Testing for Race Conditions in Distributed Systems via SMT Solving

João Carlos Pereira[1](✉), Nuno Machado[2], and Jorge Sousa Pinto[1]

[1] HASLab - INESC TEC & U. Minho, Braga, Portugal
joao.c.pereira@inesctec.pt, jsp@di.uminho.pt
[2] Teradata Iberia, Madrid, Spain
nuno.machado@teradata.com

Abstract. Data races, a condition where two memory accesses to the same memory location occur concurrently, have been shown to be a major source of concurrency bugs in distributed systems. Unfortunately, data races are often triggered by non-deterministic event orderings that are hard to detect when testing complex distributed systems.

In this paper, we propose Spider, an automated tool for identifying data races in distributed system traces. Spider encodes the causal relations between the events in the trace as a symbolic constraint model, which is then fed into an SMT solver to check for the presence of conflicting concurrent accesses. To reduce the constraint solving time, Spider employs a pruning technique aimed at removing redundant portions of the trace.

Our experiments with multiple benchmarks show that Spider is effective in detecting data races in distributed executions in a practical amount of time, providing evidence of its usefulness as a testing tool.

1 Introduction

Distributed systems are at the core of a wide range of applications nowadays, namely large-scale processing and storage, service synchronization, and cluster management [18]. Unfortunately, their inherent heterogeneity and complexity renders testing and debugging notoriously hard. As a consequence, bugs often surface in production, hampering the availability of services that are used everyday by millions of people which leads to huge economic costs [28,32].

A recent study has shown that, among the different types of distributed system bugs, *data races* are particularly challenging to find and debug, as they are non-deterministic and rarely manifest [18]. A data race consists in two *concurrent* accesses to the same memory location, where at least one access is a write. Such races in distributed systems typically stem from unpredictable message arrivals that violate the order or the atomicity of the protocols [18,20].

Over the last years, there have been multiple efforts to test and debug data races, although prior work has mostly focused on multithreaded programs [7,10,13,19]. Alongside, there has also been an increasing interest in applying

https://doi.org/10.1007/978-3-030-50995-8_7

formal verification techniques to prove correctness properties of distributed protocols, including the absence of race conditions [9,35]. However, these techniques are not yet suitable for mainstream usage because they require writing lengthy correctness proofs, which becomes a daunting task for complex systems [36].

More recently, Liu et al. proposed DCatch [20], a tool to discover distributed concurrency bugs that operates by employing a *happens-before* (HB) analysis on traces captured at runtime. DCatch was effective in finding races in popular applications, such as Apache Cassandra and ZooKeeper, even when monitoring correct executions. To keep the trace analysis tractable, DCatch relies on static analysis and hints provided manually by the programmer to capture solely events that lead to explicit failures. Despite that, its approach scales poorly, as the experimental results in the paper revealed that DCatch consumes GBs of memory for processing traces with a few MBs.

In this paper, we make the observation that distributed protocols typically involve inter-node communication steps that occur repeatedly along the execution (e.g. the leader election protocol in Zookeeper or the node heartbeats in Cassandra). Such redundant patterns, although useful to accurately understand the behavior of the system, not only produce large event traces that are prohibitively expensive to process, but also typically do not contribute to the occurrence of new data races. We thus believe that removing redundant events from traces can improve the performance and scalability of distributed system testing solutions without compromising their accuracy.

This paper proposes Spider, an automated approach to detect data races in distributed systems using redundancy pruning and symbolic constraint solving. Given a trace of a distributed system under test, Spider starts by performing a trace analysis aimed at eliminating events that appear recurrently in the execution and whose absence does not lead to any missed races. To this end, we leverage prior work on redundancy pruning for single-machine multithreaded applications [11] and extend it to message-passing systems.

After trimming the trace, Spider builds a causality model by encoding the HB relationships between events into a system of constraints over logical order variables. Finally, Spider resorts to an off-the-shelf SMT solver to compute the pairs of conflicting events that can run concurrently and, thus, form a data race.

Prior work has shown that SMT constraint solving can be successfully applied to reproduce [12], expose [24], and isolate [23,31] concurrency bugs in multithreaded programs. Alongside, SMT solving has also been employed to detect message races in models of distributed systems that are partially synchronous [33] or written as BPEL processes [5]. However, to the best of our knowledge, this is the first application of SMT solvers for race detection in arbitrarily large traces of distributed executions captured when testing unmodified source code.

We conducted an experimental evaluation of Spider using multiple benchmarks with distributed data races. Our results show that Spider is effective in detecting the bugs and that our redundancy pruning algorithm dramatically reduces the size of the traces (especially for distributed protocols based on rounds of message exchanges), which is paramount to scale our constraint solving approach. In fact, our redundancy pruning strategy was able to remove between 22% and 48% of the total amount of events in our experiments (Sect. 4.3).

In summary, this paper makes the following contributions.

– We present an algorithm, which draws on prior work [11], to eliminate redundant events from distributed system traces without hampering the race detection accuracy.
– We propose SPIDER, a tool that leverages redundancy pruning and SMT constraint solving for finding data races in distributed systems.
– We assess the performance and effectiveness of SPIDER on several benchmarks and show that our tool is capable of finding distributed races in a practical amount of time, even for executions with thousands of events.

The rest of the paper is organized as follows. Section 2 discusses some background concepts relevant to this work. Section 3 presents SPIDER and details both its architecture and *modus operandi*. Section 4 describes the experimental evaluation of SPIDER. Section 5 overviews the related work. Finally, Sect. 6 concludes the paper by summarizing its main findings.

2 Background

This section discusses some background aspects relevant to this paper, namely the types of data races in distributed systems and Satisfiability Modulo Theories.

2.1 Data Races in Distributed Systems

In general, a data race occurs when two accesses compete for the same resource in a non-synchronized fashion and at least one is modifying the resource. Since there is no causal relationship enforced between the two accesses, their ordering can vary across executions, which in some cases leads to failures.

