Efficient Testing of Concurrent Programs with Abstraction-Guided Symbolic Execution

Neha Rungta, Eric G Mercer and Willem Visser*

Dept. of Computer Science, Brigham Young University, Provo, UT 84602, USA
*Division of Computer Science, University of Stellenbosh, South Africa

Abstract. In this work we present an abstraction-guided symbolic execution technique that quickly detects errors in concurrent programs. The input to the technique is a set of target locations that represent a possible error in the program. We generate an abstract system from a backward slice for each target location. The backward slice contains program locations relevant in testing the reachability of the target locations. The backward slice only considers sequential execution and does not capture any inter-thread dependencies. A combination of heuristics are to guide a symbolic execution along locations in the abstract system in an effort to generate a corresponding feasible execution trace to the target locations. When the symbolic execution is unable to make progress, we refine the abstraction by adding locations to handle inter-thread dependencies. We demonstrate empirically that abstraction-guided symbolic execution generates feasible execution paths in the actual system to find concurrency errors in a *few seconds* where exhaustive symbolic execution fails to find the same errors in an hour.

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 [9, 25, 7, 12, 17].

Explicit state model checking enumerates all possible thread schedules and input data values of a program in order to check for errors [9, 25]. To partially mitigate the state space explosion from data input values, symbolic execution techniques substitute data input values with symbolic values [12, 24, 17]. 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.

In this work we present an abstraction-guided symbolic execution technique that efficiently detects errors caused by a combination of thread schedules and data values in concurrent programs. 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.

A set of target locations that represent a possible error in the program is provided as input to generate an abstract system. The input target locations are either generated from static analysis warnings, imprecise dynamic analysis techniques, or user-specified reachability properties. The abstract system is constructed with program locations contained in a static interprocedural backward slice for each target location and synchronization locations that lie along control paths to the target locations [10]. The static backward slice contains call sites, conditional branch statements, and data definitions that determine the reachability of a target location. The backward slice only considers sequential control flow execution and does not contain data values or inter-thread dependencies.

We systematically guide the symbolic execution toward locations in the abstract system in order to reach the target locations. A combination of heuristics are used to automatically pick thread identifiers and input data values at points of thread and data non-determinism respectively. We use the abstract system to guide the symbolic execution and do not verify or search the abstract system like most other abstraction refinement techniques [3, 8]. At points when the program execution is unable to move further along a sequence of locations (e.g. due to the value of a global variable at a particular conditional statement), we refine the abstract system by adding program statements that re-define the global variables. The refinement step adds the inter-thread dependence information to the abstract system on a need-to basis. The contributions of this work are as follows:

1. An abstraction technique that uses static backward slicing along a sequential control flow execution of the program to generate relevant locations for checking the reachability of certain target locations.
2. A guided symbolic execution technique that generates a feasible execution trace corresponding to a sequence of locations in the abstract system.
3. A novel heuristic that uses the information in the abstract system to rank data non-determinism in symbolic execution.
4. A refinement heuristic to add inter-thread dependence information to the abstract system when the program execution is unable to make progress.

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.

2 Overview

A high-level overview of the technique is shown in Fig. 1.

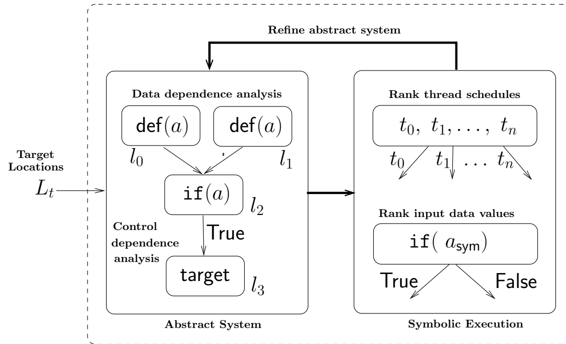

Fig. 1. Overview of the abstraction-guided symbolic execution technique

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 analysis, for example, reports program locations where lock acquisitions by unique threads may lead to a deadlock [5]. 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_3.

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 conditional 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

(a)	(b)	(c)

1: **Thread** `A`{
2: ...
3: public void run(Element *elem*){
4: `lock`(*elem*)
5: check(*elem*)
6: `unlock`(*elem*)
7: }
8: public void check(Element *elem*)
9: **if** *elem.e* > 9
10: **Throw Exception**
11: }}
	1: **Thread** `B` {
	2: ...
	3:
	public void run(Element *elem*){
	4: int x /* *Input Variable* */
	5: **if** $x > 18$
	6: `lock`(*elem*)
	7: *elem*.reset()
	8: `unlock`(*elem*)
	9: }}
		1: **Object** `Element`{
		2: int e
		3: ...
		4: public Element(){
		5: $e := 1$
		6: }
		7: public void reset(){
		8: $e := 11$
		9: }}

Fig. 2. An example of a multi-threaded program with two threads: `A` and `B`.

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 Program Model and Semantics

To simplify the presentation of the guided symbolic execution we describe a simple programming model for multi-threaded and object-oriented systems. The restrictions, however, do not apply to the techniques presented in this work and the empirical analysis is conducted on Java programs. Our programs contain conditional branch statements, procedures, basic data types, complex data types supporting polymorphism, threads, exceptions, assertion statements, and an explicit locking mechanism. The threads are separate entities. The programs contain a finite number of threads with no dynamic thread creation. The threads communicate with each other through shared variables and use explicit locks to perform synchronization operations. The program can also seek input for data values from the environment.

