explores program paths based on program causality. The intuition is that the path with the same causal relationship with already explored paths does not reveal additional program behavior. Figure 1 shows a program that may raise *division-by-zero* exceptions at Lines 11 and 13. Table I gives all eight possible paths of the program. For brevity we only list branch statements of a path. As indicated by *Column* 3, paths π_5 and π_7 can raise the exception at Line 11, and paths π_3 and π_4 can raise the exception at Line 13. *Columns* 4-8 mark the necessary paths for kinds of coverage criteria, respectively. For statement coverage $T_{\text{Statement}}$, path π_1 is sufficient. For branch coverage T_{Branch}, two paths π_1 and π_8 are satisfiable. In order to achieve the path coverage T_{Path}, all eight paths have to be executed. As expected, statement and branch coverage require much fewer test cases than path coverage, at the cost of missing both exceptions at Lines 11 and 13. *Column* 7 lists the paths that are covered by our approach. Note that with half of the paths to be explored, our approach can still catch both exceptions. Comparing π_5 and π_7, we can observe that the decision at Line 8 makes no difference on the potential exception at Line 11. Therefore a testing that covers either π_5 or π_7 can report the exception at Line 11. Similarly, it is not necessary to explore both π_3 and π_4 for detecting the exception at Line 13. This paper addresses a more general question: without considering any particular type of fault, is it possible to achieve the same testing effect as path coverage without exploring all paths? By exploring four instead of eight paths, Table I shows the answer is positive. In our approach we exploit causal relationship to achieve the path reduction. However, the existing program dependence definitions are not sufficient for causal relationship. Therefore in this paper we introduce a new type of program dependence called **Correlation Dependence**. The last column in Table I shows why correlation dependence is indispensable, as without it the exception at Line 13 cannot be detected.

```
void Test(int x, int y, int m) {
1:  int a=1;
2:  int b=1;
3:  int c=2;
4:  int d=3;
5:  int first_out, second_out;
6:  if(x+y < 0)
7:     a=3;
8:  if(m > 2)
9:     d=4;
10: if(x-y > 1)
11:    c=1/(a-b);
12: first_out=c;
13: second_out=1/(a-d);
}
```

Figure 1. A Running Program.

As illustrated by the example, the key insight of our approach is to prevent some irrelevant statements combination from generating redundant paths. We first extract causal dependencies of a program, which consist of control, data, potential dependence and newly proposed correlation dependence. Then, we conduct symbolic execution to selectively explore program paths through exploiting the

TABLE I. TEST CASES FOR KINDS OF COVERAGE CRITERIA

* The corresponding author is Ting Liu (tliu@sei.xjtu.edu.cn)

The work was supported in part by National Science Foundation of China under Grant (91118005, 91218301, 61221063, 61203174, U1301254), National High Technology Research and Development Program 863 of China under Grant (2012AA011003), Key Projects in the National Science and Technology Pillar Program of China (2012BAH16F02) and the Fundamental Research Funds for the Central Universities.

No	Execution	Error	$T_{Statement}$	T_{Branch}	T_{Path}	T_{Causal}	$T_{w/o}$
π_1	<6T,8T,10T>		√	√	√	√	√
π_2	<6T,8T,10F>				√		
π_3	<6T,8F,10T>	13			√	√	
π_4	<6T,8F,10F>	13			√		
π_5	<6F,8T,10T>	11			√	√	√
π_6	<6F,8T,10F>				√		
π_7	<6F,8F,10T>	11			√		
π_8	<6F,8F,10F>			√	√	√	√

causal dependencies. We have implemented a prototype of our approach within the framework of JPF-SE [1] and conducted experiments on six public benchmarks. The results validate the effectiveness of our approach. Compared with the original JPF-SE, our approach can reduce the number of explored paths by 46.28% to 98.84%, and achieve a speedup of 2.47X to 49.12X. We also compare our approach with FlowTest [8], where our approach can reduce the number of explored paths by 49.37% to 98.86%, and achieve a speedup of 1.05X to 61.40X. Additional experiments on the growth rates of the number of explored paths and usage time indicate that our approach can achieve orders of magnitude improvement.

Our main contributions are summarized as follows.