Addressing data races in multithreaded applications has been the subject of extensive research over the years [7,10,13,19]. Unfortunately, data races in distributed systems are much more challenging than their single-machine counterparts. A distributed system comprises multiple nodes that interact with each other by exchanging messages, therefore concurrency occurs not only at the thread level but also at the node level and in a much larger scale. As message handlers often change the node's local state and trigger additional actions (e.g. sending a new message to another node), the timing in which messages are delivered and processed plays a decisive role in the correct execution of distributed protocols. In fact, most concurrency bugs in real-world distributed systems stem from the untimely delivery of messages [20]. Since those problematic execution interleavings are typically rare, they go unnoticed during testing and only surface in production with serious consequences.

According to the TaxDC study [18], distributed data races can be classified into two categories based on their message timing conditions:

- **Order violation:** An order violation occurs when the correct execution of a protocol in a node N requires that two events e_1 and e_2 run in a determined order (say, e_1 should execute before e_2) but the program code wrongly permits an execution interleaving in which e_2 occurs before e_1, thus causing an error. At one node, order violations can occur due to races between: *i)* two message arrivals, *ii)* a message arrival and a message sending, and *iii)* a message arrival and a local computation. In turn, across multiple nodes, they are caused by races between two message arrivals at different nodes.
- **Atomicity violation:** An atomicity violation occurs when the correct execution of a protocol in a node N requires that a critical region of events, denoted as $e_1, e_2, ..., e_n$, executes atomically but the program code wrongly permits an execution interleaving in which an external event x executes in-between e_1 and e_n, thus causing an error. The error would not manifest if x happens either before or after the critical region.
  At one node, atomicity violations can occur due to data races between a message arrival and an atomic local computation, whereas across multiple nodes they stem from races between a message arrival and an atomic global computation.

Figure 2 illustrates several of the aforementioned scenarios of order and atomicity violations, which were implemented as micro-benchmarks for the experimental evaluation conducted in this paper.

We now discuss SMT constraint solving, which is at the heart of our approach to detect distributed data races.

2.2 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is the decision problem of determining whether a first-order logical formula is satisfiable with respect to a *background theory*. A background theory provides interpretations for function and predicate symbols. For example, the theory of integers T_Z provides interpretations for the symbols 0, 1, $+$, $-$ and $\leq$. It is possible to devise theories to reason about varied kinds of objects, from real numbers to data structures such as arrays [30].

The SMT problem can be seen as a generalization of the SAT [3] problem where, in the place of propositional boolean variables, formulas may have predicates over non-binary variables (i.e. binary-valued functions of non-binary variables) whose interpretations are given by a background theory. As an example, consider the following two formulas:

$$x + y \leq z \land z \leq x - y \tag{1}$$

$$x \leq 0 \land 1 \leq x \tag{2}$$

Assuming the T_Z theory (also called *Presburger arithmetic*), formula (1) is satisfiable, for example with the assignment $\{x = 1,\ y = -1,\ z = 1\}$. In turn, formula (2) is *unsatisfiable* because there is no assignment of variables that evaluates the formula to *true*.

Programs which take as input a set of first-order formulas written in the context of a background theory and determine the satisfiability of the set are called *SMT solvers*. Modern SMT solvers, like Z3 [25], are already capable of solving formulations with thousand of constraints in a timely manner. Nevertheless, there is extensive ongoing research aimed at further improving their performance and features. SMT solvers have been employed in a wide range of applications, from program synthesis [6] to testing and debugging [10,12,23,24,31], as seen on this paper.

3 SPIDER

This section details Spider, a scalable approach to detect data races in distributed systems via SMT solving. We start by providing an overview of the solution, then describe the redundancy pruning algorithm and the happens-before SMT constraint model.

3.1 Overview

Spider assumes the existence of a trace with events captured from the execution of a distributed system either during testing or in production. We assume that traces contain the following events of interest, already considered in previous work [21,26]:

- **Intra-node thread events:** *fork*, *join*, *start* and *end* events which respectively represent the spawn of a new thread in a node, the termination of a thread, and the start and end of a thread's execution;
- **Inter-node communication events:** events *send* or *receive* representing respectively the sending and receiving of a message through sockets;
- **Intra-node events:** *read* or *write* accesses to shared variables, as well as *lock* and *unlock* events;
- **Message handling region delimiters:** events signaling the beginning and the end of a message handler.

Given an execution trace, Spider operates in three steps (see Fig. 1):

1. **Redundancy Pruning:** Spider employs a trace analysis to identify patterns of shared-memory accesses and message exchanges that appear replicated in the trace. These events, once removed, do not affect the causal dependencies of the remaining ones. In other words, these events are not relevant to the occurrence of new races and, therefore, can be safely excluded from the trace in order to reduce the size of the search space.
2. **HB Model Generation:** The pruned trace is then used to generate an SMT model that represents the events' logical clocks as symbolic variables and encodes the causality dependencies as constraints over those variables. This way, Spider is able to search for races over the entire set of possible logical time orderings of events, regardless of the execution recorded at runtime.

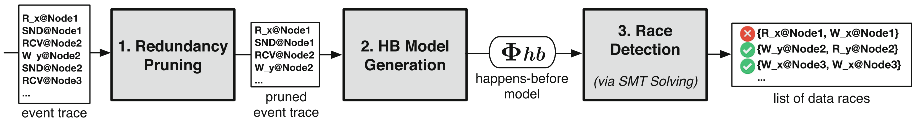


Fig. 1. Execution flow of SPIDER.

3. **Race Detection:** Once the HB model is generated, SPIDER produces a list of potential *data race candidates*. For each candidate, SPIDER then resorts to an off-the-shelf SMT solver to check whether the two accesses can have the same logical clock, meaning they execute concurrently and thus form an actual race. The list of valid data races is output at the end of the verification procedure.

The next sections describe each step in more detail, starting with a definition of the system model and its terminology.

3.2 System Model

For the purposes of this paper, a distributed system is modeled as a set of *nodes*, with at least one *thread* running at each node. Different threads communicate through message sending, with no further assumptions on message losses and network delays. Each thread can be viewed as a sequence of *events* of different types as defined in Sect. 3.1.

