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
*SEVEN Networks, 901 Marshall Street, Redwood City, CA 94063, USA

Abstract. Exhaustive search techniques such as model checking and symbolic execution are insufficient to detect errors in concurrent programs. In this work we present an abstraction-guided symbolic execution technique that quickly detects errors in concurrent programs that arise from thread schedules and input data values. An abstract system is generated that contains a set of key program locations that are relevant in testing the feasibility of a possible error in the program. We guide a symbolic execution along locations in the abstract system in an effort to generate a corresponding feasible execution trace to the error location. A combination of heuristics are used to automatically rank thread and data non-determinism in order to guide the execution. 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 [19, 41, 15, 22, 27].

Explicit state model checking enumerates all possible thread schedules and input data values of a program in order to check for errors [19, 41]. Whereas symbolic execution techniques substitute certain data values with symbolic values while all other values are concrete [22, 40, 27]. 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 automatically identifies a set of key program locations that can potentially lead to an error state in the system.

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 that are more likely to contain an error and decrease the total time required to detect errors.

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. We automatically generate an abstract system containing locations that are relevant in verifying the reachability of the target locations using control and data dependence analyses. The abstract system contains call sites, conditional branch statements, data definitions, and synchronization points in the program that lie along control paths from the start of the program to the target locations. The program locations in the abstract system do not contain any thread information or data values.

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. The abstract system simply provides locations of interest in reaching the target locations. We do not verify or search the abstract system like most other abstraction refinement techniques [3, 18]. At points in the execution when we are unable to guide the program execution further along a sequence of locations (e.g. a particular conditional statement) we refine the abstract system by adding program statements that re-define the variables of interest.

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.

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 [11]. In this case, the target locations are the set of lock acquisition locations generated by the lockset analysis.

Abstract System: An abstraction of the program is generated using standard control and data dependence analyses on control flow graphs. 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 variable a at locations l_0 and l_1 reach the conditional statement l_2; hence,

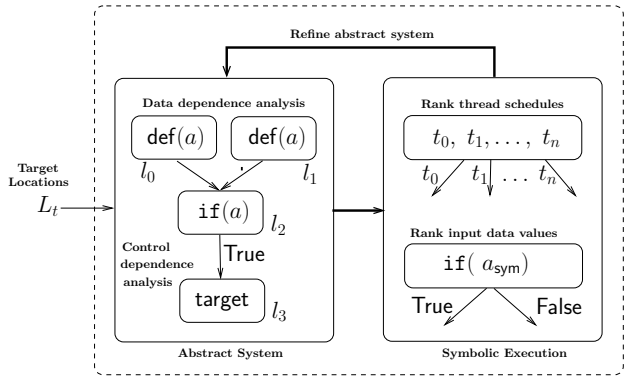

Fig. 1. Overview of the abstraction-guided symbolic execution technique

locations l_0, l_1, and l_2 are part of the abstract system because they 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 in an abstract trace, we refine the abstract system and the abstract trace by adding program statements that re-define the variables in the branch predicate. If 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 $\langle l_1, l_0, l_2, l_3 \rangle$. It is possible that different threads define the variable a at locations l_1 and l_0.

Output: When the guided symbolic execution technique discovers a feasible execution path, the path describes various conditions that lead to the target locations.

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

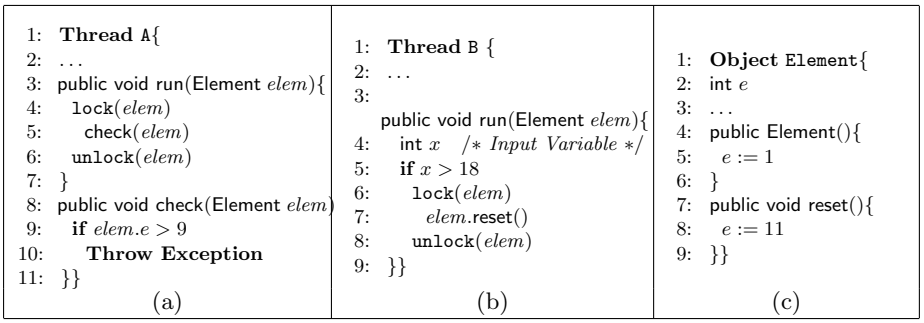

Fig. 2. An example of a multi-threaded program with two threads: `A` and `B`.

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 $\langle \mathcal{C}_s, V_c, D_{sym} \rangle$ where $\mathcal{C}_s$ is a set of threads $\{\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_{u-1}\}$, V_c is a finite set of concrete variables, and D_{sym} is a finite set of all data input variables in the system. Each thread, $\mathcal{T}_i$, has an unique identifier *id* where $id \rightarrow \{0, 1, \ldots, u-1\}$ and a set of local variables.