- We propose a novel approach to reduce test cases through exploiting the causality exhibited in program paths. Such reduction is sound.
- We extract causality in terms of control dependence, data dependence, potential dependence and newly proposed correlation dependence. This can help to comprehensively and systematically understand the causal dependencies among program statements.
- We have implemented a prototype of our approach within the framework of JPF-SE and conducted experiments on six public benchmarks. The results show that our approach can dramatically reduce the number of explored paths and the symbolic execution time. Additional experiments on growth rates of the explored path size and usage time indicate that our approach can offer orders of magnitude improvement.

II. REDUCING TEST CASES WITH CAUSALITY PARTITIONS

A. Relevant Definitions

Control flow graph (CFG) of a procedure is a directed graph that can be represented by CFG=<*N, E*>, where *N* is the set of nodes and $E \subseteq N \times N$ represents possible execution flow between the nodes. $Br \subseteq N$ is the set of branches. Also, we use $\overline{n} \in Br$ to represent the opposite branch of $n \in Br$.

Definition 1 **Control Dependence** *is a map controlD: $Br \times N \rightarrow \{T, F\}$ that returns true for a pair of nodes (n_i, n_j) if node n_i has more than one exit paths, and one of the exit paths must result in n_j being executed but the other exit path may result in n_j not being executed; Otherwise it returns false*.

Definition 2 **Data Dependence** *is a map dataD: $N \times N \rightarrow \{T, F\}$ that returns true for a pair of nodes (n_i, n_j) if there exists a path π from n_i to n_j where n_i defines a variable v, n_j uses the variable v and any n_k ($i<k<j$) on π does not redefine the variable v; Otherwise it returns false.*

Definition 3 **Potential Dependence** *[9] is a map potentialD: $Br \times N \rightarrow \{T, F\}$ that returns true for a pair of nodes (n_i, n_j), if (1) there exists a path π from n_i to n_j where n_j uses a variable v and the definition of v occurs before n_i; (2) n_j is not control dependent on n_i; (3) a different definition of v could potentially reach n_j if n_i is evaluated differently; Otherwise it returns false.*

We define ***Traditional Dependence*** as a map *traditionalD*: $N \times N \rightarrow \{T, F\}$ that returns true for a pair of nodes (n_i, n_j) if $controlD(n_i, n_j) \vee dataD(n_i, n_j) \vee potentialD(n_i, n_j)$; Otherwise it returns false.

Definition 4 **Correlation Dependence** *is a map correlationD: $N \times N \rightarrow \{T, F\}$ that returns true for a pair of nodes (n_i, n_j) if there exists a path $\pi = <\ldots n_i \ldots n_j \ldots n_k \ldots>$, and n_i, n_j, n_k satisfy (1) $traditionalD(n_i, n_k) \wedge (traditionalD(n_j, n_k) \vee correlationD(n_j, n_k))$; or (2) $(traditionalD(n_i, n_k) \vee correlationD(n_i, n_k)) \wedge traditionalD(n_j, n_k)$; Otherwise it returns false.*

We define ***Causal Dependence*** as a map *causalD*: $N \times N \rightarrow \{T, F\}$ that returns true for a pair of nodes (n_i, n_j) if $traditionalD(n_i, n_j) \vee correlationD(n_i, n_j)$; Otherwise it returns false. Table II lists all causal dependencies of the example.

TABLE II. DEPENDENCIES OF THE PROGRAM IN FIGURE 1

Type	Dependent Pairs
ControlD	(6T,7) (8T,9) (10T,11)
DataD	(1,11) (1,13) (2,11) (3,12) (4,13) (7,11) (7,13) (9,13) (11,12)
PotentialD	(6F,11) (6F,13) (8F,13) (10F,12)
CorrelationD	(1,2) (1,4) (1,6F) (1,8T) (1,8F) (1,9) (1,10T) (2,6T) (2,6F) (2,7) (2,10T) (3,10F) (4,6T) (4,6F) (4,7) (4,8F) (6F,8T) (6F,8F) (6F,9) (6F,10T) (7,8T) (7,8F) (7,9) (7,10T)

Definition 5 *A* ***covered causal chain*** *c_π of path π is a sequence $V=[n_1, \ldots, n_m]$ such that*

- $\exists n_i \in \pi$, $causalD(n_i, n_1)$, *and*
- $\forall n_j \in V$, $traditionalD(n_i, n_j) \rightarrow n_i \in V$, *and*
- $\forall n_j, n_{j+1} \in V$, $causalD(n_j, n_{j+1})$.

