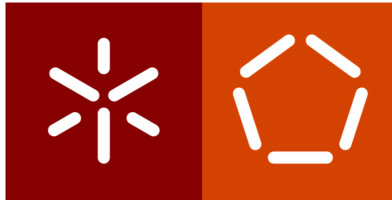


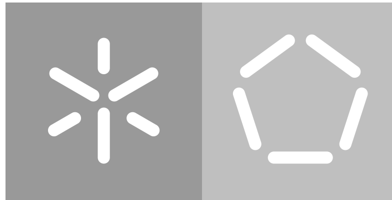
Universidade do Minho
Escola de Engenharia
Departamento de Informática

João Carlos Mendes Pereira

Scalable Trace Analysis of Distributed Systems

Finding data races

April 2020



Universidade do Minho
Escola de Engenharia
Departamento de Informática

João Carlos Mendes Pereira

Scalable Trace Analysis of Distributed Systems

Finding data races

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Jorge Miguel de Matos Sousa Pinto

April 2020

Despacho RT - 31 /2019 - Anexo 3

Declaração a incluir na Tese de Doutoramento (ou equivalente) ou no trabalho de Mestrado

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho

Atribuição-CompartilhaIgual
CC BY-SA

https://creativecommons.org/licenses/by-sa/4.0/

ACKNOWLEDGEMENTS

This dissertation would not be feasible hadn't I had the great support from Jorge Sousa Pinto (my supervisor) and Nuno Machado, who originally had the idea for Spider and co-developed it. I owe a lot to them for our countless meetings and discussions and for being available to guide and help me on matters not only related to my dissertation but also my future steps and life in general. Without their guidance, I would surely be more uncertain of what I want to do with my life. I also want to acknowledge José Orlando Pereira's work on *Minha* which provided the initial motivation and means to develop the work presented in this dissertation.

To my family as a whole and to parents in particular, I want to express my gratitude for always providing me a stable and caring environment that allowed me to flourish both academically and personally without a care in the world. To them, I owe all my past and future successes and none of my failures or shortcomings.

On a more symbolic note, the writing and submission of this dissertation marks the end of a long, unpredictable and tiring, but varied, fulfilling and fantastic journey. I couldn't submit this document without a mention to the endless time I spent in cafés, BP and CESIUM with my friends, having a small chat or purging inner demons; to the ramblings and excursions in all kinds of subjects, from mundane everyday things to hours discussing philosophy, politics and other (pseudo) intellectual activities that I enjoyed with people from all walks of life; to the nights spent roaming in the streets or just chilling in someone's house; to the Professors and lab-mates who not only thaught but also educated me and provided me with unique opportunities; to the hardest of hours spent on the library working hard, the long nights in Departamento de Informática and Salas 24, where I even slept on the floor; and finally, to all the people that I met all over the world throughout this journey that started almost 6 (!) years ago, from those that I met on day 1 of college to those I met on the final months and days. All of them have left a mark on me and most of them have shaped some of my most cherished memories. My only wish is to continue to live a life filled to the brim with new and amazing experiences, the same way I have been doing until now.

Despacho RT - 31 /2019 - Anexo 4

Declaração a incluir na Tese de Doutoramento (ou equivalente) ou no trabalho de Mestrado

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

Distributed Systems and Protocols are widely employed in the infrastructure that supports the Internet and the services available online such as streaming services and social networks. At the same time, they are well known for usually being hard to implement correctly, even when this task is left to experienced programmers. Consequently, Distributed Systems are prone to suffer from distributed concurrency bugs, which are a frequent source of significant service outages. Thus, it is of the utmost importance to ensure that widely-used distributed systems are reliable and do not suffer from this kind of bugs.

Formal Verification looks like a promising way to achieve this. However, we argue that the currently available techniques require too much of an investment in order to verify correctness of *implementations* of complex distributed systems. Instead, we defend the usage of clever testing techniques and tools for all but the most critical of contexts. In this dissertation, we present one such tool – Spider – designed to automatically detect *data races* from traced executions of distributed systems. Data races originate when two memory accesses to the same memory location occur concurrently and they 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.

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.

Keywords: distributed systems, satisfiability modulo theories, software testing, offline monitoring, dynamic race detection

RESUMO

Os sistemas e protocolos distribuídos são amplamente utilizados na infraestrutura que suporta a Internet e os serviços disponíveis online tais como, por exemplo, serviços de streaming e redes sociais. Ao mesmo tempo, os sistemas distribuídos são reconhecidamente difíceis de implementar corretamente e tendem a sofrer de bugs de concorrência distribuída, mesmo quando são desenvolvidos por programadores experientes. Este tipo de bugs é uma causa frequentemente de falhas nos serviços e, por esta razão, é da maior importância garantir que os sistemas distribuídos amplamente utilizados são confiáveis e não sofrem deste tipo de erros.

A área de verificação formal fornece ferramentas poderosas que permitem evitar que estes bugs cheguem a código de produção. No entanto, consideramos que as técnicas do estado da arte disponíveis atualmente exigem grandes investimentos caso se pretenda verificar que uma implementação de um sistema distribuído está correta. Em vez disso, defendemos o uso de técnicas e ferramentas sofisticadas de teste para assegurar a confiabilidade das implementações de sistemas distribuídos excepto nos contextos mais críticos, onde se devem empregar técnicas de verificação formal.

Nesta dissertação, apresentamos a Spider, uma ferramenta de teste e debug de sistemas distribuídos desenvolvida para detectar automaticamente *data races* a partir de *traces* obtidos aquando da execução de sistemas distribuídos. As *data races* surgem quando dois acessos ao mesmo endereço de memória ocorrem concorrentemente. A existência de *data races* é uma das principais causas de erros de concorrência em sistemas distribuídos. Ao mesmo tempo, são extremamente dificeis de detetar uma vez que se manifestam raramente e de forma não determínistica.

A ferramenta Spider codifica as relações causais entre os eventos no *trace* como um modelo de restrições, e, através de um *SMT solver*, é capaz de inferir que pares de instruções podem levar a acessos concorrentes ao mesmo endereço de memória. Para reduzir o tempo da análise, o Spider emprega uma técnica de eliminação de eventos que remove partes redundantes do *trace*. Com recurso a vários benchmarks, mostramos experimentalmente que o Spider é eficaz a detetar *data races* a partir de traces de sistemas distribuídos e que tal é exequível em tempo útil, indiciando que a Spider é útil como ferramenta de teste.

Palavras-chave: sistemas distribuídos, satisfiability modulo theories, teste de software, monitorização offline, deteção dinâmica de races

CONTENTS

1 INTRODUCTION 1
1.1 Context 1
1.2 Motivation 3
1.3 Goals and Contributions 4
1.4 Document Structure 5
2 BACKGROUND 6
2.1 Distributed Systems 6
2.2 Data Races in Distributed Systems 8
2.3 SAT and SMT 9
2.4 Runtime Verification 12
3 INTRODUCING SPIDER 14
3.1 Tracing a running distributed system 15
3.2 Building the causal relation between the traced events 15
3.3 Detecting concurrent accesses using an SMT solver 18
3.3.1 Handling inter-thread data races. 18
3.3.2 Handling intra-thread data races. 18
3.4 A running example 19
4 CHALLENGE: SCALING SPIDER'S APPROACH 25
5 EVALUATION 32
5.1 Benchmarks 32
5.1.1 TaxDC Micro-Benchmarks 32
5.1.2 Peer Sampling Service. 35
5.2 Effectiveness 36
5.3 Efficiency 36
5.4 Soundness and precision of the approach 38
6 RELATED WORK 39
6.1 Formal Verification 39
6.1.1 An overview of formal verification of computer programs 39
6.1.2 Formal verification of distributed systems 42
6.2 Other approaches 46
7 CONCLUSION AND FUTURE WORK 47

A SPIDER'S REPOSITORY ORGANIZATION 56

LIST OF FIGURES

Figure 1 Output of minisat after running with Formula 1 10
Figure 2 Output of minisat after running with Formula 2 10
Figure 3 Output of Z3 after running with Formula 3 12
Figure 4 Output of Z3 after running with Formula 4 12
Figure 5 Spider's intended workflow 14
Figure 6 Example1 source code 20
Figure 7 Possible output of Example1 (1) 20
Figure 8 Possible output of Example1 (2) 20
Figure 9 File example1.log - a traced execution of Example1 21
Figure 10 Abstract representation of the Happens-Before relation built from the events of example1.log 22
Figure 11 Invoking Spider with the file example1.log 23
Figure 12 Spider's workflow including redundant event pruning 26
Figure 13 Overview of the TaxDC micro-benchmarks with distributed data races 34
Figure 14 Class Diagram of Spider 57

LIST OF TABLES

Table 1 Data races in Example1 24
Table 2 Race detection results without redundancy pruning 37
Table 3 Race detection results with redundancy pruning 38
Table 4 Usage of *TLA+* and *PlusCal* to verify parts of some of the most complex systems from *Amazon* as presented by Newcombe et al. (2013) 43

ACRONYMS

AWS Amazon Web Services. 1

BFT Byzantine Fault Tolerance. 1, 46
BMC Bounded Model Checking. 1, 40, 41

CDCL Conflict Driven Clause Learning. 1, 11
CTI Counterexample to Induction. 1, 45

DC bugs distributed concurrency bugs. 1, 2, 42
DS Distributed Systems. 1, 2
DSL Domain Specific Language. 1, 13

HB Happens-Before. 1, 14

PTDTL Past Time Distributed Temporal Logic. 1, 13

RML Relational Modeling Language. 1, 45
RPC Remote Procedure Call. 1, 3

SAMC Semantic Aware Model Checking. 1
SAT Satisfiability. 1, 11, 40
SMC Software Model Checking. 1, 2, 42, 44
SMT Satisfiability Modulo Theories. 1, 6, 11, 25, 26, 32, 46, 47

TLA+ Temporal Logic of Actions+. 1, 42

VST Verified System Transformer. 1, 44

1

INTRODUCTION

1.1 CONTEXT

Distributed Systems (DS) are ubiquitously used in critical contexts such as electronic voting systems [Riemann and Grumbach (2017)], cryptographic coins [Nakamoto (2008)] and avionic systems [Pike et al. (2013)]. Large-scale distributed systems play a major role in the design of the infrastructure that supports the Internet and the services available online. They are employed in the deployment of scalable computing frameworks, storage systems, synchronization and cluster management services as reported by Leesatapornwongsa et al. (2016). As a result, bugs in the design and implementation of distributed systems can significatly hamper the availability and the quality of services which are used everyday by millions of people and are considered indispensable to modern life. This statement is corroborated by episodes of service outtages seen before, like the failure of Amazon Web Services (AWS) [The AWS Team] in April 2011 that caused availability issues in services like *Reddit* and *Netflix*, and the independent failures in 2015 which grounded *United Airlines* flights for two hours and halted the *New York Stock Exchange* for four hours [Popper (2015)].

Distributed systems are notoriously hard to get right, even when developed by teams of experienced programmers. Besides the typical bugs associated with non-concurrent software and those that are characteristic of single-machine multi-threaded software (caused by unsychronized memory accesses), these systems tend to suffer from distributed concurrency bugs (DC bugs) caused by unexpected orderings of distributed events like message arrivals, crashes and timeouts. Leesatapornwongsa et al. (2016) analysed various DC bugs from "four widely-deployed cloud-scale datacenter distributed systems[1]" and grouped them in *TaxDC*, a taxonomy based on their triggering conditions, some of them quite common and complex. This study also showed that, among the different types of distributed system bugs, *data races* (§2.2) are particularly challenging to find and debug, as they occur non-deterministically and manifest rarely. A data race consists of two concurrent accesses to the same memory location, where at least one of them is a Write. Such races in distributed systems typically

1 *Cassandra* [Apache Cassandra], *Hadoop MapReduce* [Apache Hadoop], *HBase* [Apache HBase], and *Zookeeper* [Apache ZooKeeper]

stem from unpredictable message exchanges that violate atomicity and order assumptions [Leesatapornwongsa et al. (2016), Liu et al. (2017)].