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 path constraint is represented in some first-order logic formula that *can be solved* using a theorem prover or a constraint solver. 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, $\{s_0, s_1, \ldots, s_n\}$. To generate all successors of a given state, the runtime environment systematically generates the successor states 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.
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 program locations and use the abstraction to guide the symbolic execution. The control flow of the system and dependence relationships between the program locations and the target locations are used to construct the abstract system.

4.1 Background definitions

A control flow graph (CFG) of a procedure in a system is a directed graph $\langle L, E \rangle$ 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 $l_{end} \in L$.

For a system with p procedures the control flow of the entire system is $\langle \mathcal{L}, \mathcal{E} \rangle$ where $\mathcal{L} := \bigcup_{0 \leq i \leq p} L_i$ and $\mathcal{E} := \bigcup_{0 \leq i \leq p} E_i$. We define the following functions to access information about the locations and edges in the CFGs:

- $\texttt{start}(l)$ returns true iff l is a start location.
- $\texttt{end}(l)$ returns true iff l is an end location.
- $\texttt{callSite}(l)$ returns true iff l is a call site that invokes a procedure.
- $\texttt{branch}(l)$ returns true iff l is a conditional branch statement.
- $\texttt{aquireLock}(l)$ returns true iff l acquires a lock.
- $\texttt{releaseLock}(l, l')$ returns true iff l releases a lock that is acquired at l'.
- $\texttt{callEdge}(l, l')$ returns true iff $\texttt{callSite}(l) \wedge \texttt{start}(l') \wedge$ l invokes l'.

There are two kinds of paths that denote the control flow in the system: a path within a CFG (intraprocedural path) and a path across different CFGs (interprocedural path). An intraprocedural or interprocedural path is simply a non-zero sequence of locations ($q := \langle l_i \ldots, l_{i+n} \rangle$) that satisfies certain conditions. The intraprocedural and interprocedural paths are defined using the following functions respectively:

- $\texttt{cfgPath}(l_i, l_n)$ returns true iff $\exists \langle l_i, \ldots, l_n \rangle,\ (l_j, l_{j+1}) \in \mathcal{E}$ for $i \leq j \leq n-1$.
- $\texttt{icfgPath}(l_i, l_n)$ returns true iff $\exists \langle l_i, \ldots, l_n \rangle, (l_j, l_{j+1}) \in \mathcal{E} \vee \texttt{callEdge}(l_j, l_{j+1})$ for $i \leq j \leq n-1$.

The `icfgPath` function describes a path between two locations across different procedures that does not contain any return (l_{end}) locations; it is a sequence of calls required to reach a particular procedure.

Data and control dependences are an integral part in constructing the abstract system. The dependence analyses are defined along intraprocedural paths in the CFG. Data dependence is primarily based on *reaching definitions*; whereas control dependence is determined by whether the outcomes of branch predicates in conditional statements affect the reachability of certain locations. We define the following functions for the dependence analyses as:

- $\texttt{postDom}(l_i, l_n)$ returns true iff $\forall \langle l_i, \ldots, l_m \rangle,\ \texttt{cfgPath}(l_i, l_m) \wedge \texttt{end}(l_m) \wedge \exists (l_j == l_n)$ for $i+1 \leq j \leq m$.
- $\texttt{defines}(l, v)$ returns true iff l defines variable v.
- $\texttt{uses}(l, v)$ returns true iff l uses variable v.
- $\texttt{reaches}(l_i, l_n)$ returns true iff $\exists \langle l_i, \ldots, l_n \rangle,\ \texttt{cfgPath}(l_i, l_n) \wedge \texttt{defines}(l_i, v) \wedge \neg \texttt{defines}(l_j, v) \wedge \texttt{uses}(l_n, v)$ for $i+1 \leq j \leq n-1$.
- $\texttt{controlD}(l_i, l_n)$ returns true iff $\exists \langle l_i, \ldots, l_n \rangle,\ \texttt{cfgPath}(l_i, l_n) \wedge \texttt{branch}(l_i) \wedge \texttt{postDom}(l_j, l_n) \wedge \neg \texttt{postDom}(l_i, l_n)$ for $i+1 \leq j \leq n-1$.

4.2 Abstract System

The locations in an abstract system, $\mathcal{A}$, are a subset of the locations in the control flow of the entire system. The abstract system contains locations that are call sites, conditional branch statements, data definitions, and synchronization operations that lie along control paths from the start of the program to the target locations.

