Reducing Test Cases with Causality Partitions

Haijun Wang¹, Xiaohong Guan¹, Qinghua Zheng¹, Ting Liu¹,², Xiangyang Li¹, Lechen Yu¹, Zijiang Yang²,³
1. MOE Key Lab. for Intelligent Networks and Network Security, Xi’an Jiaotong University, Xi’an, China
2. College of Computer and Technology, Xi’an University of Technology, Xi’an, China
3. Western Michigan University, Kalamazoo MI, U.S.A.
{hjwang, xguan, tliu, xyli, lcyu}@sei.xjtu.edu.cn; qzheng@mail.xjtu.edu.cn; ziji 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 \( \pi_3 \) and \( \pi_7 \) can raise the exception at Line 11, and paths \( \pi_3 \) and \( \pi_6 \) can raise the exception at Line 13. Columns 4-8 mark the necessary paths for kinds of coverage criteria, respectively. For statement coverage \( T_{stat} \), path \( \pi_1 \) is sufficient. For branch coverage \( T_{branch} \), two paths \( \pi_1 \) and \( \pi_4 \) 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 \( \pi_2 \) and \( \pi_7 \), 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 \( \pi_2 \) or \( \pi_7 \) can report the exception at Line 11. Similarly, it is not necessary to explore both \( \pi_1 \) and \( \pi_4 \) 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.

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

---

* 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.
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.

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 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 number of explored paths and usage time indicate that our approach can achieve 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 $CFG = \langle N, E \rangle$, 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 $\bar{n}$ to represent the opposite branch of $n \in Br$.

**Definition 1 Control Dependence** is a map $\text{controlD}: Br \times N \rightarrow \{T, F\}$ 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 $n_i$, not being executed; Otherwise it returns false.

**Definition 2 Data Dependence** is a map $\text{dataD}: N \times N \rightarrow \{T, F\}$ that returns true for a pair of nodes $(n_i, n_j)$ if there exists a path $\pi$ 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 $\pi$ does not redefine the variable $v$; Otherwise it returns false.

**Definition 3 Potential Dependence** [9] is a map $\text{potentialD}: Br \times N \rightarrow \{T, F\}$ that returns true for a pair of nodes $(n_i, n_j)$ if (1) there exists a path $\pi$ from $n_i$ to $n_j$ where $n_i$ uses a variable $v$ and the definition of $v$ occurs before $n_j$; (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 $\text{traditionalD}: N \times N \rightarrow \{T, F\}$ that returns true for a pair of nodes $(n_i, n_j)$ if $\text{controlD}(n_i, n_j) \lor \text{dataD}(n_i, n_j) \lor \text{potentialD}(n_i, n_j)$; Otherwise it returns false.

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

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Dependent Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlD</td>
<td>(6,7) (6,9) (10,1,1)</td>
</tr>
<tr>
<td>DataD</td>
<td>(1,11) (1,13) (2,11) (3,12) (4,13)</td>
</tr>
<tr>
<td>PotentialD</td>
<td>(6,F,11) (6,F,13) (8,F,13) (10,F,12)</td>
</tr>
<tr>
<td>CorrelationD</td>
<td>(1,2) (1,4) (1,F,5) (1,8) (1,8)</td>
</tr>
<tr>
<td></td>
<td>(1,9) (1,10) (2,6) (2,6) (2,7)</td>
</tr>
<tr>
<td></td>
<td>(2,10) (3,10) (4,6) (4,6) (4,7)</td>
</tr>
<tr>
<td></td>
<td>(4,8) (6,F,8) (6,F,9) (6,F,10)</td>
</tr>
<tr>
<td></td>
<td>(7,8) (7,F,8) (7,9) (7,10)</td>
</tr>
</tbody>
</table>

**Definition 5 A covered causal chain** $c_x$ of path $\pi$ is a sequence $V=\{n_i, \ldots, n_m\}$ such that

- $\exists n_i \in \pi$, causalD$(n_i, n_j)$, and
- $\forall n_j \in V$, traditionalD$(n_i, n_j) \rightarrow n_j \in V$, and
- $\forall n_j, n_j+1 \in V$, causalD$(n_j, n_j+1)$.