The difficulty associated with finding and fixing data races suggests that it is not enough to reason informally in order to build reliable distributed systems, free of DC bugs. Indeed, as Lamport et al. (1982) stated when talking about distributed algorithms,

> *"we strongly advise the reader to be very suspicious of such non-rigorous reasoning. Although this result is indeed correct, we have seen equally plausible "proofs" of invalid results. We know of no area in computer science or mathematics in which informal reasoning is more likely to lead to errors than in the study of this type of algorithm."*

Consequently, there have been efforts to apply *Formal Software Verification* techniques to distributed systems. Formal software verification consists in the use of rigorous mathematical reasoning to prove that a computer program has the intended properties by showing that it conforms to a given high-level specification. It compreends automated techniques such as *Software Model Checking (SMC)* [Jhala and Majumdar (2009)] as well as manual techniques based on axiomatic semantics and logics for reasoning about programs deductively, e.g. *Hoare Logic* [Hoare (1969)], possibly aided by *proof-assistants* like *Coq* and *Agda*.

The aforementioned efforts have culminated in promising results. *Model Checking* has been successfully used for years in academia and industry to verify high-level designs of distributed systems. For example, Zave (2012) demonstrated that no version of the *Chord* protocol [Stoica et al. (2003)] published before was correct, and Newcombe et al. (2013) described how Model Checking is effectively used at Amazon to detect subtle flaws in the design of their systems. More recently, the focus of researchers has shifted somewhat to the development and verification of the actual implementations of such systems. For example, Woos et al. (2016) provided a verified implementation of *Raft* [Ongaro and Ousterhout (2014)], a protocol conceived to mantain a consistent view of a state machine across different computers and which supports machine failures. Rahli et al. (2018) presented "the first machine-checked proof of a crucial safety property" of an implementation of *PBFT* [Castro and Loskov (1999)], a protocol that provides *bizantine fault-tolerant state machine replication*. Section 6.1 offers a more thorough overview of formal software verification of distributed systems.

In spite of these efforts, there are reasons to believe that formaly verified implementations of distributed systems will not become mainstream in the next few years. On one hand, despite its usefulness in verifying high-level designs, automatic techniques like SMC are still far from being tractable for verifying implementations of DS [Leesatapornwongsa et al. (2014)]. The sheer ammount of possible event interleavings, e.g. memory accesses and message exchanges, renders this method unusable due to state-space explosion, i.e. a combinatorial explosion on the number of possible states of the system. On the other hand, deductive methods require a significant investment. For example, the verified implementation of Raft

presented by Woos et al. (2016) consists of 50.000 lines of proof scripts. It assumed "several nontrivial invariants of the Raft protocol" and was written using Coq, a language and toolset appropriate to write verified programs from the ground up.

1.2 MOTIVATION

Despite the setbacks mentioned in the previous section, we believe that formal verification should be employed in the development of critical distributed systems and we look forward to further developments in the field. At the same time, we think that there is room for automated tools with a strong theoretical basis for *testing* and *debugging* existing distributed programs, making it easier to detect complex bugs before they enter production code. Such tools cannot offer the same correctness guarantees as actually verifying the programs against a formal specification, but they can be automated and integrated in the development and test stages, which makes them much easier to apply in practice. In fact, over the last years, there have been multiple efforts to test and debug data races, although prior work has mostly focused on multithreaded programs [Flanagan and Freund (2009); Kasikci et al. (2012); Huang (2015)].

More recently, Liu et al. (2017) proposed *DCatch*, a tool that discovers distributed concurrency bugs 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, i.e. executions where the bugs were not triggered. In order to decrease the size of the trace and improve the performance of the analysis, DCatch filters events before tracing them. This filtering focuses mainly on memory accesses; DCatch only traces accesses to heap objects and static variables in particular types of functions:

1. Remote Procedure Call (RPC) functions;
2. functions that conduct socket operations;
3. event-handler functions;

All other memory accesses are not traced and thus, are not used for data race detection. Despite that, DCatch's approach scales poorly, as the experimental results in the paper revealed that it consumes GBs of memory for processing traces with a few MBs.

In this dissertation, 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 the traces can improve the performance and scalability of distributed system testing solutions without compromising their accuracy.

1.3 GOALS AND CONTRIBUTIONS

In this dissertation, we propose Spider (§3), an automated tool to detect data races from traced executions of distributed systems using redundancy pruning and symbolic constraint solving. Given a trace of a distributed system under test (whose format is specified in §3.1), 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 [Huang and Rajagopalan (2017)] and extend it to remove even more events from the traces.

After trimming the trace, Spider builds a causality model by encoding the HB relations between events into a system of constraints over logical order variables. Finally, Spider resorts to an off-the-shelf *SMT solver* (§2.3) to compute which pairs of conflicting events can run concurrently and, thus, form a data race. SMT constraint solving has been successfully used in prior work to reproduce [Huang et al. (2013)], expose [Machado et al. (2016)], and isolate [Machado et al. (2015); Terra-Neves et al. (2019)] concurrency bugs in multithreaded programs. However, to the best of our knowledge, this is the first application of SMT solvers to detect race conditions in distributed systems.

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 ammount of events in our experiments (§5.3). In summary, this paper makes the following contributions:

1. we present an algorithm, which draws on prior work by Huang and Rajagopalan (2017), to eliminate redundant events from distributed system traces without hampering race detection accuracy;

2. we propose Spider, a tool that leverages redundancy pruning and SMT constraint solving for finding data races from traces of distributed systems;

3. 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.

1.4 DOCUMENT STRUCTURE

The rest of this document is structured as follows:

- Chapter 2 includes an overview of the foundational topics which underly the work presented in this document.
- Chapter 3 contains an extensive presentation of SPIDER. It concludes with a brief practical demonstration of the tool.
- Chapter 4 discusses the problem of scalling our approach and describes at length how we have dealt with traces containing a big number of events.
- Chapter 5 briefly mentions the implementation of SPIDER and presents the benchmarks used to attest the effectiveness and efficiency of our approach, as well as the results of said benchmarks.
- Chapter 6 summarizes other research areas with similar goals to ours.
- Chapter 7 concludes the dissertation and provides a critical analysis of the developed work. Besides, it discusses the prospects for future work, identifying possible paths to improve the work presented in this dissertation.

2

BACKGROUND

The work developed throughout this dissertation is built upon elementary concepts such as distributed systems and Satisfiability Modulo Theories (SMT). This section contains a brief and tentative explanation of the topics that underly our work. It is, by no means, a complete treatment of these topics. Instead, it is meant as introductory material for the readers not familiar with the literature on the presented subjects, together with references to relevant publications in each field.

2.1 DISTRIBUTED SYSTEMS

A distributed system consists of multiple software components, possibly running on different computers, which operate as a single system. They are often characterized by the lack of a global clock and by the fact that their components run concurrently and can fail independently of other components. For the purposes of this dissertation, we will model a distributed system as a set of *nodes* (or processes), with at least one *thread* running in each node. Different threads communicate by exchanging messages, with no assumptions on message losses and network delays. We also assume that two threads running in the same node can share memory by using *shared-variables*, defined as variables that are accessible in both threads. The accesses to shared-variables can be synchronized with the usual synchronization primitives such as *locks* and *monitors*. Each thread can be viewed as a sequence of *events*. An event is an atomic transition of the local state of a thread. Examples of events include *message sending*, *message receiving*, and *shared-variable access* for either reading or writing the value of the variable. The *location of an event* e is as a pair (n, t) meaning that event e occurs at the thread t of node n.

Distributed systems often implement one or more *distributed protocols*[1]. The *distinction between a distributed protocol and one implementation of such protocol* is indeed important: while a protocol describes at a high-level the actions that each node must perform in order to solve a distributed problem, an implementation of a protocol corresponds to the code that

1 Even though they have slightly different semantics, the terms *protocol* and *algorithm* are used interchangeably throughout this document, a common practice in the literature.

each component runs that conforms to the high-level description provided by the protocol. As such, bugs may arise from a poorly designed protocol or from a wrong *implementation* of a distributed protocol. This separation of concerns allows for a modular approach to building correct distributed systems – the programmers of the system should not worry about whether the protocol obeys its specification. Similarly, the designer of a protocol does not need to be concerned with low-level implementation-specific details.

As hinted before, distributed protocols are designed to tackle *distributed problems*. These problems are often stated for a particular *fault model*. A fault model is a description of which parts of the system, including processes and network, may fail and how they fail. Schneider (1993) identified some common fault models appearing in distributed systems literature such as:

- *Failstop* - Any process may halt and if so, remain in that state; other components can detect the failure.
- *Crash* - Any process may halt and if so, remain in that state; unlike Failstop, other components might not be able to detect the failure.
- *Omission* - Any process fails by not receiving a message sent to it, or by not succeeding to send some message or by halting.
- *Byzantine Failure* - A process fails when it displays a behaviour not specified by the algorithm. This is the most general kind of failure [Lamport et al. (1982)].

Each fault model listed above subsumes all those that appeared before. The fault model greatly influences wether a problem can be solved by a distributed algorithm and, if so, how complex the algorithm is. For example, Schneider (1993) states the following problem:

> *"Two processes, A and B, communicate by sending and receiving messages on a bidirectional channel. Neither process can fail. However, the channel can experience transient failures, resulting in the loss of a subset of the messages that have been sent. Devise a protocol where either of two actions α and β are possible, but (i) both processes take the same action and (ii) neither takes both actions."*

This problem is called the *Two Generals' Problem* and it has been proven that no distributed algorithm can solve it given the specified fault model. It is a particular case of a *consensus problem* which consists in choosing a single value from a set of values proposed by a collection of processes. Generally, a consensus algorithm is designed to have the following properties:

- *Agreement* – All correct processes choose the same value;
- *Validity* – A process can only choose a value that has been previously proposed;
- *Termination* – Each correct process eventually decides on a value.

Algorithms designed to reach consensus such as Paxos and Raft are complex. It is not easy to assert that these algorithms and its implementations have the stated properties. Consequentely, efforts to formally verify consensus algorithms often tackle the verification of variations of the correctness properties listed above.

2.2 DATA RACES IN DISTRIBUTED SYSTEMS

In general, a data race occurs when two accesses compete for the same resource in a unsynchronized fashion and at least one is modifying the resource. Since there is no causal relation 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 [Flanagan and Freund (2009); Kasikci et al. (2012); Huang (2015); Li et al. (2019)]. Unfortunately, data races in distributed systems are much more challenging than their single-machine counterparts. 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 [Liu et al. (2017)]. 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 [Leesatapornwongsa et al. (2016)], 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 data races between:

  1. two message arrivals,
  2. a message arrival and a message sending,
  3. 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 13 illustrates several of the aforementioned scenarios of order and atomicity violations, which we implemented as testbeds for our experimental evaluation (§5). We now discuss SMT constraint solving, which is at the heart of our approach to detect distributed data races.