In Fig. 2 we present an example of such a multi-threaded program with two threads `A` and `B` that communicate with each other through a shared variable, *elem*, of type `Element`. Thread `A` essentially checks the value *elem.e* at line 9 in Fig. 2(a) while thread `B` resets the value of *elem.e* in Fig. 2(b) at line 7 by invoking the `reset` function shown in Fig. 2(c). We use the simple example in Fig. 2 through the rest of the paper to demonstrate how the guided symbolic execution technique works.

A multi-threaded program, $\mathcal{M}$, is a tuple $\langle \{\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_{u-1}\}, V_c, D_{sym} \rangle$ where each $\mathcal{T}_i$ is a thread with a unique identifier $id \rightarrow \{0, 1, \ldots, u-1\}$ and a set of local variables; V_c is a finite set of concrete variables; and D_{sym} is a finite set of all input data variables in the system. An input data variable is essentially any variable that seeks a response from the environment.

A runtime environment implements an interleaving semantics over the threads in the program. The runtime environment operates on a program state s that contains: (1) valuations of the variables in V_c, (2) for each thread, $\mathcal{T}_i$, values of its local variables, runtime stack, and its current program location, (3) the symbolic representations and values of the variables in D_{sym}, and (4) a path constraint, ϕ, (a set of constraints) over the variables in D_{sym}. The runtime environment provides a set of functions to access certain information in a program state s:

- `getCurrentLoc`(s) returns the current program location of the most recently executed thread in state s.
- `getLoc`(s, i) returns the current program location of the thread with identifier i in state s.
- `getEnabledThreads`(s) returns a set of identifiers of the threads enabled in s. A thread is *enabled* if it is not blocked (not waiting to acquire a lock).

Given a program state, s, the runtime environment generates a set of successor states, $\{s_0, s_1, \ldots, s_n\}$ based on the following rules $\forall i \in \texttt{getEnabledThreads}(s) \wedge l := \texttt{getLoc}(s, i)$:

1. If l is a conditional branch with symbolic primitive data types in the branch predicate, P, the runtime environment can generate at most two possible successor states. It can assign values to variables in D_{sym} to satisfy the path constraint $\phi \wedge P$ for the target of the true branch or satisfy its negation $\phi \wedge \neg P$ for the target of the false branch.
2. If l accesses an uninitialized symbolic complex data structure o_{sym} of type T, then the runtime environment generates multiple possible successor states where o_{sym} is initialized to: (a) null, (b) references to new objects of type T and all its subtypes, and (c) existing references to objects of type T and all its subtypes [11].
3. If neither rule 1 or 2 are satisfied, then the runtime environment generates a single successor state obtained by executing l in thread $\mathcal{T}_i$.

In the initial program state, s_0, the current program location of each thread is initialized to its corresponding start location while the variables in D_{sym} are assigned a symbolic value $v_\perp$ that represents an uninitialized value.

A state s_n is reachable from the initial state s_0 if using the runtime environment we can find a non-zero sequence of states $\langle s_0, s_1, \ldots, s_n \rangle$ that leads from s_0 to s_n such that $\forall \langle s_i, s_{i+1} \rangle$, s_{i+1} is a successor of s_i for $0 \leq i \leq n-1$. Such a sequence of program states represents a feasible execution path through the system. The sequence of program states provides a set of concrete data values and a valid path constraint over the symbolic values. The reachable state space, S, can be generated using the runtime environment where $S := \{s \mid \exists \langle s_0, \ldots, s \rangle\}$.

4 Abstraction

In this work we create an abstract system that contains program locations relevant in checking the reachability of the target locations. We then use the locations in the abstract system to guide the symbolic execution. The abstract

system is constructed with program locations contained in a static interprocedural backward slice for each target location. The abstract system also contains synchronization locations that lie along control paths to the target locations. A backward slice of a program with respect to a program location l and a set of program variables V consists of all statements and predicates in the program that may affect the value of variables in V at l and the reachability of l.

4.1 Background definitions

Definition 1. *A control flow graph (CFG) of a procedure in a system is a directed graph $G := \langle L, E \rangle$ where L is a set of uniquely labeled program locations in the procedure while $E \subseteq L \times L$ is the set of edges that represents the possible flow of execution between the program locations. Each CFG has a start location $l_{start} \in L$ and an end location $l_{end} \in L$.*

Definition 2. *An interprocedural control flow graph (ICFG) for a system with p procedures is $\langle \mathcal{L}, \mathcal{E} \rangle$ where $\mathcal{L} := \bigcup_{0 \leq i \leq p} L_i$ and $\mathcal{E} := \bigcup_{0 \leq i \leq p} E_i$. Additional edges from a call site to the start location of the callee and from the end location of a procedure back to its caller are also added in the ICFG.*

Definition 3. `icfgPath`(l, l') *describes a path in the ICFG and returns true iff there exists a sequence $q := \langle l, \ldots, l' \rangle$ such that $(l_i, l_{i+1}) \in \mathcal{E}$ where $0 \leq i \leq$* $\mathsf{length}(q) - 1$

Definition 4. `postDom`(l, l') *returns true iff for each path in a CFG between l and l_{end}, $q := \langle l, \ldots, l_{end} \rangle$, where l_{end} is an end location, and there exists an i such that $l_i = l'$ where $1 \leq i \leq \mathsf{length}(q) - 1$.*

4.2 Abstract System

The abstract system is a directed graph $\mathcal{A} := \langle L_\alpha, E_\alpha \rangle$ where $L_\alpha \subseteq \mathcal{L}$ is the set of program locations while $E_\alpha \subseteq L_\alpha \times L_\alpha$ is the set of edges. The abstract system contains target locations; call sites, conditional branch statements, and data definitions in the backward slice of each target location; and all possible start locations of the program. It also contains synchronization operations that lie along control paths from the start of the program to the target locations.