SPIDER is able to model multiple distributed execution orderings from the same event trace by leveraging the *Happens-Before (HB)* $\prec_{hb}$ relationship between events. This relationship states that, for two events e_1 and e_2 in the trace, if $e_1 \prec_{hb} e_2$ then event e_1 occurs before event e_2 at runtime [16]. In other words, the $\prec_{hb}$ relation encodes event causal dependencies in a *strict partial order*, which means that it has the following properties: *i) irreflexivity* – no event can happen before itself; *ii) transitivity* – if an event a happens before an event b and b happens before another event c, then a happens before c; *iii) asymmetry* – no event a can simultaneously happen before and after another event b.

The HB relation is commonly captured by means of *logical clocks* (also known as *Lamport clocks*) [16], which are integer values that indicate the logical time in which events occur in the execution. If an event e_1 happens-before an event e_2, then their respective logical clocks $C(e_1)$ and $C(e_2)$ will reflect that dependency: $e_1 \prec_{hb} e_2 \rightarrow C(e_1) < C(e_2)$.

SPIDER casts the problem of assigning logical clocks to events as an SMT constraint solving problem. However, since the time necessary to solve an SMT formulation increases proportionally to its number of constraints, it is of paramount importance to reduce them as much as possible in order to obtain a solution in a practical amount of time. In the next section, we describe how SPIDER employs redundancy pruning to achieve this goal.

3.3 Redundancy Pruning

The performance of SPIDER's constraint solving approach is mainly determined by the number of events present in the trace. Thus, by reducing the trace length, one is able to decrease the time necessary to discover data races. Alas, blindly removing events can affect the causality originally present in the execution and lead to both false negatives and false positives during race detection [11].

To address this issue, we leverage the ReX algorithm proposed by Huang et al. [11], which allows eliminating redundancy in multithreaded traces. In this context, a memory access is deemed *redundant* if its removal from the trace does not hamper the soundness or precision of race detectors.

ReX identifies redundant events using the concept of *concurrential-subsume equivalence*. Let e_i and e_j be two memory accesses made by a thread t, then e_i concurrentially-subsumes e_j when the following three conditions hold:

i) *Lexical Equivalence:* e_i and e_j originate from the same program instruction and have the same access type (i.e., both are reads or both are writes);
ii) *Memory Equivalence:* e_i and e_j access the same dynamic memory location;
iii) *HB-subsume:* for every event e_k such that $t_{e_k} \neq t_{e_i} \wedge t_{e_k} \neq t_{e_j}$: $e_k \prec_{hb} e_i \rightarrow e_k \prec_{hb} e_j$ and $e_i \prec_{hb} e_k \rightarrow e_j \prec_{hb} e_k$.

According to this concept, an event is considered redundant if the trace contains one concurrential-subsuming event from the same thread or two concurrential-subsuming events from different threads. ReX's redundancy pruning algorithm thus consists of checking, for each event in the trace, whether it is concurrential-subsumed by other events already in the trace and, if so, the event is eliminated.

To improve the efficiency of the analysis (especially for assessing condition *iii*), ReX computes the *concurrency context* for each thread while processing the trace in a stream-based fashion. The concurrency context of a thread consists in the sequence of *send* and *unlock* events observed up to a certain point. Since these events are the ones generating inter-thread HB dependencies, one can easily check whether condition *iii)* holds for e_i and e_j simply by comparing their threads' concurrency context. If the concurrency contexts match, then e_j is concurrential-subsumed by e_i, otherwise, it is not. We defer further details on the ReX algorithm to the original paper by Huang et al. [11].

Inspired by this work, we have implemented a redundancy pruning strategy that improves the performance of SPIDER's data race detection approach while maintaining its accuracy. Our strategy consists in a sequence of two passes over the traces that filter redundant events.

First, we apply a version of the ReX algorithm adapted to our system's model, namely by augmenting the concurrency context with *fork* events.[1] After this pass, the trace will be left with all but redundant *read* and *write* events.

[1] The original ReX algorithm does not take into consideration the existence of *fork* and *join* events signaling the creation and joining of threads.

We then perform a second pass on the trace designed to filter out *redundant message handlers* and *redundant threads*. These two terms can be defined by generalizing the redundancy criterion for a block of events: a block β of contiguous events occurring in the same thread is redundant if the removal of every event in β from the trace does not change the number of unique races detected. This means that, in order to be redundant, β must exhibit the following properties:

i) it does not contain non-redundant *read* or *write* events[2];
ii) it does not contain *send* or *receive* events;
iii) it does not contain *fork* events that spawn non-redundant child threads;
iv) all locks acquired in the block are released within its boundaries.

Based on the definition of redundant block, we can now define a *redundant message handler* as a redundant block that starts (resp. ends) with an event signaling the beginning (resp. the end) of a message handler. Alongside, a *redundant thread* is defined as a redundant block comprising all events of a thread.

Note that the second pass does not remove any memory access nor does it modify the HB relation between non-redundant events. As such, the number of unique data races computed by an HB-based race detector after applying the two filters will not differ from those obtained using the full original trace.

3.4 Happens-Before Model Generation

After pruning the trace, Spider builds the HB model, denoted Φ_{hb}, by *i)* representing each event's logical clock as a symbolic integer variable and *ii)* encoding their causal dependencies $\prec_{hb}$ as constraints over those symbolic variables. Considering the types of the events, the Φ_{hb} model can be defined as a conjunction of following sub-formulae:

- **Program Order:** Let E_1 and E_2 be the logical clocks of two events e_1 and e_2 occurring in the same thread context (meaning that either e_1 and e_2 are outside of any message handler or both are inside the same handler). If e_1 appears before e_2 in the trace, then: $E_1 < E_2$.
- **Thread Synchronization:** assuming that $Fork_t$, $Start_t$, End_t, and $Join_t$ represent, respectively, the logical clocks of the creation, beginning, end, and join operations of a thread t, then:

$$Fork_t < Start_t \tag{3}$$
$$End_t < Join_t \tag{4}$$
$$Start_t < End_t \tag{5}$$

- **Message Exchange:** let Snd_{m,l_1} and Rcv_{m,l_2} represent the logical clocks of the events of sending a message m on location l_1 and receiving m on location l_2, respectively. Then:

[2] Note that the second pass is performed after ReX, so any memory access existing in the block is guaranteed to be non-redundant. Thus, condition *i)* is automatically satisfied when block β does not contain any memory accesses.

$$Snd_{m,l_1} < Rcv_{m,l_2} \tag{6}$$

Simply put, a message can only be received if it was previously sent.

