Systematic Testing of Asynchronous Reactive Systems
MSR-TR-2015-25

Ankush Desai
UC Berkeley

Shaz Qadeer
Microsoft

Sanjit Seshia
UC Berkeley

Abstract

We introduce the concept of a delaying explorer with the goal of performing prioritized exploration of the behaviors of an asynchronous reactive program. A delaying explorer stratifies the search space using a custom strategy, and a delay operation that allows deviation from that strategy. We show that prioritized search with a delaying explorer performs significantly better than existing prioritization techniques. We also demonstrate empirically the need for writing different delaying explorers for scalable systematic testing and hence, present a flexible delaying explorer interface.

We introduce two new techniques to improve the scalability of search based on delaying explorers. First, we present an algorithm for stratified exhaustive search and use efficient state caching to avoid redundant exploration of schedules. We provide soundness and termination guarantees for our algorithm. Second, for the cases where the state of the system cannot be captured or there are resource constraints, we present an algorithm to randomly sample any execution from the stratified search space. This algorithm guarantees that any such execution that requires d delay operations is sampled with probability at least $1/L^d$, where L is the maximum number of program steps. We have implemented our algorithms and evaluated them on a collection of real-world fault-tolerant distributed protocols.

1 Introduction

Asynchronous reactive systems are ubiquitous across domains like distributed systems, device drivers, web applications, and operating systems. Processes in these systems communicate by exchanging messages asynchronously and react to environment input continuously. The system reliability depends critically on correct handling of asynchrony and reactivity using stateful protocols. Testing and debugging of these systems is notoriously difficult due to the nondeterministic nature of their computation; an error could result from a combination of some choice of inputs and some interleaving of event handlers. This paper is concerned with the problem of systematic testing of such such systems by automatically enumerating all sources of nondeterminism, both from environment input and from scheduling of concurrent processes.

The main challenge in scaling systematic testing to real-world programs is the large number of behaviors that explode exponentially with the number of steps in the program. Techniques such as state caching [12] and partial-order reduction [10] have been developed to combat this explosion, yet their worst-case complexity remains exponential. In practice, the search often takes too long and has to be terminated because of a time bound, thereby giving no information to the programmer. Therefore, researchers have been motivated to investigate prioritized search techniques, both deterministic [13, 9] and randomized [3], to provide partial coverage information. However, all of these techniques have been developed for shared-memory multithreaded programs. In asynchronous reactive programs, the primary mechanism for com-

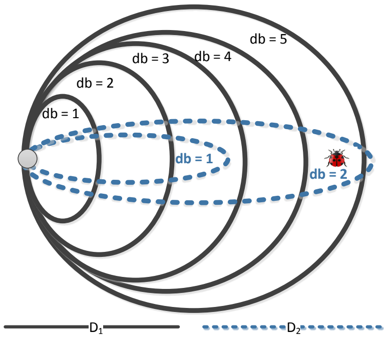

Figure 1: Stratification using delaying explorers

munication among concurrent processes is message-passing rather than shared-memory. We have discovered empirically (Section 6) that prioritization techniques developed for multithreaded programs are not effective when applied to message-passing programs.

In this paper, we introduce a new technique for systematic testing of asynchronous reactive programs. Our technique is inspired by the notion of a delaying scheduler [9] for multithreaded programs. A delaying scheduler is a *deterministic* thread scheduler equipped with a delay operation whose invocation changes the default scheduling strategy. For asynchronous reactive programs, we generalize this notion to a delaying explorer of *all* nondeterministic choices (Section 2), both from input and from the interleaving of event handlers. The key observation that makes a delaying explorer suitable for systematic testing is that every execution can be produced by introducing a finite number of delays in the default deterministic execution prescribed by the explorer. We show that appropriately designed delaying explorers are significantly better than existing prioritization techniques in searching for errors in executions of asynchronous message-passing systems.

A delaying explorer induces stratification in the search space of all executions. A stratum is the set of executions that require the same number of delays. Figure 1 represents the stratification pictorially; $db = 1$ is the set of executions with one delay, $db = 2$ is the set of executions with two delays, and so on. A delaying explorer specifies a prioritized search that explores these strata in order. Since the number of possible executions increases exponentially with the delay budget, exploration for high budget values becomes prohibitively expensive. Therefore, a delaying explorer is effective only if bugs are uncovered at low values of the delay budget. Figure 1 shows the stratification induced by two different delaying explorers. The explorer D_2 is more effective than D_1 at discovering a particular bug if that bug lies in a lower stratum for D_2 than for D_1.

The difference in stratification induced by different delaying explorers has practical consequences. We have observed empirically that there is considerable variance in the speed of detecting errors across different delaying explorers for different test problems[1]. Motivated by this observation, we have designed a general delaying explorer interface that helps programmers quickly write custom search strategies in a small amount of code, typically less than 50 LOC. Delaying explorers also provides developers and testers with a simple and elegant mechanism to express domain-specific knowledge regarding parts of the search space to prioritize. We have written several delaying explorers using our framework and used them to find bugs in implementations of distributed protocols that could not be discovered using any other method. We describe a particular case study in Section 6.

Given a delaying explorer, we need techniques for effectively exploring the strata induced by the explorer. In this paper, we also present two algorithms —Stratified Exhaustive Search (SES) and Stratified Sampling (SS)— for solving this problem. SES performs stratified search by iteratively incrementing the delay budget and exhaustively enumerating all schedules that can be explored with a given delay budget. Inspired by model checking techniques, we incorporate state caching to avoid redundant exploration of schedules. By caching the states visited along an execution, we can prune the search if an execution generated subsequently leads to a state in the cache. Incorporating state caching in delaying exploration is non-

[1] A test problem is the combination of a program and a specification.

trivial because search is performed over executions of the composition of the program and the delaying explorer, both reading and updating their private state in each step of the execution. The naive strategy of caching the product of the program and the explorer state does not work because the delaying explorer can be an arbitrary program with a huge state space of its own. Instead, our algorithm caches only the program state yet guarantees that in the limit of increasing delay budgets, all executions of the program are covered. Our evaluation shows that SES finds bugs orders of magnitude faster than prior prioritization techniques on our benchmarks (Section 6).

Even though state caching is an important optimization, it is not a panacea to the explosion inherent in systematic testing. The complexity of the algorithm mentioned in the previous paragraph still grows exponentially with the number of allowed delays. Consequently, if a delaying explorer is unable to find a bug quickly within a few delays, the search must be stopped because of the external time bound. To further scale search to large delay budgets, we present the SS algorithm which performs stratified sampling of the search space with probabilistic guarantees. Our algorithm guarantees that any execution that is visited with *db* delays is sampled with probability at least $1/L^{db}$, where L is the maximum number of program steps. SS is useful because it allows even distribution of the limited time resource over the entire search space. Furthermore, since each sample is generated independently of every other sample, random exploration can be easily and efficiently parallelized or distributed. Finally, for some systems state caching may not be possible because of the difficulty of taking a snapshot of the entire system state. In this situation, search based on random sampling could be very useful. We empirically show (Section 6) that on our benchmarks, SS can find bugs faster, often by an order-of-magnitude, compared to the prior best technique [3] for random sampling of executions of multithreaded programs.