Definition 6 *A* ***discovered causal chain*** *d_π of path π is a sequence $V=[n_1, \ldots, \overline{n}_m]$ that is a* ***covered causal chain*** *of path π' generated by flipping the branch n_m to $\overline{n}_m$ in path π.*

B. Computation of Causal Dependence

Since control, data and potential dependence are widely applied in the software engineering and can be obtained by the existing tools [8, 9], in this paper we focus on the computation of newly proposed correlation dependence.

We first compute the reachability of node pair (n_i, n_j) using *Equations* (1), (2) and (3). Similar to the data flow analysis via graph reachability [11], we use the node pair in the computation. Note that n_0 represents *start node* of CFG, which has no incoming node pair. The outgoing node pairs of n_j

consist of two parts: the node pairs generated at n_j, and the subtraction of the incoming node pairs by the node pairs killed at n_j. Given the incoming nodes N_j at n_j, obtained through data flow analysis [11], the node pair (n'_j, n_j) is generated at n_j through pairing every $n'_j \in N_j$ with n_j. A node pair (n_p, n_q) is killed at n_j if either n_p or n_q is redefined by n_j. We consider n_p or n_q is redefined by n_j, if n_p or n_q is the definition statement and the variable they define is the same with the variable defined at n_j, or n_p or n_q is the conditional statement and they no longer directly or indirectly control n_j. Assuming N_{pre} is the set of nodes that are immediate predecessors of n_j in CFG, the incoming node pairs of n_j are the union of the outgoing node pairs of the nodes in N_{pre}.

$$in[n_0] = \{\} \quad (1)$$
$$out[n_j] = gen[n_j] \cup (in[n_j] - kill[n_j]) \quad (2)$$
$$in[n_j] = \cup_{n_i \in N_{prc}} out[n_i] \quad (3)$$

We then compute the correlation dependence. For each node n_j with an incoming node pair (n_p, n_q), if either *traditionalD*(n_p, n_j)$\wedge$*causalD*(n_q, n_j) or *causalD*(n_p, n_j) $\wedge$ *traditionalD*(n_q, n_j), it is considered that n_q is correlation dependent on n_p.

C. Reducing Test Cases with Causality Partitions

Overview. Algorithm 1 gives a high level overview of our approach. It maintains two global data structures *C* and *D*. *C* is the set of covered causal chains and *D* is the set of discovered causal chains that are yet to be covered. The algorithm first starts symbolic execution with a random input (Line 2). Each subsequent path exploration is directed by a discovered causal chain *d* removed from *D* (Line 5). At Line 6, if *d* is satisfiable, we will obtain a new input such that *d* can be covered by the execution under the newly acquired input. When we obtain an input, we use the procedure *reducedPathExploration* to handle the sets *C* and *D* (Line 8), which are used to drive the future path explorations. The algorithm terminates when *D* becomes empty. The computation of the input based on a discovered causal chain *d* (Line 6) and the algorithm for procedure *reducedPathExploration* (Line 8) will be discussed later.

Table III gives a complete procedure of our approach in the example program. *Column* 1 shows the discovered causal chains that are yet not covered and then used to guide further path explorations. The generated inputs and corresponding paths are given in *Columns* 2 and 3, respectively. *Columns* 4 and 5 list the set of covered causal chains and discovered causal chains. Note that although we pick only *one* discovered causal chain in each iteration, the newly explored path may cover many previously discovered causal chains that are still not covered. Therefore it may reduce the size of *D* significantly. As illustrated in Table III, four paths can cover all possible causal chains instead of eight paths in standard symbolic execution.

Table IV explains why neither π_3 nor π_4 is explored without considering correlation dependence and hence the exception at Line 13 is missed. The exception at Line 13 can only be raised by the path that executes both 6T and 8F. Since there does not exist any control, data or potential dependence between 6T and 8F, symbolic execution without correlation dependence will not combine 6T and 8F to generate a new path after exploring π=<6T,8T,10T>, which leads to incompleteness.