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 is constructed based on the set of input target locations L_t and the CFGs of the system $\langle \mathcal{L}, \mathcal{E} \rangle$. We initialize the set of abstract locations, L_α, with the set of target locations L_t and the set of all possible start locations of the program L_s. The set L_s contains the start location of each thread, $\mathcal{T}_i$, in the system. We initialize the set $L_\alpha := L_t \cup L_s$ and iteratively add locations, $l \in \mathcal{L}$, to L_α if one of the following four equations is satisfied. We continue to add locations until we reach a fixpoint, re-evaluating the four equations each time a location is added.

$$\begin{aligned} &\exists l_t \in L_t, l_s \in L_s, l' \in \mathcal{L}, [\texttt{icfgPath}(l, l_t) \wedge \texttt{icfgPath}(l', l_t)] \wedge \\ &[\texttt{icfgPath}(l_s, l) \wedge \texttt{icfgPath}(l_s, l')] \wedge [\texttt{callEdge}(l, l') \vee \texttt{callEdge}(l', l)] \end{aligned} \quad (1)$$

The call sites are added to L_α one at a time by satisfying Eq. (1); the call sites are part of method sequences such that invoking a particular sequence leads from the start of the program to a procedure containing a target location. In addition to the call sites, start locations of the procedures invoked by the call sites are also added one at a time to the set of locations L_α when Eq. (1) evaluates to true.

$$\begin{aligned} &\exists l_\alpha \in L_\alpha,\ l' \in \mathcal{L},\ [\texttt{cfgPath}(l, l_\alpha) \wedge \texttt{cfgPath}(l', l_\alpha)] \wedge \\ &\big\{ [\texttt{cfgEdge}(l, l') \wedge \texttt{controlD}(l, l') \wedge \texttt{controlD}(l, l_\alpha)] \vee \\ &\quad [\texttt{cfgEdge}(l', l) \wedge \texttt{controlD}(l', l) \wedge \texttt{controlD}(l', l_\alpha)] \big\} \end{aligned} \quad (2)$$

Conditional branch statements that determine the reachability of the locations that are already present in the abstract system are added to L_α whenever Eq. (2) is satisfied. The branch statements that result from any nested control dependence are also added. Furthermore, the immediate target of a conditional branch statement is added when Eq. (2) evaluates to true where the execution of the target depends on the same branch outcome as l_α. This allows the desired target of the branch to be encoded in the abstract trace.

$$\begin{aligned} &\exists l_\alpha \in L_\alpha,\ \texttt{cfgPath}(l, l_\alpha) \wedge \texttt{defines}(l, v) \wedge \\ &\texttt{isBranch}(l_\alpha) \wedge \texttt{uses}(l_\alpha, v) \wedge \texttt{reaches}(l, l_\alpha) \end{aligned} \quad (3)$$

Locations that define variables used in branch predicates at the conditional statements in L_α are added if Eq. (3) is satisfied. To compute the reaching definitions we conservatively compute the alias information based on the notion of *maybe an alias*. If two variables in a given procedure can be aliases of one another we assume they are aliases. In the `reaches` function the alias information is only propagated along a path in the CFG; in other words at an intraprocedural level (single procedure). The alias computation does not consider aliases among variables passed as parameters to different procedures. This restriction allows us to generate a smaller set of locations in the abstract system and we rely on refinement to find other definitions if/when they are needed.

$$\begin{aligned} &\exists l_\alpha \in L_\alpha,\ [\texttt{cfgPath}(l, l_\alpha)\ \wedge \texttt{acquireLock}(l)] \vee \\ &\qquad [\texttt{cfgPath}(l_\alpha, l) \wedge \texttt{releaseLock}(l, l_\alpha)] \end{aligned} \quad (4)$$

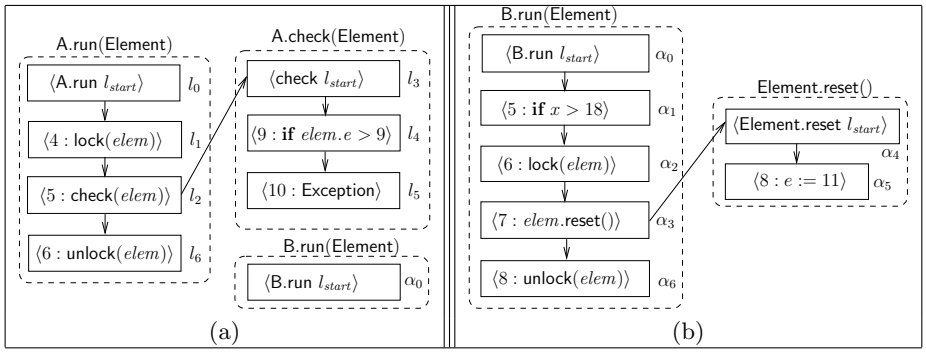

Fig. 3. The abstract system for Fig. 2: (a) Initial Abstract System. (b) Additions to the abstract system after refinement.

