Automated Systematic Testing of Open Distributed Programs

Koushik Sen and Gul Agha
Department of Computer Science
University of Illinois at Urbana-Champaign, USA.
{ksen,agha}@cs.uiuc.edu

Abstract. We present an algorithm for automatic testing of distributed programs, such as Unix processes with inter-process communication, Web services, etc. Specifically, we assume that a program consists of a number of asynchronously executing concurrent processes or actors which may take data inputs and communicate using asynchronous messages. Because of the large numbers of possible data inputs as well as the asynchrony in the execution and communication, distributed programs exhibit very large numbers of potential behaviors. Our goal is two fold: to execute all reachable statements of a program, and to detect deadlock states. Specifically, our algorithm uses simultaneous concrete and symbolic execution, or *concolic execution*, to explore all distinct behaviors that may result from a program's execution given different data inputs and schedules. The key idea is as follows. We use the symbolic execution to generate data inputs that may lead to alternate behaviors. At the same time, we use the concrete execution to determine, at runtime, the partial order of events in the program's execution. This enables us to improve the efficiency of our algorithm by avoiding many tests which would result in equivalent behaviors. We describe our experience with a prototype tool that we have developed as a part of our Java program testing tool jCUTE.

1 Introduction

Open distributed programs consist of asynchronous processes which communicate with each other using asynchronous messages and which may also receive data inputs from the environment. Unix process and web services are two examples of open distributed programs. The problem of testing such programs is a difficult one because of the large number of potential behaviors that they may exhibit, both because the number of possible inputs is unbounded and because there are many possible orders of execution of distributed events.

In this paper, we focus on the problem of testing for reachability of statements in distributed programs. Determining whether a statement is reachable is, in some cases, undecidable. Our goal is to automatically and efficiently find inputs and orderings which cover a large subset of the reachable statements in a program. Note that our algorithm may also detect some deadlock states during testing. Our testing algorithm builds on two ideas: *concolic execution* and *runtime partial order reduction*.

Concolic testing extends symbolic execution based testing [14, 16, 25, 17, 24] as follows. In symbolic execution, a program is executed using symbolic variables in place of concrete values for inputs. Each conditional expression in the program

represents a constraint that determines an *execution path*. Observe that the feasible executions of a program can be represented as a tree, where the branch points in a program are internal nodes of the tree. The goal is to explore all feasible execution paths of a program [16]. The classic approach is to use depth-first exploration of the paths by backtracking [24]. Unfortunately, for large or complex program, it is computationally intractable to precisely maintain and solve the constraints required for test generation.

Concolic testing removes the limitations of symbolic execution based testing [12, 21]. Specifically, the algorithm uses simultaneous concrete and symbolic execution, or *concolic execution*, to explore distinct behaviors that may result from a program's execution given different data inputs. The key idea is as follows. We use the symbolic execution to generate data inputs that may lead to alternate behaviors. At the same time, we use the concrete execution to guide the symbolic execution along a distinct execution path. The concrete execution is also used to simplify symbolic expression that cannot be handled by our constraint solver.

For the purpose of testing for reachability and deadlocks, the behavior of a distributed program may be defined by the partial order of events at the processes, where an event is defined as the execution of a statement by a process. Testing must account for nondeterminism in the order of events, not just indeterminacy of data inputs. Moreover, the nondeterminism in the order of events arises both from the asynchrony in scheduling of processes and the delay in message delivery. Our testing algorithm forces the computation along an execution schedule which represents a particular choice for both kinds of nondeterminism.

Two difficult problems have to be addressed by our algorithm. First, concolic testing has to incorporate efficient control of the execution schedules. We use concrete executions to not only guide the symbolic execution, but also to compute the *happens before partial order relation* [7] on the events. This relation is used to determine a distinct schedule which, in general, corresponds to a different partial order. Second, we have to track symbolic expressions and constraints across process boundaries in a distributed setting.

Note that the runtime partial reduction technique we use is more involved than the standard partial order reduction [23, 18, 10, 8]: we track both symbolic constraints and the "happens-before" relation at runtime. Moreover, our partial order reduction is dynamic as the partial order is computed at runtime. This helps us to track the partial order accurately by eliminating some of the approximations required in a static analysis technique for standard partial order reduction [11].

Because our algorithm is designed to explore execution paths of a distributed program, we term our approach *Explicit Path Model Checking*. To the best of our knowledge, our algorithm is the first to consider *both inputs and schedules* for message-passing distributed programs. While other approaches [11, 13, 15, 8] have considered testing for different schedules, they use either finite domains or random values for the inputs. Moreover, our algorithm is always sound – any bugs that it reports are real. Our algorithm is complete only under certain

assumptions – namely, when our constraint solver can handle all constraints that are generated and every execution is finite. More importantly, it can significantly increase coverage as compared to testing using random inputs.

In Section 3, we describe a simple model of distributed programs which we use in Section 5 and 6 to describe our algorithm. Essentially, the model corresponds to actors [1, 2]. This allows us to describe the algorithm independent of any particular programming language. We have implemented our algorithm as a part of the tool jCUTE [20], which we have developed to test general multithreaded programs written in Java. Section 7 describes some preliminary experiments using this tool. Note that the algorithm can also be used for C programs by extending CUTE [21] with Unix processes and IPC libraries.