We have implemented our framework and algorithms for systematic testing of applications written in P [6], a domain-specific language for asynchronous event-driven programming currently used for developing device drivers and distributed services. The P compiler generates both executable C code and Zing code which can be explored with the Zing model checker. The generated C code is used for executing the application either locally on a single computing node or distributed across a collection of nodes. We have implemented in P a fault-tolerant transaction management system (TMS) that internally comprises many protocols such as two-phase commit [2], multi-paxos [4], and chain replication [21]. Using our test framework, we found many bugs, caused by protocol-level race conditions, in our implementation of TMS. The suite of protocols in TMS form the benchmark set for evaluating our algorithms.

We conclude this section by summarizing our contributions below.

⇒ We introduce delaying explorers as a foundation for systematic testing of asynchronous reactive programs. We empirically demonstrate that for the domain of message-passing programs, delaying explorers are better, often by an order-of-magnitude, than existing prioritization techniques.

⇒ We observe that the efficacy of a delaying explorer depends on the test problem. To enable programmers to easily write custom explorers, we have created a flexible interface for specifying explorers. We have written four delaying explorers, each in less than 50 LOC, using our interface.

⇒ We present the SES algorithm that uses state-caching for efficiency while prioritizing search using a delaying explorer. The algorithm guarantees soundness even without caching the state of the delaying explorer.

⇒ We present the SS algorithm to efficiently sample executions with a fixed number of delays. Our algorithm guarantees that if a buggy execution exists with *db* delays for a given delaying explorer, then each sample triggers the bug with probability at least $1/L^{db}$ where L is the maximum number of steps in the program.

2 Delaying explorers

In this section, we provide intuition for delaying explorers and their use in systematic testing of asynchronous reactive systems. We begin by formally

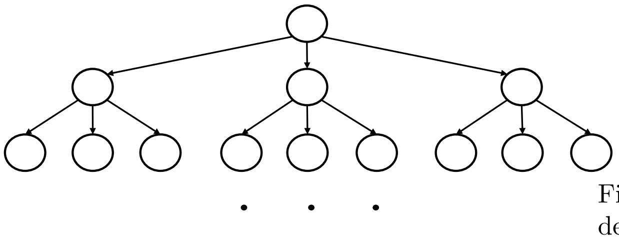

Figure 2: A concurent program

stating our model of programs and explorers.

A program $\mathcal{P}$ is a tuple $(S, \mathit{Cid}, T, s_0)$:

1. S is the set of states of $\mathcal{P}$.
2. Cid is a finite set of nondeterministic choices that $\mathcal{P}$ can make during execution. This set includes both choices due to scheduling of concurrent processes in $\mathcal{P}$ and choices due to nondeterministic input received by each process.
3. $T \in \mathit{Cid} \times S \rightharpoonup S$ is the transition function of $\mathcal{P}$. If $s' = T(c, s)$, we say that (s, s') is a transition of $\mathcal{P}$. We define $\mathit{Choices}(s) = \{c \mid \exists s'.\ T(c, s) = s'\}$.
4. s_0 is the initial state of $\mathcal{P}$.

A sequence of states $s_0, s_1, s_2, \ldots, s_n$ is an *execution* of $\mathcal{P}$ if (s_i, s_{i+1}) is a transition of $\mathcal{P}$ for all $i \in [0, n)$. A state $s \in S$ is *reachable* if it is the final state of some execution. An infinite sequence of states $s_0, s_1, s_2, \ldots$ is an *infinite execution* of $\mathcal{P}$ if (s_i, s_{i+1}) is a transition of $\mathcal{P}$ for all $i \geq 0$. We assume that $\mathcal{P}$ is *terminating*, i.e., it does not have any infinite executions.

The formalization of the nondeterministic transition graph of an asynchronous reactive program is standard in the literature; it is depicted pictorially in Figure 2. The exploration algorithms popularized by model checking tools, e.g. SPIN [12], view the transitions coming out of a state as unordered; the order in which those transitions are explored is considered an implementation-level detail. A delaying explorer, formalized below, instead considers the order of transitions an important concern for efficient exploration. It provides a general interface for specifying this order based on the entire history of the program execution.

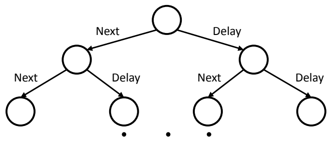

Figure 3: A concurrent program composed with a delaying explorer

A delaying explorer $\mathcal{D}$ is a tuple $(D, \mathit{Next}, \mathit{Step}, \mathit{Delay}, d_0)$:

1. D is the set of states of $\mathcal{D}$. The state of the explorer typically includes a data structure, e.g. stack or queue, to maintain an ordering among the choices available to the program.
2. $\mathit{Next} \in D \to \mathit{Cid}$ is a total function. Given a explorer state d, the choice $\mathit{Next}(d)$ is prescribed by the explorer to be taken next.
3. $\mathit{Step} \in S \times D \to D$ is a total function. Suppose we have a program state s and a explorer state d, and we execute the choice $\mathit{Next}(d)$ at s. Then $\mathit{Step}(s, d)$ yields the explorer state corresponding to the program state $T(\mathit{Next}(d), s)$. The *Step* function enables building explorers which change their state in response to specific events that occur during execution of the program, such as sending or receiving of messages, creation of new processes, etc.
4. $\mathit{Delay} \in D \to D$ is a total function. Given a explorer state d, the application $\mathit{Delay}(d)$ yields a new explorer state. The *Delay* function provides a mechanism to change the next choice to be explored.
5. d_0 is the initial state of $\mathcal{D}$.