2.3 SAT AND SMT

Satisfiability (SAT)

The *propositional satisfiability problem*, also known as *SATISFIABILITY* or *SAT*, is the problem of determining if, given a propositional formula ϕ, there exists an assignment that satisfies ϕ. To put it another way, SAT consists in deciding if it is possible to assign a boolean value to each variable in a propositional formula such that the formula is satisfied, i.e. it is *true*. In that case, the formula is considered *satisfiable*. Otherwise, it is considered *unsatisfiable*. For example, formula 1 is satisfiable – assigning *true* to a and b and $false$ to c makes the formula have a truth-value of *true* – whereas formula 2 is unsatisfiable – no assignment can make a and $\neg a$ be *true* at the same time.

$$a \wedge b \wedge \neg c \tag{1}$$

$$a \wedge \neg a \tag{2}$$

Programs that take a set of logical formulas as input and determine wether the set is satisfiable[2] are called *SAT Solvers*. Most modern SAT solvers such as *Minisat* and *Glucose* receive its input in the *DIMACS* format[3] and output $SATISFIABLE$ if the set is satisfiable or $UNSATISFIABLE$ otherwise. Fig. 1 and Fig. 2 demonstrate the output of minisat when it is passed a file with the Formula 1 and a file with the Formula 2, respectively.

SAT was the first problem to be proven *NP-complete* [Cook (1971)]. As such, there is no known algorithm capable of efficiently solving all instances of the SAT problem. In other words, generally speaking, there is no known *polynomial-time algorithm* which can decide if a propositional formula is satisfiable. Notwithstanding SAT's complexity, current SAT solvers

2 By abuse of notation, we consider that a set of propositional formulas S is satisfiable if there exists an assignment which satisfies all the formulas in S.

3 A presentation of the DIMACS format is not required and it is out of the scope of this dissertation.

```
$ minisat formula1.sat
============================[ Problem Statistics ]=============================
|                                                                             |
|   Number of variables:            3                                         |
|   Number of clauses:              0                                         |
|   Parse time:                  0.00 s                                       |
|   Simplification time:         0.00 s                                       |
|                                                                             |
============================[ Search Statistics ]==============================
| Conflicts |          ORIGINAL          |          LEARNT          | Progress |
|           |    Vars  Clauses Literals  |    Limit  Clauses Lit/Cl |          |
===============================================================================
===============================================================================
restarts              : 1
conflicts             : 0              (0 /sec)
decisions             : 1              (0.00 \% random) (221 /sec)
propagations          : 3              (664 /sec)
conflict literals     : 0              (-nan \% deleted)
Memory used           : 14.00 MB
CPU time              : 0.004515 s

SATISFIABLE
```

Figure 1: **Output of** minisat **after running with Formula 1** – the formula is satisfiable.

```
$ minisat formula2.sat
============================[ Problem Statistics ]=============================
|                                                                             |
|   Number of variables:            1                                         |
|   Number of clauses:              0                                         |
|   Parse time:                  0.00 s                                       |
|   Simplification time:         0.00 s                                       |
|                                                                             |
===============================================================================
Solved by simplification
restarts              : 0
conflicts             : 0              (0 /sec)
decisions             : 0              (-nan \% random) (0 /sec)
propagations          : 1              (667 /sec)
conflict literals     : 0              (-nan \% deleted)
Memory used           : 14.00 MB
CPU time              : 0.0015 s

UNSATISFIABLE
```

Figure 2: **Output of** minisat **after running with Formula 2** – the formula is unsatisfiable.

achieve acceptable performance for many practical applications thanks to algorithms such as *Conflict Driven Clause Learning (CDCL)* [Silva and Sakallah (1996)]. SAT solvers have a wide range of practical applications which include, but are not limited to, model checking and planning in artificial intelligence [Marques-Silva (2008)].

Satisfiability Modulo Theories (SMT)

SMT is the decision problem of determining wether 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 [SMT-Lib: Logics].

An SMT instance can be seen as a generalization of a Satisfiability (SAT) instance where, in the place of propositional variables, there are predicates over non-binary variables (i.e. binary-valued functions of non-binary variables) whose interpretations are given by a background theory.

In the context of the already mentioned theory T_Z (also called *Presburger arithmetic*), formula 3 is satisfiable – the assignment $x = 1$, $y = -1$ and $z = 1$ satisfies the formula – but formula 4 is not.

$$x + y \leq z \wedge z \leq x - y \tag{3}$$

$$x \leq 0 \wedge 1 \leq x \tag{4}$$

T_Z is a *decidable* theory – there is a decision procedure capable of determining the satisfiability of formulas written in this theory [Stansifer (1984)]. That is generally not the case. For example, the theory $T_Z^{\times}$, also known as Peano arithmetic, consists in the theory T_Z expanded with the operator $\times$ and is undecidable.

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*. Most modern SMT solvers like *Z3* [Z3 Github Page] support the *SMT-LIB2* format[4] [Barrett et al. (2010)] for representing sets of formulas and print *sat* or *unsat* depending on the satisfiability of the input. Figures 3 and 4 exemplify the usage of an SMT solver when it is passed a file with the formulas 3 and 4, respectively. SMT solvers have been employed in a wide range of applications, from program synthesis [Feng et al. (2018)] to testing, as seen on this dissertation.

4 A presentation of the SMT-LIB2 format is not required and it is out of the scope of this dissertation.

```
$ z3 --smt2 formula3.smt
sat
(model
  (define-fun z () Int
    0)
  (define-fun y () Int
    0)
  (define-fun x () Int
    0)
)
```

Figure 3: **Output of** Z3 **after running with Formula 3** – the formula is satisfiable. Besides, Z3 provides an assignment which satisfies the formula: all variables are assigned to 0.

```
$ z3 --smt2 formula4.smt
unsat
```

Figure 4: **Output of** Z3 **after running with Formula 4** – the formula is unsatisfiable.

2.4 Runtime Verification

The approaches to verification presented previously are performed either before or during compilation. Because of this, they are considered static verification techniques. There are other approaches which rely on runtime checks to detect property violations. Accordingly, such techniques commonly receive the label of *Runtime Verification* even though they do not offer the same guarantees as traditional (static) verification techniques – generally, a program cannot be proved correct using runtime verification techniques alone because the information used by such techniques is often limited to particular executions of the system; for example, runtime verification does not provide a way to verify the code that is not exercised in the particular execution under consideration.

Runtime techniques such as *monitoring* [Leucker and Schallhart (2009)] are often employed to reason about a single execution of a computer program. It is also possible to exploit information about one execution of the program to reason about multiple possible executions. Section 3 describes one way to achieve this.

Monitoring is a method where a component called *monitor* runs along the evaluated system. The monitor is responsible for analysing the behaviour of the program in order to reach a *verdict*, *i.e.*, a truth-value, indicating whether a property holds in the program or not. Monitoring can be performed in different ways – Cassar et al. (2017) present a spectrum of monitoring techniques regarding the way monitors are coupled with the monitored system and the level of control they have over it. If a monitor uses a set of logged executions to reach a verdict, then it is called an *offline monitor*. Otherwise, the monitor checks the current execution of the system and is considered an *online monitor*. Online monitors can

be used to detect violations of properties and react to them accordingly, e.g. by calling an error-handling function in the monitored system.

There are monitoring tools which permit the encoding of the properties to be monitored in a Domain Specific Language (DSL). The properties are automatically translated into the monitor's code to be run alongside the monitored system. For instance, *Copilot* [Pike et al. (2013)] provides a stream-based DSL embedded in *Haskell* to specify properties to be monitored in embedded systems. Properties are encoded as streams of booleans. Each value in the stream corresponds to the truth-value of the property in each sampled point. The specification is utilized to generate the monitor's code written in *C* to be cross-compiled with the monitored system.

Depending on the implementation, a monitor is capable of reading either a finite log of the running system or its internal state. Besides, work has been carried out to build monitors to reason locally (i.e., in each node of the system) about the global state of a distributed system. For example, Sen et al. (2004) proposed Past Time Distributed Temporal Logic (PTDTL), a logic to specify properties in distributed systems using epistemic operators to reason about what each process knows about other processes. Simply put, a process is associated with a set of temporal formulas about its local state, which includes the last known states of the other processes in the system. Each process is run with a local monitor generated from the set of formulas stated for that process.

3

INTRODUCING SPIDER

In the work leading up to this dissertation, we developed SPIDER, a tool for detecting data races in executions of distributed systems by analysing *traces* obtained at runtime using a tool such as *Minha* [Machado et al. (2019)] or Falcon [Neves et al. (2018)]. A trace consists in a log that has entries for each relevant event occurring during the system's execution. For this reason, SPIDER can be considered an offline monitoring tool (§2.4). The idea for SPIDER came from the the need to analyse traces produced by Minha (in fact, it was originally called *Minha-checker*) but it grew into an independent project capable of handling traces captured from any system, as long as the traces conform to the specified trace format (§3.1).

SPIDER explores the idea presented by Lamport (1978) that, in distributed systems, it is often impossible to tell which one of two events (e.g. message sending, message receving, memory access) happened first - even if the events have timestamps associated with them, there's no guarantee that the clocks on the nodes where the events ocurred are synchronized. Hence, it may be impossible to establish a unique total-order on the events ocurring in a distributed system. However, it is always possible to establish a partial-order on the events based on which events cause or affect other events. In other words, it is possible to build a *causality relation*, also known as a Happens-Before (HB) relation which is a partial ordering. Given such an ordering on the events pertaining to one execution of a system, it is possible to determine wether two events occur concurrently. In particular, it is possible to infer that two accesses to the same memory location are concurrent, which often translates into a race condition.

Figure 5 summarizes the worflow intended for SPIDER. Each of the steps is further described below.

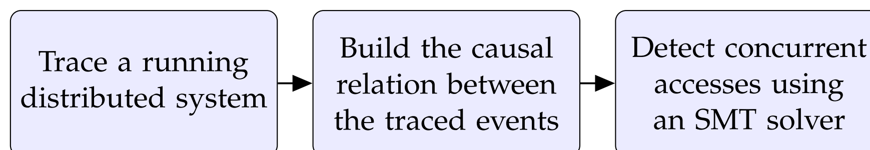


Figure 5: **Spider's intended workflow**

3.1 TRACING A RUNNING DISTRIBUTED SYSTEM

SPIDER relies on external tools such as Minha or Falcon to capture a trace of a distributed system's execution. Such tools must be able to detect and log the following kinds of events when a distributed system is running:

1. **Intra-node shared memory accesses**: accesses to shared variables including their types (Read or Write), as well as Lock Acquire and Lock Release events;
2. **Intra-node thread events**: Fork, Join, Thread Start and Thread End events representing, respectively, the creation of a new thread in a node, the termination of a thread, the start of a thread execution and the end of a thread execution;
3. **Inter-node communication events**: Send and Receive events representing message sending and receiving through sockets;
4. **Message handling region delimiters**: events signaling the beggining and the end of a message handler.

The tracing mechanism is not required to detect custom synchronization protocols based on combinations of RPC/socket communication and intra-node computation [Liu et al. (2017)]. If the traced system relies on such mechanisms, the results returned by SPIDER will not be precise (§5.4).

The tracing mechanism should produce either a file with all the traced events or else, a file per node with the events relative to each node. In both cases, the file has to be *locally sorted*. A trace file is locally sorted when, given two events e_1 and e_2 in the trace, if they occur in the same thread and e_2 appears in the trace after e_1, then e_2 does not causally precede e_1. SPIDER currently uses the *Falcon Event Trace API* [Falcon TAZ] to parse trace files.

3.2 BUILDING THE CAUSAL RELATION BETWEEN THE TRACED EVENTS

SPIDER is able to model multiple distributed execution orderings from the same event trace by leveraging the *Happens-Before (HB)* $\prec_{hb}$ (binary) relation between events. This relation states that, for two events e_1 and e_2 in the trace, if $e_1 \prec_{hb} e_2$ then event e_1 must have run before event e_2 at runtime [Lamport (1978)]. The $\prec_{hb}$ relation encodes causal dependencies between events in a *strict partial order*, which means that it has the following properties:

1. *irreflexivity* – no event can happen before itself:

$$\forall a \,.\, \neg\, a \prec_{hb} a \tag{5}$$

2. *transitivity* – if an event a happens before an event b and b happens before another event c, then a happens before c:

$$\forall a, b, c \,.\, a \prec_{hb} b \wedge b \prec_{hb} c \;\rightarrow\; a \prec_{hb} c \tag{6}$$

3. *asymmetry* – no event a can simultaneously happen before and after another event b:

$$\forall a, b \,.\, a \prec_{hb} b \rightarrow \; \neg b \prec_{hb} a \tag{7}$$

The HB relation is commonly captured by means of *logical clocks* (also known as *Lamport clocks*) [Lamport (1978)], 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 chapter 4, we discuss this issue at length and describe how Spider employs redundancy pruning to achieve this goal.

