Symbolic Analysis of Concurrent Programs with Polymorphism

Neha Rungta
NASA Ames Research Center, Moffett Field, CA 94035, USA
neha.s.rungta@nasa.gov

1. INTRODUCTION

The current trend of multi-core and multi-processor computing is causing a paradigm shift from inherently sequential to highly concurrent and parallel applications. Certain thread interleavings, data input values, or combinations of both often cause errors in the system. Systematic verification techniques such as explicit state model checking and symbolic execution are extensively used to detect errors in such systems [7, 9].

Explicit state model checking enumerates possible thread schedules and input data values of a program in order to check for errors [3, 9]. To partially mitigate the state space explosion from data input values, symbolic execution techniques substitute data input values with symbolic values [5, 7, 6]. Explicit state model checking and symbolic execution techniques used in conjunction with exhaustive search techniques such as depth-first search are unable to detect errors in medium to large-sized concurrent programs because the number of behaviors caused by data and thread non-determinism is extremely large.

We present an overview of abstraction-guided symbolic execution for concurrent programs that detects errors manifested by a combination of thread schedules and data values [8]. The technique generates a set of key program locations relevant in testing the reachability of the target locations. The symbolic execution is then guided along these locations in an attempt to generate a feasible execution path to the error state. This allows the execution to focus in parts of the behavior space more likely to contain an error.

2. ABSTRACTION-GUIDED SYMBOLIC EXECUTION

A high-level overview of the abstraction-guided symbolic execution technique is shown in Fig. 1.

Input: The input to the technique is a set of target locations, \( L_t \), that represent a possible error in the program. The target locations can either be generated using a static analysis tool or a user-specified reachability property. The lockset

Figure 1: Overview of the abstraction-guided symbolic execution technique

analysis, for example, reports program locations where lock acquisitions by unique threads may lead to a deadlock [2]. The lock acquisition locations generated by the lockset analysis are the input target locations for the technique.

Abstract System: An abstraction of the program is generated from backward slices of the input target locations and synchronization locations that lie along control paths to the target locations. Standard control and data dependence analyses are used to generate the backward slices. Location \( l_3 \) is a single target location in Fig. 1. The possible execution of location \( l_3 \) is control dependent on the true branch of the conditional statement \( l_2 \). Two definitions of a global variable \( a \) at locations \( l_0 \) and \( l_1 \) reach the conditional statement \( l_2 \); hence, locations \( l_0 \), \( l_1 \), and \( l_2 \) are part of the abstract system. These locations are directly relevant in testing the reachability of \( l_1 \).

Abstraction-Guided Symbolic Execution: The symbolic execution is guided along a sequence of locations (an abstract trace: \( \langle l_0, l_2, l_3 \rangle \)) in the abstract system. The program execution is guided using heuristics to intelligently rank the successor states generated at points of thread and data non-determinism. The guidance strategy uses information that \( l_3 \) is control dependent on the true branch of location \( l_2 \) and in the ranking scheme prefers the successor representing the true branch of the conditional statement.

Refinement: When the symbolic execution cannot reach the desired target of a conditional branch statement containing a global variable we refine the abstract system by adding inter-thread dependence information. Suppose, we cannot generate the successor state for the true branch of the con-
itional statement while guiding along \(\langle l_0, l_2, l_3 \rangle\) in Fig. 1, then the refinement automatically adds another definition of \(a\) to the abstract trace resulting in \(\langle l_1, l_0, l_2, l_3 \rangle\). The new abstract trace implicitly states that two different threads need to define the variable \(a\) at locations \(l_1\) and \(l_0\). Note that there is no single control flow path that passes through both \(l_1\) and \(l_0\).

**Output:** When the guided symbolic execution technique discovers a feasible execution path we output the trace. The technique, however, cannot detect infeasible errors. In such cases it outputs a “Don’t know” response.