2 Other Related Work

A number of approaches [11, 13, 15, 8] for testing distributed programs explore all possible distinct partial orders for fixed inputs. Specifically, the approaches in [11, 13] use static partial order reduction to avoid exploring some of the different executions that have the same partial order. A reason for redundant explorations is that there are approximations associated with static analysis. One way to address this problem is to use dynamic analysis to guarantee that exactly one interleaving from each partial order is explored [15]. The approach involves storing partial orders that have already been explored; this can become a memory bottleneck. Dynamic partial order reduction [8] removes the memory bottleneck in [15] at the cost of possibly exploring more than one interleaving for each partial order. The approach in [8] works for programs which have no data input and use persistent sets. On the other hand, our dynamic partial order reduction approach is based on the macro-step Actor semantics [2]. A clear distinction between the two approaches would require a study of the adaptation of the approach in [8] to asynchronous message-passing distributed systems. The adaptation of dynamic partial order reduction to multithreaded programs is shown in [8]. In our recent work [20] extending concolic testing to multithreaded programs, we develop a new technique for dynamic partial order reduction. We believe the technique, called *race-flipping*, can be more efficient than that in [8].

Model checking tools [6, 22] based on static analysis have been developed to detect bugs in shared memory concurrent programs. These tools employ partial order reduction techniques to reduce search space. Testing shared memory multi-threaded programs using symbolic execution [24] has been developed by extending Java Pathfinder.

A number of approaches [5, 4, 9] have been developed to explore all possible global states of program that can be inferred by observing a single execution of a distributed program. These techniques are orthogonal to our algorithm and may be combined with our testing tool to enable it to also explore all reachable global states.

3 Programming Model

In order to simplify the description of our testing approach, we define a simple asynchronous message-passing imperative concurrent language MPIL (Figure 1).

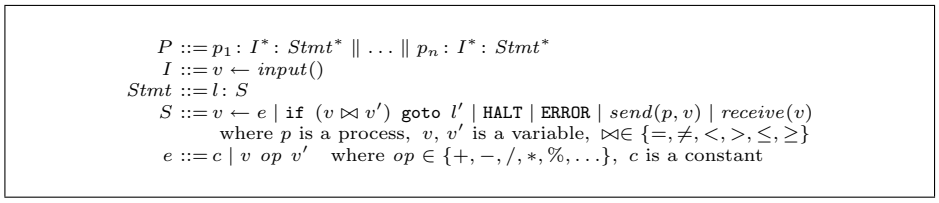

Fig. 1. Syntax of MPIL

MPIL extends the simple language presented in [21] with message-passing primitives. An MPIL program is a set of processes that are executed concurrently, where each process executes a sequence of statements. Processes in a program communicate by passing messages asynchronously. The semantics of the language is closely related to the actor semantics–each process implements an actor [2]. However, we assume that all executions terminate or the program has deadlocked.

For brevity and simplicity, we assume that new processes are not created during an execution of a program. The extension to handle these is fairly straightforward and, in fact, our implementation handles it. Moreover, an MPIL program may receive data inputs from its environment. We assume that all such inputs are available as needed; again this assumptions simplifies the description of our algorithm: our Java programs can get data inputs at any time during an execution.[1] To further simplify our exposition we assume that an MPIL program has no pointers – in fact, we assume that all variables are of type integers. However, as in [21], our algorithm can be extended to programs with pointers and complex data structures, and this is done in the implementation.

3.1 Interleaving Semantics

We now informally describe the semantics of MPIL. Consider an MPIL program P consisting of a set of processes $\mathcal{P} = \{p_1, \ldots, p_n\}$, where p_i first gets a sequence of inputs and then executes a sequence of statements, each of which is labeled. If l is the label of a statement in some process, then $l + 1$ is the label of the next statement in that process, unless the statement labeled by l is a `HALT` or an `ERROR`. The label of the initial statement of a process p is given by l_0^p. To simplify the description of the macro-step semantics (see Section 3.2), we assume that the initial statement of each process is always of the form $receive(v)$.

A program may only have variables of type integer. Variables are always local to a process; they cannot be shared among processes. A process in a program can communicate with another process by sending messages using the primitive *send*. $send(p, v)$ sends the content of the variable v to the process p. In the semantics of MPIL, an execution of the statement $send(p, v)$ by a process p' adds the content of v to the message queue of process p. The message queue of a process is a list of values. We will use Q_p to denote the message queue of process p, $|Q_p|$ to denote

[1] The reason this assumption does not reduce the generality of our algorithm is easy to see. Inputs are essentially unconstrained messages. Since we test for all potential external behaviors, any values of the data inputs are possible in response to any output of the program. Thus considering values as available from the beginning of the execution does not constrain the contexts in which the program is tested.

the number of elements in the queue, and $Q_p[i]$ to denote the i^{th} element in the message queue. We assume that at the beginning of execution the message queue of process p_1 contains a message with content 0. A process can receive a message by calling the primitive *receive*(v). On executing *receive*(v), a process waits if its message queue is empty. Otherwise, the process *non-deterministically* picks a message from its message queue, removes the message from the queue, assigns the content of the message to v, and continues executing the next statement. The non-determinism in picking the message models the asynchrony associated with message passing.

Before executing any statement, an MPIL program gets input using the command $v \leftarrow$*input*(). This command assigns the input data to the variable v. Observe that *input*() captures the various functions through which a program may receive data from its external environment. We assume that the execution of a command of the form $v \leftarrow$*input*() is always non-blocking.

A process is said to be *active* if it has not already executed a `HALT` or an `ERROR` statement. A process is said to be *enabled* if the process is active, and the processes' message queue is non-empty if the next statement to be executed by the process is *receive*.

The operational semantics of a program in MPIL is given using a (default) scheduler which represents the choices made in a distributed execution of a program. The pseudo-code for the default scheduler can be found in [19]. We use the term *schedule* to refer to the sequence of choices.

