M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Hyperconcolic

Finding parallel bugs in Java programs, using concolic execution

Christian Gram Kalhauge (s093273)

Kgs. Lyngby, Denmark 2015

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Preface

Abstract

Parallel errors are an increasingly greater problem. Of these problems are the data race the easiest problem to solve, but it still remains a difficult problem. Static analyses finds to many and dynamic analysis finds too few. I have created a new sound way to extract schedules from parallel programs, based of concolic execution, called hyperconcolic. A new mathematical notation for describing the coverage of dynamic analyses, called schedule classes, is introduced. Hyperconcolic has better schedule class coverage than concolic, because hyperconcolic is able to chose the scheduler for each execution. A new language for containing schedules has been specified to enable multiple programs to work on the same schedules without interfering. State of the art data race detectors has been implemented to analyse these schedules. I have also made probable that combinations of hyperconcolic and some dynamic data race techniques are sound and complete for programs which produce bounded length schedules for all inputs.

Special Thanks

I want to thank Mahdi Eslamimehr and Jens Palsberg for their continues help on the subject, and for supporting my crazy Ideas. I want to thank Christian Probst for the guidance throughout the project and for the comfier meetings and detailed instructions in how to make coffee. I what to thank my dad, Tobias Bertelsen and Kasper Laursen, for proof reading the report. Lastly I want to thank my mom for bringing me cookies, and my dog for walking me now and then.

Structure

Each chapter in this thesis holds a short description and the substructure of the chapter. First Chapter 1, in which the problems and the motivation for the thesis is described. Chapter 2 is about on what a schedule is and defines a language which can describe it. This chapter also introduce a notion call schedule classes which the fundamental element for the thesis. Chapter 3 handles how these schedules can be used for detecting data races. Chapter 4 describes how to generate schedules and how to expand coverage of the dynamic analysis, even in parallel programs. Chapter 5 is a description of the implementation of the systems, and the engineering problems

of the thesis. The implementation is then valued and compared to other data race detectors in Chapter 6. Chapter 7 emphasises the contributions of this thesis, and relates it to previous and future work.

Contents

CHAPTER 1

Introduction

During the last decade there has been an explosion in the number of cores in computer. The best way to harness this parallelism is by using threads, and executing the code asynchronous. In programing languages with side effects the parallel threads might access and disturb each others executions. Even before the multi core systems where introduced, many programing languages supported *concurrent* execution of code. Concurrency is good at handling user inputs or events from the outside world in real-time. Without concurrency this should be handled in an event loop which can be hard to maintain. Concurrency is also great at distributing resources over multiple separate actions which has the same priority. This is especially so for web servers, and therefore, as the most used web server language in the world, Java has also build-in concurrency. With this concurrency, new errors also introduced them selfs, which in some case had fatal consequences, but also extremely hard to detect.

Java is chosen as the primary focus of this investigation. It is easy to find benchmarks and test cases that contain parallel errors that can be detected, because of the popularity and the ease in which people build parallel programs Other languages where possible, especially C++11 which introduced concurrency statements. Most of the techniques used in this thesis are language independent, and should therefore be transferable to any imperative language with some notion of concurrency.

1.1 Parallel Errors

There are many different parallel errors, but the most fundamental is race conditions [17]. There exist two types of races, *data races* and *general races*. A data race can occur when two or more threads can access the same variable, without being ordered. Because the order of the accesses might change from each run through, unordered accesses may result in unintended or impossibles states, as seen in Figure 1.1. The example shows two threads both updating the variable `v` by incrementing it. Even though this access might look atomic, in fact it is not, it is a separation of two events `r = v` and `v = r + 1` where `r` is a register or local variable to the thread. One interleaving of these events result in a final value of 2, and another can result in a final value of 3. If the example where about the amount of insulin a patient gets instead of an integer, the example becomes scary.

The most common way to fix data races is to introduce synchronization statements. Theses statements, can through contracts with the operation system, grantee that some code is executed atomically, without interleaving. This is where the general