- **Message Handling:** let $Rcv_{m,l}$ denote the event logical clock for receiving m on location l, and let H_Begin_m and H_End_m represent, respectively, the logical clocks signaling the beginning and the end of m's message handler. Then:

$$Rcv_{m,l} = H\_Begin_m \tag{7}$$
$$H\_Begin_m < H\_End_m \tag{8}$$

Assuming that the handler is the region of the program responsible for processing the message, the first constraint states that a message m cannot be processed before it was received, as m's handler can only begin when m arrives. Moreover, the constraint also guarantees that no other message m' can be processed in-between $Rcv_{m,l}$ and H_Begin_m.

The second constraint ensures that the event signaling the beginning of an handler occurs before the event signaling its end.

- **Mutual Exclusion:** let $Lock_{t,v,l_1}$ and $Unlock_{t,v,l_2}$ represent, respectively, the logical clocks of the lock acquisition and release operations by thread t on a synchronization variable v at locations l_1 and l_2. Then:

$$Lock_{t,v,l_1} < Unlock_{t,v,l_2} \tag{9}$$

Moreover, when different threads compete to execute the same critical region, we need additional constraints to ensure mutual exclusion, i.e., that only one thread at a time accesses the variables encompassed by the lock.

Let P denote the set of locking pairs on a synchronization variable and let (L, U) and (L', U') be any two different locking pairs in P. The constraint encoding the mutual exclusion between locking pairs is as follows:

$$\forall_{(L,U),(L',U')\in P} : \; U < L' \;\vee\; U' < L \tag{10}$$

Solving the constraint model thus consists in assigning an integer value to each symbolic variable (i.e. to each logical clock), such that all constraints are satisfied. In other words, by solving the model, Spider is able to obtain a *feasible execution interleaving*, in which events are guaranteed to be ordered according to their happens-before relations.

3.5 Race Detection via SMT Solving

The last step of Spider's approach consists in using an SMT solver to identify race conditions. Let (e_1, e_2) represent a pair of *conflicting accesses* (i.e., read-write events to the same variable on the same node, with at least one write), and let E_1 and E_2 be the respective logical clocks of e_1 and e_2. The pair (e_1, e_2) is considered a data race iff it verifies the following *race* property:

$$race(e_1, e_2) \equiv \Phi_{hb} \wedge (E_1 = E_2) \tag{11}$$

The data race property Φ_{race} requires that the logical clocks E_1 and E_2 to have identical values while satisfying all other constraints in Φ_{hb}, which can only occur when the events e_1 and e_2 are not causally ordered. In other words, e_1 and e_2 form a data race because they do not have a happens-before relationship.

Spider resorts to an SMT solver to check whether Eq. 11 holds for each *candidate pair* (e_1, e_2). If the solver returns *satisfiable*, then (e_1, e_2) is considered an actual data race. Conversely, if the formula is *unsatisfiable*, then e_1 and e_2 cannot execute concurrently, hence (e_1, e_2) is not reported as a race.

After validating all candidate pairs of conflicting accesses, Spider outputs the list with the data races detected in the execution trace. It should be noted that the checking procedure is *embarrassingly parallel*, as each pair can be checked independently from the others.

Handling Intra-thread Data Races. Contrary to shared-memory programs on a single machine, in which data races can only occur in the presence of multiple threads, distributed systems can suffer from race conditions in a single thread. This scenario happens when there is an order violation due to a race between the arrival of two messages processed by the same thread, where at least one of the message handlers changes the node's state (see Fig. 2b for an example).

Spider addresses these type of data races in a two-fold fashion. First, it identifies message races in each thread. This is done by applying Eq. 11 to pairs of send events. Let m_1 and m_2 be two different messages processed by thread t and let Snd_{m_1} and Snd_{m_2} be the logical clocks of their sending events. If $race(Snd_{m_1}, Snd_{m_2})$ is satisfiable, then both messages are racing.

Second, Spider detects conflicting accesses in the message handlers by computing the intersection of their read-write sets. Let rw_1 and rw_2 be two events belonging to the handlers of m_1 and m_2, respectively, that access the same variable and at least one is to write. If m_1 and m_2 are racing, then (rw_1, rw_2) form a intra-thread data race.

4 Evaluation

To assess the benefits and limitations of Spider, we conducted an experimental evaluation focused on answering the following four questions:

– How effective is Spider in finding data races in distributed executions? (Sect. 4.2)
– How does the Spider's efficiency vary with the size of the execution trace? (Sect. 4.3)
– How does redundancy pruning affect Spider's effectiveness and efficiency? (Sect. 4.3)
– Is Spider sound and precise? (Sect. 4.4).

Our prototype of Spider was implemented in Java in around 1.9K lines of code and is publicly available at https://github.com/jcp19/SPIDER.