Consider a program $\mathcal{P}$ and a delaying explorer $\mathcal{D}$. We denote by $\mathcal{P} \| \mathcal{D}$ the composition of $\mathcal{P}$ and $\mathcal{D}$. Figure 3 shows the nondeterminism in the unfolding of $\mathcal{P} \| \mathcal{D}$; we formalize this intuition below. A finite sequence $(s_0, d_0) \xrightarrow{x_0} (s_1, d_1) \xrightarrow{x_1} (s_2, d_2) \xrightarrow{x_2} \cdots$ is an execution of $\mathcal{P} \| \mathcal{D}$ if for all $i \geq 0$, either (1) $x_i = \mathit{Next}$ and $T(\mathit{Next}(d_i), s_i) = s_{i+1}$ and $\mathit{Step}(s_i, d_i) = d_{i+1}$, or (2) $x_i = \mathit{Delay}$ and $s_i = s_{i+1}$ and $\mathit{Delay}(d_i) = d_{i+1}$. In this execution, a transition $\xrightarrow{\mathit{Next}}$ is a *Next*-transition and $\xrightarrow{\mathit{Delay}}$ is a *Delay*-transition. A pair (s, d) is a reachable state of $\mathcal{P} \| \mathcal{D}$ if it occurs on an execution. A *db-delay execution* of $\mathcal{P} \| \mathcal{D}$ is one in which the number of *Delay*-transitions is *db*. Thus, a delaying explorer $\mathcal{D}$ induces a stratification of the executions of a program $\mathcal{P}$ such that the i-th stratum contains exactly the set of i-delay executions.

In order to ensure that all choices are covered (*soundness*), the delaying explorer must ensure that all choices are indeed generated by successive applications of *Delay*. To formalize this requirement, we define Delay^k (for $k \geq 0$) inductively as

$$\begin{aligned} \mathit{Delay}^0(d) &= d \\ \mathit{Delay}^{k+1}(d) &= \mathit{Delay}(\mathit{Delay}^k(d)) \end{aligned}$$

and Next^k (for $k \geq 0$) inductively as

$$\begin{aligned} \mathit{Next}^0(d) &= \{\} \\ \mathit{Next}^{k+1}(d) &= \mathit{Next}^k(d) \cup \{\mathit{Next}(\mathit{Delay}^k(d))\}. \end{aligned}$$

A delaying explorer $\mathcal{D}$ is *sound* with respect to a program $\mathcal{P}$ if for every reachable state (s, d) of $\mathcal{P} \| \mathcal{D}$, we have that $\mathit{Delay}^{|\mathit{Choices}(s)|}(d) = d$ and $\mathit{Choices}(s) = \mathit{Next}^{|\mathit{Choices}(s)|}(d)$. The following theorem says that reachability analysis on $\mathcal{P} \| \mathcal{D}$ is equivalent to reachability analysis on $\mathcal{P}$.

Theorem 1 *Consider a program $\mathcal{P}$ and a delaying explorer $\mathcal{D}$ that is sound with respect to $\mathcal{P}$. A state s is reachable in $\mathcal{P}$ iff (s, d) is reachable in $\mathcal{P} \| \mathcal{D}$ for some d.*

Proof sketch: The proof is by induction on the length of an execution of $\mathcal{P}$. The base case for the initial state of $\mathcal{P}$ is trivial. For the inductive case, suppose s is reachable in k steps and s has a transition to s'. From the induction hypothesis, (s, d) is reachable in $\mathcal{P} \| \mathcal{D}$ for some d. The soundness of $\mathcal{D}$ with respect to $\mathcal{P}$ allows us to explore all outgoing choices at the cost of a suitable number of delays. Therefore, we can take a sequence of transtions in $\mathcal{P} \| \mathcal{D}$ to get to (s', d') for some d'.□

Example: Let us consider a simple program in which the only source of nondeterminism is the scheduling of concurrent processes. An example of a delaying explorer for this program is a round-robin process scheduler. The state D of this scheduler is a queue of process ids initialized to contain the id of the initial process. *Next* returns the process id at the head of the queue. *Step* instruments the program's execution so that the id of a new process is added to the tail, the id of a terminated process is removed, and the id of a blocked process is moved to the tail. *Delay* moves the process id at the head to the tail. This explorer maintains the invariant that the ids of all enabled processes are present in the queue. By applying the *Delay* operation at most n times, where n is the size of the queue, any enabled process can be moved to the head and be returned by a subsequent call to *Next*. Therefore, this explorer is sound with respect to the program.

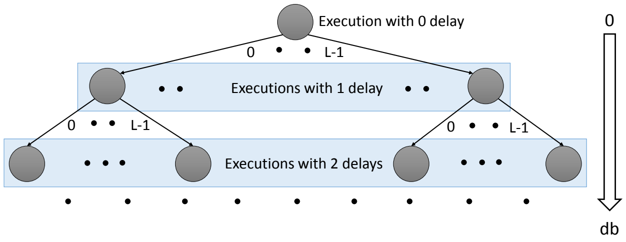

Figure 4: Stratified exhaustive search

3 Stratified Exhaustive Search

Figure 4 shows a pictorial representation of stratified exhaustive search of a program with respect to a delaying explorer. In this picture, L is the maximum number of steps in the program. In contrast to the graphs in Figures 2 and 3 where a node represents the program state, each node in Figure 4 is a complete execution of the program. The root node is the execution with no delays. This execution presents at most L positions to insert a delay operation, each yielding another complete execution with a single delay operation. These executions are indicated by the nodes at the end of the edges coming out of the root node. This process can be continued until all executions have been generated. It is clear that there can be at most L^{db} executions with no more than db delays. Thus, for small values of db, it is feasible to enumerate all executions even for large values of

L. This observation suggests our stratified exhaustive search algorithm (SES) which generates executions level by level, exploring all executions in a level before moving to the next level. A delaying explorer induces a stratification of the executions of a program; in general, different delaying explorers induce different stratification for the same program. Thus, a delaying explorer is a mechanism to bias the search performed by our SES algorithm to different parts of the execution space.

The algorithm in Figure 5 takes as input a program $\mathcal{P}$, a delaying explorer $\mathcal{D}$, and a parameter $\delta > 0$. It uses three global variables. The integer *db*, initialized to 0 and iteratively incremented by δ, contains the current delay bound. During the search, a frontier of pending executions, that go beyond the current delay bound, is maintained in the dictionary *Frontier*. For each state s in the frontier, *Frontier* contains a pair (d, i), where d is the explorer state just prior to the the execution of i-th transition from state s. The mapping from s to (d, i) is put into the frontier because execution of the i-th transition would require more delays than the current bound. Finally, we optimize the search by using a cache of (hashes of) visited states maintained in the set *Cache*.

The workhorse of our algorithm is *DelayBoundedDFS*, a procedure with four parameters—program state s, explorer state d, transition count i, and delay count n. The goal of *DelayBoundedDFS* is to continue exploration from state s. The transition count i is the number of transitions already explored from s. The delay count n is the number of delays required, starting from the initial state, to execute the next transition out of s. *DelayBoundedDFS* iterates through the transitions from s by repeatedly invoking the *Next* operation of the delaying explorer to find out which transition to execute and incrementing i to indicate the execution of another transition. For each discovered state s', if s' is not present in *Cache* then it is added to *Cache* and *DelayBoundedDFS* is called recursively on s'. To move to the next transition, the *Delay* operation of the delaying explorer needs to invoked. If the current delay count n has already reached the current delay bound *db* and there is at least one more transition to be executed, then exploration cannot continue from s and work for the remainder of exploration from s is added to the frontier. Otherwise, the *Delay* operation is used to update d and the delay count n is incremented.