To compute an interprocedural backward slice, a backwards reachability analysis can be performed on a system dependence graph [10]. Note that the backward slice only considers sequential execution and ignores all inter-thread dependencies. Intuitively a backward slice contains: (1) call sites and the start locations of the corresponding callees such that invoking the sequence of calls leads to a procedure containing the target locations, (2) conditional statements that affect the reachability of the target locations determined using control dependency analyses, (3) data definitions that affect the variables at target locations determined from data dependence analyses, and (4) all locations generated from the transitive closures of the control and data dependences.

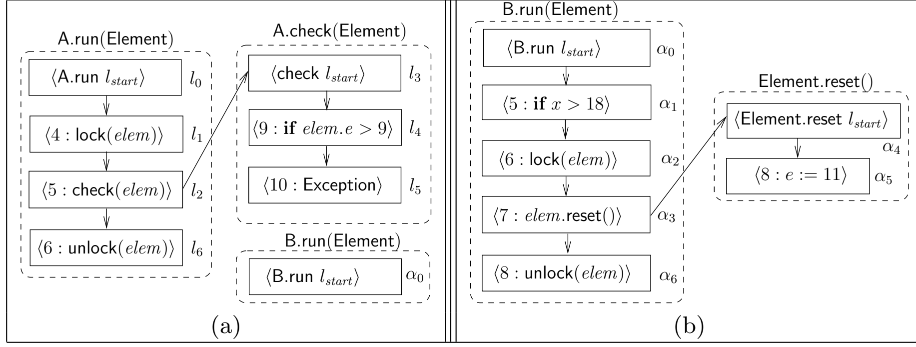

Fig. 3. The abstract system for Fig. 2: (a) Initial Abstract System. (b) Additions to the abstract system after refinement.

In order to add the synchronization locations we define the auxiliary functions $\texttt{acqLock}(l)$ that returns true iff l acquires a lock and $\texttt{relLock}(l, l')$ that returns true iff l releases a lock that is acquired at l'. For each $l_\alpha \in L_\alpha$ we update $L_\alpha := L_\alpha \cup l$ if Eq. (1) is satisfied for l.

$$[\texttt{icfgPath}(l, l_\alpha)\ \wedge \texttt{acqLock}(l)] \vee [\texttt{icfgPath}(l_\alpha, l) \wedge \texttt{relLock}(l, l_\alpha)] \quad (1)$$

After the addition of the synchronization locations and locations from the backward slices we connect the different locations. Edges between the different locations in the abstract system are added based on the control flow of the program as defined by the ICFG. To map the execution order of the program locations in the abstract system to execution order in the ICFG we check the post-dominance relationship between the locations while adding the edges. An edge between any two locations l_α and l'_α in L_α is added to E_α if Eq. (2) evaluates to true.

$$\forall (l''_\alpha \in L_\alpha)\ \textit{such that}\ \neg\texttt{postDom}(l_\alpha, l''_\alpha) \vee \neg\texttt{postDom}(l''_\alpha, l'_\alpha) \quad (2)$$

The abstract system for the example in Fig. 2 where the target location is line 10 in the `check` method in Fig. 2(a) is shown in Fig. 3(a). Locations l_0 and α_0 in Fig. 3(a) are the two start locations of the program. The target location, l_5, represents line 10 in Fig. 2(a). Location l_2 is a call site that invokes start location l_3 that reaches target location l_5. The target location is control dependent on the conditional statement at line 9 in Fig. 2(a); hence, l_4 is part of the abstract system in Fig. 3(a). The locations l_1 and l_6 are the lock and unlock operations. The abstract system shows Thread `B` is not currently relevant in testing the reachability of location l_5.

4.3 Abstract Trace Set

The input to the guided symbolic execution is an abstract trace set. The abstract trace set contains sequences of locations generated on the abstract system, $\mathcal{A}$,

from the start of the program to the various target locations in L_t. We refer to the sequences generated on the abstract system as *abstract traces* to distinguish them from the sequences generated on the CFGs. To construct the abstract trace set we first generate intermediate abstract trace sets, $\{P_0, P_1, \dots P_{t-1}\}$, that contain abstract traces between start locations of the program (L_s) and the input target locations (L_t); hence, $P_i := \{\pi | \pi \textit{ satisfies Eq. } (3) \textit{ and Eq. } (4)\}$. We use the array indexing notation to reference elements in π, hence, $\pi[i]$ refers to the ith element in π.

$$\exists l_0 \in L_s, l_t \in L_t \textit{ such that } \pi[0] == l_0 \wedge \pi[\mathsf{length}(\pi) - 1] == l_t \quad (3)$$

$$(\pi[i], \pi[i+1]) \in E_\alpha \wedge (i \neq j \implies \pi[i] \neq \pi[j]) \textit{ for } 0 \leq i, j \leq \mathsf{length}(\pi) - 1 \quad (4)$$

Eq. (4) generates traces of finite length in the presence of cycles in the abstract system caused by loops, recursion, or cyclic dependencies in the program. Eq. (4) ensures that each abstract trace generated does not contain any duplicate locations by not considering any back edges arising from cycles in the abstract system. We rely on the guidance strategy to drive the program execution through the cyclic dependencies toward the next interesting location in the abstract trace; hence, the cyclic dependencies are not encoded in the abstract traces that are generated from the abstract system.