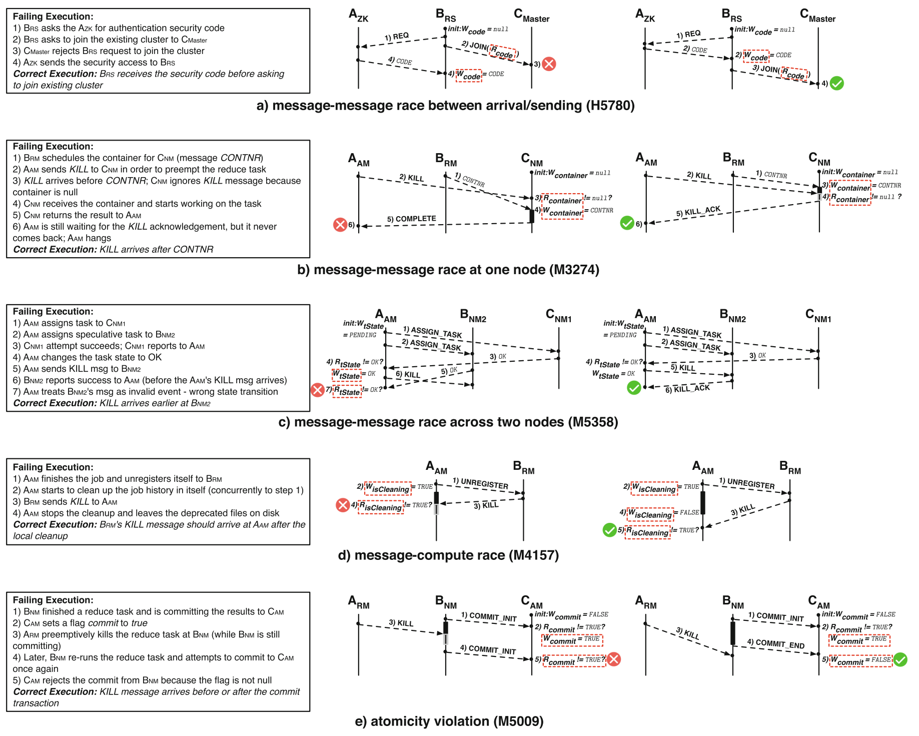
In the experiments, we used testing framework Minha [21] to collect the execution traces, and the SMT solver Z3 (version 4.4.1) to solve the constraints. We assumed a timeout of 2 h for constraint solving, after which the Z3 process was killed. All the experiments were ran on commodity hardware equipped with an Intel Core i7-8550U CPU and 16 GB of RAM.

The next sections describe the benchmarks used to evaluate Spider and discuss the results obtained.

4.1 Benchmarks

We used the following test cases to evaluate Spider's race detection approach.

TaxDC Micro-benchmarks. We designed five micro-benchmarks that were inspired by real-world races on popular distributed systems, namely HBase [2] and Hadoop MapReduce [1], as described in the TaxDC database [18]. These micro-benchmarks contain different types of data races (see Sect. 2) and are



Failing Execution:
1) B_{RS} asks the A_{ZK} for authentication security code
2) B_{RS} asks to join the existing cluster to C_{Master}
3) C_{Master} rejects B_{RS} request to join the cluster
4) A_{ZK} sends the security access to B_{RS}
Correct Execution: *B_{RS} receives the security code before asking to join existing cluster*

a) message-message race between arrival/sending (H5780)

Failing Execution:
1) B_{RM} schedules the container for C_{NM} (message *CONTNR*)
2) A_{AM} sends *KILL* to C_{NM} in order to preempt the reduce task
3) *KILL* arrives before *CONTNR*; C_{NM} ignores *KILL* message because container is null
4) C_{NM} receives the container and starts working on the task
5) C_{NM} returns the result to A_{AM}
6) A_{AM} is still waiting for the *KILL* acknowledgement, but it never comes back; A_{AM} hangs
Correct Execution: *KILL arrives after CONTNR*

b) message-message race at one node (M3274)

Failing Execution:
1) A_{AM} assigns task to C_{NM1}
2) A_{AM} assigns speculative task to B_{NM2}
3) C_{NM1} attempt succeeds; C_{NM1} reports to A_{AM}
4) A_{AM} changes the task state to OK
5) A_{AM} sends KILL msg to B_{NM2}
6) B_{NM2} reports success to A_{AM} (before the A_{AM}'s KILL msg arrives)
7) A_{AM} treats B_{NM2}'s msg as invalid event - wrong state transition
Correct Execution: *KILL arrives earlier at B_{NM2}*

c) message-message race across two nodes (M5358)

Failing Execution:
1) A_{AM} finishes the job and unregisters itself to B_{RM}
2) A_{AM} starts to clean up the job history in itself (concurrently to step 1)
3) B_{RM} sends *KILL* to A_{AM}
4) A_{AM} stops the cleanup and leaves the deprecated files on disk
Correct Execution: *B_{RM}'s KILL message should arrive at A_{AM} after the local cleanup*

d) message-compute race (M4157)

Failing Execution:
1) B_{NM} finished a reduce task and is committing the results to C_{AM}
2) C_{AM} sets a flag *commit* to *true*
3) A_{RM} preemptively kills the reduce task at B_{NM} (while B_{NM} is still committing)
4) Later, B_{NM} re-runs the reduce task and attempts to commit to C_{AM} once again
5) C_{AM} rejects the commit from B_{NM} because the flag is not null
Correct Execution: *KILL message arrives before or after the commit transaction*

e) atomicity violation (M5009)

Fig. 2. Overview of the TaxDC micro-benchmarks with distributed data races. Boxes on the left describe the steps of the failing executions, as well as how the bugs are prevented. Message diagrams containing, respectively, the failing and correct executions are depicted on the right of the figure. Data races detected by Spider are represented by red dashed boxes. (Color figure online)

publicly available [22], therefore we believe they can be useful for the community to evaluate similar testing tools in the future.

Figure 2 depicts the distributed data races considered in our micro-benchmarks. Following TaxDC's notation, in Fig. 2, each race condition is associated with a label that indicates the real-world bug on which the test case is inspired: the starting letter indicates the system (H stands for HBase, whereas M stands for MapReduce) and the number denotes the issue identifier (e.g. *H5780* represents the issue 5780 in HBase's issue tracking system). In turn, the node subscript indicates the system component present in the original buggy scenario: ZK stands for ZooKeeper, RS for region server, $Master$ for master node, AM for application master, RM for resource manager, and NM for node manager.