The top-level procedure of our algorithm is *SES*. This procedure initializes *db* to 0 and *Frontier* and *Cache* to $\emptyset$. It then executes two nested loops. The outer loop iterates over the value of *db* incrementing it by δ each time around. The goal of each iteration of this loop is to restart each pending exploration in the current frontier. To do this task, a copy of *Frontier* is made in *Frontier*$'$ and *Frontier* is reset to $\emptyset$. The inner loop then picks each work item in *Frontier*$'$ and invokes *DelayBoundedDFS* with it. The execution of the inner loop refills *Frontier* which is again emptied in the next iteration of the outer loop.

Theorem 2 *Consider a program $\mathcal{P}$ and a delaying explorer $\mathcal{D}$ that is sound with respect to $\mathcal{P}$. The* SES *algorithm (Figure 5) terminates and visits a state s' iff s' is reachable from s_0.*

Proof sketch: We argue that *SES* terminates for any program $\mathcal{P}$. Since $\mathcal{P}$ does not have any infinite executions and $Choices(s) \subseteq Cid$ is finite for any state s, any invocation of *DelayBoundedDFS* terminates. The inner loop in *SES* terminates because each iteration removes one entry from *Frontier*$'$. The termination of the outer loop is based on the observation that the inner loop adds a state s to *Frontier* only if there is an execution reaching s with more delays than db'. The number of delays in an execution is bounded by $L \times Cid$, where L is the maximum number of steps in $\mathcal{P}$. Since the outer loop increments db' in each iteration and the number of delays for an execution is bounded, eventually *Frontier* will become empty.

Next, we argue that *SES* is safe and eventually visits all reachable states of $\mathcal{P}$. This argument depends on the crucial assumption that the delaying explorer $\mathcal{D}$ is sound with respect to $\mathcal{P}$. Because of this property, by applying delays repeatedly in a state all outgoing transitions are taken.□

Neither the termination nor the safety argument for our algorithm depends on *Cache*. The only role of *Cache* is to optimize the search by avoiding redundant executions. Therefore, there is considerable flexibility in how much memory is devoted to the storage for *Cache*. The two extreme cases are when *Cache* is not used at all and when all visited states are put into *Cache*. But, it is possible and our implementation supports imposing a bound on the memory consumption for *Cache* beyond which states are either not added to *Cache* or added with replacement.

An important consideration in our use of *Cache* is that we store only the program state in it and avoid storing the explorer state. This design has the advantage that we get the maximum pruning out of the use of state caching. If a state s is first visited with explorer state d and later with explorer state d', the second visit is ignored even if it happened with fewer delays compared to the first visit. As a result, we can avoid re-exploration for the second visit. However, it may be possible that a state is discovered with a higher delay than the minimum delay required to visit it. We believe that this trade-off is good because the primary goal of a delaying explorer is to bias the search rather than enforce strict priority.

Finally, we note that it is enough to store only a hash of a state in *Cache*. But it is important to store the full state both when it is passed as a parameter to *DelayBoundedDFS* or when it is stored in *Frontier* since the program needs to be executed from it. For the latter uses, a state could either be cloned or reconstructed by re-executing the program from the beginning.

4 Stratified Sampling

In the previous section, we described the SES algorithm to perform stratified exhaustive search over the executions of an asynchronous reactive program. In this section, we describe a complementary algorithm that enables stratified exploration via near-uniform random sampling of executions from the strata induced by a delaying explorer; we call this algorithm the stratified sampling algorithm (SS).

To motivate why random sampling is beneficial, we note that the complexity of the SES algorithm grows exponentially with the upper bound on the number of allowed delays. Consequently, if a delaying explorer is unable to find a bug quickly within a few delays, the search often takes more time than the programmer is willing to wait for. To deal with this common problem, a time bound is usually supplied in addition to the number of delays. When an external time bound could stop the search before the delay limit has been reached, random sampling has certain advantages over exhaustive deterministic exploration. First, unlike deterministic exploration, random sampling can sample *every* execution with a non-zero probability, making it possible to distribute the limited time resource over the entire search space. Second, since each sample is generated independently of every other sample, random exploration can be easily and efficiently parallelized, an important advantage in an era where parallelism is abundantly available via multicore and cloud computing.

Figure 6 shows how our algorithm samples an execution with two delay operations. First, the *ExecutePath* function (defined later in Figure 7) executes the program using a custom strategy defined by the delaying scheduler without introducing any delays. The *ExecutePath* function returns the length of the execution L_0 from the start state to the terminal state. Using **choose**(L_0) we uniformly pick a value n_0 in the range $[0, L_0)$ to insert the first delay. When *ExecutePath* is invoked again, it introduces a delay at n_0, deterministically executes the program upto termination, and returns L_1, the length of the path since the last delay. Using **choose**(L_1) we uniformly pick a value n_1 in the range $[0, L_1)$ to insert the second delay. Finally, the execution $S_{0,0} \rightarrow^* S_{n_0,0} \rightarrow S_{0,1} \rightarrow^* S_{n_1,1} \rightarrow S_{0,2} \rightarrow^* S_{L_2,2}$ represents a random execution with two delays.

Given a program $\mathcal{P}$, a delaying explorer $\mathcal{D}$, and a delay bound db, an invocation of *DelayBoundedSample* (Figure 7) produces a terminating execution of $\mathcal{P}$ with no more than db delays. The random exploration performed by our algorithm is very different in spirit from the classical random walk algorithm on a state-transition graph (Figure 2) which starts from the initial state and executes the program by randomly selecting a transition out of the current state. This naive random walk, although it guarantees a non-zero probability for sampling any execution, suffers from the problem that the probability of sampling long executions decreases exponentially with the execution length. Instead, our algorithm performs a random walk, not on the state-transition graph, but on a different graph (Figure 4) induced by the delaying explorer $\mathcal{D}$. In this graph, each node is a complete terminating execution (as opposed to a state) and an edge is a position in the execution for inserting a delay (as opposed to transition). We show later that the probability of sampling any execution requiring *db* delays is at least $\frac{1}{L^{db}}$. Unlike the naive random walk, the probability of sampling an execution is exponential in the number of required delays rather than the number of steps. A long execution has just as much chance to be produced as a short execution with the same number of delays, thereby eliminating the bias towards short executions.

The algorithm in Figure 7 uses a single global variable *path*, a sequence of natural numbers. This sequence represents a path as follows. For each i starting from 0 and up to $path.Length - 1$, execute $\mathcal{P}$ for $path[i]$ steps followed by a delay. Finally, execute $\mathcal{P}$ until it terminates. The procedure *ExecutePath* performs the execution encoded by *path* and returns the number of steps performed after the last delay.