Eq. (4) adds locations for synchronization operations. Locations where locks are relinquished corresponding to the lock acquisition locations are also added to L_α using the `releaseLock` function.

Termination: Loops and recursive procedure calls can cause cyclic dependencies between different branch statements, data definitions, and call sites. Program locations that are part of cyclic dependencies, however, are added only once to L_α since L_α is a set of unique locations; this guarantees termination.

Finally, edges are added between the different locations in the abstract system. An edge between any two locations l_α and l'_α in L_α is added to E_α if either Eq. (5) or Eq. (6) evaluates to true.

$$\exists \langle l_\alpha, l_i, \ldots, l_n, l'_\alpha \rangle,\ \texttt{cfgPath}(l_\alpha, l'_\alpha)\ \wedge l_j \notin L_\alpha \textit{ for } i \leq j \leq n \tag{5}$$

$$\texttt{callEdge}(l_\alpha, l'_\alpha) \tag{6}$$

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, $\{P_0, P_1, \ldots P_{t-1}\}$, that contain abstract traces between start locations of the program (L_s) and the input target locations (L_t). For each target location, $l_i \in L_t$, we use Eq. (7) to create a set of abstract traces, P_i, where each abstract trace leads from the start of the program to l_i.

$$\forall l_0 \in L_s,\ P_i := \{\langle l_0, l_1, \ldots, l_i\rangle \mid l_j \neq l_k,\ \langle l_l, l_{l+1}\rangle \in E_\alpha \\ \textit{for } j \neq k \textit{ and } 0 \leq j, k \leq i \textit{ and } 0 \leq l \leq i-1\} \tag{7}$$

Eq. (7) 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. (7) 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:

$$\Pi_{\mathcal{A}} := \{\{\pi_0, \ldots, \pi_{t-1}\} | \pi_0 \in P_0, \ldots, \pi_{t-1} \in P_{t-1}\}$$

Each $\Pi_\alpha \in \Pi_{\mathcal{A}}$ contains a single abstract trace from the start of the program to a target location. In essence, $\Pi_\alpha := \{\pi_{\alpha_0}, \pi_{\alpha_1}, \pi_{\alpha_{t-1}}\}$ where each $\pi_{\alpha_i} \in \Pi_\alpha$ is an abstract trace from the start of the program to a unique $l_i \in L_t$ and $|\Pi_\alpha| = |L_t|$.

The input to the guided symbolic execution technique is $\Pi_\alpha \in \Pi_{\mathcal{A}}$. The different abstract trace sets in $\Pi_{\mathcal{A}}$ 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 Π_α 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, $\Pi_\alpha := \{\pi_0, \pi_1, \ldots, \pi_{t-1}\}$, to construct a corresponding feasible execution path, $\Pi_s := \langle s_0, s_1, \ldots, s_n\rangle$. For an abstract trace set, the guided symbolic execution tries

```
 1: /* backtrack := ∅, A_α := Π_α, s := s_0, trace := ⟨s_0⟩ */
procedure main()
 2: while within_time_bound() and ⟨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)) 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) /* Update Trace */
17:    s := s', trace := trace ∘ s'
18: return ⟨s', Π_α, trace⟩ ∈ backtrack
```

Fig. 4. Guided symbolic execution pseudocode.

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. Intermediate program locations are 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).

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 is immediately refined when the program execution is unable to execute the program along the desired target of a conditional branch statement in an abstract trace π_i on line 15 in Fig. 4. The refinement is invoked when the current values of the variables in the branch predicate l_α do not result in the desired successor s' of s. 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. The runtime environment generates a set of successors $\{s'_0, s'_1, \ldots, s'_x\}$ for the program state s. For each successor state s'_i, we compute its heuristic value using a two-tier and data ranking scheme. The two-tier ranking scheme has been described in earlier works [32, 33] while ranking the data non-determinism is new to this work.