In the scheduler, a variable pc_p represents the program counter of the process p. For each process p, pc_p is initialized to the label of the first statement of the process p (i.e. l_0^p) and Q_p is initialized to the empty list (except for Q_1). The scheduler then starts a loop. Inside the loop, the scheduler *non-deterministically* chooses an enabled process p from the set $\mathcal{P}$. It executes the next statement of the process p, where the next statement is obtained by calling *statement_at*(pc_p). During the execution of the statement the program counter pc_p of the process p is incremented by one, unless the statement is of the form `if` p `goto` l' and the predicate p in the statement evaluates to true, in which case pc_p is set to l'. The loop of the scheduler terminates when there is no enabled process in $\mathcal{P}$. The termination of the scheduler indicates either the normal termination of a program execution, or a deadlock state (when at least one process in $\mathcal{P}$ is active).

3.2 Macro-step Semantics

As shown in [2], the interleaving semantics of MPIL presented in Section 3.1 is equivalent to the *macro-step* semantics given in the form of a *macro-step scheduler* in Figure 2. In the macro-step scheduler, the execution of a process from a *receive* statement up to the next *receive* statement takes place consecutively without interleaving with any other process. The consecutive execution of all statements of a process from a *receive* statement up to the next *receive* statement is called a *macro-step* and an execution following the macro-step semantics is called a *macro-step execution*. An execution of MPIL program can be seen as a sequence of macro-steps, where at the beginning of each macro-step, using the function *choice*, the scheduler non-deterministically picks an enabled

process p to be executed next and an index *msg_id* indicating that the message $Q_p[msg\_id]$ must be consumed by the next *receive* statement. The sequence of pairs of processes and message indices chosen during a macro-step execution is called a *macro-step schedule*.

```
scheduler_macro_step(P)
  pc_{p_1} = l_0^{p_1}; ...; pc_{p_n} = l_0^{p_n};
  Q_{p_1} = [0]; Q_{p_2} = [ ]; ...; Q_{p_n} = [ ];
  for each p ∈ P initialize input variables
  while (∃p ∈ P such that p is enabled)
    (p,msg_id)=choose(P);
    s =statement_at(pc_p);
    execute_concrete(p, s,msg_id);
    s =statement_at(pc_p);
    while (p is active and s ≠ receive(v))
      execute_concrete(p, s,msg_id);
      s =statement_at(pc_p);
  if (∃p ∈ P such that p is active)
    warning "Deadlock detected";

choose(P)
  pick non-deterministically a p from P
       such that p is enabled
  pick a j non-deterministically from [1..|Q_p|]
  return (p, j);
```

Fig. 2. Macro-step Scheduler for MPIL

Observe that during a macro-step execution, whenever the macro-step scheduler invokes the function *choice*, a pair of process and message index is non-deterministically picked from a set of possible choices. The set of possible choices can be formally defined as follows:

$$Choices = \{(p, j) \mid p \text{ is an enabled process in } \mathcal{P} \text{ and } 1 \leq j \leq |Q_p|\}$$

The elements of this set can be lexicographically ordered as follows. We say $(p, j) < (p', j')$ iff one of the following holds:

- The index of p is less than that of p', i.e., if i and i' are such that $p = p_i$ and $p' = p_{i'}$, then $i < i'$.
- $p = p'$ and $j < j'$.

Definition 1 (next). *Given the above ordering relation $<$ over the elements of the set Choices, we can define a function next: $Choices \cup \{(\perp, \perp)\} \rightarrow Choices \cup \{(\perp, \perp)\}$ as follows. The elements of the set Choices can be ordered using the relation $<$ to get a linear sequence. If (p, j) is an element of the sequence except the last element, next(p, j) is defined as the element next to (p, j) in the sequence. Otherwise, if (p, j) is the last element in the sequence, then next(p, j) is defined as $(\perp, \perp)$. next$(\perp, \perp)$ is defined as the first element of the sequence.*

3.3 Execution Model

We represent the execution of a statement labeled l in a process p as the *event* (p, l), and use $e, e', e_1, \ldots$ to denote events. A macro-step execution of a distributed program can be seen as a sequence of events $\tau = e_1 e_2 \ldots e_m$, such that τ is the concatenation of a sequence of sub-sequences. Each such sub-sequence has the following property. Only the first event in the sub-sequence is a receive event and each event in the sub-sequence happens at the same process. Thus each sub-sequence represents a macro-step in the execution. Note that this definition requires that the first statement of each process is a *receive*. Let $\mathcal{E}$ be the set of all macro-step executions that can be exhibited by a program on all possible

inputs and schedules. In the simple testing algorithm (Section 5), our goal will be to systematically and automatically explore all executions in $\mathcal{E}$ exactly once. Later, we will refine the algorithm to avoid exploring 'equivalent' executions as much as possible (see Section 6).

We now formally define this equivalence, based on a "happens-before" relation [7]. Given an execution of a distributed program, let E be the set of events that happened during the execution. We can define a relation $\preccurlyeq \subseteq E \times E$, called "happens-before" relation, among the events of the execution as follows:

1. $e \preccurlyeq e$,
2. $e \preccurlyeq e'$ if e and e' are events of the same process and e happens before e' in the execution,
3. $e \preccurlyeq e'$ if e is the send event of a message and e' is the receive event that consumes the message sent during the event e, and
4. $e \preccurlyeq e'$ if there is a e'' such that $e \preccurlyeq e''$ and $e'' \preccurlyeq e'$.

Thus the "happens-before" relation is a partial order relation.

Given two executions τ and τ' in $\mathcal{E}$, we say that τ and τ' are *causally equivalent*, denoted by $\tau \equiv_{\preccurlyeq} \tau'$, iff τ and τ' have the same set of events and they are linearizations of the same "happens-before" relation. We use $[\tau]_{\equiv_{\preccurlyeq}}$ to denote the set of all executions in $\mathcal{E}$ that are causally equivalent to τ.