The procedure *DelayBoundedSample* invokes the procedure *ExecutePath* repeatedly to randomly sample an execution with *db* delays. If the initial state s_0 does not have any transitions, there is nothing to do. Otherwise, it sets *path* to the empty sequence and calls *ExecutePath* which executes $\mathcal{P}$ without any delays. The algorithm chooses a step at random from the number of steps returned by *ExecutePath* as the position to execute a delay operation. It extends *path* with it and invokes *ExecutePath* again to create a new execution. It continues to do so iteratively until the number of delays in the execution has reached *db*. A single invocation of *DelayBoundedSample* samples a single execution with *db* delays. To calculate this sample, it must re-execute the program *db* times and perform *db* random choices.

Theorem 3 *Consider a program $\mathcal{P}$ and a delaying explorer $\mathcal{D}$ that is sound with respect to $\mathcal{P}$. Let L be the maximum number of steps along any execution of $\mathcal{P}$. For any integer $db \geq 0$ and any execution τ of $\mathcal{P} \| \mathcal{D}$ with db delays, the* SS *algorithm (Figure 7) generates τ with probability at least $\frac{1}{L^{db}}$.*

Proof sketch: The SS algorithm performs a random walk on the graph in Figure 4. The branching factor of this graph is bounded by L, the maximum number of steps in $\mathcal{P}$ and its depth is bounded by the delay bound *db*. Since the number of terminal nodes in the graph is bounded by L^{db}, the probability of sampling any execution requiring *db* delays is at least $\frac{1}{L^{db}}$.□

Figure 7 also shows a procedure *SS* that repeatedly invokes *DelayBoundedSample* to implement a stratified sampling algorithm. This procedure has an (timeout-terminated and infinite) outer loop that repeatedly increases the delay bound *db*. The inner loop samples *NumSamples*(*db*) executions from the set of executions with exactly *db* delays by invoking *DelayBoundedSample* repeatedly. Our algorithm is parameterized by a function *NumSamples* that specifies the number of executions to be sampled for each delay bound. As we have explained before, the number of executions increases exponentially with the number of available delays. Therefore, we believe that a practical *NumSamples* function should also have an exponential dependency on the delay bound. For our evaluation (Section 6), we chose $c_1 + c_2^{db}$ to be the shape for *NumSamples*(*db*); through trial and error, we found that $c_1 = 100$ and $c_2 = 3$ work well for the benchmarks we studied in this paper.

5 Implementation

In this section, we provide an overview of the framework we used for the evaluation of delaying explorer based systematic testing. Note that the idea of using delaying explorers is *independent* of the asynchronous programming language used and the explorer used for searching the state space.

P ***programs***: We wrote our programs in the P programming language [6], a domain-specific language for implementing asynchronous event-driven systems. A P program is a collection of state machines, each with an input message queue, communicating with each other by sending and receiving messages. The

P compiler generates from the input program both C code and Zing code. The generated C code is used for executing the application either locally on a single computing node or distributed across a collection of nodes; the P runtime supports both local and distributed execution. The generated Zing code is provided as input to the Zing explorer [1] for systematic testing.

There are two kinds of nondeterminism in the semantics of P programs. First, P has interleaving nondeterminism because the language provides a primitive for dynamic machine creation. As a result, multiple machines can be executing concurrently. In each step, one machine can be chosen nondeterministically to execute and it can either compute on local state or dequeue a message or send a message to another machine. This nondeterminism implicitly creates nondeterminism in the order in which messages are delivered to a machine. The code of a machine has to be programmed robustly and tested so that it continues to perform safely regardless of the reordering. Second, a P program may also make an explicit nondeterministic choice by using the special expression `$` whose evaluation results in a nondeterministic *Boolean* choice. This feature is extremely useful for modeling the environment of reactive systems; like nondeterministic component failure or message loss.[2]. For scalable[3] systematic testing of such programs, it is essential to control both these sources of non-determinism.

Implementing a delaying explorer: We have implemented the algorithms in Sections 3 and 4 using the infrastructure in the Zing model checker. The component of Zing most pertinent to our implementation is state caching and the explorer that orchestrates the depth-first search of the state-transition graph of the input Zing program. We modified the explorer to query an external object implementing the `IZingDelayingScheduler` interface. The explorer invokes the method `Next` to determine the process whose transition it should explore and the method `Delay` to inform the scheduler of its decision to delay the next process.

```
interface IZingDelayingScheduler
{
/* Next is called by ZING to get the next process to be executed */
    int Next ();

/* Delay is called by ZING to delay the delaying explorer state */
    void Delay ();

/* Start is called by ZING when a new process is created */
    void Start  (int processId);

/* Finish is called by ZING when a process is terminated */
    void Finish (int processId);

/* Step is called to communicate information to the
delaying explorer for e.g. change priority, process blocked.*/
    void Step (params object[] P);
}
```

The methods `Start`, `Finish`, and `Step` together implement the capability formalized by the *Step* function described in Section 2; these methods inform the delaying scheduler of important events occurring during the execution. The method `Start` is invoked whenever a new process is created and the method `Finish` whenever a process terminates. The method `Step` is used to implement a general mechanism for instrumenting the program's execution for updating the scheduler state.

Controlling non-determinism: The general approach of controlling schedules in systematic testing frameworks [3, 13, 11] is to instrument the program at every synchronization points. In the context of asynchronous message passing programs like P, the only synchronization points are at enqueue of a message, blocking at dequeue and creation of a new machine (more details in [6]). The P compiler automatically instruments the program at these three points and passes the information to the delaying explorer using the `Step` function. In addition to prioritizing interleaving nondeterminism, a delaying explorer must also prioritize explicit nondeterministic choice. We simply adopt the convention that *false* is ordered before *true*. For a language that provides nondeterministic choice over types other than *Boolean*, the choices may be controlled by expanding the `IZingDelayingScheduler` interface.

[2] Environment models are erased during compilation to C code and replaced with hand-written C code.

[3] In this context, scalable means finding bugs faster.

6 Evaluation

Our evaluation was directed towards the following goals:

- ✓ Evaluate the performance of SES and SS in comparison with the best known approaches, preemption bounding [13] and probabilistic concurrency testing [3], respectively (Section 6.1).
- ✓ Evaluate the performance of different delaying explorers in finding bugs, and demonstrate the need for flexible delaying explorer interface (Section 6.2).
- ✓ Demonstrate the benefit of writing custom explorer with a case study of chain replication protocol(Section 6.3).