Since the purpose of these benchmarks is to allow evaluating Spider's ability to automatically detect different types of distributed data races rather than mimicking real-world workloads and code complexity, we developed them focusing solely on the aspects that contribute to the occurrence of the bug. As such, we represent local state queries and updates respectively as reads and writes on shared variables, and confine the behavior of each node to its message handlers.

a) Message-message race between arrival/sending (H5780). B_{RS} attempts to join the cluster by sending C_{Master} a JOIN message. However, since it does so before receiving the security-key message from A_{ZK}, the value null is sent to C_{Master}, thus causing an error.

b) Message-message race at one node (M3724). B_{RM} schedules a container for C_{NM} to work on a reduce task by sending the message $CONTNR$. Concurrently, A_{AM} sends a $KILL$ message to C_{NM} in order to preempt the reduce task. Since the two messages race with each other, the $KILL$ message can arrive before $CONTNR$ and be ignored by C_{NM} because no container exists yet (i.e. container = null). This untimely message arrival will cause C_{NM} to later reply to A_{AM} with a task-completion message, instead of the expected ACK.

c) Message-message race across two nodes (M5358). A_{AM} assigns a task to C_{NM1} along with a backup speculative task to B_{NM2}. When receiving the success confirmation from C_{NM1}, A_{AM} changes the state of the task to succeeded (tState = OK) and sends a $KILL$ message to B_{NM2}. However, if B_{NM2} manages to finish the task and also send the confirmation message OK to A_{AM} prior to receiving the $KILL$ signal, A_{AM} will consider B_{NM2}'s message as a wrong state transition and throw an exception.

d) Message-compute race (M4157). In the original bug, after finishing the task, A_{AM} unregisters itself to B_{RM} and starts removing its local temporary files. Concurrently to the local cleanup, B_{RM} sends a $KILL$ message to A_{AM} for stopping its execution. As a consequence, A_{AM} does not finish removing all files, which might cause storage space issues in the future.

This error is illustrated in our benchmark by means of a flag isCleaning in A_{AM}. In particular, A_{AM} spawns a worker thread to perform the local cleanup. This thread sets flag isCleaning to true (resp. true) at the beginning (resp. end) of the cleaning task. If A_{AM} receives B_{RM}'s $KILL$ before its working thread completes the cleanup, an error will occur.

e) ***Atomicity violation (M5009).*** After finishing a reduce task, B_{NM} starts committing the results to C_{AM} (which sets the flag `commit` to true). Simultaneously, A_{RM} sends a $KILL$ message to B_{NM}, thus preempting the task without resetting the commit states on C_{AM}. As a result, when later B_{NM} reruns the task and attempts to initiate a new commit transaction, C_{AM} fails due to a double-commit exception. The error does not manifest if the $KILL$ message arrives either before or after the transaction.

Peer Sampling Service. To assess how Spider's constraint solving time varies with the increase in the number of events in the execution, we used the implementation of a popular peer sampling service (PSS), named Cyclon [34], already used by prior work [21]. The goal of a PSS is to provide a gossip-based application with a churn-tolerant logical overlay for message dissemination.

Briefly, the Cyclon protocol operates as follows. For each node of the system, Cyclon maintains a *view*, which is a set of references to other nodes in the network associated with a timestamp. To ensure that this view remains consistent with the nodes alive at each moment, Cyclon performs periodic *shuffle cycles*, in which a node A sends a subset of randomly sampled peers to another node B, and receives a random subset of B's entries in return. Upon receiving a shuffle response, A replaces the oldest entries in its view by those received from B.

As noted by Machado et al. [21], the atomicity of the shuffle operation is not guaranteed by the original description of the Cyclon. This scenario happens when a node A requests a shuffle to a node B and, before receiving the response from B, A receives a shuffle request from another node C. As a result, the state of A's view upon receiving the references from B will not be the expected, as it was already updated with the entries sent by C. In the long term, this atomicity violation may generate corrupted views and break the connectivity of the dissemination overlay provided by Cyclon.

We picked the Cyclon PSS to evaluate Spider due to the possibility of obtaining arbitrarily large traces simply by changing the number of nodes and cycles used by the protocol. Moreover, we note that Cyclon is an *adversarial example* for race detection, as the message race scenario described above might not manifest in every the execution of the protocol and, when it does, the nodes involved and the cycles in which the violation occurs might vary across test runs.

4.2 Effectiveness

Table 1 reports the results of running Spider over traces captured from the benchmarks' execution. The experiments show that Spider successfully found all the pairs of racing instructions that caused the concurrency bugs. In particular, for test case H5780, Spider detects that there is a data race between the read of and the write to variable *code*, in steps 2 and 4. For M3724, Spider finds the data race on variable `container`. For M5358, Spider is able to detect that the state of variable `tState` can be concurrently modified by the message handlers of

Table 1. Race detection results without redundancy pruning. Column "Actual Races (Unique)" reports the number of data race candidate pairs that were confirmed by the SMT solver (the value within parenthesis indicates the amount of data races with unique code locations). Benchmarks whose names are of the form *Cyclon-XN-YC* indicate that the trace was obtained from runs of the protocol with X nodes and Y cycles. "-" means that SPIDER did not output any results due to timeout.

Benchmark	Trace size	#Trace events	#Constraints	#Race candidates	#Actual races (unique)	Solving time
h5780	3 KB	15	12	3	1 (1)	<1 s
m3274	3 KB	18	14	1	1 (1)	<1 s
m5358	5 KB	27	19	2	2 (2)	<1 s
m4157	2 KB	12	11	4	2 (2)	<1 s
m5009	4 KB	19	14	2	2 (2)	<1 s
Cyclon-5N-5C	74 KB	420	488	325	121 (1)	<1 s
Cyclon-5N-10C	145 KB	820	1464	1150	481 (1)	1.8 s
Cyclon-5N-100C	1433 KB	8020	104505	101500	49835 (1)	1 h 43 m
Cyclon-10N-5C	147 KB	840	976	650	243 (1)	<1 s
Cyclon-10N-10C	290 KB	1640	2920	2300	969 (1)	7.8 s
Cyclon-10N-100C	2869 KB	16040	6031	203000	-	Timeout
Cyclon-100N-5C	1486 KB	8401	9800	6500	2394 (1)	2 min 53 s
Cyclon-100N-10C	2934 KB	16401	29298	23000	9651 (1)	1 h 03 min
Cyclon-100N-100C	29076 KB	160400	60301	2030000	-	Timeout

C_{NM1} and B_{NM2}. For M4157, SPIDER correctly signals the flag set in the worker thread and the flag check in the message handler as a data race. Alongside, for M5009, SPIDER warns that the write to flag `commit` on step 2 and the read of the same variable on step 5 are not causally ordered (because they occur on two independent message handlers) and thus form a data race.

Finally, for Cyclon test cases, SPIDER is also effective in discovering problematic data races in the different execution scenarios. We note, however, that all of the races actually refer to the same unique pair of instructions in the source code. The reason why SPIDER reports them individually is that they correspond to events on different nodes and at different cycles. As shown in previous work [11], not all data race candidates with lexical equivalence are true data races.