Spider builds the HB model, denoted Φ_{hb}, in two steps:

1. encode each event's logical clock as a symbolic integer variable,

2. encode the 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 the following sub-formulae:

1. **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 they occur in the same thread and 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: $E_1 < E_2$.

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:

$$Fork_t < Start_t \tag{8}$$

$$End_t < Join_t \tag{9}$$

$$Start_t < End_t \tag{10}$$

3. **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:

$$Snd_{m,l_1} < Rcv_{m,l_2} \tag{11}$$

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

4. **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:

$$Rcv_{m,l} = H_Begin_m \tag{12}$$

$$H_Begin_m < H_End_m \tag{13}$$

   Assuming that the handler is the region of the program responsible for processing the message, the first constraint means that a handler for message m begins immediately after m has been received. Thus, a message m cannot be processed before it was received. The second constraint ensures that the event signaling the beginning of a handler occurs before the event signaling its end.

5. **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:

$$Lock_{t,v,l_1} < Unlock_{t,v,l_2} \tag{14}$$

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:

$$\forall_{(L,U),(L',U') \in P} : \; U < L' \;\vee\; U' < L \tag{15}$$

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.

3.3 DETECTING CONCURRENT ACCESSES USING AN SMT SOLVER

The last step of SPIDER's approach consists in using an SMT solver to identify race conditions. More precisely, we use an SMT solver to detect two different kinds of data races:

1. **inter-thread data races** resulting from concurrent accesses to shared memory originating from two different threads. These races are characteristic of multi-threaded software.
2. **intra-thread data races** resulting from racing messages whose handlers modify the same portions of a node's state. These races are unique to distributed systems.

3.3.1 *Handling inter-thread data races.*

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 the following *race* property is satisfiable:

$$race(e_1, e_2) \equiv \Phi_{hb} \wedge (E_1 = E_2) \tag{16}$$

The data race property Φ_{race} requires 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 are no related via the happens-before relation.

SPIDER resorts to an SMT solver to check whether equation 16 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.

3.3.2 *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 Figure 13b for an example).

Spider addresses these types of data races in a two-fold fashion. First, it identifies message races in each thread. This is done by applying equation 16 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 the sets of shared variables that each handler accesses. 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 of them be a *Write*. If m_1 and m_2 are racing, then (rw_1, rw_2) forms an intra-thread data race.

3.4 A running example

In this section, it is shown how to use Spider in practice with a simple but elucidating example. Figure 6 presents Example1, a Java program[1] that suffers from two distinct data races. When the program starts, it launches a child thread. Then, the main thread writes on the shared variable counter by incrementing its value (line 7) before trying to read the value in order to print it (line 8). At the same time, the child thread writes on the shared variable counter by also incrementing it (line 12). Given that no synchronization mechanism is employed, the increment operation in the child thread is concurrent with both the increment and read operations in the main thread. Because of these data races, there are two possible outputs that Example1 may produce, shown in fig. 7 and fig. 8. The increment and read operations on the main thread are not concurrent because they happen in the same thread and the increment operation has to happen first.

1 Even though Example1 is written in Java, Spider could, in principle, be utilized to detect data races in programs written in other programming languages, as long as there is an effective way to trace their executions.

```
class Example1 implements Runnable {
    static int counter = 0;

    public static void main(String[] args) {
        Thread t1 = new Thread(new Ex1());
        t1.start();
        counter++;
        System.out.println("The value of counter is " + counter);
    }

    public void run() {
        counter++;
    }
}
```

Figure 6: **Example1 source code** (written in *Java*)

```
The value of counter is 1
```

Figure 7: **Possible output of Example1 (1)**

```
The value of counter is 2
```

Figure 8: **Possible output of Example1 (2)**

```
{
    "thread": "main@10.0.0.1",
    "type": "START",
    "timestamp": 1525270020050
}{
    "thread": "main@10.0.0.1",
    "type": "FORK",
    "child": "Thread-1@10.0.0.1",
    "timestamp": 1525270020069
}{
    "thread": "Thread-1@10.0.0.1",
    "type": "START",
    "timestamp": 1525270021812
}{
    "thread": "main@10.0.0.1",
    "loc": "demos.Example1.main.7",
    "variable": "demos.Example1.counter",
    "type": "WRITE",
    "timestamp": 1525270021837
}{
    "thread": "main@10.0.0.1",
    "loc": "demos.Example1.main.8",
    "variable": "demos.Example1.counter",
    "type": "READ",
    "timestamp": 1525270021852
}{
    "thread": "Thread-1@10.0.0.1",
    "loc": "demos.Example1.run.12",
    "variable": "demos.Example1.counter",
    "type": "WRITE",
    "timestamp": 1525270021875
}{
    "thread": "Thread-1@10.0.0.1",
    "type": "END",
    "timestamp": 1525270022188
}{
    "thread": "main@10.0.0.1",
    "type": "END",
    "timestamp": 1525270022200
}
```

Figure 9: **File example1.log** - a traced execution of *Example1*

To detect the data races using Spider, the program must be traced while it runs using tools such as Minha or Falcon. Figure 9 shows the contents of file example1.log, containing a traced execution of the program in the format specified by the Falcon Event Trace API.

Each entry contains the relevant data pertraining to one event, including its type and the thread where it occurs. Spider can then be invoked with the collected trace in order to find the data races in Example1. Internally, it will build a causality relation, shown as a directed graph in fig. 10, and then it will use it to determine which conflicting memory addresses are concurrent.

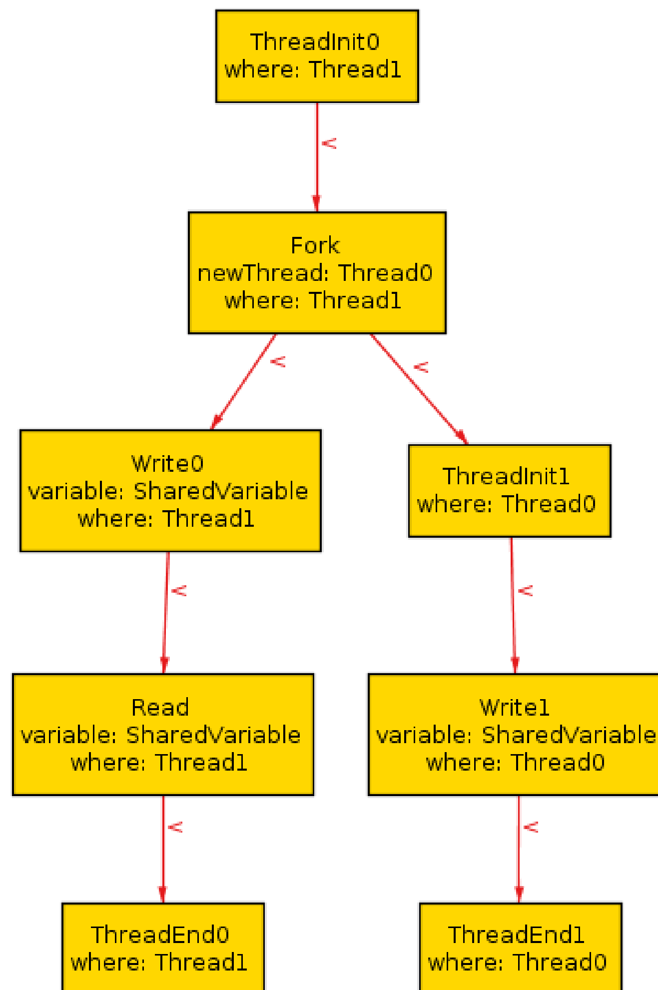


Figure 10: **Abstract representation of the Happens-Before relation built from the events of** example1.log - this figure was obtained from an Alloy specification of the possible traces. As such, it uses the generic terms instead of concrete ones: it uses SharedVariable instead of demos.Example1.counter, Thread1 instead of main@10.0.0.1 and Thread0 instead of Thread-1@10.0.0.1.

```
$ spider -f example1.log
[main] Loading events from example1.log
[main] Trace successfully loaded!
[RaceDetector] Generate program order constraints
[RaceDetector] Generate fork-start constraints
[RaceDetector] Generate join-end constraints
[RaceDetector] Generate wait-notify constraints
[RaceDetector] Generate locking constraints
[RaceDetector] Generate communication constraints
[RaceDetector] Generate message handling constraints

Data Race Candidates:
-- (W_demos.Example1.counter_main@10.0.0.1_3@demos.Example1.main.7,
    W_demos.Example1.counter_Thread-1@10.0.0.1_5@demos.Example1.run.12)
-- (R_demos.Example1.counter_main@10.0.0.1_4@demos.Example1.main.8,
    W_demos.Example1.counter_Thread-1@10.0.0.1_5@demos.Example1.run.12)

Actual Data Races:
-- (W_demos.Example1.counter_main@10.0.0.1_3@demos.Example1.main.7,
    W_demos.Example1.counter_Thread-1@10.0.0.1_5@demos.Example1.run.12)
-- (R_demos.Example1.counter_main@10.0.0.1_4@demos.Example1.main.8,
    W_demos.Example1.counter_Thread-1@10.0.0.1_5@demos.Example1.run.12)

=======================
        RESULTS
=======================
> Number of events in trace:          8
> Number of constraints in model:     6
> Time to generate constraint model:  0.001 seconds

## DATA RACES:
> Number of data race candidates:     2
> Number of actual data races:        2
> Time to check all candidates:       0.007 seconds
```

Figure 11: **Invoking Spider with the file example1.log**

Fig. 11 shows the output presented by SPIDER when it runs with file example1.log. The output is structured in the following manner:

1. first, some logging messages reporting the generation of constraints are shown;
2. then, it presents the candidate pairs of events which may form a data race;
3. next, it lists the actual data races, i.e. the candidate pairs whose events are concurrent;
4. finally, some metrics collected during SPIDER's execution are shown.

Even though the lists of pairs of events in SPIDER's output look hard to read, they are actually simple - each pair in the list contains two event identifiers. Each identifier starts with a 'W' if it is a write or with 'R' if it is a read and is followed by the accessed variable, the thread where it occurs, the node where the thread is running, and the line of code responsible for the event. Table 1 summarizes the actual data races found in fig. 11, which are exactly those reported in the beggining of this section, demonstrating that SPIDER is effective at finding the data races that are present in Example1.

Table 1: Data races in Example1

Race Pair	**Instruction #1**	**Instruction #2**
(W_demos.Example1.counter_main@10.0.0.1_3@demos.Example1.main.7, W_demos.Example1.counter_Thread-1@10.0.0.1_5@demos.Example1.run.12)	Line 7 in class Example1	Line 12 in class Example1
(W_demos.Example1.counter_main@10.0.0.1_3@demos.Example1.main.7, W_demos.Example1.counter_Thread-1@10.0.0.1_5@demos.Example1.run.12)	Line 8 in class Example1	Line 12 in class Example1

4

CHALLENGE: SCALING SPIDER'S APPROACH

From empirical observations (Table 2), it became clear that Spider did not scale sufficiently well to analyse long traces of a distributed application. Runtime tracing of distributed systems usually results in traces which, due to their size, are not amenable to being analysed using SMT solvers. Explicitly selecting the variables to be tracked and ignoring accesses to all other variables could significantly improve the performance of the analysis but even in that case, the performance would be severely degraded if there were multiple and repetitive accesses to the selected variables. Such degradation is, above all, a consequence of the assimptotic complexity of the trace-analysis algorithm which is bounded below by $\mathcal{O}({n_{events}}^3)$, where n_{events} stands for the number of events in the trace file(s). This bound factors in the following observations:

- The worst case occurs when reading a trace after having read n traced memory accesses to a shared variable. In such conditions, when a new Write is read in a thread that did not contain any accesses to the shared variable before, there will be n new conflicting pairs of memory accesses, one for each access performed in other threads and read by the trace analysis algorithm before the current access. For each conflicting pair of acesses, there will be a call to an SMT solver to determine wether the accesses are concurrent, as seen in Fig. 5. Thus, the number of queries to the SMT solver is quadratic in the number of memory accesses. It is also quadratic in the number of traced events.