First-level Rank: $h_1(s'_i) := \sum_{\pi_j \in \Pi_\alpha} \texttt{length}(\pi_j)$ is the first level rank of the program state. Intuitively, program states along execution paths that correspond to more locations from the input abstract trace set are ranked better than others [32]. States with lower heuristic values are ranked higher. The second-level rank is used to prioritize the states that have the same first-level rank

Second-level Rank: The second-level rank of s'_i is an estimate of the distance from the program state to the next program location in any of the abstract

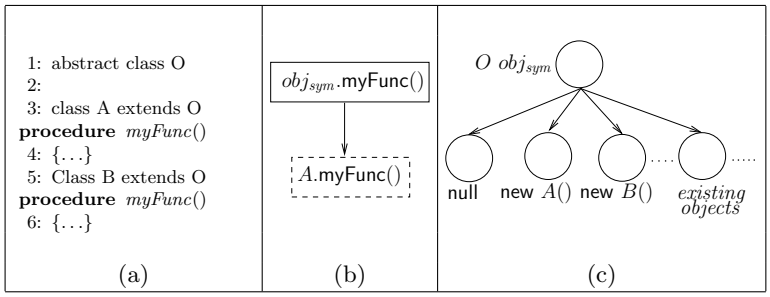

Fig. 5. Ranking data non-determinism for complex data structures. (a) Classes A and B inherit from class O. (b) Locations in an abstract trace. (c) Different non-determinism choices for obj_{sym} of type O.

traces in Π_α [33]. We get the set of next program locations of all the abstract traces and the current program location of s'_i: $L_p := \{l \mid \texttt{head}(\pi_j), \pi_j \in \Pi_\alpha\}$ and $l_a := \texttt{getCurrentLoc}(s'_i)$. The distance heuristic generates a set of all possible paths, Q, between l_a and $\forall l_b \in L_p$ on the CFGs $\langle \mathcal{L}, \mathcal{E} \rangle$.

The paths in q satisfy: $\forall q_k \in Q,\ \forall (l, l'),\ \texttt{cfgEdge}(l, l') \vee \texttt{callEdge}(l, l') \vee \texttt{end}(l)$ and l returns to l'). The last disjunct essentially accounts for paths along return statements. The distance heuristic generates a finite number of paths even in the presence of looping constructs and recursive procedures by short-circuiting the path when it detects a cycle. The final estimate minimizes the estimates across the different paths: $h_2(s'_i) := \texttt{length}(q_{min} \in Q)$. The details of the heuristic computation are provided in [33].

Ranking Data Non-Determinism: New in this work we rank the non-deterministic choices that are generated 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 $T := \texttt{getClass}(l')$. 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, $h_3(s'_i) := 1$. The information in the abstract trace allows us to intelligently pick objects of types that lead to the target location.

Consider the program shown in Fig. 5(a). Two classes A and B inherit from the abstract base class O and implement the `myFunc` method. Suppose the sequence shown in Fig. 5(b) is part of an abstract trace where obj_{sym} is a symbolic object of type O that invokes the `myFunc` method in class A. Consider the example shown in Fig. 5(c). There is a non-deterministic choice for a new instance of class A or B when accessing an uninitialized object of type O. The new object can also be existing objects of type A and B to account for aliasing. The example in Fig. 5(b) shows that the obj_{sym}.`myFunc` call needs to invoke the `myFunc` method in class A and assign a lower heuristic value to the state.

procedure refine_trace(A_α, π_i)
1: $l_{branch} :=$ `head`(π_i)
2: $L_v = \{l_v \mid$ `defines`$(l_v, v)\}$
3: $L_\alpha := L_\alpha \cup L_v$, Recompute the fixpoint for $\mathcal{A}$ (Section 4.2)
4: $\pi_v :=$ get_abstract_trace(L_v)
5: $\pi_{pre} := \langle l_0, \ldots, l_k \rangle$ such that $\exists \langle l_0 \ldots l_k \rangle \circ \pi_i \in A_\alpha$
6: **if** $\exists l_a \in \pi_{pre}, l_b \in \pi_v,$ `same_lock`(l_a, l_b) **then**
7: $\quad \pi_v := \pi_v \circ l'_b$ where `releaseLock`(l'_b, l_b)
8: $\pi_{new} := \pi_v \circ \pi_{pre}$
9: A_α.replace_trace$(\pi_{pre} \circ \pi_i, \pi_{new} \circ \pi_i)$

Fig. 6. Refinement pseudocode.

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 variables in the branch predicate can either be concrete, symbolic, global, or thread-local. In an effort to execute the needed branch condition in the abstract trace we add locations in the abstract system that redefine variables in the branch predicate. The algorithm for `refine_trace`(A_α, π_i) is shown in Fig. 6. We define some additional functions that are used to describe the refinement process.

- `same_lock`(l_a, l_b) returns true iff `acquireLock`$(l_a) \wedge$ `acquireLock`(l_b) such that l_a and l_b acquire the lock on the same object.
- `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 the abstract trace set 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 we select locations where thread-local and global variables in the branch predicate are redefined. We use an *interprocedural data flow analysis* to generate L_v. We compute an alias analysis through different procedures to check if variables passed to different procedures can alias the same object. We also compute the `reaches` definition along interprocedural paths. Note this more expensive data flow analysis is invoked on a *need-to* basis in the refinement phase when the original intraprocedural data flow analysis was insufficient to generate the key locations in determining the reachability of the target locations.