Experimental setup: All the experiments are performed on Intel Xeon E5-2440, 2.40GHz, 12 cores (24 threads), 160GB machine running 64 bit Windows Server OS. The ZING model checker can exploit multiple cores during exploration as its iterative depth-first search algorithm is parallel [20]. We do not report the time taken to find bugs as it is dependent on the degree of parallelism and the parallel explorer implementation, but instead we report the number of distinct states explored (in the case of SES) and number of schedules explored (in the case of SS) before finding the bug. Time taken to find the bug is directly proportional to these parameters. The numbers reported for the evaluation of stratified sampling algorithm in Table 1 are a median over 5 runs of the experiment.

Benchmarks: We have used P [6] to implement a fault tolerant Transaction Management System (TMS) and a Windows driver communicating with an OSR [7] device. We used P because its compiler provides a translation both to C code for execution on the Microsoft Azure cluster and to Zing code for systematic testing. Our implementations are not abstract models; they are detailed enough to be deployed as a distributed service. TMS uses various protocols like *Two-Phase Commit* protocol [2] for atomicity of transactions, *Chain Replication* protocol [21] for fault-tolerant replication of state machines, *Multi-Paxos* protocol [4] for consistent log replication and consensus. The buggy programs used for evaluation in this paper were collected during the development of this protocol suite. Each row in Table 1 represents a different bug. We only consider hard-to-find bugs that led to unhandled-event exceptions (system crash) and violation of global safety specifications (written as monitors).

6.1 Evaluation of SES and SS

Evaluating SES: We applied the iterative SES algorithm with different delaying explorers to the set of buggy programs (incrementing the value of *db* by 1 after each iteration). For evaluating the performance of SES, we implemented iterative preemption bounding [13] (PB) with state-caching in ZING. Table 1 shows the number of distinct states explored before finding the bug by both the approaches. It can be seen that PB fails to find the bug in most of the cases, and in cases where PB succeeds, SES with some delaying explorer is able to find the bug orders of magnitude faster (except for TMS_1 and ChainRep_8). Also, there is a lot of variance in the performance of SES when combined with different delaying explorers, which motivates the need for a flexible interface to write custom delaying explorers.

Evaluating SS: We implemented naive random walk as the baseline for comparison. For random walk, the probability of finding a bug decreases exponentially with length of buggy execution. Random walk fails to find most of the bugs, confirming that these bugs are hard to find with very low % of executions leading to the bug. We compared the iterative SS algorithm described in Section 4 with the PCT [3] algorithm, which is considered as state of the art in probabilistic concurrency testing. PCT provides probabilistic guarantees of finding a bug with *bug-depth* d, by randomly inserting d priority inversions. Most of the concurrency bugs using PCT were found with bug depth of less than 3 in [3, 15]. The PCT algorithm makes an assumption about the maximum length of program execution (k), which is hard to compute statically in the case of asynchronous reactive programs. We use $k = 5000$ and $d = 5$ for our experiments. Table 1 shows that PCT fails to find most of the bugs, confirming that the bugs in asynchronous programs generally have a larger *bug-depth*.

Programs	Stratified Sampling					Stratified Exhaustive Search			
	No. of schedules explored before finding bug					No. of states explored before finding bug			
	Random	PCT $d = 5$	SS + Delaying Explorer			PB	SES + Delaying Explorer		
	Walk (RW)	$k = 5000$	RR	RTC	PRR		RR	RTC	PRR
2pc_1	124356	22341	8278	**331**	5183	7934221	88451	6571	**6512**
2pc_2	*	*	10943	6378	**6300**	*	*	17690	**9090**
2pc_3	*	*	5023	**1018**	4109	48321	1898	**1123**	2189
2pc_4	*	*	*	**8874**	*	*	*	**5101**	77212
ChainRep_1	998445	9832	5607	9999	**1985**	*	92178	9913	**936**
ChainRep_2	*	*	34034	**7829**	28221	74231	32166	**8821**	88732
ChainRep_3	*	13283	2032	**1711**	6093	*	19731	**3452**	8981
ChainRep_4	*	4439	3452	4249	**1238**	59234	**672**	5441	11742
ChainRep_5	*	55	**53**	110	101	*	3973	**521**	6652
ChainRep_6	*	*	*	*	*	*	78443	**44331**	54981
ChainRep_7	*	*	*	*	*	*	*	**3538**	*
ChainRep_8	*	*	5513	*	**2201**	8342	9791	*	**8218**
ChainRep_9	*	*	66381	**9425**	16559	*	37222	**7812**	37213
ChainRep_10	*	**74**	129	331	888	812561	5431	1944	**1781**
MultiPaxos_1	*	9934	7821	**765**	5819	*	**82114**	89341	88129
MultiPaxos_2	*	*	9872	**8873**	11239	*	15563	9983	**1934**
Multipaxos_3	*	*	19881	15023	**9589**	*	18831	8923	**1198**
Paxos_1	*	592	122	**53**	233	3320	2233	**1098**	4312
Paxos_2	*	*	14463	**1831**	19874	77834	4912	**833**	8831
Paxos_3	*	*	*	*	*	*	*	**14832**	*
TMS_1	*	9427	**12**	305	34	**553**	2220	660	8965
TMS_2	*	*	*	*	*	*	*	**44832**	*
TMS_3	#	#	#	#	**3009214**	#	#	#	#
TMS_4	#	#	#	**5530042**	#	#	#	#	#
OSR_1	44235	332	**75**	122	1009	5532	4421	**683**	55392
OSR_2	7756	4331	115	**66**	224	12864	12931	**1634**	3212

*** → the search ran out of memory budget of $60GB$ or exceeded the time budget of 2 hours.**
# → the search exceeded the time budget of 5 hours (running for longer duration).

Table 1: Evaluation Results for SS and SES using various delaying explorers

In the cases where PCT succeeds in finding the bug, SS with some delaying explorer is orders of magnitude faster. Similar to the behavior of SES, for SS also we see variance in performance of different delaying explorers across different problems.

Comparison between SES **and** SS: We have extensively used both SES and SS for finding bugs in our implementations. In our experience, the SES algorithm is able to find bugs faster than SS in most of the cases as it uses state-caching to prune redundant explorations. Furthermore, SES can find low-probability bugs[4] that occur at smaller values of delay budget faster than SS. In the case of ChainRep_6 and Paxos_3 there was a low probability bug at small delay budget; SS fails to find it whereas SES finds it.

As the delay bound increases, search space explodes exponentially. If there is a bug that requires large delay budget for a given stratification strategy, then SES may fail to find it due to running out of memory. We came across scenarios (TMS_3 and TMS_4 in Table 1) where SES ran out of memory but after running SS for a long time we uncovered a bug. SS can be kept running for a long time without any memory constraints. Since it performs sampling with probabilistic guarantees, it may find a bug at larger delay budget where SES fails.