### 3. Ranking Data Non-determinism

Generalized symbolic execution (GSE) algorithm is a powerful technique to analyze programs with symbolic input data structures [4]. During GSE, when an un-initialized symbolic object reference of type \(T\) is accessed, the reference is non-deterministically initialized to the following choices: (a) null, (b) a new instance of class \(T\), and (c) all previously initialized symbolic references of type \(T\). Step (c) allows the algorithm to account for aliasing. The initialization is termed as lazy since the initialization is delayed until the symbolic reference is first accessed during symbolic execution. The GSE approach is further extended to handle polymorphism [6, 8]. Step (b) is extended to also non-deterministically initialize instances of all sub-classes that inherit from class \(T\). Similarly, step (c) is extended to where the symbolic reference is non-deterministically initialized to all previously initialized symbolic references of type \(T’\) where \(T’\) is a subtype of \(T\). A systematic initialization with polymorphism causes a combinatorial explosion in the number of choices that are explored in the generalized symbolic execution algorithm.

We use the information in the abstract trace to rank data non-determinism choices generated in GSE when lazily initializing a reference with a polymorphic class hierarchy. We rank a program state, \(s\), at a point of complex data non-determinism for some object \(obj_{sym}\). If there exists in an abstract trace the following sequence of instructions:

\[
\begin{align*}
 l : obj_{sym}.foo() & \rightarrow l' : some_{insn} \\
\end{align*}
\]

The object \(obj_{sym}\) invokes the procedure \(foo\) at the call site, \(l\), such that \(some_{insn}\) is the first instruction to be execution at location \(l'\) in \(foo\).

The heuristic computation assigns the lowest heuristic value to a successor state, \(s' \in successors(s)\), in which \(obj_{sym}\) is initialized to objects of type \(T := getClass(l')\). The \(getClass\) function returns the class containing the program location \(l'\). The term class and type are used interchangeably. The heuristic computation allows us to exploit information about the class hierarchies in the abstract traces.

### 4. Experimental Results

We demonstrate in an empirical analysis on benchmarked multi-threaded Java programs and the JDK 1.4 concurrent libraries that locations in the abstract system can be used to generate feasible execution paths to the target locations. We show that the abstraction-guided technique can find errors in multi-threaded Java programs in a few seconds where exhaustive symbolic execution is unable to find the errors within a time bound of an hour. We use Symbolic Pathfinder to conduct the experimental study [6].

### Table 1: Effort in error discovery and abstract trace statistics.

<table>
<thead>
<tr>
<th>Model</th>
<th>States</th>
<th>Time (sec)</th>
<th>Memory (MB)</th>
<th>Total trace Length</th>
<th>Total Refinements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reorder (9.1)</td>
<td>205</td>
<td>1.57</td>
<td>76MB</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>Reorder (10.1)</td>
<td>296</td>
<td>1.67</td>
<td>85MB</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>Airline (15.3)</td>
<td>1210</td>
<td>3.23</td>
<td>5MB</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>Airline (20.2)</td>
<td>3279</td>
<td>7.46</td>
<td>65MB</td>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>Airline (20.1)</td>
<td>3099</td>
<td>7.46</td>
<td>65MB</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>VecDeadlock0</td>
<td>1570</td>
<td>4.56</td>
<td>65MB</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>VecDeadlock1</td>
<td>2948</td>
<td>6.28</td>
<td>69MB</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>VecRace</td>
<td>3120</td>
<td>7.98</td>
<td>65MB</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

VecDeadlock0, VecDeadlock1, and VecRace shown in Table 1 are examples that use the JDK 1.4 synchronized Vector library in accordance with the documentation. We use Jlint to automatically generate warnings on possible deadlocks and race-conditions in the synchronized Vector library [4]. Exhaustive symbolic execution using a depth-first search is unable to discover the errors in these models within a time bound of one hour. In the VecDeadlock0, the abstraction-guided symbolic execution only generates 1370 states and takes about 4.5 seconds to find the deadlock in the program. Similarly in the VecDeadlock1 and VecRace programs, the guided symbolic execution only generates a few thousand states before generating a concrete trace to the error.

Using the information from the abstract trace set, the heuristic to rank the non-determinism of complex data structures, and refining the abstract trace when required allows us to achieve this dramatic improvement in error discovery over exhaustive symbolic execution.

### 5. References