Each intermediate abstract trace set, P_i, contains several abstract traces from the start of the program to a single target location $l_i \in L_t$. We generate a set of final abstract trace sets as:

$$\mathit{\Pi}_{\mathcal{A}} := \{\{\pi_0, \dots, \pi_{t-1}\} | \pi_0 \in P_0, \dots, \pi_{t-1} \in P_{t-1}\}$$

Each $\mathit{\Pi}_\alpha \in \mathit{\Pi}_{\mathcal{A}}$ contains a set of abstract traces. $\mathit{\Pi}_\alpha := \{\pi_{\alpha_0}, \pi_{\alpha_1}, \dots, \pi_{\alpha_{t-1}}\}$ where each $\pi_{\alpha_i} \in \mathit{\Pi}_\alpha$ is an abstract trace leading from the start of the program to a unique $l_i \in L_t$. Since there exists an abstract trace in $\mathit{\Pi}_\alpha$ for each target location in L_t, $|\mathit{\Pi}_\alpha| == |L_t|$.

The input to the guided symbolic execution technique is $\mathit{\Pi}_\alpha \in \mathit{\Pi}_{\mathcal{A}}$. The different abstract trace sets in $\mathit{\Pi}_{\mathcal{A}}$ allow us to easily distribute checking the feasibility of individual abstract trace sets on a large number of computation nodes. Each execution is completely independent of another and as soon as we find a feasible execution path to the target locations we can simply terminate the other trials.

In the abstract system shown in Fig. 3(a) there is only a single target location—line 10 in `check` procedure shown in Fig. 2(a). Furthermore, the abstract system only contains one abstract trace leading from the start of the program to the target location. The abstract trace $\mathit{\Pi}_\alpha$ is a singleton set containing $\langle l_0, l_1, l_2, l_3, l_4, l_5 \rangle$.

5 Guided Symbolic Execution

We guide a symbolic program execution along an abstract trace set, $\mathit{\Pi}_\alpha := \{\pi_0, \pi_1, \dots, \pi_{t-1}\}$, to construct a corresponding feasible execution path, $\mathit{\Pi}_s :=$

```
1: /*  backtrack := ∅,  A_α := Π_α,  s := s_0,  trace := ⟨s_0⟩ */
procedure main()
 2: while ⟨s, Π_α, trace⟩ ≠ null do
 3:    ⟨s, Π_α, trace⟩ := guided_symbolic_execution(s, Π_α, trace)
 4:
procedure guided_symbolic_execution(s, Π_α, trace)
 5: while ¬(end_state(s) or depth_bound(s) or time_bound()) do
 6:    if goal_state(s) then
 7:       print trace exit
 8:    ⟨s', S_s⟩ := get_ranked_successors(s, Π_α)
 9:    for each s_other ∈ S_s do
10:       backtrack := backtrack ∪ {⟨s_other, Π_α, trace ∘ s_other⟩}
11:    if ∃ π_i ∈ Π_α, head(π_i) == getCurrentLoc(s) then
12:       l_α := head(π_i)   /* First element in the trace */
13:       l'_α := head(tail(π_i)) /* Second element in the trace */
14:       if branch(l_α) ∧ (l'_α ≠ getCurrentLoc(s')) then
15:          return ⟨s_0, A_α := refine_trace(A_α, π_i), ⟨s_0⟩⟩
16:       remove(π_i, l_α) /* This updates the π_i reference in Π_α */
17:    s := s',  trace := trace ∘ s'
18: return ⟨s', Π_α, trace⟩ ∈ backtrack
```

Fig. 4. Guided symbolic execution pseudocode.

$\langle s_0, s_1, \ldots, s_n \rangle$. For an abstract trace set, the guided symbolic execution tries to generate a feasible execution path that contains program states where the program location of the most recently executed thread in the state matches a location in the abstract trace. The total number of locations in the abstract trace is $m := \sum_{\pi_i \in \Pi_\alpha} \texttt{length}(\pi_i)$ where the `length` function returns the number of locations in the abstract trace π_i. In our experience, the value of m is a lot smaller than n, $m << n$ where n is the length of the feasible execution trace corresponding to Π_α, because the abstract traces contain large control flow gaps between two locations in the abstract trace. The intermediate program locations that are not part of the backward slice are also not included in the abstract system or the resulting abstract traces.

The pseudocode for the guided symbolic execution is presented in Fig. 4. On line 1 we initialize the *backtrack* set as empty, store a copy of the input abstract trace set Π_α in A_α, set program state s to the initial program state s_0, and add s_0 to the feasible execution trace. On line 3, `main` invokes `guided_symbolic_execution` where the values of the elements in the tuple are $\langle s_0, \Pi_\alpha, \langle s_0 \rangle \rangle$. A time and depth bound are specified by the user as the termination criteria of the symbolic execution.

The guided symbolic program execution is a greedy depth-first search that picks the best immediate successor of the current state and does not consider unexplored successors until it reaches the end of a path and needs to backtrack. The search is executed along a path in the program until it reaches an end state (a state with no successors), a user-specified depth bound (line 5), a user-specified time bound (line 2), or the goal state (line 6). In the *goal state*, s, there exists a unique thread at each target location ($\forall l_i \in L_t, \exists j \in$

getEnabledThreads(s), getLoc$(s, j) == l_i$). If the state s is the goal state (line 6) then the feasible execution *trace* is printed before exiting the search. In this scenario we are successfully able to find a corresponding execution trace that includes each location in the abstract trace set. The guided symbolic execution technique is guaranteed to terminate even if the goal state is not reachable because it is depth and time bounded.