Algorithm 1 $reducedSymbolicExecution$(Program P)

```
1: C = D = ∅
2: input = randomInput()
3: reducedPathExploration(P, input, C, D)
4: while D ≠ ∅ do
5:     d = D.remove()
6:     input = obtain an input based on d
7:     if input ≠ ∅ then
8:         reducedPathExploration(P, input, C, D)
9:     end if
10: end while
```

Causal Chain Computation during Path Exploration. Algorithm 2 focuses on computing the covered and discovered causal chains once a path π is obtained. We will explain the algorithm through Table V, which illustrates the procedures under the path π=<1,2,3,4,6T,7,8T,9,10T,11,12,13>. Let C_π and D_π be the set of covered and discovered causal chains of π, respectively. Initially, as shown in *row* 1, C_π={[1], [3]} and D_π={}. Both nodes 1 and 3 appear in π and they are not causal dependent on any other nodes in π (Line 2). D_π is empty as there is no node in π whose negated node is not causal dependent on any other nodes in π (Line 3). We illustrate Lines 5-15 in Algorithm 2 using *rows* 6 and 13 in Table V. c_π.*branch* at Line 6 represents the sub sequence of c_π that consists of all its branches. In fact, the longer causal chain is generated by extending a node into the previous shorter causal chain. In *row* 6, the previous shorter covered causal chain c_π is [1,2,6T,7,10T] that is generated at *row* 5 and *EXT* is {11}. *EXT* represents the set of extensible nodes. If a node n_j is causal dependent on the last node of previous shorter covered causal chain c_π and the set of nodes that are traditional dependent by n_j are all included in c_π, n_j can be an extensible

TABLE III. SYMBOLIC EXECUTION WITH CAUSALITY

Causal Chain	Input	Path	*C*	*D*
None	(0,-2,3)	<6T,8T,10T>	[6T] [6T,8T] [6T,10T] [8T] [10T]	[6F] [6T,8F] [8F] [10F]
[6F]	(2,0,3)	<6F,8T,10T>	[6F] [6F,8T] [6F,10T]	[6F,8F]
[6T, 8F]	(0,-2,0)	<6T,8F,10T>	[6T,8F] [8F]	
[10F]	(1,0,0)	<6F,8F,10F>	[6F,8F] [10F]	

TABLE IV. SYMBOLIC EXECUTION WITHOUT CORRELATION DEPENDENCE

Causal Chain	Input	Path	*C*	*D*
None	(0,-2,3)	<6T,8T,10T>	[6T] [6T,10T] [8T] [10T]	[6F] [8F] [10F]
[6F]	(2,0,3)	<6F,8T,10T>	[6F] [6F,10T]	
[10F]	(1,0,0)	<6F,8F,10F>	[8F] [10F]	

Algorithm 2 $reducedPathExploration(P, input, C, D)$

```
1: π = execute P under input
2: C_π = {[n_j] | n_j ∈ π ∧ ¬causalD(n_i, n_j) for any other n_i ∈ π}
3: D_π = {[n_j] | n̄_j ∈ π ∧ ¬causalD(n_i, n_j) for any other n_i ∈ π}
4: while C_π ≠ ∅ do
5:     c_π = C_π.removc()
6:     C = C ∪ {c_π.branch}
7:     EXT = {n_j | causalD(c_π.lastNode, n_j)
                  ∧∀n_i(traditionalD(n_i, n_j) → n_i ∈ c_π)}
8:     for each node ext ∈ EXT do
9:         c'_π = c_π.append(ext)
10:        if ext ∈ π then
11:            C_π = C_π ∪ {c'_π}
12:        else
13:            D_π = D_π ∪ {c'_π}
14:        end if
15:    end for
16: end while
17: D = (D ∪ D_π.branch) − C
```

node of c_π (Line 7). Note that c_π.*lastNode* at Line 7 refers to the last node of c_π. Since node 11 is causal dependent on branch 10T and the set of nodes {2,7,10T} that are traditional dependent by node 11 are all included in c_π, node 11 is an extensible node. Obviously, $ext \in \pi$ and finally c_π' is added into C_π. In *row* 13, the previous shorter covered causal chain c_π is [3] and *EXT* is {10F}. c_π is appended as c_π'=[3,10F] and c_π' is added into D_π because $ext \notin \pi$.