- All models passed to the SMT solver have the same size, i.e., the same number of variables and constraints. Every query to the SMT solver includes the complete model of the happens-before relation and one concurrency constraint.

- The number of constraints in the model is linear in the number of traced events – each event appears, at most, in three constraints. As such, there cannot be more than $3 * n_{events}$ constraints. Besides, the number of variables in the model is also linear in the number of traced events – for each event, there is a variable in the model representing the relative ordering of the event.

- For each conflicting pair of memory accesses, the SMT solver will be called with the model containing a number of constraints and variables that is linear in n_{events}. Spider uses an SMT solver to determine the satisfiability of formulas in QF_IDL (sec. 3.3). In turn, the solver makes use of efficient decision procedures to determine the satisfiability of the constraints. However, in order to simplify the complexity analysis of the problem that Spider solves and to provide a lower-bound to the time complexity in the worst case of the problem, we only consider the complexity of decision procedures used independently of SMT solvers. An example of this is the *Bellman-Ford algorithm* which reduces the problem of consistency checking for difference logic to the detection of negative cycles in weighted digraphs in $\mathcal{O}(n \cdot m)$ time, where n is the number of variables and m is the number of constraints. Ge et al. (2016) realized that it is not necessary to use the full QF_IDL for this purpose, and devised the *theory of Ordering Constraints*, a fragment of QF_IDL which includes conjunctions and disjunctions of boolean expressions over ordering comparisons such as $O_1 \square O_2$, with $\square \in \{<, \leq, >, \geq, =, \neq\}$. They also provided a decision procedure that can be used with or without an SMT solver and showed that, in the latter case, the consistency of conjunctions of ordering constraints can be decided in $\mathcal{O}(n + m)$ time. Given that the number of variables n and the number of constraints m are linear in n_{events}, the complexity of verifying consistency of conjunctions of ordering constraints in this context can be simplified to $\mathcal{O}(n_{events})$.

To sum up, the number of traced events is the major factor influencing the performance of Spider.

Handling too many events

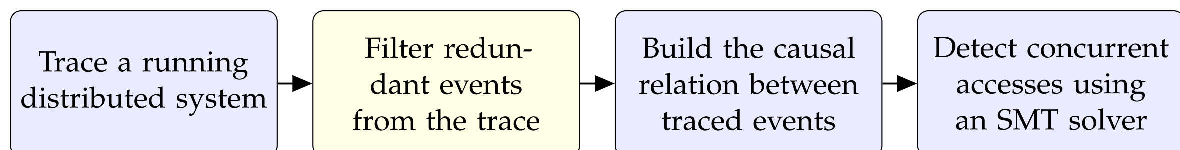


Figure 12: **Spider's workflow including redundant event pruning** – the new step is highlighted in yellow.

Having shown that the performance of Spider depends mainly on the number of traced events, we now observe that Spider's run time can decrease substantially by eliminating events from the trace. Furthermore, it is possible to remove events from the trace without changing the results obtained by the trace analysis. For example, Huang and Rajagopalan (2017) observed that, in multi-threaded systems, most accesses to shared variables arise typically from the same locations in the code, often leading to races being reported multiple

times due to repeated execution of the same faulty instruction. Subsequently, a repetitive memory access which does not reveal new races, dubbed a *redundant event*, can be ignored for race detection purposes. Huang and Rajagopalan (2017) designed *ReX*, an algorithm to identify and remove redundant events from multi-threaded systems' traces. They tested ReX against several benchmarks and found out that almost all of the resulting traces consisted of more than 90% of redundant events.

Inspired by the amount of redundant events present in the traces and by the effectiveness of ReX, we implemented an extended version of this algorithm to detect and remove redundant events from traces. However, unlike single-node multi-threaded systems, distributed systems tend to use messaging primitives quite extensively. As such, our algorithm uses an extended notion of redundant event applicable not only to memory accesses but to any kind of event and is capable of removing redundant message handlers from the traces. This algorithm was integrated in Spider to reduce the size of the traces before analysing them. The next section presents the algorithm.

Redundancy pruning

As noted by Huang et al., previous reasearch on how to decrease the size of traces focused mainly on dynamic sampling [Bond et al. (2010); Marino et al. (2009)] and static analysis [Rhodes et al. (2017); Biswas et al. (2015)] but these approaches have the disadvantage of producing too many false positives and leading to races being missed from the analysis. Instead, they propose a way to prune events based on a *redundancy criterion*: a memory access is deemed *redundant* and can be removed from the trace if its removal does not affect the results of data race detection [Huang and Rajagopalan (2017)]. To this end, they introduce the concept of *Concurrential-Subsume Equivalence*: for two memory accesses e_i and e_j, e_j is concurrentially-subsumed by e_i when

1. both e_i and e_j are caused by the same program instruction and have the same access type (i.e. both are Reads, or both are Writes)
2. they access the same memory location
3. for every event e_k such that $t_{e_k} \neq t_{e_i} \wedge t_{e_k} \neq t_{e_j}$,

$$e_k \prec_{hb} e_i \rightarrow e_k \prec_{hb} e_j \tag{17}$$

$$e_i \prec_{hb} e_k \rightarrow e_j \prec_{hb} e_k \tag{18}$$

Besides, they prove that an event is redundant if the trace contains one concurrential-subsuming equivalent event from the same thread, or two concurrential-subsuming events

from different threads; finally, they present ReX, an algorithm designed to efficiently prune redundant events based on this observation.

The inner workings of ReX are out of the scope of this dissertation. Interested readers should look into the original paper on ReX. Nonetheless, we present a brief description on how it works. ReX keeps track of the *concurrency contexts* of each thread. The concurrency context of a threadt t, represented by $Gamma_t$, is continuously updated and encodes the history of Send, Lock Release and Fork events observed by t. The concurrency context of an event e generated by thread t is the value of Γ_t at the time e is observed. For each thread, its concurrency context has a stack associated. Based on the contents of this stack, there are four cases to consider when an event e occurs in thread t:

1. the stack is empty – this particular concurrency context was not seen in any of the accesses so far, and thus the event is not redundant;
2. the stack contains t – e can be eliminated because there is already a concurrential-subsuming event from the same thread;
3. the stack does not contain the thread t – the event is not redundant;
4. the stack has size two – e can be eliminated because there is already a concurrential-subsuming event from two different threads;

Inspired by this work, we have implemented an event filter in Spider that runs before any trace analysis and significatly improves its performance on large traces while mantaining the results of data race detection. It results from composing two other filters. First, we apply a version of the ReX algorithm adapted to our systems model[1] (algorithm 1). After ReX has been applied, there will be no redundant memory accesses left in the trace but the presence of all other kinds of events will remain unchanged.

We then perform a second filtering on the events designed to prune *redundant message handlers* and *redundant threads* (algorithm 3). Redundant message handlers and redundant threads are defined in terms of a generalization of redundancy for block of events: a block ϕ of events ocurring in the same thread is redundant if the removal of every event in ϕ from the trace does not change the results of the HB race detection. Furthermore, an event is redundant if it is an element of a redundant block. In order for a block to be redundant, it must not have any non-redundant memory access and its events must not enforce any particular ordering on events occurring in different threads, e.g., a redundant block must not contain a Send event because it enforces an event ordering where the events ocurring before the Send event in the sending thread occur before the events occurring after the corresponding Receive event in the receiving thread. To determine if a block is redundant, it suffices to check that the block has the following properties:

1 The original ReX algorithm does not take into consideration the existence of Fork and Join events signaling the creation and joining of threads.

Algorithm 1: ReX (Trace trace)

```
for Event e : trace do
    t ← e.getThread();
    loc ← e.getLineOfCode();
    switch e.getType() do
        case Read, Write do
            if isRedundant(e, t, Γt) then
                remove event e from the trace;
        case Unlock do
            add e.getLock() to Γt;
        case Send do
            add e.getSocketId() to Γt;
        case Fork do
            add e.getChildThreadID() to Γt;
```

Algorithm 2: isRedundant(Event e, Thread t, ConcurrencyCtxt Γ_t)

```
stack ← obtain the stack associated with event e location and concurrency context Γt;
if stack is empty then
    // add thread t to the stack
    stack.push(t);
else if stack.contains(t) then
    return true;
else if stack.size() = 1 then
    stack.push(t);
    return false;
else
    // the stack is full
    return true;
```

1. it does not contain a non-redundant message access[2];
2. it does not contain a Send or Receive event;
3. it does not contain a Fork or Join operation whose child thread is not redundant.
4. all locks acquired in the block are released in the block, and all locks released in the block have been previously acquired in the block.

We can now define a redundant message handler as a message handler whose events constitute a redundant block, and a redundant thread as a redundant block that contains all the events of that thread. In the case of redundant threads, it is also possible to delete the corresponding Fork and Join events without affecting the results of data race detection. In the case of redundant message handlers, it is not correct to remove the corresponding Send and Receive events because its removal may lead to the removal of constraints between two different non-redundant threads in the generated model and this can, in principle, lead to false positives.

The second filter does not remove any memory access, neither does it modify the generated HB relation between non-redudant events. The two filters in conjunction do not cause the removal of non-redudant memory accesses neither do they modify the generated happens-before relation between non-redundant events. Consequently, the results of data race detection when run after filtering the events will not differ from those obtained when data race detection is run without previously filtering the events.

Algorithm 3: RemoveRedundantBlocks(Trace trace)

```
for Event e : trace do
    switch e.getType() do
        case Receive do
            messageHandler ← events in the corresponding message's handler;
            if isRedundantBlock(messageHandler) then
                remove all events in messageHandler from the trace;
        case Fork do
            childThreadID ← e.getChildThreadID();
            threadEvents ← events in thread childThreadID;
            if isRedundantBlock(threadEvents) then
                remove all events in threadEvents from the trace;
                remove corresponding Fork event;
                remove corresponding Join event;
```

2 This filtering is performed after ReX, so any memory access in the block is non-redundant.

Algorithm 4: IsRedundantBlock(Block block)

```
acquiredLocks ← ∅;
for Event e : block do
    switch e.getType() do
        case Send, Receive, Write, Read, Join do
            return false;
        case Fork do
            childThreadID ← e.getChildThreadID();
            eventsThread ← events in thread childThreadID;
            if ¬ isRedundantBlock(eventsThread) then
                return false;
        case Lock do
            acquiredLocks ← acquiredLocks ∪ {e.getLock()};
        case Unlock do
            acquiredLocks ← acquiredLocks \ {e.getLock()};
return acquiredLocks = ∅;
```

5 EVALUATION

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. Currently, Spider only supports the Z3 SMT solver but there are plans to make Spider agnostic of the underlying solver, as long as the solver supports the *SMT-LIB2* format. Appendix A briefly summarizes how the implementation of Spider is structured.