We define *the representative set of executions* $\mathcal{E}_{\equiv} \subseteq \mathcal{E}$ as the set that contains exactly one candidate from each equivalence class $[\tau]_{\equiv_{\preccurlyeq}}$ for all $\tau \in \mathcal{E}$. Formally, $\mathcal{E}_{\equiv}$ is the set such that following properties hold: $\mathcal{E}_{\equiv} \subseteq \mathcal{E}$, $\mathcal{E} = \bigcup_{\tau \in \mathcal{E}_{\equiv}} [\tau]_{\equiv_{\preccurlyeq}}$, and for all $\tau, \tau' \in \mathcal{E}_{\equiv}$, it is the case that $\tau \not\equiv_{\preccurlyeq} \tau'$.

The following result shows that a systematic and automatic exploration of each element in $\mathcal{E}_{\equiv}$ is sufficient for testing.

Proposition 1. *If a statement is reachable in a program P for some input and schedule, then there exists a $\tau \in \mathcal{E}_{\equiv}$ such that the statement is executed in τ.*

The proof of this proposition is straight-forward. If a statement is reachable then there exists an execution τ in $\mathcal{E}$ such that the execution τ executes the statement. By the definition of $\equiv_{\preccurlyeq}$, any execution in $[\tau]_{\equiv_{\preccurlyeq}}$ executes the statement. Hence, the execution in $\mathcal{E}_{\equiv}$ that is equivalent to τ executes the statement.

The "happens-before" relation among the events can be tracked efficiently at runtime using vector clocks [7]. A vector clock $V \colon \mathcal{P} \to \mathbb{N}$ is a map from processes to natural numbers (also known as logical time). For each process p, let us associate a vector clock denoted by VC_p with p. Let $V = max(V_1, V_2)$ iff for all $p \in \mathcal{P}$, $V(p) = max(V_1(p), V_2(p))$. Let $V \leq V'$ iff for all $p \in \mathcal{P}$, $V(p) \leq V(p')$.

At the beginning of an execution, for all p and p' in $\mathcal{P}$, let $VC_p(p') = 0$. During the execution, at every event, the vector clock of a process is updated as follows.

1. If e is a send event executed by process p, then $VC_p(p) \leftarrow VC_p(p) + 1$ and attach VC_p with the message.
2. If e is a receive event executed by process p and if V is the vector clock attached with the message received, then $VC_p \leftarrow max(V, VC_p)$. This is followed by $VC_p(p) \leftarrow VC_p(p) + 1$.

We can associate a vector clock with every event e, denoted by VC_e as follows. If e is executed by p and if VC_p is the vector clock of p just before the event e, then $VC_e = VC_p$.

Given the above update rules for vector clocks during an execution, the following theorem [4] holds:

Theorem 1. *For any two events e and e', $e \preccurlyeq e'$ iff $VC_e \leq VC_{e'}$.*

We say that two events e and e' are *independent* iff $e \not\preccurlyeq e'$ and $e' \not\preccurlyeq e$. Therefore, by Theorem 1, e and e' independent iff $VC_e \not\leq VC_{e'}$ and $VC_{e'} \not\leq VC_e$.

Since we are interested only in exploring macro-step executions, henceforth, we will use the terms execution and schedule to refer to macro-step execution and macro-step schedule, respectively.

4 An Illustrative Example

We now illustrate our testing methodology by means of the simple program in Figure 3. For brevity, we omit the first *receive* statement of process p_1 in the program. We perform testing on the program by generating inputs and schedules one by one and executing the program both concretely and symbolically on these inputs and schedules. We assume that a program executes according to the macro-step semantics described above. We represent an execution diagrammatically using a lifeline where each circle on the lifeline represents a program state and each line segment between two circles represents the execution of a statement by a process. We always label such a line segment by a pair of the form (p, l) denoting the execution of the statement labeled l by the process p. We assume that time increases from top to bottom in the diagram.

Figure 4.a shows the execution of the program on a random input and a random schedule. In the execution there are three states s_1, s_2, s_3 where the program can possibly backtrack (or continue with a different path) if we can generate a different schedule or different input. For example, at the s_1, there are two other possible choices that the scheduler may make – it may execute p_3 by receiving the value sent by the second *send* statement of p_1 or by receiving the value sent by the third *send* statement of p_1. Similarly, at s_2, the scheduler may make another choice – it may execute p_3 by receiving the message sent by the third *send* statement of p_1. At s_3, the program may take the *then* branch of the program if the input is chosen such that it satisfies the constraint $2 * y + 1 == 4$, which is generated using the simultaneous symbolic execution and constraint solving.

p_1 :	p_2 :	p_3 :
$x \leftarrow input()$		$y \leftarrow input()$
1: $send(p_2, 1)$	1: $receive(z)$	1: $receive(u)$
2: $send(p_3, 4)$		2: `if` $(u! = 2 * y + 1)$ `goto` 4
3: $send(p_3, x)$		3: `ERROR`
		4: $receive(u)$

Fig. 3. Simple Distributed Program Example

In our simple testing algorithm (described in Section 5), we generate the next input or schedule by exploring other alternatives at these backtracking points in a depth-first manner. We cannot generate an input such that, in the next execution, the program takes the *then* branch at s_3. This is because the

equation $2 * y + 1 == 4$ is unsatisfiable assuming that y is an integer. Therefore, in the next execution we execute the program by taking the alternative scheduler choice at the s_2. The execution is shown in Figure 4.b. After this execution we try to backtrack at s_3 and generate the input $\{x = 1, y = 0\}$ by solving the constraint $x == 2 * y + 1$, which is generated during the simultaneous symbolic execution. Figure 4.c gives the third execution.