4.3 Efficiency

We assessed the efficiency of SPIDER's data race detection technique by measuring its time and space overhead, respectively in terms of constraint solving time and trace sizes. To this end, we executed SPIDER with multiple configurations of Cyclon, varying the number of nodes in the system and the number of cycles of the protocol between the values $\{5, 10, 100\}$. The different configurations show how the constraint solving approach scales with the increase in the number of events in the execution and, consequently, the constraints in the

Table 2. Race detection results with redundancy pruning.

Benchmark	#Redundant events	#Constraints	#Candidate data races	#Actual races (unique)	Solving time (speed up)
Cyclon-5N-5C	122 (29%)	196	62	27 (1)	<1 s
Cyclon-5N-10C	296 (36%)	339	99	45 (1)	<1 s
Cyclon-5N-100C	3875 (48%)	2179	135	65 (1)	8.0 s (↓ 777.5x)
Cyclon-10N-5C	207 (24%)	446	173	78 (1)	<1 s
Cyclon-10N-10C	520 (32%)	838	348	149 (1)	<1 s (↓ 7.8x)
Cyclon-10N-100C	7466 (47%)	5180	1028	509 (1)	4 min 44 s
Cyclon-100N-5C	1980 (24%)	4437	1668	753 (1)	30.3 s (↓ 5.7x)
Cyclon-100N-10C	3719 (22%)	11893	6615	3134 (1)	22 min 25 s (↓ 2.8x)
Cyclon-100N-100C	47800 (30%)	47601	350202	-	Timeout

model. Table 1 reports the results of our experiments. The columns of the table indicate, respectively, the benchmark name, the size of the trace, the number of events in the trace, the number of constraints in the SMT model, the number of candidate data race pairs (i.e. the number of pairs of events with conflicting memory accesses in the trace), the number of confirmed pairs of events which contain data races, and the time the SMT solver took to check all candidate pairs.

The results show that, as expected, the constraint solving time increases with the number of events in the trace. From our experiments, it also became clear that the traces contain a large portion of redundant events, varying between 22% and 48% of the total number of events. Table 2 summarizes our observations. The columns of the table indicate, respectively, the benchmark that was run, the number of redundant events and its percentage of the total trace, the number of constraints in the generated SMT model after removing redundant events, the number of candidate data race pairs, the number of confirmed pairs of instructions which contain data races, and the time the SMT solver took to check all candidate pairs. Table 2 shows that removing redundant events before looking for data races can lead to big speedups in the time that the analysis takes. Despite this fact, there's still a *timeout* when SPIDER runs with the largest benchmark (Cyclon-100N-100C). We believe that this problem can be mitigated in future versions of SPIDER by optimizing the number of queries that are performed: instead of determining wether the events are concurrent for all pairs of candidates, we can analyse only the pairs whose corresponding code locations haven't yet been shown to produce concurrent events. Finally, we observe that even though the elimination of redundancy causes a decrease in the number of data race candidate pairs that were confirmed by the SMT solver, the number of data races with unique code locations remains unchanged and thus, no race was missed by removing redundant events.

4.4 Discussion About the Soundness and Precision of the Approach

In this section, we discuss why the results of data race analysis using SPIDER are to be trusted. First, we observe that SPIDER is *sound* in the sense that, given any trace, SPIDER is always able to find all pairs of instructions which lead to data races present in the trace. The analysis performed by SPIDER always terminates because, for each trace, there is a finite number of data race candidates, and the SMT constraints used to encode the causality model, and to find which pairs of instructions are concurrent, are encoded in *Quantifier-Free Integer Difference Logic (QF_IDFL)*, a decidable fragment of first-order logic.

Furthermore, we claim, without giving a formal proof, that redundant events are indeed of no importance for data race detection. As such, the redundancy pruning algorithm does not affect the soundness of Spider.

Assuming that the tracing mechanism captures all relevant synchronization events, no false positives will be reported by SPIDER, i.e. SPIDER will only report pairs of instructions if they can indeed produce non-synchronized (and thus, concurrent) memory accesses. Given that the redundancy pruning algorithm does not modify the HB relation between non-redundant events, it cannot lead to false positives being introduced in the results. As such, the elimination of redundant events does not affect the precision of the results.

It is important to stress that SPIDER should be used with traces captured during executions which exercise as much code as possible from the traced program, since SPIDER can only detect a race between two instructions if there are events in the trace pertaining to both instructions. Alternatively, SPIDER can be used with multiple traces to achieve a considerable coverage of the code of the traced program.

5 Related Work

SMT constraint solving has been successfully employed in the past to test and debug concurrent programs. For instance, CLAP [12] uses SMT solving to replay failing interleavings, MCR [10] and Cortex [24] to uncover latent concurrency bugs, and Symbiosis [23,31] to isolate their root cause.

Prior research efforts have also shown that SMT constraint solving can be useful to find races in distributed systems. However, contrary to SPIDER, these solutions assume that the system is either partially synchronous [33] or modeled as BPEL processes [5].

Like SPIDER, DCatch [20] also aims at detecting distributed concurrency bugs based on an HB model. This work abstracts the causality of events into HB rules and builds a graph representing the timing relationships of several distributed concurrency and communication mechanisms. However, DCatch does not attempt to remove redundant portions of the state space, thus incurring unnecessary slowdowns during the analysis of the trace.

Another approach for testing distributed systems is model checking. Model checkers, such as MaceMC [14], Demeter [8], MoDist [37], dBug [29] and SAMC

[17], systematically explore different execution orderings by permuting message arrivals and injecting node crashes and timeouts. Despite being effective in discovering failures, this approach falls short for large distributed systems due to the exponential increase of the state space [17].

The verification of the correctness of distributed systems can also be achieved through formal methods, typically through soundness proofs based on the notion of *inductive invariant* [9,27]. Coq [4] and TLA+ [15] are frameworks that have been used to build formal models of distributed systems and prove their correctness. Verdi [35] is another verification framework based on Coq, that supports automatic transformation of soundness proofs to assume different fault network models. Verification techniques are useful to prove the absence of errors in executions. Alas, they require a thorough formal model of the system, which may be time-consuming to write and significantly longer than the implementation code.