The fixpoint for the abstract system, $\mathcal{A}$, is recomputed to add the locations that are relevant in checking the reachability of all $l_v \in L_v$. The abstract system is now modified and contains a new set of locations and edges. The `get_abstract_trace` returns an abstract trace in the abstract system from the start of the program to some location in L_v. 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.

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 reach the desired target.

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 $A_\alpha := \{\pi_v \circ \pi_{pre} \circ \pi_i\}$.

The refinement process can be invoked repeatedly for the same branch condition it is possible the same definition of the variable is added multiple times. Such a scenario allows us to handle the cases where the variable needs to be over a certain value in the branch predicate and its value is incremented by some variable or constant in the definition. The refinement strategy is in itself a heuristic. Developing and evaluating other refinement strategies is a work in progress.

	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.

7 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 [27]. JPF uses a modified JVM that operates on Java bytecode. The abstraction-refinement and the heuristic computation processes are also performed on the Java bytecode. This allows us to model any libraries used by the program as part of the multi-threaded system.

We present an empirical study on five multi-threaded Java programs shown in Table 1. The `Reorder` and the `Airline` model are benchmarked examples while the `VecDeadlock0`, `VecDeadlock1`, and `VecRace` are examples that use the JDK 1.4 synchronized `Vector` library in accordance with the documentation. The programs are part of the JPF examples repository and can be obtained by downloading JPF from [21]. We use Jlint to automatically generate warnings on possible deadlocks and race-conditions in the synchronized `Vector` library [2]. The `Reorder` and `Airline` programs have user-defined reachability properties. The errors in the `Vector` library are caused by a combination of data values and thread schedules. For each program, in Table 1, 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). The parameters with the program names indicate the thread configuration of a particular program. Each parameter represents the total number of symmetric threads in the system.

Picking Abstract Traces: A large number of abstract trace sets can be generated from the abstract system. 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. Several abstract trace sets may contain the smallest number of call sites. In Table 1, the number of abstract trace sets reported are the ones generated for the smallest number of call sites. For the programs used in this empirical study, we were able to discover the goal state by simply using the initial abstract trace sets.

We present in Table 2 the results for generating a concrete feasible execution trace corresponding to an abstract trace for the programs used in our empirical study. The guided symbolic execution trials for the different abstract trace sets reported in Table 1 are launched in parallel on different computation nodes since

Model	States	Time secs	Memory MB	Total trace Length	Total Refinements
Reorder (9,1)	205	0.874	7MB	13	1
Reorder (10,1)	239	0.875	7MB	13	1
Airline (15,3)	619	0.548	5MB	3	13
Airline (20,2)	1230	0.521	6MB	3	19
Airline (20,1)	1248	0.498	6MB	3	20
VecDealock0	37	1.003	66MB	14	1
VecDeadlock1	294	1.005	69MB	15	2
VecRace	312	1.032	65MB	12	1

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

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. 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 results in Table 2 indicate that the guided symbolic execution technique can quickly generate a concrete feasible execution to a corresponding abstract trace. In the `VecDeadlock0`, the technique only generates 37 states and takes about 1 second to find the deadlock in the program. Similarly in the `VecDeadlock1` and `VecRace` programs, the guided symbolic execution only generates a few hundred states before generating a concrete trace to the error. Exhaustive symbolic execution using a depth-first search is unable to discover the errors in the programs used in the empirical study within a time bound of one hour.

In the examples shown in Table 2 most models require only one or two refinements to find the goal state. In these models one or two perturbations to global variables were required to elicit the errors in the system. A related heuristic, the iterative context-bounding approach bounds the number of preemptions along a certain path in order to reach the error faster [23]; however, for the configurations of the models used in this study the iterative context-bounding approach with a preemption bound of two, the Chess concurrency testing tool was unable to find the error in a time-bound of one hour even with no data non-determinism [24]. Note that we created corresponding $C\#$ programs. 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. In this case even with the correct number of preemptions as the input the iterative context-bounding approach was unable to discover an error. ConTest, [12], was also unable to find an error in the models with the thread configurations used in the empirical study in a 1000 trails (these experiments too were conducted in the absence of data non-determinism).