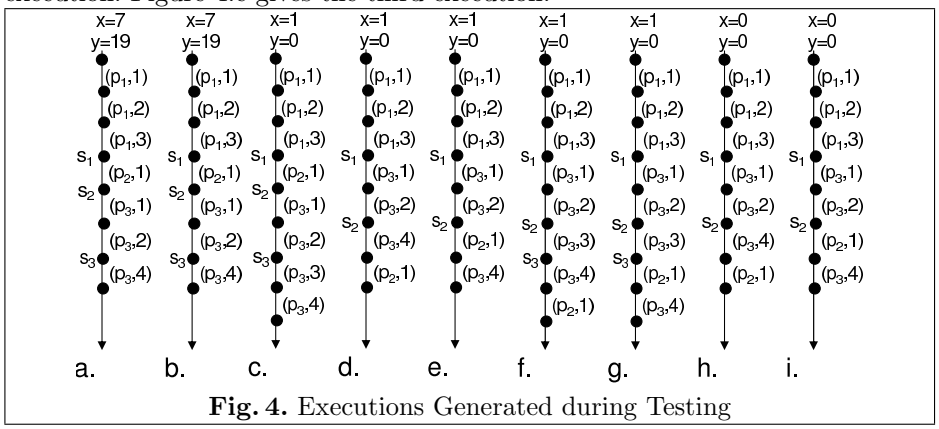

Fig. 4. Executions Generated during Testing

In this way, our simple testing algorithm proceeds in a depth-first manner either by generating an input by solving a constraint at a backtracking point or by generating different schedule by making an alternative scheduler choice at a backtracking point. The remaining executions of the program are shown in Figure 4. Note that our simple algorithm, which considers all possible scheduler choices at a backtracking point, results in many redundant executions. We can get rid of most of the redundant executions using our efficient testing algorithm (described in Section 6) which performs *runtime partial order reduction* by computing a "happens-before" relation (described in Section 3.3) among various events in an execution.

Our efficient testing algorithm only generates the first three executions in Figure 4. This is far less than the number of executions generated by our simple testing algorithm. Our efficient testing algorithm avoided the redundant executions and yet was able to hit the error statement. In particular, at the backtracking point denoted by s_1 in Figure 4.c, our efficient algorithm does not consider the other two possible alternative choices of executing the p_3. This is because the execution of the first *receive* statement by p_2 after s_1 does not effect the execution of any future statement. Therefore, delaying the execution of the p_2 after s_1 will result in an execution that will have the same "happens-before" relation as the current execution. Considering two executions having the same "happens-before" relation is redundant since we are concerned with statement reachability (see Theorem 1).

5 Simple Algorithm

We present a simple systematic search algorithm in which our goal is to explore all macro-step execution paths of a program P by generating inputs and macro-step schedules. As in earlier work [12, 21], our algorithm uses concrete

```
// input: P is the program to test
run_CUTE(P)
  completed=false; I = path_c = branch_hist=[ ];
  while not completed
    scheduler(P);
```

Fig. 5. Testing Algorithm Calls Scheduler in a Loop

values as well as symbolic values for the inputs, and executes the program both concretely and symbolically. During the course of the execution, it collects the constraints over the symbolic values over each branching point (i.e., the *symbolic constraints*). At the end of the execution of a path, the algorithm has computed a sequence of symbolic constraints corresponding to each branching point. We call the conjunction of these constraints *path constraint*. Observe that all input values that satisfy a given path constraint will explore the same execution path given that we follow the same schedule.

The algorithm first generates a random input and a macro-step schedule which specifies the order of execution of processes. Then the algorithm does the following in a loop: it executes the code with the generated input and the schedule, and the same time records the process and message index pairs chosen by the scheduler as well as the symbolic constraints. The algorithm backtracks and alters these choices to systematically explore all possible macro-step execution paths using a depth-first search strategy. Specifically, the algorithm does one of the following to find the new data values or schedule for the next execution:

1. It picks a constraint from the symbolic constraints that were collected along the execution path and negates the constraint to define a new path constraint. The algorithm then finds some concrete values, if such values exist, that satisfy the new path constraint.
2. It generates a new schedule such that at some state where the scheduler makes a choice, the *next* possible choice is picked instead of the current choice.

The algorithm continues the loop until it sweeps all feasible execution paths.

There is one complication arising from the fact that for some symbolic constraints, our constraint solver may not be powerful enough to compute concrete values that satisfy the constraints. To address this difficulty, such symbolic constraints are simplified by replacing some of the symbolic values with concrete values. Because of this, our algorithm is sound but not complete.

We now provide the details of the algorithm. The algorithm is described using a centralized interpreter for programs in MPIL. This is to simplify the description. In fact, jCUTE instruments distributed programs and uses a centralized scheduler to control the distributed processes.

The pseudo-code for our algorithm is in Figure 5. Before starting the execution loop, the algorithm initializes the logical input map $\mathcal{I}$ (which maps each input variable to a value) to an empty map [21], the sequences *path_c* (which maintains scheduler choices and symbolic constraints for a given execution), and *branch_hist* (which maintains the history of branches taken) to the empty sequences. Each element of the list *path_c* has the following fields:

1. *constraint*: stores the constraint generated on the execution of a conditional statement. At the end of an execution, the conjunction of all the constraints stored

in the elements of *path_c*, for which the field *hasConstraint* is true, gives the path constraint for the given execution path. (Since in [21] we were not concerned about distributed events, each element of *path_c* was used to store only a constraint).