From Discovered Causal Chain to Input. Given a discovered causal chain *d*, we will compute an input so that the execution under the newly acquired input can cover *d*. In this section we use an example to illustrate the computation.

Assume *d* = [10F] and we would like to obtain a path π of the program in Figure 1 such that π can cover *d*. Treating inputs as symbolic variables we start a symbolic execution. At the first conditional statement (Line 6) we allow symbolic execution to choose any branch because *d* does not specify a particular branch at Line 6. The fact that *d* does not specify Line 6 means that there is no statement in *d* that is causal dependent on Line 6. Therefore symbolic execution can take any branch at Line 6. Next symbolic execution encounters the conditional statement Line 8. Since the branch at Line 8 is also not specified by *d*, symbolic execution still executes randomly. Finally symbolic execution reaches the conditional statement Line 10. Since 10F is specified in *d*, symbolic execution takes the *false* branch and adds term x–y≤1 into the path constraints ξ to match the requirement of 10F in *d*. If ξ is unsatisfiable, the procedure terminates and no input can be found to match *d*, otherwise symbolic execution continues. As there is no more conditional constraint in *d*, we pass ξ to SMT solver. If ξ is satisfiable, the solution forms an input that can cover *d*; otherwise there is no valid path to cover *d*.

Theorem 1. Given a program *P*, our approach can cover all feasible causal chains of *P*.

Example. Assume a causal chain *V*=[1,4,6T,7,8F,13] of the program in Figure 1 is required to be covered, which can detect the exception at Line 13. As shown in *rows* 9 and 10 of Table V, the initial path π=<1,2,3,4,6T,7,8T,9,10T,11,12,13> is able to cover the causal chain *V'*=[1,4,6T,7] and discover the

TABLE V. CAUSAL CHAINS OF Π = <1,2,3,4,6T,7,8T,9,10T,11,12,13>

No.	c_π	*ext*	C_π	D_π
1		1	[1]	
		3	[3]	
2	[1]	2	[1,2]	
		4	[1,4]	
		6F		[1,6F]
		8T	[1,8T]	
		8F		[1,8F]
		10T	[1,10T]	
3	[1,2]	6T	[1,2,6T]	
		6F		[1,2,6F]
		10T	[1,2,10T]	
4	[1,2,6T]	7	[1,2,6T,7]	
5	[1,2,6T,7]	10T	[1,2,6T,7,10T]	
6	[1,2,6T,7,10T]	11	[1,2,6T,7,10T,11]	
7	[1,2,6T,7,10T,11]	12	[1,2,6T,7,10T,11,12]	
8	[1,4]	6T	[1,4,6T]	
		6F		[1,4,6F]
		8F		[1,4,8F]
9	[1,4,6T]	7	[1,4,6T,7]	
10	[1,4,6T,7]	8T	[1,4,6T,7,8T]	
		8F		[1,4,6T,7,8F]
		10T	[1,4,6T,7,10T]	
11	[1,4,6T,7,8T]	9	[1,4,6T,7,8T,9]	
12	[1,4,6T,7,8T,9]	13	[1,4,6T,7,8T,9,13]	
13	[3]	10F		[3,10F]

causal chain *V''*=[1,4,6T,7,8F] through extending the node 8F to *V'*. As stated in our algorithm, the discovered causal chain *V''* then is used to direct symbolic execution to generate an input *input*=<*x*=0, *y*=-2, *m*=0> which can cover *V''*, as shown in Table III. Simultaneously, *input* can cover the required causal chain *V*.