States are assigned a heuristic rank in order to intelligently guide the program execution. The get_ranked_successors function returns a tuple $\langle s', S_s \rangle$ on line 8 in Fig. 4 where s' is the best ranked successor of state s while all the other successors are in set S_s. Each $s_{other} \in S_s$ is added to the backtrack set with the abstract trace set and the feasible execution trace (lines 9 and 10). The feasible execution trace added to the backtrack set with s_{other} denotes a feasible execution path from s_0 to s_{other}. The best-ranked state s' is assigned as the current state and the feasible execution trace is updated by concatenating s' to it using the $\circ$ function (line 17). The $\circ$ function returns the concatenation of two input lists.

In order to match a location in the abstract trace set to a program state, the algorithm checks whether the program location of the most recently executed thread in state s matches the first location in an abstract trace, $\pi_i \in \Pi_\alpha$ (line 11). The head function returns the first element of the input abstract trace. The tail function returns the input abstract trace without its head. Location l_α is the first location in π_i while l'_α is the immediate successor of l_α. Location l_α is removed from the abstract trace (line 16) if refinement is not needed. Removing l_α updates π_i and in turn updates Π_α. The execution now attempts to match the location of the most recently executed thread in the current state toward the next location in π_i by directing the search.

The abstract trace set on line 15 is immediately refined when the program execution is unable to reach the desired target of a conditional branch statement that contains a global variable in its predicate. The refinement is performed on the abstract trace set A_α (a copy of the original unmodified abstract trace set Π_α). After the refinement the search is restarted from the initial program state s_0 and the updated abstract trace set A_α. The details on the refinement process are given in Section 6.

The get_ranked_successors(s, Π_α) in Fig. 4 takes as input a program state s and the abstract trace set Π_α. For each successor state s'_i of s we compute its heuristic value using a two-tier and data ranking scheme. The two-tier ranking scheme has been described in earlier works [19, 20]. In the first-tier rank program states along execution paths that correspond to more locations from the input abstract trace set are ranked better than others [19]. The second-level rank is an estimate of the distance from the program state to the next program location in any of the abstract traces in Π_α [20].

The abstract trace contains conditional branch statements where the outcome of its branch predicate determines the reachability of the input target locations. When the path constraints for both outcomes of a conditional branch with primitive symbolic data types is satisfiable, the second-level heuristic uses

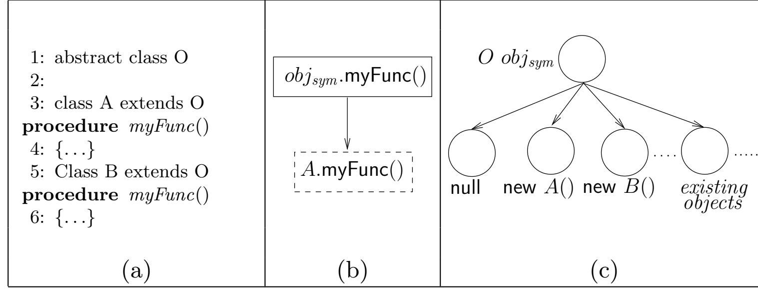

Fig. 5. Ranking data non-determinism for complex data structures. (a) Classes A and B inherit from class O. (b) Locations in an abstract trace. (c) Different non-determinism choices for obj_{sym} of type O.

information from the abstract trace to assign a better rank to the state at the desired outcome of the conditional branch.

New in this work, we use the information in the abstract trace to rank data non-determinism choices generated in the symbolic execution for complex input data structures. We rank s'_i at a point of complex data non-determinism for some object obj_{sym}. If there exists in an abstract trace in Π_α a call site l where obj_{sym} is the object that invokes the procedure containing the start location l', then we prefer successor states where obj_{sym} is initialized to objects of type $T :=$ `getClass`(l'). The `getClass` function returns the class containing the program location l'. The $h_3(s'_i) := 0$ if obj_{sym} points to an object of type T; otherwise, $h_3(s'_i) := 1$.

In Fig. 5(a), two classes A and B inherit from the abstract base class O and implement the `myFunc` method. Fig. 5(b) is an abstract trace where obj_{sym} is a symbolic object of type O that invokes the `myFunc` method in class A. Fig. 5(c) shows the non-deterministic choices: (1) null, (2) new instance of class A or B, and (3) existing objects of type A and B to account for aliasing [11]. The information in Fig. 5(b) indicates that the obj_{sym}.`myFunc` call needs to invoke the `myFunc` method in class A. This allows us to pick a state where the complex data structure is of type A. Information in the abstract trace about types of the objects required to reach target locations allows us to guide the symbolic execution to the target locations.