6 Conclusion

In this paper, we propose Spider, a tool that relies on SMT constraint solving to detect data races in execution traces captured during the testing of distributed systems. To reduce the time necessary to solve the constraints and scale to executions with thousands of events, Spider employs a redundancy pruning step aimed at eliminating portions of the trace that are not relevant to the occurrence of new races.

Our experiments with multiple benchmarks show that Spider is capable of discovering different types of distributed data races in a timely fashion and that our redundancy pruning algorithm is effective at reducing the size of the trace with no consequences to the accuracy of our tool.

Acknowledgements. This work is financed by the ERDF - European Regional Development Fund through the North Portugal Regional Operational Programme - NORTE2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project NORTE-01-0145-FEDER-028550-PTDC/EEI-COM/28550/2017.

References

1. Apache Hadoop. http://hadoop.apache.org
2. Apache HBase. http://hbase.apache.org
3. Cook, S.A.: The complexity of theorem-proving procedures. In: STOC 1971. ACM (1971)
4. The Coq proof assistant. https://coq.inria.fr/
5. Elwakil, M., Yang, Z., Wang, L., Chen, Q.: Message race detection for web services by an SMT-based analysis. In: Xie, B., Branke, J., Sadjadi, S.M., Zhang, D., Zhou, X. (eds.) ATC 2010. LNCS, vol. 6407, pp. 182–194. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16576-4_13
6. Feng, Y., Martins, R., Bastani, O., Dillig, I.: Program synthesis using conflict-driven learning. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018. ACM (2018)

7. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: PLDI 2009 (2009)
8. Guo, H., Wu, M., Zhou, L., Hu, G., Yang, J., Zhang, L.: Practical software model checking via dynamic interface reduction. In: SOSP 2011 (2011)
9. Hawblitzel, C., et al.: IronFleet: proving practical distributed systems correct. In: SOSP 2015. ACM (2015)
10. Huang, J.: Stateless model checking concurrent programs with maximal causality reduction. In: PLDI 2015. ACM (2015)
11. Huang, J., Rajagopalan, A.K.: What's the optimal performance of precise dynamic race detection? - a redundancy perspective. In: ECOOP (2017)
12. Huang, J., Zhang, C., Dolby, J.: CLAP: recording local executions to reproduce concurrency failures. In: PLDI 2013. ACM (2013)
13. Kasikci, B., Zamfir, C., Candea, G.: Data races vs. data race bugs: telling the difference with portend. In: ASPLOS 2012. ACM (2012)
14. Killian, C., Anderson, J.W., Jhala, R., Vahdat, A.: Life, death, and the critical transition: finding liveness bugs in systems code. In: NSDI 2007. USENIX Association (2007)
15. Lamport, L.: The TLA+ home page. https://lamport.azurewebsites.net/tla/tla.html. Accessed 10 Oct 2019
16. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
17. Leesatapornwongsa, T., Hao, M., Joshi, P., Lukman, J.F., Gunawi, H.S.: SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In: OSDI 2014. USENIX Association (2014)
18. Leesatapornwongsa, T., Lukman, J.F., Lu, S., Gunawi, H.S.: TaxDC: a taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In: ASPLOS 2016. ACM (2016)
19. Li, G., Lu, S., Musuvathi, M., Nath, S., Padhye, R.: Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In: SOSP 2019. ACM (2019)
20. Liu, H., et al.: DCatch: automatically detecting distributed concurrency bugs in cloud systems. SIGOPS Oper. Syst. Rev. **51**(2), 677–691 (2017)
21. Machado, N., Maia, F., Neves, F., Coelho, F., Pereira, J.: Minha: large-scale distributed systems testing made practical. In: OPODIS 2019. Leibniz International Proceedings in Informatics (LIPIcs) (2019)
22. Machado, N.: TaxDC Micro-benchmarks Repository (2018). https://github.com/jcp19/micro-benchmarks
23. Machado, N., Lucia, B., Rodrigues, L.: Concurrency debugging with differential schedule projections. In: PLDI 2015. ACM (2015)
24. Machado, N., Lucia, B., Rodrigues, L.: Production-guided concurrency debugging. In: PPoPP 2016. ACM (2016)
25. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
26. Neves, F., Machado, N., Pereira, J.: Falcon: a practical log-based analysis tool for distributed systems. In: DSN 2018 (2018)
27. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. SIGPLAN Not. **51**(6), 614–630 (2016). https://doi.org/10.1145/2980983.2908118

28. Popper, N.: The stock market bell rings, computers fail, wall street cringes (2015). https://www.nytimes.com/2015/07/09/business/dealbook/new-york-stock-exchange-suspends-trading.html. Accessed 06 Aug 2019
29. Simsa, J., Bryant, R., Gibson, G.: DBug: Systematic evaluation of distributed systems. In: Proceedings of the 5th International Conference on Systems Software Verification, SSV 2010, p. 3. USENIX Association, Berkeley (2010)
30. SMT-LIB: Logics. http://smtlib.cs.uiowa.edu/logics.shtml
31. Terra-Neves, M., Machado, N., Lynce, I., Manquinho, V.: Concurrency debugging with MaxSMT. In: AAAI 2019. AAAI Press (2019)
32. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region (2011). https://aws.amazon.com/message/65648/. Accessed 03 Mar 2019
33. Tekken Valapil, V., Yingchareonthawornchai, S., Kulkarni, S., Torng, E., Demirbas, M.: Monitoring partially synchronous distributed systems using SMT solvers. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 277–293. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_17
34. Voulgaris, S., Gavidia, D., van Steen, M.: CYCLON: inexpensive membership management for unstructured P2P overlays. J. Netw. Syst. Manag. **13**(2), 197–217 (2005)
35. Wilcox, J.R., et al.: Verdi: A framework for implementing and formally verifying distributed systems. In: PLDI 2015. ACM (2015)
36. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.: Planning for change in a formal verification of the Raft consensus protocol. In: CPP 2016. ACM (2016)
37. Yang, J., et al.: MODIST: transparent model checking of unmodified distributed systems. In: NSDI 2009. USENIX Association (2009)