III. RELATED WORK

Symbolic execution [1-3] has emerged as a popular technique for testing real-world software applications. However, symbolic execution suffers from the inherent path explosion problem. Many techniques have been proposed to alleviate this problem, and they can be roughly divided into four categories. The first category addresses the path explosion problem, e.g., [5], through integrating abstraction to reduce the search space. In the second category (e.g. [4]), researchers generate method summaries to enable more efficient constraint solving. The approaches of the third category, e.g., [8, 9, 12, 13], adopt the notion of the path equivalence to avoid full path exploration. Finally the techniques in the fourth category, e.g., [14, 15], limit the analysis scope to avoid irrelevant path exploration.

Our approach shares similarities with the third category. The technique in [9] partitions the paths based on the program symbolic output. Two paths are placed in the same partition if the symbolic expressions connecting the input of a program with the output are the same. During symbolic execution only one path from each partition is explored. The main difference from our approach is that [9] ignores program crashes that do not come from the unexpected outputs, while our approach can avoid such situation. In work [13], a program is statically decomposed into some path families, where each path family contains several paths that share similar program behaviors.

TABLE VI. SYMBOLIC EXECUTION WITH AND WITHOUT CAUSALITY REDUCTION

Subject	LoC	Time(s)			#Explored Paths			#Infeasible Paths			#Avg. PC		
		Causal	Flow	JPF	Causal	Flow	JPF	Causal	Flow	JPF	Causal	Flow	JPF
WBS	231	196	539	519	4123	13840	13824	0	0	0	28.8	56.0	56.1
Jtcas	236	734	1906	1813	101	216	188	214	1169	1138	20.2	23.9	24.4
Totinfo	375	7946	36224	34998	37	225	204	306	3536	3151	54.2	93.8	93.7
Siena	1529	39	41	596	40	79	1728	104	104	471	7.2	6.7	20.2
Wc	1557	25	1535	1228	163	2626	2596	130	2630	2590	4.3	23.2	23.5
Sum	1257	37	720	696	17	1490	1464	23	1460	1458	6.5	17.8	18.0

Authors argue that a program can be analyzed at the granularity of path family instead of individual path. A significant difference from our approach is that [13] does not consider the interaction of statements that are correlation dependent. The technique in [12] partitions paths based on program states. Two program paths are equivalent if symbolic states of all live variables are the same.

The work closest to our approach is FlowTest [8], where the inputs of a program are partitioned into non-interfering blocks such that symbolically solving an input block while keeping other blocks assigned with concrete values can find the same set of assertion violations as symbolically solving for the entire inputs. Two inputs are considered of interfering, if they are jointly relevant to the same statement or they are relevant to the statements that have dependent relationship in some program path. By partitioning the program inputs, FlowTest can avoid full path exploration as our approach. In this paper, we have re-implemented FlowTest within the framework of JPF-SE and empirically compared it with our approach in Section IV. The main difference between two techniques is the different partition criteria. The technique FlowTest partitions the paths based on the dependencies of the program inputs, while our approach partitions the paths based on the dependencies of the program statements. Our approach enjoys finer granularity as if two statements are dependent, the inputs that are relevant to these two statements must be dependent; it may be not true for the opposite situation.

IV. EMPIRICAL STUDY

We implemented a prototype of our approach within the framework of JPF-SE [1] and used the constraint solver Z3 [16] to check the satisfiability of path conditions. To compute the causal dependence we used Indus [10], an inter-procedural analysis tool based on Soot for Java byte code. Our benchmark suite consists of six programs with lines of code ranging from 200 to 2000(*Column LoC* in Table VI). WBS is a synchronous reactive component from the automotive domain. Its Java implementation is based on a Simulink model derived from the WBS case example presented in ARP 4761 [15]. Jtcas, Totinfo and Siena are obtained from the SIR repository [17, 18]. Jtcas is an aircraft collision avoidance system. Totinfo is used to compute the statistics of the given input data. Siena implements partial event notification architecture. WC and SUM are obtained from the GNU Coreutils suite [2]. WC calculates the number of bytes, words, and lines in a file. SUM checksums and counts the number of blocks in a file. Totinfo, WC and SUM are written in C and we manually translated them to Java.