6 Refinement

The refinement process is invoked when the symbolic execution cannot reach the target of the branch statement in an abstract trace, π_i. The branch predicate contains global variables that can be possibly redefined by other threads. The global variables can either be concrete or symbolic. In an effort to execute the needed branch condition a location that redefines a global variable in the branch predicate is added to the abstract trace. This allows us to account for inter-thread

```
procedure refine_trace(A_α, π_i)
1: l_branch := head(π_i)
2: L_v := {l_v | defines(l_v, global_vars(l_branch))}
3: Update the 𝒜 /* generate backward slices for l_v ∈ V and synchronization locs(Section 4.2) */
4: π_v := get_abstract_trace(L_v)
5: π_pre := ⟨l_0, ..., l_k⟩ such that ∃⟨l_0 ... l_k⟩ ∘ π_i ∈ A_α
6: if ∃l_a ∈ π_pre, l_b ∈ π_v, same_lock(l_a, l_b) then
7:     π_v := π_v ∘ l'_b where releaseLock(l'_b, l_b)
8: π_new := π_v ∘ π_pre
9: A_α.replace_trace(π_pre ∘ π_i, π_new ∘ π_i)
```

Fig. 6. Refinement pseudocode.

dependencies that affect the reachability of the target locations. We define some additional functions that are used to describe the refinement process.

- `same_lock`(l_a, l_b) returns true iff `acqLock`$(l_a) \wedge$`acqLock`(l_b) such that l_a and l_b acquire the lock on the same object determined using a *may-alias* algorithm.
- `get_abstract_trace`(L_v) returns an abstract trace from the start of a program to $l_v \in L_v$.
- Π_α.`replace_trace`(π_i, π_j) substitutes π_i with π_j in Π_α.

The refinement process is shown in Fig. 6. The first element of the abstract trace, π_i, is a branch statement as assumed on line 1 of Fig. 6. To generate a set of program locations, L_v, on line 2 the `defines` function returns a set of program locations where global variables in the branch predicate of l_{branch} are redefined. The abstract system, $\mathcal{A}$, is updated from locations in backward slices that affect the reachability of $l_v \in L_v$ and additional synchronization locations. In essence, the process to generate locations and edges for the target location is repeated now with locations in $l_v \in L_v$. The `get_abstract_trace` returns an abstract trace in the abstract system from the start of the program to some location in L_v.

There can be many threads in the program that define a particular global variable of interest. We randomly pick a $l_v \in L_v$ and generate an abstract trace from the start of the program to l_v in $\mathcal{A}$. When there are multiple abstract traces to l_v then we, again, randomly pick an abstract trace. This refinement strategy is a heuristic that is forcing the symbolic execution to try and reach a program location where a global variable of interest is redefined and then *again* check whether the desired target location of l_{branch} is reachable.

The abstract trace set A_α is updated with a new abstract trace that contains additional locations leading to the definition of a variable used in the branch predicate. In Fig. 6, $\pi_v := \langle l_0, \ldots, l_v \rangle$ is an abstract trace from the start of the program to location, l_v, that defines a variable in the branch predicate. The abstract trace π_{pre} is the prefix of the trace π_i in the original abstract trace set. The prefix denotes the sequence of locations from the start of the program up to, and not including, the conditional branch statement that cannot

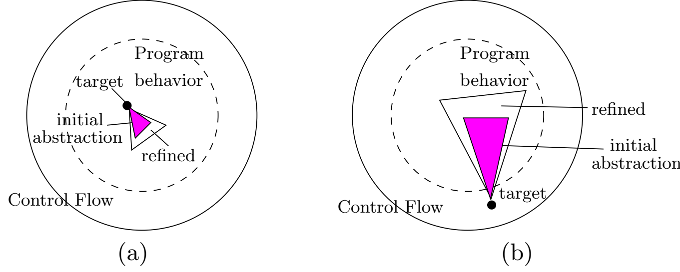

Fig. 7. Abstraction and refinement in context of the program behavior and control flow. (a) Target is reachable. (b) Target is not reachable.

reach the desired target. This allows the refinement heuristic to add inter-thread dependence information that is ignored in the original abstraction.

In order to generate the replacement abstract trace we check the lock dependencies between π_{pre} and π_v. If π_{pre} and π_v acquire the lock on the same object (line 6), then we add the corresponding lock relinquish location to π_v (line 7). Adding the lock relinquish location ensures that if one thread acquires a lock to define a variable in the branch predicate, then after the definition another thread is not blocked trying to acquire the same lock to reach the conditional statement. A new prefix, π_{new}, is essentially created by combining π_v and π_{pre}. This operation adds to the abstract trace the definition of a variable in the branch predicate before the conditional statement.

Finally we replace in the abstract trace set A_α the abstract trace corresponding to $\pi_{pre} \circ \pi_i$ with $\pi_{new} \circ \pi_i$ (line 9). The guided symbolic execution is now restarted from the initial program state s_0 and guided along the updated abstract trace set.

Suppose, $A_\alpha := \{\langle l_0, l_1, l_2, l_3, l_4, l_5 \rangle\}$ and $\pi_i := \langle l_4, l_5 \rangle$ for the example in Fig. 2. In the runtime environment we have found a feasible execution trace that visits locations l_0 to l_3, but at the conditional branch l_4 the execution cannot reach the desired target location l_5. The refinement process shown in Fig. 6 adds new locations and edges shown in Fig. 3(b) to the abstract system in addition to the ones shown in Fig. 3(a). In Fig. 3(b) location α_5 defines the integer field, e, of the shared variable *elem*; $\pi_v := \langle \alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5 \rangle$ such that the sequence leads from the start of the program to α_5 in Thread `B`. The prefix of π_i is $\pi_{pre} := \langle l_0, l_1, l_2, l_3 \rangle$. Locations l_1 and α_2 in Fig. 3(a) and Fig. 3(b) respectively acquire the lock on the same object *elem*; hence, we add the lock release location to $\pi_v := \pi_v \circ \alpha_6$. Finally the guided symbolic execution is restarted from s_0 and $A_\alpha := \{\pi_v \circ \pi_{pre} \circ \pi_i\}$.