To assess the potential and limitations of our prototype, we conducted an experimental evaluation focused on answering the following four questions:

- How effective is Spider in finding data races in distributed executions? (§5.2)
- How does the Spider's efficiency vary with the size of the execution trace? (§5.3)
- How does redundancy pruning affect Spider's effectiveness and efficiency? (§5.3)

Besides, this chapter ends with a brief discussion on the soundness and precion of Spider (§5.4). In our experiments, we used Minha [Machado et al. (2019)] to collect the execution traces, and the Z3 SMT solver Z3 (v4.4.1) to solve the constraints. We assumed a timeout of 2 hours 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 16GB of RAM.

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

5.1 BENCHMARKS

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

5.1.1 *TaxDC Micro-Benchmarks*

We tested Spider agains five micro-benchmarks developed by Nuno Machado that were inspired in real-world races on popular distributed systems, namely HBase [Apache HBase]

and Hadoop MapReduce [Apache Hadoop], as described in the TaxDC database [Leesat-apornwongsa et al. (2016)]. These micro-benchmarks contain different types of data races (§2.2) and are publicly available in github.com/jcp19/micro-benchmarks. We believe they can be useful for the community to evaluate similar testing tools in the future.

Figure 13 depicts the distributed data races considered in our micro-benchmarks. Following TaxDC's notation, in Figure 13, 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 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.

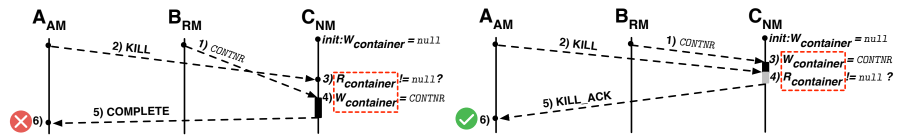
c) **Message-message race across two nodes (M5358)** – A_{AM} assigns a task to C_{NM1} along with a backup speculative task to B_{NM2}. When receiving the success confirmation from C_{NM1}, A_{AM} changes the state of the task to succeeded (tState = OK) and sends a *KILL* message to B_{NM2}. However, 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

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*

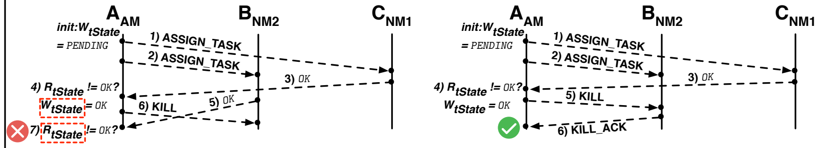
a) message-message race between arrival/sending (H5780)

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}*

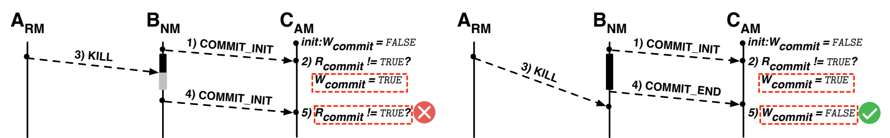


c) message-message race across two nodes (M5358)

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)

Figure 13: **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.

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.

5.1.2 *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 [Voulgaris et al. (2005)], already used in prior work by Machado et al. (2019). 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. (2019), the atomicity of the shuffle operation is not guaranteed by the original description of 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 execution of the protocol and, when it does, the nodes involved and the cycles in which the violation occurs might vary across test runs.

5.2 EFFECTIVENESS

Table 2 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 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.

5.3 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 model. Table 2 reports the results of our experiments. The columns of the table indicate, respectively, the benchmark that was run, the size of the trace, the number of events in the trace, the number of constraints in the generated 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.

Table 2: **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	3KB	15	12	3	1 (1)	<1s
m3274	3KB	18	14	1	1 (1)	<1s
m5358	5KB	27	19	2	2 (2)	<1s
m4157	2KB	12	11	4	2 (2)	<1s
m5009	4KB	19	14	2	2 (2)	<1s
Cyclon-5N-5C	74KB	420	488	325	121 (1)	<1s
Cyclon-5N-10C	145KB	820	1464	1150	481 (1)	1.8s
Cyclon-5N-100C	1433KB	8020	104505	101500	49835 (1)	1h43m
Cyclon-10N-5C	147KB	840	976	650	243 (1)	<1s
Cyclon-10N-10C	290KB	1640	2920	2300	969 (1)	7.8s
Cyclon-10N-100C	2869KB	16040	6031	203000	-	Timeout
Cyclon-100N-5C	1486KB	8401	9800	6500	2394 (1)	2m53s
Cyclon-100N-10C	2934KB	16401	29298	23000	9651 (1)	1h03m
Cyclon-100N-100C	29076KB	160400	60301	2030000	-	Timeout

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 3 summarizes our observations. The columns of the table indicate, respectively, the benchmark that was run, the number of redundant events and their ratio 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 3 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.

Table 3: **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)	<1s
Cyclon-5N-10C	296 (36%)	339	99	45 (1)	<1s
Cyclon-5N-100C	3875 (48%)	2179	135	65 (1)	8.0s (↓ 777.5x)
Cyclon-10N-5C	207 (24%)	446	173	78 (1)	<1s
Cyclon-10N-10C	520 (32%)	838	348	149 (1)	<1s (↓ 7.8x)
Cyclon-10N-100C	7466 (47%)	5180	1028	509 (1)	4m44s
Cyclon-100N-5C	1980 (24%)	4437	1668	753 (1)	30.3s (↓ 5.7x)
Cyclon-100N-10C	3719 (22%)	11893	6615	3134 (1)	22m25s (↓ 2.8x)
Cyclon-100N-100C	47800 (30%)	47601	350202	-	Timeout

5.4 SOUNDNESS AND PRECISION OF THE APPROACH

In this section, we argue why the results of data race analysis using SPIDER are trustworthy. 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.

6

RELATED WORK

This chapter presents research that, like Spider, aims at improving the reliability of distributed systems. This section foccuses mainly on formal software verification and testing of distributed systems.

6.1 formal verification

6.1.1 *An overview of formal verification of computer programs*

Static program analysis, commonly referred to as *static analysis*, offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program on a computer [Nielson et al. (1999)]. This branch of Computer Science is historically related to code optimization and compiler design and subsumes fields like *Abstract Interpretation*, *Data Flow Analysis* and *Control Flow Analysis*.

Formal Verification grew independently from Program Analysis leading to techniques like *Model Checking*, an automated way of checking whether a property holds in a formal model of a system by systematically exploring the state-space of the model. Traditionally, models are described as finite-state transition systems and the properties to be checked are expressed in *temporal logics* [Ben-Ari et al. (1981)]. Checked properties are usually divided into two main categories:

- *safety properties* specify the states that should never be reached. For example, in a model of a crossroad with traffic lights on each entrance of the crossroad, a critical safety property would be

  P1: No two traffic lights are green at the same time

  Any violation of a safety property is representable as a finite execution of the system leading to the undesired state. In the given example, if $P1$ is invalidated the model

checker will give a finite execution leading to a state where more than one traffic light is green.

- *liveness properties* specify the properties that are desired to eventually hold in the model. In the example of the crossroad model, it is expected that

  P2: Each traffic light will always become green eventually

  Liveness properties can only be invalidated with an infinite execution where the desired property is never reached. $P2$ can be invalidated with an infinite run of the system where a traffic light never becomes green. $P2$ can also be invalidated in an execution where a traffic light becomes green and then returns to red without ever becoming green again.

Model Checking has the advantage of providing a counter-example if a property is found to be invalid. However, this approach is limited in the sense that only finite-state systems can be verified and, even in that case, the size of the state-space may render this technique intractable - the famous *state explosion problem*. Readers interested in a complete treatment of Model Checking are directed to the book by Baier and Katoen (2008). A weaker form of model checking, Bounded Model Checking (BMC) [Biere et al. (2003)], was introduced to search for counterexamples in executions whose length is bounded by some integer. Implementations of Bounded Model Checkers often work in two main steps: (1) generate propositional formulas which are satisfiable iff the property holds for executions with a length not bigger than the bound (2) dispatch the generated formulas to SAT solvers. BMC cannot be used to prove the absence of property violations since the analyis is limited to executions whose length is limited by a pre-defined bound. It can however be useful to detect early violations of properties.

Formal Software Verification results from the intersection of static analysis and formal verification. The holy grail of formal software verification would be to have algorithms both *sound* and *complete* to fully automate the verification of properties about computer programs in the sense that no additional input besides the program and the properties to verify are required. A verification algorithm is said to be *complete* iff for every program P and every valid property of the program ψ, the algorithm returns *True*. An algorithm is said to be *sound* iff for every program P and every property ψ the algorithm returns *True* only when ψ is valid in the program. Any attempt to develop sound and complete algorithms is however doomed to fail: Henry Gordon Rice demonstrated in his PhD thesis that all non-trivial semantic properties of programs are undecidable[1]. This result is known as *Rice's Theorem*

1 A semantic property is a property about the program's behavior e.g. the program terminates on all inputs. A property is non-trivial if it is neither true nor false for every computer program.

and implies that there cannot be a fully automated, sound and complete algorithm which for all properties and programs written in any sufficiently powerful programming language returns *True* if and only if the property is valid in the program. This does not mean that it is impossible to solve particular instances of the verification problem but that one has to be aware that sacrifices must be made when conceiving theoretical and practical frameworks to tackle it.

The initial research in the area focused on creating formal semantics for sequential programs in order to manually and deductively reason about them. *Floyd-Hoare logic* [Hoare (1969)] was conceived to allow system verifiers to specify the *pre-conditions* P and *post-conditions* Q of a computer program C as a triplet of the form

$$\{P\}C\{Q\}$$

and to prove, through well defined inference rules, that for any initial state satisfying P, the program C would run and, in case of termination, would reach a state satisfying Q. The proof system of *Floyd-Hoare logic* is sound and relatively complete, i.e. it is complete if there is a complete proof system for proving assertions in the underlying logic[2]. Proofs in *Floyd-Hoare logic* are difficult and cannot be automated due to the need of finding *loop invariants* for each loop in the program. A loop invariant is a property that is preserved during all iterations of the loop and, after the loop terminates, implies the post condition of the loop.

Manual proofs rapidly grew too long and complex, albeit repetitive. Consequently, researchers started to look for ways to obviate and automate parts of software verification giving birth to the area of *Software Model Checking*. The goal of the research in this area is to increase the scope of automated techniques for reasoning about programs, both in power and applicability. Tool designers have to choose either completeness or soundness when they design algorithms for software model checking. Jhala and Majumdar (2009) state in their survey about Software Model Checking that algorithms tend to preserve soundness and give up on completeness. There are two approaches to achieve this:

- *software model checking geared towards falsification* – a subset of the computations is explored in order to find a violation of the property. If one is found, it proves that the program does not satisfy the property. Otherwise, no conclusion can be drawn. BMC can be seen as a particular case of this.

- *software model checking geared towards verification* – a superset of the possible computations is explored. If a violation is found, there is no conclusion to be drawn. Otherwise, it is certain that the property is valid in the program.

2 First-order logics are incomplete and are commonly used as the underlying logic.

A proper coverage of Software Model Checking is out of the scope of this document. Nevertheless, the surveys by D'Silva et al. (2008) and Jhala and Majumdar (2009) are highly recommended.

Research on SMC didn't stop the use of deductive techniques for verifying complex programs; in fact, those techniques have been used in conjunction with proof assistants to verify a wide range of non-trivial programs such as compilers [Leroy (2009)], operating systems' kernels [Gu et al. (2016)] and distributed systems (discussed in the next section).