2. *hasConstraint*: set to true if the field *constraint* stores a constraint. It is set to false if the field *constraint* contains a scheduler choice.
3. *schedule*: stores a pair of process and message index, which is the choice made by the scheduler before executing a $receive(v)$ statement.
4. *next_schedule*: stores the scheduler choice $next(schedule)$.

```
scheduler(P)
  pc_{p_1} = l_0^{p_1}; ...; pc_{p_n} = l_0^{p_n};
  Q_{p_1} = [0]; Q_{p_2} = [ ]; ...; Q_{p_n} = [ ];
  i = 0;
  for each p ∈ P initialize
       input variables using I
  while (∃p ∈ P such that p is enabled)
    (p,msg_id)=choose_simple_systematic(P);
    path_c[i].hasConstraint = false;
    i = i + 1;
    s =statement_at(pc_p);
    execute_concolic(p, s,msg_id);
    s =statement_at(pc_p);
    while (p is active and s ≠ receive(v))
      execute_concolic(p, s,msg_id);
      s =statement_at(pc_p);
  if (∃p ∈ P such that p is active)
    warning "Deadlock detected";
  compute_next_input_and_schedule();

choose_simple_systematic(P)
  if (i ≤ |path_c|)
    (p,msg_id) =path_c[i].schedule;
  else
    path_c[i].schedule = (p,msg_id) = next(⊥, ⊥);
  path_c[i].next_schedule = next(p,msg_id);
  return (p,msg_id);
```

Fig. 6. Simple Scheduler for Testing MPIL

```
execute_concolic(p, s, j)
  pc_p = pc_p + 1;
  match(s)
    case send(p', v):
      Q_{p'} = (S_p(v), A_p(v)) :: Q_{p'} ;
    case receive(v):
      (val, sval) = Q_p[j];
      S_p = S_p[v ↦ val]; A_p = A_p[v ↦ sval];
      Q_p = remove_element(Q_p, j);
    .
    .
compute_next_input_and_schedule()
  j = i − 1;
  while j ≥ 0
    if path_c[j].hasConstraint
      if (branch_hist[j].done == false)
        branch_hist[j].branch=¬branch_hist[j].branch;
        if (∃I' that satisfies neg_last(path_c[0 ... j]))
          path_c = path_c[0 ... j];
          branch_hist=branch_hist[0 ... j];
          I = I';
          return;
    else if path_c.next_schedule ≠ (⊥, ⊥)
      path_c.schedule=path_c.next_schedule;
      path_c = path_c[0 ... j];
      branch_hist=branch_hist[0 ... j];
      return;
    j = j − 1;
  if (j < 0) completed=true;
```

Fig. 7. Concolic Execution and Compute Next Schedule and Input.

The non-deterministic function *choice* given in Figure 2 is replaced by the function *choice_simple_systematic* (see Figure 6). The simple scheduler first initializes the program counters pc_p and Q_p for each process $p \in \mathcal{P}$. In addition, the simple scheduler also initializes the global counter variable i to 0. At any point of execution, i contains the sum of the number of choices made by the scheduler thus far, as well as the number of conditional statements executed. The input variables of each process are also initialized using the logical input map $\mathcal{I}$ (cf. [21]). If $\mathcal{I}(v)$ is undefined for an input variable v, then v is initialized randomly. In the function *choose_simple_systematic*, the scheduler picks the same schedule as the previous execution as long as i is less than the number of elements of *path_c*. The list *path_c* is truncated appropriately at the end of the previous execution to perform a depth-first search of the execution paths. Otherwise, the scheduler picks a pair of process and message index such that the pair is the smallest pair in the set of possible choices.

5.1 Computing Next Schedule and Input

The function *compute_next_input_and_schedule*, described in Figure 7, computes the schedule and the input that will direct the next program execution along

an alternative execution path. It first picks an element of *path_c* from the end. If the element contains a constraint and if it is not negated before, then constraint solving is invoked to generate a new input (see [21]). Otherwise, if the element contains a scheduler choice and if not all scheduler choices at the execution point denoted by the element have been exercised, then a new schedule is generated. Specifically, if the pair (p, m) is chosen at the execution point denoted by $path\_c[j].schedule$, then $next(p, m)$, which is stored in $path\_c[j].next\_schedule$, is assigned to $path\_c[j].schedule$. In the next execution, at that particular execution point, the scheduler will pick $next(p, m)$, a choice which was not exercised before at that execution point. This ensures that in subsequent executions all the choices are selected one by one.

5.2 Concolic Execution

Concolic execution [21, 12] performs both symbolic and concrete execution of a program side by side in a cooperative way. The concolic execution technique will be important for efficiently testing distributed programs: the availability of concrete values for all memory locations in addition to the symbolic values helps us to accurately determine the partial order of a distributed execution (as described later in Section 6). Determining the partial order is important to avoid exploring redundant executions. On the other hand, the symbolic execution part of the concolic execution helps us perform symbolic execution as much as possible. This symbolic execution combined with constraint solving is essential to generate data inputs for the next execution.

The details of the procedure *execute_concolic* can be found in [21]. A brief pseudo-code of the procedure is given in Figure 7. At runtime, for each process p concolic execution maintains a symbolic state $\mathcal{A}_p$ mapping memory locations to symbolic expressions over symbolic input values in addition to the concrete state $\mathcal{S}_p$ mapping memory locations to concrete values. During concolic execution, every statement is executed concretely using the function *evaluate_concrete* and symbolically using the function *evaluate_symbolic*. In addition to performing symbolic execution, the function *evaluate_symbolic* simplifies any complex (e.g. non-linear) symbolic expressions in the symbolic state by replacing some symbolic values in the expression by their corresponding concrete values.