We can fruitfully combine both approaches as follows. Perform SES first to find all shallow (few delays) bugs quickly and get strong coverage guarantees. Once SES has uncovered all shallow bugs and has almost consumed the memory budget, perform SS from the frontier states and get probabilistic guarantees. We leave the evaluation of this combination for future work.

6.2 Experience with delaying explorers

We have implemented three different delaying explorers. In this section, we explain the construction of each explorer and the reasons for the variance in their performance. The source code for these explorers is available at the following website: [https://zing.codeplex.com/SourceControl/latest#DelayingSchedulers/].

Run-to-completion explorer (RTC**)**: The run to completion explorer was introduced in prior work [6] for testing device drivers written in P. The default strategy in RTC is to follow the causal sequence of events, giving priority to the receiver of the most recently sent event. When a delay is applied, the highest priority process is moved to the lowest priority position. Even for small values of delay bound, this explorer is able to explore long paths in the program since it follows the chain of generated events. In our experience, this explorer is able to find bugs that are at large depth better than any other explorer. For example, bugs in ChainRep_7 and TMS_2 were found were found by RTC at depth greater than 1500 and delay budget less than 4 while other explorers could not find these bugs.

Round-robin explorer (RR**)**: The round-robin delaying explorer, explained earlier in Section 2, cycles through the processes in process creation order. It moves to the next task in the list only on a delay or when the current task is completed. Round-robin explorer has been used in the past ([9, 19]) to test multi-threaded programs. In our experience, in most of the cases (Table 1) other delaying explorers perform better than RR. RR can be used for finding bugs that manifest through a small number of preemptions or interleaving between processes. Our evaluation shows that most bugs in asynchronous programs do not fall in that category.

Probabilistic round-robin explorer (PRR**)**: A probabilistic delaying explorer is one in which the *Step* operation is allowed to make random choices. While a deterministic delaying explorer induces a fixed stratification over the executions of a program, a probabilistic delaying explorer induces a probability space over stratification. We have experimented with a cannibalistic version of the round-robin explorer (PRR). We believe that the culprit behind the poor performance of the round-robin explorer is its default process scheduling order which is based on the order of process creation. The simplest way to change this default order is to randomize it. Instead of inserting a freshly-created process at the tail of the queue, insert it at a random position in the queue; everything

[4] Low-probability bugs are those for which the fraction of executions leading to the bug is small.

else carries over from the round-robin explorer. The probabilistic round-robin explorer is still sound since the definitions of *Next* and *Delay* do not change. Table 1 indicates that PRR typically performs better than RR.

6.3 Writing a custom delaying explorer

After testing the chain replication protocol using the three delaying explorers explained earlier, we tested it for more specific scenarios. One such scenario is testing the system against random node failures. We provide a brief description of the chain replication protocol. Next, we show how we wrote a custom explorer to test for the node failure scenario and found a previously unknown bug in our implementation.

The chain replication protocol [21] is a distributed fault-tolerant protocol for replicating state machines. Consider an instance of a chain replication system with 6 machines—4 instances of `Server` machine ($S_1, \ldots, S_4$) connected in a chain, 1 instance of `Master` machine (M), and 1 instance of `Fault` machine (F). $S_1, \ldots, S_4$ communicate with each other to implement replication. M periodically monitors the health of $S_1, \ldots, S_4$ to detect if any of them has failed. If it detects a fault in S_i, it tells the neighbors of S_i to reconfigure. F is a machine that models fault injection. It maintains a set of numbers initialized to $\{1, \ldots, 4\}$. F repeatedly and nondeterministically removes a number i from this set and sends a failure message to S_i until the size of the set becomes 1. The chain replication protocol is expected to behave correctly for N servers as long as at most $N - 1$ fail.

When a distributed system starts up, there is an initialization phase involving exchange of messages between nodes for setting up the network topology and other system configuration. Bugs during the initialization phase are straight forward, infrequent, and get discovered quickly. Subtle bugs are generally encountered after the system is initialized and has reached an interesting global state. Since we want to test our system against a specific scenario of failure occurring after the system has stabilized, the new delaying explorer should not spend a lot of time injecting failures or monitoring the system during the initialization phase. We need stratification that gives less priority to certain interleaving in the the initial phase.

To capture this intuition with a delaying explorer, we wrote a customized delaying explorer (*CustExplorer*). The explorer maintains an ordering of all dynamically-created machine and cycles through them based on the ordering. The program can change the ordering by invoking *ChangeOrder* callbacks (implemented using `Step`). Using *ChangeOrder* callback in the initialization phase, the machines $S_1, \ldots, S_4$ are ordered before machines M and F. After the initialization phase, the machines M and F are moved ahead in the ordering as compared to machines $S_1, \ldots, S_4$. Thus, *CustExplorer* helps in stratifying the search by giving less priority to interleaving the failure and monitor machines, until the system has stabilized.

Using *CustExplorer* we were able to find a previously unknown bug in chain replication, which occurred when the failure was injected simultaneously at two neighboring nodes after the initialization phase. *CustExplorer* was able to find the bug with SES by exploring 220103 states and with SS by exploring 193442 schedules. We applied the same strategy to ChainRep_6 as it had similar bug related to node failure and we were able to find the bug in 10445 states which is nearly 4 times faster than the next best.

7 Related work

Model checking [12, 22] is a classic technique applied to prove temporal properties on programs whose semantics is an arbitrary state-transition graph. Our use of state caching to prune search is inspired by model checking. Partial-order reduction [10] is another technique to prune search. Combining partial-order reduction with schedule prioritization techniques is known to be a challenging problem [14]. Coons et al. [5] have proposed a technique to combine preemption-bounding with partial-order reduction. In future work, we would like to investigate the feasibility of combining delayed exploration with partial-order reduction.

There is prior work on random sampling of concurrent executions. Sen [16] provides an algorithm for sampling partially-ordered multithreded executions. Similar to our work, the PCT algorithm [3] also exploits prioritization techniques to effectively sample multithreaded executions. The PCT algorithm characterizes a concurrency bug according to its depth and guarantees that the probability of finding a bug with depth d in a program with L steps and n threads is at least $1/nL^{d-1}$. The mathematical techniques underlying PCT and our sampling algorithm are different. PCT provides a custom algorithm for a particular notion of bug depth whose definition has a deep connection with the proof for the probability bound. On the other hand, our algorithm does not depend on a characterization of bugs. Rather, it is parameterized by a delaying explorer, a mechanism used by the programmer to stratify the search space. Consequently, the proof for our probability bound is a straightforward combinatorial argument on a bounded tree in terms of its branching factor and depth.