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

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 \( n_p \), and the subtraction of the incoming node pairs by the node pairs killed at \( n_j \). Given the incoming nodes \( N_i \) at \( n_j \), obtained through data flow analysis [11], the node pair \((n_j', n_j)\) is generated at \( n_j \) through pairing every \( n_j' \in N_i \) 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 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 \( N_{pre} \).

\[
\begin{align*}
in[n_0] = \{ \} \quad & (1) \\
out[n_j] = gen[n_j] \cup (in[n_j] - kill[n_j]) \quad & (2) \\
in[n_j] = \cup_{n_i \in N_{pre}} out[n_i] \quad & (3)
\end{align*}
\]

We then compute the correlation dependence. For each node \( n_j \) with an incoming node pair \((n_p, n_q)\), if either traditional\(D(n_p, n_j)\) \& causal\(D(n_q, n_j)\) or causal\(D(n_p, n_j)\) \& traditional\(D(n_q, n_j)\), 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 \textit{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 \textit{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 \( \pi_3 \) nor \( \pi_4 \) 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 \( \pi=\langle 6T,8T,10T \rangle \), which leads to incompleteness.

Causal Chain Computation during Path Exploration.

Algorithm 2 focuses on computing the covered and discovered causal chains once a path \( \pi \) is obtained. We will explain the algorithm through Table V, which illustrates the procedures under the path \( \pi=\langle 1,2,3,4,6T,7,8T,9,10T,11,12,13 \rangle \). Let \( C_\pi \) and \( D_\pi \) be the set of covered and discovered causal chains of \( \pi \), respectively. Initially, as shown in row 1, \( C_\pi = \{[1], [3]\} \) and \( D_\pi = \{\} \). Both nodes 1 and 3 appear in \( \pi \) and they are not causal dependent on any other nodes in \( \pi \) (Line 2). \( D_\pi \) is empty as there is no node in \( \pi \) whose negated node is not causal dependent on any other nodes in \( \pi \) (Line 3). We illustrate Lines 5-15 in Algorithm 2 using rows 6 and 13 in Table V. \textit{extensible branch} at Line 6 represents the sub sequence of \( c_e \) 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 \( c_e \) is \([1,2,6T,7,10T]\) that is generated at row 5 and \textit{extensible} \( EXT \) is \{11\}. \textit{extensible} represents the set of extensible nodes. If a node \( n_j \) is causal dependent on the last node of previous shorter covered causal chain \( c_e \) and the set of nodes that are traditional dependent by \( n_j \) are all included in \( c_e \), \( n_j \) 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,8T] | [6F,8F] |
| | | | [8F] | [10F] |
| [6F] | (2,0,3) | <6F,8T,10T> | [6F] | [6F,8F] |
| | | | [6F,8T] | [6F,10T] |
| | | | [6F,8F] | [6F,10T] |
| [6F,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] | [6F] |
| | | | [6T,10T] | [8F] |
| | | | [10T] | [10F] |
| [6F] | (2,0,3) | <6F,8T,10T> | [6F] | [6F,10F] |
| [10F] | (1,0,0) | <6F,8F,10F> | [6F] | [10F] |
Algorithm 2 reducedPathExploration(P, input, C, D)
1: \( \pi = \text{execute P under input} \)
2: \( C_e = \{ n_i | n_j, n_i, n_j \in \pi \land -\text{causalD}(n_i, n_j) \text{ for any other } n_k \in \pi \} \)
3: \( D_e = \{ n_i | n_j, n_i, n_j \in \pi \land -\text{causalD}(n_i, n_j) \text{ for any other } n_k \in \pi \} \)
4: while \( C_e \neq \emptyset \) do
5: \( c_e = \text{ext_c}_e \text{ remove()} \)
6: \( C = C \cup \{c_e \text{ branch} \} \)
7: \( EXT = \{ n_i | \text{causalD}(c_e, \text{lastNode}_e, n_j) \land \forall n_k | \{\text{traditionalD}(n_i, n_j) \rightarrow n_i \in c_e \} \} \)
8: for each node \( ext \in EXT \) do
9: \( \xi = c_e \text{ append}(ext) \)
10: if \( ext \in \pi \) then
11: \( C_e = C_e \cup \{ \xi \} \)
12: else
13: \( D_e = D_e \cup \{ \xi \} \)
14: end if
15: end for
16: end while
17: \( D = (D \cup D_e, \text{branch}) - C \)