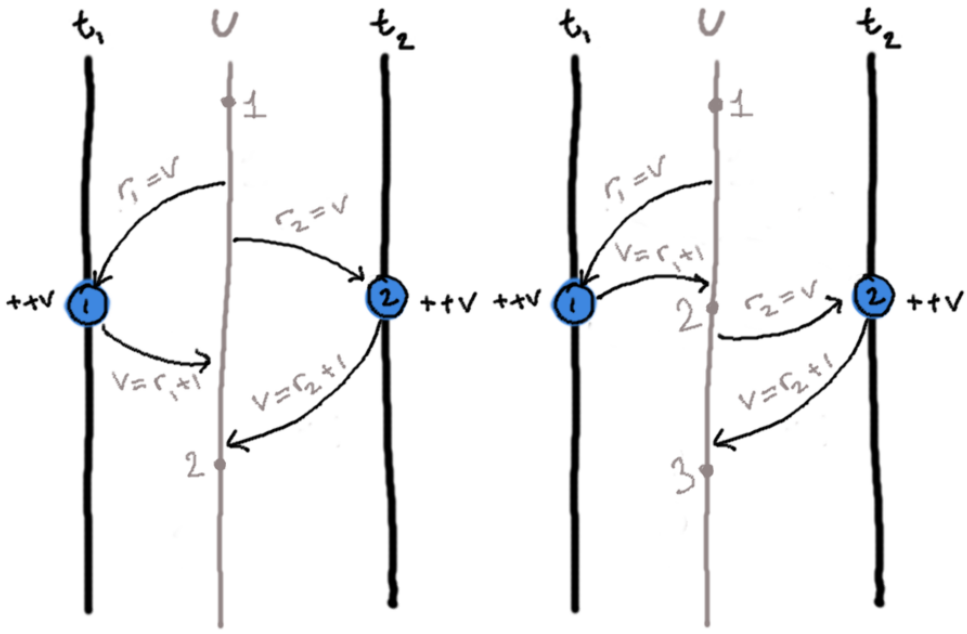


Figure 1.1: A data race. Two processes acessing a single variable `v` in different orderings.

races occurs. General races happens when the atomic sections is ordered. This makes the execution nondeterministic, because depending on the ordering of the atomic sections different outputs might occur. This nondeterminism can result in two atomics sections both have locked a resource the other need. This is called a *deadlock*, referring to that both threads in essences refuses to continue execution. To avoid this some code might release its atomic section, and yield for the other thread. But both threads might do this, which then causes a *livelock*, in which the computer would look like its preforming computations, but really is just accessing and releasing the same atomic sections, over and over.

A large real life study of concurrency bugs, concludes that most (97%) of all bugs can be parted into two patterns *atomicity violation* and *order violation* [15]. Atomicity violations are bugs where the order of the two pieces of code were irrelevant to the execution, but none or bad placed atomic sections made, some sections interleave that should not [14]. Order violations are bugs in which there where implicit order of the two pieces of code, e.g. use before initialization, and use after destruction, but it where not enforced by the code.

I have focused on *data races* as target of this thesis, but the techniques can be reused to find *deadlocks* as well [4]. Data races does have some limitations in terms of

real life cases. Data races can be a strong indicator of atomicity or order violations in the programs, but some races might be benign [18], meaning that the effect of them racing is expected. On the same time, a program without data races might have badly placed atomic sections, so that concurrency bugs still exits in the program.

1.2 Current Solutions

It is almost impossible to debug and test programs that are nondeterministic. The biggest problem is that a data race in itself does not cause an error, but the inconsistent state may be parsed on for some time before the bug is detected. Bug reports might therefore indicate problems in an other part of the code base, in which the data race occurs. Even when the location of a data race is actually found, it might not be enough to fix the bug, because realizing how the situation occurred is as big of a challenge as actually detecting them.

There exist two approaches in finding data races, a static sound analysis or dynamic complete analysis. In the context of this thesis, complete means that the analysis only find real data races and sound means that all the data races are found. It is mathematical impossible for any analysis on a Turning complete language (Halting problem) to be both sound and complete, that is, unless you cheat. Inside a context though, it is possible to put up claims that analyses are sound and complete.

Static Analyses

There has been made a lot of work in making static analysis to detect data races, most of these techniques are sound and not complete. This means that static analyses can be used to guarantee that no data races exist in the program. In real life, static analyses result in a wast amount of possible data races. Because data races are hard to debug, even with the location of racing statements. It can quickly become unfeasible to try to fix all theses reported errors, when they sometimes does not exist.

The most common approach for finding data races using static methods are the lockset analysis (See Section 3.2). It checks that all accesses to shared data is inside atomic section which are guaranteed by the same lock. If not, they are reported as possible data races. Other versions uses type-based systems, which proves or disproves the existence of the program using a type-checker [5], or stateful model checkers, which models the code and tries to prove if some proposition is true [19].

One example of a static data race detector is JChord [16]. It uses a complex system of analyses to perform the most precise static lockset analysis possible.

Dynamic Analyses

Dynamic analyses work on actual schedules, which makes them able to produce complete data race detectors.

Completeness is often more essential than soundness for debugging purposes. Code without data races, is not a grantee that the code is without atomicity violations, while

code with data races is often erroneous. Therefore guaranteeing data race free code, though possible checking a lot of false positives, is often less appealing, than checking the code for true data races, which could be used as an indicator that the program is faulty.