Predictive testing [17, 18, 23, 24] follows the basic recipe of executing the program, collecting information from the execution, constructing a model of the program from the collected information, and then re-executing the program based on new predicted interleavings likely to reveal errors. The various techniques differ in the information collected and the targeted class of errors. The search performed by predictive techniques is goal-driven but typically does not provide coverage guarantees. On the other hand, our search technique is not goal-driven but provides coverage guarantees.

Concurrit [8] proposes a domain specific language for writing debugging scripts that help the tester specify thread schedules for reproducing concurrency bugs. The search is guided by the script without any prioritization. In contrast, our work is focused on finding rather than reproducing bugs. Instead of a debugging script, a tester writes a domain-specific scheduler with appropriate uses of sealing; iterative deepening with delays automatically prioritizes the search with respect to the given scheduler.

8 Conclusion

We have demonstrated how delaying explorers help in systematic testing of asynchronous reactive programs. We also showed that using delay bounding [9] with a single default scheduler is not scalable for finding bugs. Different delaying explorers induce different stratification, and hence, writing custom delaying explorers as unit test strategies can make testing complex asynchronous protocols scalable. We also presented and evaluated two algorithms, (1) SES for exhaustive search with strong coverage guarantees and showed how state-caching can be used efficiently for pruning, (2) SS for sampling executions with probabilistic guarantees. We evaluated both these algorithms on real implementation of distributed protocols and showed that our techniques perform orders of magnitude better than state-of-art search prioritization techniques like preemption bounding and PCT.

References

[1] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proceedings of CAV*. 2004.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.

[3] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of ASPLOS*, 2010.

[4] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of PODC 2007.*

[5] K. E. Coons, M. Musuvathi, and K. S. McKinley. Bounded partial-order reduction. In *Proceedings of OOPSLA 2013.*

[6] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: Safe asynchronous event-driven programming. In *Proceedings of PLDI*, 2013.

[7] Drivers. OSR USBFX2 (http://code.msdn.microsoft.com/windowshardware/kmdfusbfx2-e785110a).

[8] T. Elmas, J. Burnim, G. Necula, and K. Sen. CONCURRIT: A domain specific language for reproducing concurrency bugs. In *Proceedings of PLDI*, 2013.

[9] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *Proceedings of POPL*, 2011.

[10] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag, 1996.

[11] P. Godefroid. Model checking for programming languages using Verisoft. In *Proceedings of POPL*, pages 174–186, 1997.

[12] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 1997.

[13] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of PLDI*, 2007.

[14] M. Musuvathi and S. Qadeer. Partial-order reduction for context-bounded state exploration. Technical Report MSR-TR-2007-12, Microsoft Research, 2012.

[15] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proceedings of PLDI 2012.*

[16] K. Sen. Effective random testing of concurrent programs. In *Proceedings of ASE, 2007.*

[17] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of PLDI*, pages 11–21, 2008.

[18] F. Sorrentino, A. Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *Proceedings of FSE*, 2010.

[19] P. Thomson, A. F. Donaldson, and A. Betts. Concurrency testing using schedule bounding: An empirical study.

[20] A. Udupa, A. Desai, and S. Rajamani. Depth bounded explicit-state model checking. In *Proceedings of SPIN*, 2011.

[21] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI 2004*.

[22] W. Visser and P. C. Mehlitz. Model checking programs with Java Pathfinder. In *Proceedings of SPIN*, 2005.

[23] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *Proceedings of FM 2009.*

[24] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *Proceedings of ICSE 2011.*

```
**var** $db : \mathbb{N}$;
**var** $Frontier : Dictionary\langle S, (D \times \mathbb{N})\rangle$;
**var** $Cache : Set\langle S\rangle$;
$DelayBoundedDFS(s : S, d : D, i : \mathbb{N}, n : \mathbb{N})$ {
    **var** $s' : S$;
    **while** $(i < |Choices(s)|)$ {
        $s', i := T(Next(d), s), i + 1$;
        **if** $(s' \notin Cache)$ {
            $Cache.Add(s')$;
            $DelayBoundedDFS(s', Step(s, d), 0, n)$;
        }
        **if** $(n = db \wedge i < |Choices(s)|)$ {
            $Frontier(s) := (d, i)$;
            **break**;
        }
        $d, n := Delay(s, d), n + 1$;
    }
}
$SES()$ {
    **var** $db' : \mathbb{N}$;
    **var** $Frontier' : Dictionary\langle S, (D \times \mathbb{N})\rangle$;
    $db, Frontier, Cache := 0, \emptyset, \emptyset$;
    $Cache.Add(s_0)$;
    $DelayBoundedDFS(s_0, d_0, 0, 0)$;
    **while** $(Frontier \neq \emptyset)$ {
        $Frontier', Frontier := Frontier, \emptyset$;
        $db', db := db, db + \delta$;
        **foreach** $((s, d, i) \in Frontier')$
            $DelayBoundedDFS(s, Delay(s, d), i, db' + 1)$;
    }
}
```

Figure 5: SES algorithm: Stratified exhaustive search

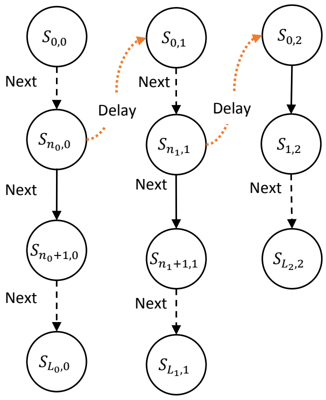

Figure 6: A run of SS algorithm

```
var $path : Sequence\langle\mathbb{N}\rangle$
$ExecutePath() : \mathbb{N}$ {
    var $i, j : \mathbb{N}$;
    var $s : S$;
    var $d : D$;
    $s, d, i := s_0, d_0, 0$;
    while ($i < path.Length$) {
        $j := 0$;
        while ($j < path[i]$) {
            $s, d, j := T(Next(d), s), Step(s, d), j + 1$;
        }
        $d, i := Delay(s, d), i + 1$;
    }
    $j := 0$;
    while ($0 < |Choices(s)|$) {
        $s, d, j := T(Next(d), s), Step(s, d), j + 1$;
    }
    return $j$;
}
$DelayBoundedSample()$ {
    var $i, l : \mathbb{N}$;
    if ($|Choices(s_0)| = 0$) return;
    $path := \emptyset$;
    $l := ExecutePath()$;
    $i := 0$;
    while ($i < db$)
    invariant $0 < l$
    {
        $path.Append($choose$(l))$;
        $l := ExecutePath()$;
        $i := i + 1$;
    }
}
$SS()$ {
    var $i : \mathbb{N}$;
    $db := 1$;
    while ($true$) {
        $i := 0$;
        while ($i < NumSamples(db)$) {
            $DelayBoundedSample()$;
            $i := i + 1$;
        }
        $db := db + 1$;
    }
}
```

Figure 7: SS algorithm: Near-uniform random sampling