8 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 [40]. 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 [36, 35]. 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 [32]. 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 [34]. The work in [32] 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 [12]. 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 [23]. These tools only report feasible errors but they require a closed environment and cannot handle data non-determinism. Note that there are imprecise dynamic analysis techniques that also report possible errors in the system. The output of such tools can also be used to generate input for the abstraction-guided symbolic execution technique.

Check 'n' Crash, [5], is a hybrid test technique that uses a constraint solver to generate concrete test cases based on the output from the static analyzer tool—ESC/Java. Check 'n' Crash, however, only generates test cases for safety violations in sequential programs. DSD crasher [6], extends Check 'n' Crash by adding information from a runtime analysis tool to ESC/Java to improve its analysis. It too is, however, limited to generating test cases for sequential programs.

Static analysis techniques ignore the actual execution environment of the program and reason about errors by simply analyzing the source code of the program. Warlock and ESC/Java are two static analysis tools that rely heavily on program annotations to find deadlocks and race-conditions [38, 14]. Annotating existing code is cumbersome and time consuming. FindBugs and JLint look for suspicious patterns in Java programs [20, 2]. Error warnings reported by static analysis tools have to be manually verified which is difficult and sometimes not possible.

Model checking is a formal approach for systematically exploring all possible behaviors of a concurrent software system [19, 28, 41, 3]. The state space explosion problem renders it intractable in verifying medium to large-sized programs. Partial order reduction techniques have been used to partially overcome the explosion in the state-space of the system due to thread-schedules [13]. Such reductions are complementary to the technique presented in this paper. In our experimental results we rank the thread schedules with dynamic partial order reduction turned on.

Conservative abstractions are applied to high-level programming languages [18, 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. In this work, we use the abstraction to guide the execution of the program and do not verify the abstraction itself—a key difference between the abstraction-guided symbolic execution technique and other abstraction-refinement techniques.

Heuristics have extensively been used for error detection in program and system verification. Hamming distance heuristics presented by Yang and Dill use the explicit state representation to estimate a bit-wise distance between the current state and an error state [42]. Edelkamp, Lafuente, and Leue implemented a property based heuristic search which considers the minimum number of changes required to the variables in the property in order for the property to be violated. This information is used to estimate the distance to the error [8]. This heuristic was refined using a Bayesian meta-heuristic by Seppi, Jones and Lamborn [37]. The heuristics in the FLAVERS tool uses the structure of the property to guide the search [39]. The property based heuristics are not very effective for finding errors in object-oriented multi-threaded Java and C# programs because a large number of operations in the program do not directly affect the property being verified.

The heuristics in [16] exploit the properties of Java programs to find concurrency errors. A variety of domain specific heuristics are proposed to find different concurrency errors; for example, the `most-blocked` heuristic prefers states with a greater number of blocked threads in order to find deadlocks, while the `prefer-thread` heuristic allows the user to specify a set of threads whose execution will be preferred over the other threads. The `prefer-thread` heuristic is effective for error discovery in certain programs where exhaustive model checking techniques fail to find an error as shown in [31]; however, a considerable manual effort is expended while configuring the correct parameters required for error discovery in the models.

Distance heuristics are structural heuristics that have been extensively evaluated in this work, [10, 4, 29, 30]. In essence, the success of these heuristics lie in the fact that they exploit the structure of the program to drive the program execution toward a set of interesting locations either specified by the user or generated using static/dynamic analysis techniques. The combination of distance estimate heuristics with the meta heuristic that guides the search along

a sequence of program locations is considerably successful in detecting errors in multi-threaded programs [32]. The sequence of program locations generated manually essentially represents an abstract trace through the program while the distance heuristic enables us to find a corresponding feasible concrete execution path.

There are other related guided techniques that use different abstract traces and guidance strategies for either error discovery or optimal counter-example generation. The trail directed model checking generates a concrete counter-example to the error state using a depth-first search and uses the counter-example produced to guide the search toward an optimal counter-example [9]. The goal of the distance heuristic presented in this paper with the meta heuristic, [32], is to *discover errors* in programs where exhaustive search techniques such as depth-first search fail.

The deterministic execution technique uses a sequence of relevant data input to execute branch conditions, thread schedules, and method sequences generated manually by a tester to check whether an error exists in concurrent Java programs [17]. This, however, requires a significant amount of manual effort. The distance heuristic can intelligently rank thread schedules to drive certain threads along a small sequence of interesting locations.

Using an abstraction to guide a concrete execution of the system has also been explored in hardware verification. One approach generates a trace on an abstract model created using a set of initial boolean variables to represent the transition relation [25]. Next, a guided simulation using pseudo random number vectors guides the simulation of the concrete model to find a concrete counter-example. It refines the abstraction by adding more boolean variables. This work, however, is limited to verifying circuit designs and boolean programs. Other approaches use different abstraction and guidance techniques but, again, are limited to boolean programs [26].