One newer development in dynamic analysis is to produce witnesses [21]. Witnesses are real schedules, which leads up to the error produced in the case and can be used to replay the error, letting the programmer discover the problem unfold in real-time. There is a lot of dynamic analysis techniques of finding data races, most of them based of lockset analysis [22] or happens-before techniques [8], and some both in an hybrid [18]. Most notable is RVPredict [8], which loosens the happens-before notion to get a maximal data race detector, which finds more data races than any of the other techniques.

During a visit at UCLA I worked closely with Mahdi Eslamimehr and Jens Palsberg. Their research worked by combining a concolic engine [24, 10] with dynamic analysis. Concolic is the contraction of the two words *concrete* and *symbolic*. The basic idea is to concretely run a program, while symbolically recording the execution. This allows them to get a better path coverage than running the analysis once or a few times at random [3]. They used output of a static analysis as a guide for their data race detector, and this enabled them to use the concolic engine to direct the execution towards possible data races.

1.3 Contributions

During this project, I build upon Eslamimehr and Palsberg's research on multiple fronts; I have implemented a better data race detector and improved the concolic engine to get a better coverage of the program.

The data race detector used in their project, where based on the older *said* algorithm[21]. I have implemented a version of the `huang` algorithm [8], which improves the coverage.

The concolic execution used in their projects, where unable to correctly produce schedules from parallel programs, this made the coverage incomplete. In this thesis, I have devised this new hyperconcolic engine, which works in a parallel environment. To argue about the coverage of the concolic and the *hyperconcolic engine*, I have developed a notion called schedule classes. Schedule classes are abstract representations of the set of schedules that an analysis covers. The hyperconcolic class ($[h]^{hc}$) is stricter defined than the concolic class, because Hyperconcolic chooses its scheduler. Using this framework of schedule class it is possible to argue if the combination of dynamic analysis and schedule generator is sound. Using the framework, I have argued that the hyperconcolic engine paired with the `huang` data race detector is both sound and complete in programs that for all inputs terminates, and I have also proposed a technique for testing those that do not.

A schedule intermediate format has been developed, because the hyperconcolic engine covers a larger pipeline of analyses.

The main contributions of this thesis are these five points.

- A new schedule format, which language independent can represent symbolic schedules.
- A survey of dynamic analysis, especially the ones detecting parallel errors, and data races. An implementation of the newest techniques, `huang` and `said` has been made.
- Description of the first combination of partial order constraints and input constraints in the context of concolic execution, to my knowledge. Thus enabling real concurrent support to the concolic engine. The resulting concolic engine is called the *hyperconcolic engine*.
- A combination of the hyperconcolic engine and the data races analysis, which is both sound and complete for all programs with a bounded length of schedules. A mathematical framework for arguing about the soundness of the combination is also proposed.
- A symbolic virtual machine of Java programs, which produces schedules in the new language. The machine logs the output of multiple threads and is capable of forcing a special scheduling.

CHAPTER 2

Symbolic Schedule Language

This chapter introduces the symbolic schedule language, both in mathematical notations and also in a data format. The symbolic schedule will be used in the chapters, both to argue about dynamic algorithms and to prove properties for them. The schedule classes are introduced, as the last part of the chapter. The schedule classes serves as the theoretical foundation for arguing about maximality of the dynamic analyses and coverage of the concolic engines.

This section will use a functional approach to build the execution model. All functions are type annotated, to make the ideas of the functions easier to understand. The resulting schedule language is a symbolic reduction of the Java engine instruction set, but should be able to represent all programs that can be reduced to the execution model. An example of the concrete language can be found in Appendix A.

2.1 The Execution Model

We have to define the structure of the programs that we are investigating, to understand the structure of schedules. It is possible to abstract away a lot of the gritty details of the separate languages and focus on the core, using a general execution model.

The main focus of the project is bytecode languages like assembly, LLVM IR and Java bytecode. Even though they have different execution models, all of these languages are ways to represent a program (`P`) that is build up of statements (`S`). In the model, a `P` can be completely described by an initial statement.

$$\texttt{init} : \texttt{P} \rightarrow \texttt{S} \tag{2.1}$$

Statements

A statement in this model has the ability to do two things. It can choose the next statement to be executed (Equation (2.2)), or none to indicate the end of execution, and it can update the state of the execution (Σ) (Equation (2.3)). Both of these actions has knowledge of the current state.

$$\texttt{next} : \texttt{S} \to \Sigma \to \texttt{S} \cup \{\epsilon\} \tag{2.2}$$
$$\texttt{exec} : \texttt{S} \to \Sigma \to \Sigma \tag{2.3}$$