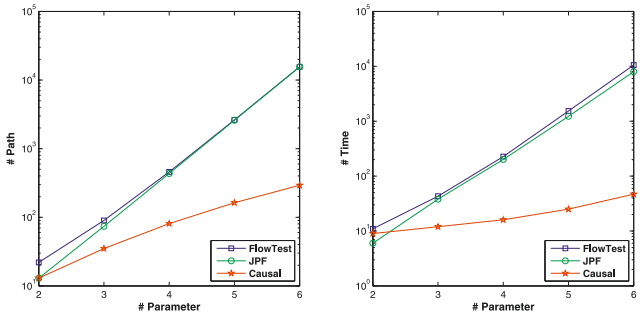
We conducted two groups of experiments. In the first group, we compared our approach against standard symbolic execution JPF-SE and FlowTest proposed in [8]. We reported the symbolic execution time, the number of explored paths and other data on six benchmark programs. In the second group, we examined the growth rates of the explored path size and usage time with respect to the number of input parameters, through using our approach and other two peer techniques. The benchmark programs used in the second group are WC and SUM, because both programs can take different numbers of input parameters and produce different numbers of explored paths.

A. Reducing Test Cases with Causality Partitions

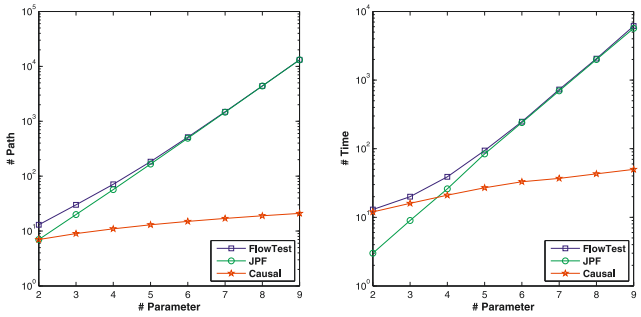
Table VI gives the experiment results of our approach and other two peer techniques. We set the number of input parameters of WC and SUM to be 5 and 7, respectively. The columns labeled with *Causal*, *Flow* and *JPF* denote the results obtained from our approach, FlowTest and standard symbolic execution JPF-SE, respectively. The column *Time(s)* gives the usage time of three approaches in second.

It can be observed that the speedup achieved by our approach is significant, ranging from 2.47X to 49.12X compared with JPF-SE, and from 1.05X to 61.40X compared with FlowTest. These results indicate the effectiveness of our approach. It is well known that the cost of symbolic execution is dominated by SMT solver callings. From this perspective, there are three main reasons that can explain the speedup achieved by our approach. The first reason is due to the reduction in the number of explored paths. This can be confirmed by the column labeled with *#Explored Paths*. The number of *#Explored Paths* is proportional to the number of SMT solver callings that are satisfiable, and our approach reduces the number of explored paths by 46.28% to 98.84% and by 49.37% to 98.86% compared with JPF-SE and FlowTest, respectively. The second reason is due to fewer numbers of SMT solver callings that are unsatisfiable. This can be confirmed by the column *#Infeasible Paths*. Our approach reduces the number of *#Infeasible Paths* by 0 to 98.42% and by 0 to 98.43% compared with JPF-SE and FlowTest, respectively. In this paper, if the path conditions ξ_i are unsatisfiable, any path conditions ξ_j prefixed by ξ_i are not counted in *#Infeasible Paths* because ξ_j cannot be generated by our approach and other two peer techniques. The third reason is that the average number of constraints in the path conditions is fewer in our approach. Although not guaranteed, in general SMT solver is more efficient with fewer numbers of constraints. This can be explained by the fact that the path

conditions in our approach must have causal relationships, while in JPF-SE and FlowTest



(a) WC



(b) SUM

Figure 2. Growth of the explored paths and usage time on WC and SUM.

all path conditions encountered during a path exploration have to be considered. This can be confirmed by the data presented in the column labeled with *#Avg.PC* that shows the average number of constraints in the path conditions.

B. Growth Rates of Explored Path Size and Usage Time

Figure 2 depicts the number of explored paths and usage time over different numbers of input parameters that range from 2 to 6 for WC and from 2 to 9 for SUM.