Note that in concolic execution, to carry out the symbolic execution, we need to track symbolic states and symbolic constraints across the process boundaries. To achieve this, both the concrete value and the symbolic value of the variable v are sent, when a process executes a statement of the form $send(p, v)$. Moreover, for each process p the message queue Q_p is modified to a list of pairs of concrete and symbolic values. An execution of the statement $send(p, v)$ by a process p' prepends a pair of the concrete and the symbolic value of the variable v to the message queue of process p.

6 Efficient Algorithm

We now provide an efficient algorithm which explores a much smaller superset of the execution paths in $\mathcal{E}_{\equiv}$. The efficient algorithm is based on the following observation. At a point where the scheduler makes a choice, often it is sufficient to consider all messages for a particular process only as possible choices by

the scheduler, instead of considering all messages for all processes as possible scheduler choices. This is because, considering all messages for all processes would result in many equivalent executions.

We now characterize the case where the scheduler has to choose between messages for different processes. Consider a prefix $\tau = e_1 e_2 \dots e_k$ of the sequence of events in an execution, such that the scheduler makes a choice after τ. Let e be the event from process p, which happens immediately after τ when the scheduler only chooses all messages for the particular process p after τ. Now if there exists an execution τ' with prefix τe such that there is a send event e' to process p, e' appears after τe in τ', and e is independent of e', then we need to delay the execution of process p after τ such that the receive event e of process p after τ consumes the message sent by the event e'. This would give a different non-equivalent execution. Thus in such situations, it is not sufficient if the scheduler only chooses all messages of process p after τ. Rather, immediately after τ, we need to consider all messages of at least one more process other than p.

```
scheduler(P)
  pc_{p_1} = l_0^{p_1}; ...; pc_{p_n} = l_0^{p_n};
  Q_{p_1} = [0]; Q_{p_2} = [ ]; ...; Q_{p_n} = [ ];
  i = 0;
  for each p ∈ P initialize
       input variables using I
  while (∃p ∈ P such that p is enabled)
    (p,msg_id)=choose_simple_systematic(P);
    path_c[i].hasConstraint = false;
    path_c[i].vclock = (p, VC_p);
    i = i + 1;
    s =statement_at(pc_p);
    execute_concolic(p, s,msg_id);
    s =statement_at(pc_p);
    while (p is active and s ≠ receive(v))
      if s is send(p', v)
        for all k ≤ i
              such that (p'', V) = path_c[k].vclock
              and p'' = p' and V ⋦ VC_p and VC_p ⋦ V
            path_c[k].needs_delay=true;
      execute_concolic(p, s,msg_id);
      s =statement_at(pc_p);
  if (∃p ∈ P such that p is active)
    warning "Deadlock detected";
  compute_next_input_and_schedule();
```

```
compute_next_input_and_schedule()
  j = i − 1;
  while j ≥ 0
    if path_c[j].hasConstraint
      if (branch_hist[j].done == false)
        branch_hist[j].branch=¬branch_hist[j].branch;
        if (∃I' that satisfies neg_last(path_c[0 ... j]))
          path_c = path_c[0 ... j];
          branch_hist=branch_hist[0 ... j];
          I = I';
          return;
    else if path_c[j].next_schedule ≠ (⊥, ⊥)
      (p, m) =path_c[j].schedule;
      (p', m') =path_c[j].next_schedule;
      if p = p' or path_c[j].needs_delay
        path_c[j].schedule=path_c[j].next_schedule;
        if p ≠ p'
          path_c[j].needs_delay = false;
        path_c = path_c[0 ... j];
        branch_hist=branch_hist[0 ... j];
        return;
    j = j − 1;
  if (j < 0) completed=true;
```

Fig. 8. Efficient Scheduler for Testing MPIL

Based on the above observation, we refine the simple scheduler described in Figure 6 by one (see Figure 8) that uses the "happens-before" relation to avoid exploring equivalent executions as much as possible. We assume that the scheduler maintains vector clocks with each process and that the vector clocks are updated using the procedure described in Section 3.3. We omit the vector clock update procedure from Figure 8 to keep the description simple.

In the efficient scheduler, we keep track of the vector clocks of each *receive* event. For every *send* event we check if the *send* event can synchronize with an already executed *receive* event in some alternative execution. This is done by checking the independence of the *send* event with any previously executed *receive* event. If such a check passes, then we flag the scheduler choice at the execution point just before the independent *receive* event. The flag indicates that

in some future execution, just before the *receive* event, the scheduler needs to consider all messages of at least one more process.

To keep track of vector clocks and the flag, we introduce two more fields to each element of *path_c* as follows.

1. *vclock*: stores a pair (p, V), where p is the process executing the *receive* event and V is the vector clock of the event.
2. *needs_delay*: stores the flag whose truth indicates that at the current execution point, the scheduler needs to consider all messages of more than one process. If the flag is false, then the scheduler only considers all messages of a single process.

Soundness of our algorithm is trivial. A bug reported by our algorithm is an actual bug because our algorithm provides a concrete input and schedule on which the program exhibits the bug. Moreover, our algorithm can be complete in some cases.

Proposition 2. ***(Completeness)*** *During testing a program with our efficient algorithm, if the following conditions hold:*

- *The algorithm terminates.*
- *The algorithm makes no approximation during concolic execution and it is able to solve any constraint which is satisfiable.*

then our algorithm has executed all executions in $\mathcal{E}_{\equiv}$ and we have hit all reachable statements of the program.

The proof of this proposition, while fairly intuitive, is beyond the scope of this paper. Next, we show that the efficient algorithm explores significantly fewer execution paths than the simple algorithm while achieving the same branch coverage.