7 Discussion

The abstraction is an under-approximation of the control flow of the system, and the refinement adds more information as needed. In Fig. 7(a) and (b) the

outer-most circle represents the control flow of the program while the dashed circle represents the actual behaviors possible in the program. The control flow of the program is an over-approximation of all possible behaviors of the program. In Fig. 7(a) and (b) the initial abstraction is represented by the shaded triangle. The initial abstraction generated from the backward slice is an under-approximation of the control flow of the system. It attempts to carve out parts of the control flow relevant in checking the feasibility of the target locations.

Consider the two possible cases. In the first case, the target location, initial abstraction, and the refined slice are all contained within the realm of the program behavior in Fig. 7(a). When the symbolic execution is unable to reach the target location based on the information in the initial abstraction, the abstraction is refined (represented by triangle enclosing the shaded triangle). More information is added to the original under-approximation. Since the target is contained within the reachable part of the program behavior the guided symbolic execution is effective in discovering the error. In the second case shown in Fig. 7(b), the target location is contained outside the program behavior. The approach in this work is unable to state its infeasibility since the refinement is a heuristic. In this case, the time bound is used as the termination criteria.

8 Experimental Results

We conduct experiments on machines with 8 GB of RAM and two Dual-core Intel Xeon EM64T processors (2.6 GHz). We use the symbolic extension of the Java PathFinder (JPF) v4.1 model checker with partial order reduction turned on [17]. The symbolic execution extension uses Choco Solver (`http:choco-solver.net`) to check the satisfiablity of the path constraints. JPF uses a modified JVM that operates on Java bytecode. This allows us to model the libraries as part of the program.

We present an empirical study on five multi-threaded Java programs. In Table 1, for each program we show the type of error, source lines of code (SLOC), total time taken in seconds to generate the set of abstract traces (Time), total number of abstract trace sets tested (Traces Sets), and total memory used (Memory). To pick the initial abstract trace sets we choose sets that contain traces with the smallest number of call sites leading from the start of the program to each target location. For the programs used in this empirical study, we were able to discover the goal state with the initial abstract trace sets.

The guided symbolic execution trials for the different abstract trace sets reported in Table 1 are launched in parallel on different computation nodes since each trial is completely independent of the other trials. When a feasible execution trace is generated along an abstract trace set, we terminate the other trials. We present the total number of states generated, total time taken, and total memory used in the trial that generates a feasible execution trace corresponding to the abstract trace set in Table 2. We also show the length of the initial trace ($\sum_{\pi_i \in \Pi_\alpha} |\pi_i|$) and total number of refinements performed on the abstract trace; Π_α is the input abstract trace set. The parameters with the program

	Error Type	SLOC	Time (secs)	Trace Sets	Memory MB
Reorder	Reachability	44	0.28	5	1.93 MB
Airline	Reachability	31	0.30	3	1.58 MB
VecDeadlock0	Deadlock	7267	1.21	5	38 MB
VecDeadlock1	Deadlock	7169	0.98	17	38 MB
VecRace	Race	7151	0.92	8	39 MB

Table 1. Information on models and abstract trace generation.

names indicate the thread configuration of a particular program. Each parameter represents the total number of symmetric threads in the system.

The `Reorder` and the `Airline` model are benchmarked examples and have user-defined reachability properties. These models do not contain any data non-determinism. The results of the models are used to demonstrate the effectiveness of the abstraction refinement technique in guiding a concrete program execution to an error state. The `Airline` model required a larger number of refinements because the reachability of the target location depends on the value of a global counter that is modified by different threads. The refinement adds the location where the global variable is defined at each iteration.

We created C# programs for the `Reorder` and `Airline` models to evaluate on CHESS, a stateless concurrency testing tool [13, 14]. The models with the thread configuration presented in this paper, CHESS was unable to find an error within a time bound of one hour. CHESS uses an iterative context-bound approach that bounds the number of preemptions along a certain path in order to reach the error faster. Note that the correct number of preemption points required to find the error was provided as input. Random walk and randomized depth-first search in JPF have also been shown ineffective to find errors in these models [18].

`VecDeadlock0`, `VecDeadlock1`, and `VecRace` 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 [2]. Each model has two symbolic variables whose specific values in addition to certain thread schedules are required to manifest errors in the `Vector` library. 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 allows us to achieve this dramatic improvement in error discovery over exhaustive symbolic execution.

Model	States	Time secs	Memory MB	Total trace Length	Total Refinements
Reorder (9,1)	205	1.67	7MB	13	1
Reorder (10,1)	236	1.67	7MB	13	1
Airline (15,3)	1210	3.23	5MB	3	13
Airline (20,2)	3279	7.46	6MB	3	19
Airline (20,1)	3609	7.46	6MB	3	20
VecDealock0	1370	4.56	66MB	14	1
VecDeadlock1	2948	6.89	69MB	15	2
VecRace	3120	7.98	65MB	12	1

Table 2. Effort in error discovery and abstract trace statistics.

9 Related Work

Recent work by Tomb *et al.* uses symbolic execution to generate concrete paths to null pointer exceptions at an inter-procedural level in sequential programs [24]. In contrast, concolic testing executes the program with random concrete values in conjunction with symbolic execution to collect the path constraints over input data values [23, 22]. The cost of constraint solving in concolic testing to achieve full path coverage in a concurrent system is extremely high. The techniques presented in this work are complementary to concolic testing. The techniques can also be used to efficiently guide concolic testing.