It can be observed that in JPF-SE both the number of explored paths and usage time grow exponentially with the number of input parameters, and there is no observable improvement in FlowTest. In both programs, FlowTest explores the paths without reduction because the input parameters of two programs are mutually dependent. On the other hand, the growth rates are much slower in our approach. In Figure 2(a), we can see that the number of explored paths and usage time in our approach still grow exponentially with the number of input parameters, but the exponents in our approach are smaller than other two peer techniques. In Figure 2(b), the number of explored paths and usage time in our approach become to grow in the order of polynomial magnitude and it greatly slows the growth rates. In program SUM, the program statements that handle input parameters are independent. Therefore, with additional input parameters our approach only additionally explores the program paths that are relevant to the added input parameters, without considering the program paths that are relevant to the previous input parameters. In addition, it can also be observed that when the number of explored paths is trivial, our approach uses a little longer time than JPF-SE. This is due to the overhead of the causal chain computation. Based on the growth rates of the explored path size and usage time, our approach has potential to offer orders of magnitude improvement over standard symbolic execution.

V. CONCLUSION AND FUTURE WORK

In this paper, we present a novel test case generation technique that selectively explores program paths based on program causality, which can generally alleviate the path explosion problem of symbolic execution. In addition, our reduction is sound, which means our approach can cover all feasible causal chains of a program. This enables us to obtain a concise test suite. We also empirically compare the efficiency of our approach with standard symbolic execution JPF-SE and another improved symbolic execution FlowTest.

Moreover, our approach can be extended for regression testing if we only focus on the program dependencies relevant to the changed statements. We will address this problem in the future work.

REFERENCES

[1] Pasareanu, C. and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. ASE, 2010. Antwerp, Belgium: ACM.

[2] Cadar, C., D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI, 2008. San Diego, California: USENIX Association.

[3] Godefroid, P., N. Klarlund, and K. Sen. DART: directed automated random testing. PLDI, 2005. Chicago, IL, USA: ACM.

[4] Anand, S., and N. Tillmann. Demand-driven compositional symbolic execution. TACAS, 2008. Budapest, Hungary: Springer-Verlag.

[5] Anand, S, Pasareanu, C, Visser, W. Symbolic execution with abstraction. Int. J. Softw. Tools Technol. Transf., 2009. 11(1): p. 53-67.

[6] Siddiqui, J.H. and S. Khurshid. Scaling symbolic execution using ranged analysis. OOPSLA, 2012. Tucson, Arizona, USA: ACM.

[7] Staats, M. and C. P. Parallel symbolic execution for structural test generation. ISSTA, 2010. Trento, Italy: ACM.

[8] Majumdar, R. and R.-G. Xu. Reducing Test Inputs Using Information Partitions. CAV, 2009. Grenoble, France: Springer-Verlag.

[9] Qi, D., H.D.T. Nguyen, and A. Roychoudhury. Path exploration based on symbolic output. FSE, 2011. Szeged, Hungary: ACM.

[10] Ranganath, V.P., et al., A new foundation for control dependence and slicing for modern program structures. ACM Trans. Program. Lang. Syst., 2007. 29(5): p. 27.

[11] Reps, T., S. Horwitz, and M. Sagiv, Precise interprocedural dataflow analysis via graph reachability. POPL, 1995, ACM: San Francisco, California, USA. p. 49-61.

[12] Boonstoppel, P., C. Cadar, and D. Engler. RWset: attacking path explosion in constraint-based test generation. TACAS, 2008. Budapest, Hungary: Springer-Verlag.

[13] Santelices and Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. ISSTA, 2010. Trento, Italy: ACM.

[14] Taneja et al. eXpress: guided path exploration for efficient regression test generation. ISSTA, 2011. Toronto, Ontario, Canada: ACM.

[15] Person S., G.Yang, Rungta, N. Directed incremental symbolic execution. PLDI, 2011. San Jose, California, USA: ACM.

[16] Moura, L.D., Bjorner N. Z3: an efficient SMT solver. TACAS, 2008. Budapest, Hungary: Springer-Verlag.

[17] Hutchins, M., Foster, H. Goradia, T, Ostrand, T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. ICSE, 1994. Sorrento, Italy: IEEE Computer Society Press.

[18] Bohme, M., Oliveira, B, Rochoudhur, A. Partition-based regression verification. ICSE, 2013. San Francisco, CA, USA: IEEE Press.