7 Implementation and Experiments

We have implemented both the simple and the efficient testing algorithm as a part of the Java testing tool jCUTE. The tool can be applied to test distributed Java programs written in the Actor language [2]. The Actor language extends Java by supporting actors or processes. The language is supported as a library in Java. In the language we assume that *Java threads are not explicitly used by the programmer*.

We report our experience of using jCUTE on a few examples, which include implementations of a *leader election* algorithm, a *distributed sorting algorithm*, and *Chandy-Misra's shortest path algorithm*. We performed all experiments on a Windows XP laptop with a 2.0 GHz Pentium M processor and 1GB RAM. The tool and the code for the case studies can be downloaded from `http://osl.cs.uiuc.edu/~ksen/cute/`.

The leader election algorithm that we considered works on a system with N processes connected using a unidirectional ring. Each process is assumed to have an unique id. We considered a general implementation where we assumed that the unique ids can be any value – in fact, they are assumed to be inputs. Such a general implementation cannot be handled by the model-checker in [3].

In the implementation, when we did not assume that the communication channels are FIFO, then our testing algorithms discovered an assertion violation

		Simple Testing Algorithm			Efficient Testing Algorithm			
Name	# of Processes	Run time in seconds	# of Executions	% Branch Coverage	Run time in seconds	# of Iterations	% Branch Coverage	Bug(s) Found
Leader	3	25.1	387	66.7	0.53	9	66.7	0
Election (FIFO)	4	> 33000	> 30000	66.7	15.92	22	66.7	0
Leader (non-FIFO)	3	0.16	5	70.0	0.24	5	70.0	1
Distributed	4	0.39	14	71.43	0.21	7	71.43	0
Sorting	5	13.3	420	71.43	1.13	35	71.43	0
	6	2152.42	64636	71.43	7.63	226	71.43	0
Chandy-	4	> 2600	> 100000	62.5	8.92	338	62.5	0
Misra	5	> 2690	> 100000	62.5	15.01	562	62.5	0

Table 1. Results of Testing Distributed Programs

that shows that there can be inputs and schedules where the algorithm fails to elect a leader.

When we assumed that the communication channels are FIFO, both of our testing algorithms terminated without reporting any error. Table 1 gives the various statistics about this testing experiment.

Similarly, we tested implementations of a distributed sorting algorithm and Chandy Misra's shortest path computation algorithm. A model of the sorting algorithm was used for model-checking using the SPIN model-checker. However, in that experiment, they assumed a fixed sequence of numbers for sorting. Instead, we made the numbers to be sorted as inputs. This enabled us to test the algorithm not only for all schedules but also for all inputs.

The experimental results show that for the implementations that we considered, the efficient algorithm explores significantly fewer execution paths than the simple testing algorithm. On the other hand, both the algorithms attain the same branch coverage. The branch coverage in most cases is less than 100% because the implementations contain a number of assert statements that were never violated and some dead branches which cannot be taken.

8 Conclusion

We presented a new algorithm and an implementation to systematically and efficiently test distributed programs with inputs. To our best knowledge, jCUTE is the first testing tool that can automatically and exhaustively explore all non-equivalent execution paths of a distributed program with data inputs. In contrast, all previous tools [11, 15, 8] were able to test distributed programs only with a small finite domain input or with random inputs.

Acknowledgment

The first author benefited greatly from interaction with Patrice Godefroid and Nils Klarlund during a summer internship. We would like to thank Timo Latvala, Darko Marinov, Abhay Vardhan, and Mahesh Viswanathan for providing valuable comments. This work is supported in part by the ONR Grant N00014-02-1-0715, the NSF Grant NSF CNS 05-09321, and the Motorola Grant Motorola RPF #23.

References

1. G. Agha. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.
2. G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.
3. R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification*, pages 340–351, 1997.

4. O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In *Distributed Systems*, pages 55–96. 1993.
5. K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.
6. M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Form. Methods Syst. Des.*, 25(2–3):199–240, 2004.
7. C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the Workshop on Parallel and Distributed Debugging (WPDD)*, pages 183–194. ACM, 1988.
8. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of the 32nd Symposium on Principles of Programming Languages (POPL'05)*, pages 110–121, 2005.
9. V. K. Garg and C. M. Chase. Distributed algorithms for detecting conjunctive predicates. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, page 423. IEEE, 1995.
10. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. 1996.
11. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *24th ACM Symposium on Principles of Programming Languages)*, pages 174–186, 1997.
12. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.
13. G. J. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
14. J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
15. Y. Lei and K.-C. Tai. Efficient reachability testing of asynchronous message-passing programs. In *8th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 35–, 2002.
16. G. J. Myers. *The Art of Software Testing*. Wiley, 1979.
17. C. S. Pasareanu, M. B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Proc. of TACAS'01*, pages 284–298, 2001.
18. D. Peled. All from one, one for all: on model checking using representatives. In *5th Conference on Computer Aided Verification*, pages 409–423, 1993.
19. K. Sen and G. Agha. Automated systematic testing of open distributed programs. Technical Report UIUCDCS-R-2005-2647, UIUC, 2005.
20. K. Sen and G. Agha. Concolic testing of multithreaded programs and its application to testing security protocols. Technical Report UIUCDCS-R-2006-2676, UIUC, 2006.
21. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*. ACM, 2005.
22. S. D. Stoller. Model-Checking Multi-Threaded Distributed Java Programs. In *Proc. of SPIN'00: SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 224–244. Springer, 2000.
23. A. Valmari. Stubborn sets for reduced state space generation. In *10th International Conference on Applications and Theory of Petri Nets*, pages 491–515, 1991.
24. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.
25. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. of the Tools and Algorithms for the Construction and Analysis of Systems*, 2005.