Another area of related work is the use of abstract databases and heuristics that are used to guide the searches in planning problems [7].

9 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. Based on a set of input target locations the technique automatically generates an abstract system that contains relevant locations in checking the reachability of the target locations. The symbolic execution is guided along traces in the abstract system to generate a corresponding feasible execution path to the target locations. Heuristics are used to efficiently rank thread and data non-determinism to guide the symbolic execution along the locations in the abstract system. We empirically show that the abstraction-guided refinement technique can find errors in multi-threaded Java programs in a few seconds where exhaustive search techniques are unable to find errors within a time bound of an hour.

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].

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. J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification. In *International Conference on Software Engineering*, pages 37–46, 2001.
5. C. Csallner and Y. Smaragdakis. Check 'n' Crash: combining static checking and testing. In *Proc. ICSE*, pages 422–431, New York, NY, USA, 2005. ACM Press.
6. C. Csallner and Y. Smaragdakis. DSD-Crasher: a hybrid analysis tool for bug finding. In *Proc. ISSTA*, pages 245–254, New York, NY, USA, 2006. ACM.
7. S. Edelkamp. Planning with pattern databases. In *Proc. European Conference on Planning*, pages 13–24, 2001.
8. S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *Proceedings of the 7th International SPIN Workshop*, number 2057 in Lecture Notes in Computer Science. Springer-Verlag, 2001.
9. S. Edelkamp, A. L. Lafuente, and S. Leue. Trail-directed model checking. In S. D. Stoller and W. Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.
10. S. Edelkamp and T. Mehler. Byte code distance heuristics and trail direction for model checking Java programs. In *Proc. MoChArt*, pages 69–76, 2003.
11. 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.
12. 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.
13. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. POPL*, pages 110–121, New York, NY, USA, 2005. ACM.
14. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245, New York, NY, USA, 2002. ACM.
15. P. Godefroid. Model checking for programming languages using verisoft. In *Proc. of POPL*, pages 174–186, New York, NY, USA, 1997. ACM.
16. A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *Proc. ISSTA*, pages 12–21, 2002.

17. C. Harvey and P. Strooper. Testing Java monitors through deterministic execution. page 61, Washington, DC, USA, 2001. IEEE Computer Society.
18. 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.
19. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
20. D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.
21. Java PathFinder Website. http://javapathfinder.sourceforge.net.
22. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
23. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6):446–455, 2007.
24. M. Musuvathi and S. Qadeer. Fair stateless model checking. In *Proc. of PLDI*, pages 362–371, New York, NY, USA, 2008. ACM.
25. K. Nanshi and F. Somenzi. Guiding simulation with increasingly refined abstract traces. In *Proc. DAC*, pages 737–742, New York, NY, USA, 2006. ACM.
26. 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.
27. 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.
28. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. *ACM SIGSOFT Software Engineering Notes*, 28(5):267–276, September 2003.
29. N. Rungta and E. G. Mercer. A context-sensitive structural heuristic for guided search model checking. In *Proc. ASE*, pages 410–413, Long Beach, California, USA, November 2005.
30. N. Rungta and E. G. Mercer. An improved distance heuristic function for directed software model checking. In *Proc. FMCAD*, pages 60–67, Washington, DC, USA, 2006. IEEE Computer Society.
31. N. Rungta and E. G. Mercer. Generating counter-examples through randomized guided search. In *Proceedings of the 14th International SPIN Workshop on Model Checking of Software*, pages 39–57, Berlin, Germany, July 2007. Springer–Verlag.
32. N. Rungta and E. G. Mercer. A meta heuristic for effectively detecting concurrency errors. In *Haifa Verification Conference*, Haifa, Israel, 2008.
33. 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.
34. K. Sen. Race directed random testing of concurrent programs. *SIGPLAN Not.*, 43(6):11–21, 2008.
35. 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.
36. 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.
37. K. Seppi, M. Jones, and P. Lamborn. Guided model checking with a bayesian meta-heuristic. *Fundamenta Informaticae*, 70(1-2):111–126, 2006.
38. N. Sterling. Warlock— a static data race analysis tool. In *USENIX Technical Conference Proceedings*, pages 97–106, 1993.

39. J. Tan, G. S. Avrunin, L. A. Clarke, S. Zilberstein, and S. Leue. Heuristic-guided counterexample search in flavers. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth International symposium on Foundations of software engineering*, pages 201–210, New York, NY, USA, 2004. ACM Press.
40. 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.
41. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.
42. C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *35th Design Automation Conference (DAC98)*, pages 599–604, 1998.