Recent work shows that guiding the concrete program execution along a sequence of manually generated program locations relevant in verifying the feasibility of the target location dramatically lowers the time taken to reach the target location [19]. The manual aspect of generating relevant program locations is tedious and sometimes intractable.

Race-directed random testing of concurrent programs uses the output of imprecise dynamic analysis tools and randomly drives threads to the input locations [21]. The work in [19] shows that guiding the search through key locations relevant in determining the target locations yields significantly better error discovery rates. Dynamic analysis tools such as ConTest use heuristics to randomly add perturbations in the thread schedules [6]. The results are similar to those obtained with just a stateless random search and it is not very effective in error discovery. Chess is a concurrency testing tool that systematically explores thread schedules in *C#* programs and supports iterative context bounding [13].

Model checking is a formal approach for systematically exploring all possible behaviors of a concurrent software system [9, 7, 25, 3]. The state space explosion problem renders it intractable in verifying medium to large-sized programs. Conservative abstractions are applied to high-level programming languages [8, 3] in order to verify programs. The abstraction is iteratively refined if it generates an infeasible counter-example to an error state. Counter-example guided abstraction refinement techniques are successful in verifying sequential programs; however, they are not effective for testing concurrent programs.

Related works in hardware verification guide the simulation of the concrete model using an abstract model of boolean variables that represent the transition

relation [15, 16]; however, these works are limited to verifying circuit designs and boolean programs. The techniques cannot be extended to verify complex concurrent software systems. Another area of related work is the use of abstract databases and heuristics that are used to guide the searches in planning problems [4].

10 Conclusions and Future Work

In this work we present an abstraction-guided symbolic execution technique that efficiently detects errors caused by thread schedules and data values in concurrent programs. Using backward slices for the input target locations the technique automatically generates an abstract system with relevant locations in checking the reachability of the target locations. The backward slices only consider sequential execution along the control flow of the program. The symbolic execution is guided along traces in the abstract system to generate a corresponding feasible execution path to the target locations. When the symbolic execution is unable to make progress we refine the abstraction by adding locations to handle inter-thread dependencies.

In the case when we are unable to discover a feasible execution path, we want to design a probabilistic measure to estimate the likelihood of the reachability of the target locations as future work. Another avenue of future work consists of studying more precise refinement techniques based on compositional symbolic execution [1].

11 Acknowledgements

This work was in part supported by Google Summer of Code 2008. We thank Patrice Godefroid at Microsoft Research and the anonymous reviewers for their comments on a previous version of this work.

References

1. S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 367–381. Springer, 2008.
2. C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Proc. ASWEC*, page 68, Washington, DC, USA, 2001. IEEE Computer Society.
3. T. Ball and S. Rajamani. The SLAM toolkit. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. CAV*, volume 2102 of *LNCS*, pages 260–264, Paris, France, July 2001. Springer-Verlag.
4. S. Edelkamp. Planning with pattern databases. In *Proc. European Conference on Planning*, pages 13–24, 2001.
5. D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proc. SOSP '03*, pages 237–252, New York, NY, USA, 2003. ACM Press.

6. Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):267–279, 2007.
7. P. Godefroid. Model checking for programming languages using Verisoft. In *Proc. of POPL*, pages 174–186, New York, NY, USA, 1997. ACM.
8. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In T. Ball and S. Rajamani, editors, *Proc. SPIN Workshop*, volume 2648 of *LNCS*, pages 235–239, Portland, OR, May 2003.
9. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
10. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 39(4):229–243, 2004.
11. S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. *Proc. TACAS*, pages 553–568, 2003.
12. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
13. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6):446–455, 2007.
14. M. Musuvathi and S. Qadeer. Fair stateless model checking. In *Proc. of PLDI*, pages 362–371, New York, NY, USA, 2008. ACM.
15. K. Nanshi and F. Somenzi. Guiding simulation with increasingly refined abstract traces. In *Proc. DAC*, pages 737–742, New York, NY, USA, 2006. ACM.
16. F. M. D. Paula and A. J. Hu. An effective guidance strategy for abstraction-guided simulation. In *Proc. DAC '07*, pages 63–68, New York, NY, USA, 2007. ACM.
17. C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proc. ISSTA*, pages 15–26, New York, NY, USA, 2008. ACM.
18. N. Rungta and E. G. Mercer. Hardness for explicit state software model checking benchmarks. In *Proc. SEFM*, pages 247–256, London, U.K, September 2007.
19. N. Rungta and E. G. Mercer. A meta heuristic for effectively detecting concurrency errors. In *Haifa Verification Conference*, Haifa, Israel, 2008.
20. N. Rungta and E. G. Mercer. Guided model checking for programs with polymorphism. In *Proc. PEPM*, pages 21–30, New York, NY, USA, 2009. ACM.
21. K. Sen. Race directed random testing of concurrent programs. *SIGPLAN Not.*, 43(6):11–21, 2008.
22. K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Proc. HVC*, volume 4383 of *LNCS*, pages 166–182. Springer, 2007.
23. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. ESEC/FSE-13*, pages 263–272, New York, NY, USA, 2005. ACM.
24. A. Tomb, G. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *Proc. ISSTA*, pages 97–107, New York, NY, USA, 2007. ACM Press.
25. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.