node of \( c_e \) (Line 7). Note that \( c_e, \text{lastNode}_e \) at Line 7 refers to the last node of \( c_e \). Since node 11 is causal dependent on branch 10F and the set of nodes \([2,7,10T]\) that are traditional dependent by node 11 are all included in \( c_e \), node 11 is an extensible node. Obviously, \( ext \in \pi \) and finally \( c_e' \) is added into \( C_e \). In row 13, the previous shorter covered causal chain \( c_e \) is \([3]\) and \( EXT \) is \([10F]\). \( c_e \) is appended as \( c_e = [3,10F] \) and \( c_e' \) is added into \( D_e \) 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 \( \pi \) of the program in Figure 1 such that \( \pi \) 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 \( x \leq y \leq 1 \) into the path constraints \( \zeta \) to match the requirement of 10F in \( d \). If \( \zeta \) 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 \( \zeta \) to SMT solver. If \( \zeta \) 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 \( \pi = <1,2,3,4,6T,7,8T,9,10T,11,12,13> \) is able to cover the causal chain \( V' = [1,4,6T,7] \) and discover the causal chain \( V'' = [1,4,6T,7,8F,13] \) 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 V. Causal Chains of \( \pi = <1,2,3,4,6T,7,8T,9,10T,11,12,13> \)

<table>
<thead>
<tr>
<th>No.</th>
<th>( c_e )</th>
<th>ext</th>
<th>( C_e )</th>
<th>( D_e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[1]</td>
<td>2</td>
<td>[1,2]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>[1,2]</td>
<td>4</td>
<td>[1,4]</td>
<td>[1,6F]</td>
</tr>
<tr>
<td>4</td>
<td>[1,2,6F]</td>
<td>6F</td>
<td>[1,6F]</td>
<td>[1,6F]</td>
</tr>
<tr>
<td>5</td>
<td>[1,2,6F]</td>
<td>10F</td>
<td>[1,10T]</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>[1,2,6F,10T]</td>
<td>10F</td>
<td>[1,10T]</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>[1,2,6F,10T,11]</td>
<td>12F</td>
<td>[1,10T,11,12]</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[1,4]</td>
<td>6F</td>
<td>[1,4F]</td>
<td>[1,4F]</td>
</tr>
<tr>
<td>9</td>
<td>[1,4,6T]</td>
<td>7</td>
<td>[1,4,6T]</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>[1,4,6T]</td>
<td>8F</td>
<td>[1,4,6T,8T]</td>
<td>[1,4,6T,8F]</td>
</tr>
<tr>
<td>11</td>
<td>[1,4,6T,7,9]</td>
<td>10F</td>
<td>[1,4,6T,7,10T]</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>[1,4,6T,7,8F,9]</td>
<td>13F</td>
<td>[1,4,6T,7,8F,9]</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>[3]</td>
<td>10F</td>
<td>[3,10F]</td>
<td></td>
</tr>
</tbody>
</table>
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 not be 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 an aircraft collision avoidance system. Totinfo and Siena are obtained from the SIR repository [17, 18]. Jtcas is an aircraft collision avoidance system.

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 $\xi_i$ are unsatisfiable, any path conditions $\xi_j$ prefixed by $\xi_i$ are not counted in #Infeasible Paths because $\xi_j$ 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.
the program paths that are relevant to the previous input parameters, without considering 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