6.1.2 *Formal verification of distributed systems*

In the last decade, there has been no shortage of research directed at the verification of distributed systems originating not only from academia but also from industry as well. Initially, the focus of research on verifying distributed systems was on proving that the high-level designs (or models) of these systems conform to a given specification. Such tools can be used to demonstrate that a high-level design of a system is free of DC bugs, including data races. More recently, researchers' focus has expanded to include not only the verification of designs of distributed algorithms but also of their implementations. In this section, we show some techniques that have been proposed to ensure that a distributed system is free of bugs, and in particular, of DC bugs. We start by discussing some automated approaches (§6.1.2) and then consider techniques which require manual intervention to complete correctness proofs (§6.1.2).

Model Checking and Software Model Checking of Distributed Systems

Model Checking techniques are often used to verify high-level designs of distributed protocols against a specification. Zave (2012) formally modeled and specified the Chord algorithm [Stoica et al. (2003)] which implements a distributed hash-table and found, using *Alloy* [Jackson (2006)], that no previously published version of the algorithm was correct. Besides, the author stated that

> *"Whether one cares about Chord or not, the significance of these results is that important research protocols such as Chord exist in a haze of uncertainty about their specifications, properties, versions, and performance characteristics. Lightweight modeling can reduce this uncertainty with relatively little effort and without specialized knowledge of verification."*

At Amazon, Newcombe et al. (2013) reported the successful adoption of two specification and modelling languages – *Temporal Logic of Actions+ (TLA+)* and *PlusCal*, both created by Lamport. By creating models and specifications in these languages and running them in the *TLC model checker*, they were able to detect serious bugs in the design of some of their

most complex systems. Besides, the authors noted that the protocols are better understood by developers when they are formally specified. Table 4 summarizes some of the results achieved by adopting verification techniques to scrutinize the design of complex distributed systems.

System	**Components**	**Line Count**	**Benefit**
S3	Fault-tolerant low-level network algorithm	804 in PlusCal	Found 2 bugs. Found further bugs in proposed optimizations.
S3	Background redistribution of data	645 in PlusCal	Found 1 bug, and found a bug in the first proposed fix.
DynamoDB	Replication & group membership system	939 in TLA+	Found 3 bugs in long runs of the system.
EBS	Volume management	102 in PlusCal	Found 3 bugs.
Internal Distributed lock manager	Lock-free data structure	223 in PlusCal	Improved confidence on the system. No bugs found.
Internal Distributed lock manager	Fault tolerant replication and reconfiguration algorithm	318 in TLA+	Found 1 bug and verified one agressive optimization

Table 4: Usage of *TLA+* and *PlusCal* to verify parts of some of the most complex systems from *Amazon* as presented by Newcombe et al. (2013)

Software Model Checking aimed at distributed programs has been the subject of improvements in recent times. Software Model Checkers such as MaceMC [Killian et al. (2007)], Demeter [Guo et al. (2011)], MoDist [Yang et al. (2009)], dBug [Simsa et al. (2010)] systematically explore different execution orderings by permuting message arrivals and injecting node crashes and timeouts. Like Spider, software model checkers of this sort (also called *execution-based model checkers*) are often used as a *test amplification* mechanism in the sense that programs under evaluation are tested against a typical workload of the system instead of running with all possible inputs, and rely on various re-executions of the target system to detect bugs. However, unlike Spider which only looks into a single execution of the program to detect possible concurrency bugs, execution-based model checkers usually require multiple runs of the program under the same workload. This is due to the underlying assumption that the non-determinism that affects a *concurrent program* comes from either the inputs from the environment or the scheduling choices made by the OS's scheduler. As such, the set of program behaviours can be identified by the set of all inputs and schedules.

This assumption dictates the way execution-based model-checkers find bugs. Jhala and Majumdar (2009) summarize:

> ”The user provides a test harness corresponding to a workload under which the program is run. The usual operating system scheduler would execute the workload under a fixed schedule, thereby missing most possible behaviors. However, the execution-based model checker's scheduler is able to systematically explore the possible executions of the same workload under different schedules, e.g., exploring what happens under different interleavings of shared operations like message sends, receives etc., and is able to find various safety errors like assertion failures, deadlocks and divergences, that are only manifested under corner-case schedules.”

Solutions based on SMC still suffer from state-space explosion. As a result, the exaustive exploration of program executions is bounded by a ”shallow depth” of interleavings. State-space reduction techniques like *partial order reduction* alleviate state-space explosion but this problem remains a major hindrance [Yabandeh and Kostic (2009); Leesatapornwongsa et al. (2014)].

The key advantage of SMC compared to SPIDER's approach is that whenever a system is found to have bugs, an execution where the bug manifests itself is also presented.

Verification Techniques requiring manual intervention

There has been a growing body of work studying how to build correct distributed systems and how to develop them correctly from scatch using proof assistants such as Coq and Agda. The following list presents a sample of relevant works published recently:

- *Verdi* [Wilcox et al. (2015)] – framework used to implement and verify distributed systems in the Coq Proof Assistant. It provides various semantics for the network and its fault model. Verdi's authors claim that it simplifies proofs by allowing the developer to first verify a system on an idealized fault model and then proving its correctness in more complex fault models without extra work. To that end, they use a Verified System Transformer (VST) to transform a correct system in a given fault model into another one that tolerates faults in different fault models while providing analogous guarantees.

- *IronFleet* [Hawblitzel et al. (2015)] – methodology for building provably correct distributed systems using *Dafny* which can later be compiled to C#. The core idea of the IronFleet methodology revolves around proving that a distributed system is correct by showing that it meets a high-level centralized specification. For example, if one can prove that the implementation of a sharded key-value store acts like a centralized

key-value store then there cannot be bugs like race conditions otherwise it would not behave like a centralized key-value store. According to the *IronFleet* methodology, a distributed system implementation and proof of correctness should be organized into three layers:

1. *specification layer* – the system is specified as a state machine, i.e. a set of initial states and a description of the allowed transitions between states. This layer defines all behaviours allowed in the system.
2. *protocol layer* – the developer specifies a distributed state machine consisting of multiple host state machines which communicate through messages and a set of packets in the network.
3. *implementation layer* – this layer contains the single-threaded, imperative code to be run on each host.

To prove the implementation correct, it must be proved that the protocol layer is a *refinement* of the specification layer and that the implementation layer is a refinement of the protocol layer[3].

- *Ivy* [Padon et al. (2016)] – tool for implementing and verifying safety properties on infinite-state systems. Ivy's main use case is the interactive construction of inductive invariants, presenting visual cues to help a human come up with one. *Ivy* can automatically check whether a property is an inductive invariant or not, terminating with either a proof of inductiveness of the invariant or a finite Counterexample to Induction (CTI) which is represented visually. In order to guarantee that Ivy will produce one of these two possible outputs, programs are written and specified using a restricted specification language – Relational Modeling Language (RML) – that guarantees that all verification conditions are in *EPR*, a decidable fragment of first order logic. When a CTI is found, one of the following three statements has to be true:

  1. there is a bug in the model;
  2. one of the conjectures of the invariant is wrong and the invariant should be weakened;
  3. the CTI is not reachable from the initial states. The invariant should be strengthened by adding conjectures that rule out the CTI and generalize from it.

  Ivy also allows the compilation of the RML program into an executable program.

- *Disel* [Sergey et al. (2017)] – separation-style logic based framework for implementing and verifying distributed systems in a compositional way inside *Coq*.

3 A state machine X is said to refine another Y if for each possible sequence of states in machine X there exists a corresponding one in Y.

- *Velisarios* [Rahli et al. (2018)] – framework based on *logic-of-events* for proving the correctness of Byzantine Fault Tolerance (BFT) protocols and their implementation in *Coq*. It provides a model of Byzantine faults as well as a set of tactics that capture frequently used patterns in proofs about BFT protocols.

6.2 OTHER APPROACHES

SPIDER is not the first tool to employ SMT constraint solving to test and debug concurrent programs. In fact, SMT solvers were successfully employed in the past to test and debug concurrent programs. For instance, CLAP [Huang et al. (2013)] uses SMT solving to replay failing interleavings, MCR [Huang (2015)] and Cortex [Machado et al. (2016)] to uncover latent concurrency bugs, and Symbiosis [Machado et al. (2015); Terra-Neves et al. (2019)] 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 [Valapil et al. (2017)] or modeled as BPEL processes [Elwakil et al. (2010)].

Like SPIDER, DCatch [Liu et al. (2017)] 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.

7

CONCLUSION AND FUTURE WORK

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. This fact supports our claim that Spider is an effective tool to test distributed systems against data races.

Notwithstanding, we believe that there are multiple paths to improve Spider. First, as we have hinted in section 5.3, we believe that it is possible to optimize Spider by restructuring the queries it makes to SMT solvers – instead of determining wether two events are concurrent for all pairs of conflicting memory accesses, it is enough to ask if a pair of program instructions that leads to conflicting memory accesses has at least a pair of conflicting accesses which are concurrent. This way, we only test if two events are concurrent while their locations haven't been shown to produce data races.

Second, we observe that some data races might not manifest during a particular execution of the system (if, for example, a branch which leads to data races is not executed). As such, Spider may benefit from the hability to analyse multiple different traces of the same system and combining the results in a single report. The goal of this would be to cover multiple execution paths, possibly uncovering more race conditions than what would be possible with a single execution.

Finally, we could extend Spider to act as an *online monitor*, capable of reacting to data races at runtime. This however would require a complete rewriting of Spider. Besides, using SMT solvers in an online monitor might prove to be slow and affect the performance of the monitored program. Thus, we would need to look into alternative algorithms to detect data races in an online setting.

BIBLIOGRAPHY

Agda. URL https://wiki.portal.chalmers.se/agda/pmwiki.php.

Apache Cassandra. Apache cassandra. URL https://cassandra.apache.org/.

Apache Hadoop. Apache hadoop. URL https://hadoop.apache.org/.

Apache HBase. Apache hbase. URL https://hbase.apache.org/.

Apache ZooKeeper. Zoo keeper. URL https://zookeeper.apache.org/.

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN 026202649X, 9780262026499.

Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 164–176, New York, NY, USA, 1981. ACM. ISBN 0-89791-029-X. doi: 10.1145/567532.567551. URL http://doi.acm.org/10.1145/567532.567551.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking, 2003.

Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Valor: Efficient, software-only region conflict exceptions. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 241–259, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336895. doi: 10.1145/2814270.2814292. URL https://doi.org/10.1145/2814270.2814292.

Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: Proportional detection of data races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 255–268, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300193. doi: 10.1145/1806596.1806626. URL https://doi.org/10.1145/1806596.1806626.

Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. In *PrePost@iFM*, 2017.

Miguel Castro and Barbara Loskov. Practical byzantine fault tolerance, 1999.

Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. doi: 10.1145/800157.805047. URL http://doi.acm.org/10.1145/800157.805047.

Coq. The coq proof assistant. URL https://coq.inria.fr/.

Dafny. Dafny: A language and program verifier for functional correctness. URL https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/.

Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification, 2008.

Mohamed Elwakil, Zijiang Yang, Liqiang Wang, and Qichang Chen. Message race detection for Web Services by an SMT-based analysis. In *Autonomic and Trusted Computing*, pages 182–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-16576-4.

Falcon TAZ. Falcon taz (event trace organizer). URL https://github.com/fntneves/falcon/tree/master/falcon-taz.

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. *SIGPLAN Not.*, 53(4):420–435, June 2018. ISSN 0362-1340. doi: 10.1145/3296979.3192382. URL http://doi.acm.org/10.1145/3296979.3192382.

Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *PLDI '09*, 2009.

Cunjing Ge, Feifei Ma, Jeff Huang, and Jian Zhang. Smt solving for the theory of ordering constraints. In Xipeng Shen, Frank Mueller, and James Tuck, editors, *Languages and Compilers for Parallel Computing*, pages 287–302, Cham, 2016. Springer International Publishing. ISBN 978-3-319-29778-1.

Glucose. Glucose's home page. URL https://www.labri.fr/perso/lsimon/glucose/.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, Savannah, GA, November 2016. USENIX

Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *SOSP '11*, 2011.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815428. URL http://doi.acm.org/10.1145/2815400.2815428.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12 (10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259.

Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *PLDI '15*. ACM, 2015.

Jeff Huang and Arun K. Rajagopalan. What's the optimal performance of precise dynamic race detection? - a redundancy perspective. In *ECOOP*, 2017.

Jeff Huang, Charles Zhang, and Julian Dolby. Clap: Recording local executions to reproduce concurrency failures. In *PLDI '13*. ACM, 2013.

Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006. ISBN 0262101149.

Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4): 21:1–21:54, October 2009. ISSN 0360-0300. doi: 10.1145/1592434.1592438. URL http://doi.acm.org/10.1145/1592434.1592438.

Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with portend. In *ASPLOS '12*. ACM, 2012.

Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 18–18, Berkeley, CA, USA, 2007. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1973430.1973448.

Leslie Lamport. The tla+ home page. URL https://lamport.azurewebsites.net/tla/tla.html. Accessed: 2019-01-10.

Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM 21, 7 (July 1978), 558-565. Reprinted in several collections, including Distributed Computing: Concepts and Implementations, McEntire et al., ed. IEEE Press, 1984.*, pages 558–565, July 1978. URL https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/.

Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982. ISSN 0164-0925. doi: 10.1145/357172.357176. URL http://doi.acm.org/10.1145/357172.357176.

Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 399–414, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa.

Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. *SIGARCH Comput. Archit. News*, 44(2):517–530, March 2016. ISSN 0163-5964. doi: 10.1145/2980024.2872374. URL http://doi.acm.org/10.1145/2980024.2872374.

Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL http://doi.acm.org/10.1145/1538788.1538814.

Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009. ISSN 1567-8326. doi: https://doi.org/10.1016/j.jlap.2008.08.004. URL http://www.sciencedirect.com/science/article/pii/S1567832608000775. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In *SOSP '19*. ACM, 2019.

Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. *SIGOPS Oper. Syst. Rev.*, 51(2):677–691, April 2017. ISSN 0163-5980. doi: 10.1145/3093315.3037735. URL http://doi.acm.org/10.1145/3093315.3037735.

N. Machado, F. Maia, F. Neves, F. Coelho, and J. Pereira. Minha: Large-scale distributed systems testing made practical. In *OPODIS' 19*. Leibniz International Proceedings in Informatics (LIPIcs), November 2019.

Nuno Machado, Brandon Lucia, and Luís Rodrigues. Concurrency debugging with differential schedule projections. In *PLDI '15*. ACM, 2015.

Nuno Machado, Brandon Lucia, and Luís Rodrigues. Production-guided concurrency debugging. In *PPoPP '16*. ACM, 2016.

Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 134–143, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542491. URL https://doi.org/10.1145/1542476.1542491.

J. Marques-Silva. Practical applications of boolean satisfiability. In *2008 9th International Workshop on Discrete Event Systems*, pages 74–80, May 2008. doi: 10.1109/WODES.2008.4605925.

Minisat. The minisat page. URL http://minisat.se/.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL http://bitcoin.org/bitcoin.pdf.

F. Neves, N. Machado, and J. Pereira. Falcon: A practical log-based analysis tool for distributed systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 534–541, June 2018. doi: 10.1109/DSN.2018.00061.

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. Use of formal methods at amazon web services, 2013.

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg, 1999. ISBN 3540654100.

Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2. URL http://dl.acm.org/citation.cfm?id=2643634.2643666.

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. *SIGPLAN Not.*, 51(6):614–630, June 2016. ISSN 0362-1340. doi: 10.1145/2980983.2908118. URL http://doi.acm.org/10.1145/2980983.2908118.

Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Copilot: Monitoring embedded systems. In *Innovations in Systems and Software Engineering: Special Issue on Software Health Management*, volume 9, pages 235–255. Springer, 2013. Preprint available at http://www.cs.indiana.edu/~lepike/pub_pages/isse.html.

Nathaniel Popper. The stock market bell rings, computers fail, wall street cringes, 2015. URL https://www.nytimes.com/2015/07/09/business/dealbook/new-york-stock-exchange-suspends-trading.html. Accessed: 2019-08-06.

Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. Velisarios: Byzantine fault-tolerant protocols powered by coq. In *Programming Languages and Systems*. Springer International Publishing, 2018.

Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. Bigfoot: Static check placement for dynamic race detection. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 141–156, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. doi: 10.1145/3062341.3062350. URL https://doi.org/10.1145/3062341.3062350.

Robert Riemann and Stéphane Grumbach. Distributed protocols at the rescue for trustworthy online voting. *CoRR*, abs/1705.04480, 2017. URL http://arxiv.org/abs/1705.04480.

Fred B. Schneider. Chapter 2: What good are models and what models are good? In Sape Mullender, editor, *Distributed Systems (2Nd Ed.)*, chapter 2. ACM Press/Addison-Wesley Publishing Co., 1993. ISBN 0-201-62427-3.

Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 418–427, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0. URL http://dl.acm.org/citation.cfm?id=998675.999446.

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.*, 2(POPL):28:1–28:30, December 2017. ISSN 2475-1421. doi: 10.1145/3158116. URL http://doi.acm.org/10.1145/3158116.

João P. Marques Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7597-7. URL http://dl.acm.org/citation.cfm?id=244522.244560.

Jiri Simsa, Randy Bryant, and Garth Gibson. Dbug: Systematic evaluation of distributed systems. In *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, page 3, USA, 2010. USENIX Association.

SMT-Lib: Logics. Smt-lib: Logics. URL http://smtlib.cs.uiowa.edu/logics.shtml.

Ryan Stansifer. Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department, September 1984. URL http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR84-639.

Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, February 2003. ISSN 1063-6692. doi: 10.1109/TNET.2002.808407. URL http://dx.doi.org/10.1109/TNET.2002.808407.

Miguel Terra-Neves, Nuno Machado, Inês Lynce, and Vasco Manquinho. Concurrency debugging with maxsmt. In *AAAI '19*. AAAI Press, 2019.

The AWS Team. Summary of the amazon ec2 and amazon rds service disruption in the us east region, 2011. URL https://aws.amazon.com/message/65648/. Accessed: 2019-01-03.

Vidhya Tekken Valapil, Sorrachai Yingchareonthawornchai, Sandeep Kulkarni, Eric Torng, and Murat Demirbas. Monitoring partially synchronous distributed systems using SMT solvers, 2017.

Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. CYCLON: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2), 2005.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *PLDI 2015: Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation*, pages 357–368, Portland, OR, USA, June 2015.

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 154–165, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4127-1. doi: 10.1145/2854065.2854081. URL http://doi.acm.org/10.1145/2854065.2854081.

Maysam Yabandeh and Dejan Kostic. Dpor-ds: Dynamic partial order reduction in distributed systems. 2009. URL http://infoscience.epfl.ch/record/139173.

Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *NSDI '09*. USENIX Association, 2009.

Z3 Github Page. The z3 theorem prover. URL https://github.com/Z3Prover/z3.

Pamela Zave. Using lightweight modeling to understand chord. *ACM SIGCOMM Computer Communication Review*, 2012.

SPIDER'S REPOSITORY ORGANIZATION

The core functionality of Spider is implemented in the classes of the package pt.haslab.spider available in Spider's repository and the package pt.haslab.taz provided in Falcon's repository. The latter contains all classes required to parse the traces, while the former contains the actual code of Spider. Figure 14 presents a class diagram of the Spider's repository.

Each class implements the following functionality:

- **CheckerParallel** – main class which connects all components of Spider;
- **RaceDetector** – class responsible for finding all conflicting memory accesses and invoking Z3 to figure out which pairs of conflicting memory accesses actually lead to a data race;
- **RedundantEventPruner** – class that contains the functionalitty required to reduce the number of events in the traces;
- **Stats** – singleton class storing metrics related to Spider's execution, including the time it took for the analysis, the number of conflicting pairs of memory accesses, and the number of data races found;
- **Z3SolverParallel** – wrapper class that spanwns Z3 processes in parallel;

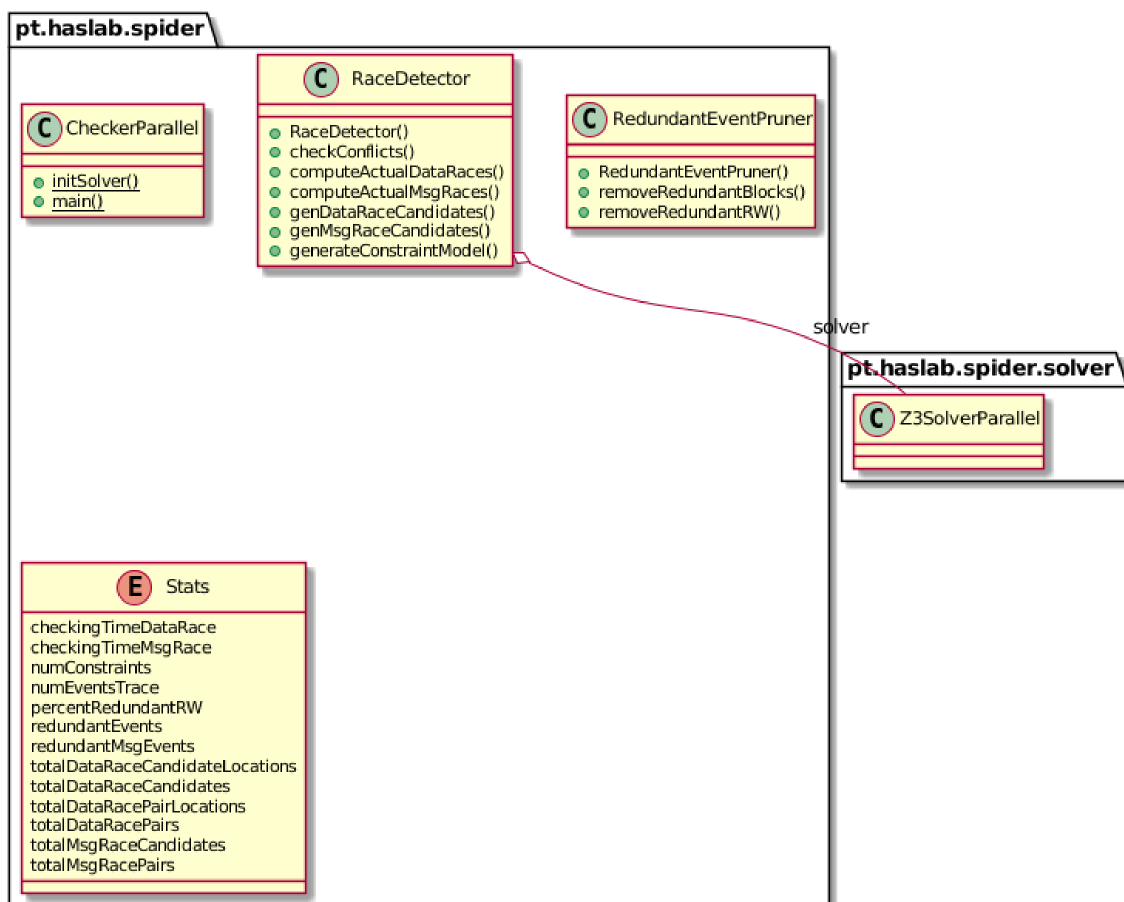


Figure 14: **Class Diagram of Spider**

Este trabalho é financiado por Fundos FEDER através do Programa Operacional Regional do Norte - NORTE 2020 e por Fundos Nacionais através da FCT - Fundação para a Ciência e a Tecnologia no âmbito do projeto NORTE-01-0145-FEDER-028550 - PTDC/EEI-COM/28550/2017.

This work is financed by the ERDF – European Regional Development Fund through the North Portugal Regional Operational Programme - NORTE 2020 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.