It is possible to build a simple sequential execution model, from these functions alone:

$$\begin{array}{lcl} \texttt{run}_{seq} & : & \texttt{P} \to \Sigma \to \Sigma \\ \texttt{run}_{seq}\ p\ \sigma & = & \texttt{run}'_{seq}\ (\texttt{init}\ p)\ \sigma \\ \\ \texttt{run}'_{seq} & : & \texttt{S} \cup \{\epsilon\} \to \Sigma \to \Sigma \\ \texttt{run}'_{seq}\ \epsilon\ \sigma & = & \sigma \\ \texttt{run}'_{seq}\ s\ \sigma & = & \texttt{run}'_{seq}\ (\texttt{next}\ s\ \sigma)\ (\texttt{exec}\ s\ \sigma) \end{array} \tag{2.4}$$

$\texttt{run}_{seq}$ is a recursive function that a program and a state returns a new state. The top level function $\texttt{run}_{seq}$ calls the helper function with the initial statement of the program $\texttt{run}'_{seq}$. The helper function takes a statement or no statement and a state and returns a state. If the execution is not done (ϵ is presented as the statement) will $\texttt{run}'_{seq}$ executed the statement on the state and find the next statement. The function then calls itself with these updated values. Notice, that there is no way to guarantee that the program will terminate, but if it does it will do so with an updated state.

Scheduler

We can describe programs with branches and loops using this model, but they are limited to sequential execution. The model does not allow us to represent concurrent programs. Concurrent execution requires different execution contexts, threads ($\texttt{T}$), which each execute statements in a serial manor. The threads can present certain parts of the state local to the thread. These substates are referred to with the thread as an index (σ_t), and to some degree be overlapping.

The execution needs to maintain a mapping from threads to the next statement to be executed at all times in the execution. If the thread has ended or not started it will map to nothing ($\texttt{J} := \texttt{T} \to (\texttt{S} \cup \{\epsilon\})$). This mapping is called the current jobs of the execution. To manage these jobs we need a scheduler (Ψ). The scheduler is the algorithm, which the computer uses to schedule the program. The schedulers responsibility is to choose the next thread to be executed from the jobs:

$$\begin{array}{lcl} \texttt{choose} & : & \Psi \to \Sigma \to \texttt{J} \to \texttt{T} \cup \{\epsilon\} \times \Psi \\ \texttt{choose}\ \psi\ \sigma\ js & = & \texttt{choose}'\ \psi\ js' \\ & & \textbf{where}\ js'[x \mapsto js\ x\ \textbf{if}\ \texttt{ready}\ (js\ x)\ \psi\ \sigma_x] \end{array} \tag{2.5}$$

The `choose` function can from a scheduler, a state and a job mapping find a tuple; the next thread to be executed, and an updated scheduler. The function is a combination of the two functions `ready` and $\texttt{choose}'$. The function `ready` indicates

with a boolean if the statement is ready to run and $\texttt{choose}'$ choose picks a thread from the active jobs to execute.

$$\texttt{ready} : \mathtt{S} \to \Psi \to \Sigma \to \{\top, \bot\} \tag{2.6}$$

$$\texttt{choose}' : \Psi \to (\mathtt{T} \to \mathtt{S}) \to \mathtt{T} \cup \{\epsilon\} \times \Psi \tag{2.7}$$

Some languages embeds requirements to the scheduler in the statements, this can be done in the terms of locks or join commands. The semantics of these statements require that they are not executed before some state or execution history has been reached, in these cases would `ready` then return false.

If and only if the statement is ready to be executed, `ready` will return true, which will include the statement in the partial ordering of active jobs.

The $\texttt{choose}'$ function returns a pair of a job and a scheduler. The scheduler is updated because the scheduler may need to react on the statements which has already been run. This is the case in a round-robin scheduler where each thread is executed in turn, the scheduler therefore has to have knowledge about which thread was executed last and for how long. The function can return the empty thread, because the $\texttt{choose}'$ function might get an empty mapping in some cases when none of the statements are read.

The mathematic scheduler is a good way to describe the nondeterminism of choices made by the computer during the execution of multi-threaded programs. It is possible to introduce nondeterminism in a mathematically sound way, by using an unknown scheduler in the equations. The scheduler is constrained by the semantics of the program. This means that the statements has the ability to determine some of the choices of the scheduler.

We only need one more function, for the execution to be complete. The statements can start new threads, this is represented by the `spawn` function. The function returns a set of statements (indicated by the power set type), because statements might be able to start multiple threads at once. The statements indicates the initial statement of the new thread.

$$\texttt{spawn} : \mathtt{S} \to \Sigma \to 2^{\mathtt{S}} \tag{2.8}$$

Execution

We can update the `run` function to also handle concurrent programs, with the help of the scheduler and the active jobs mapping:

$$
\begin{array}{lcl}
\texttt{run} & : & \texttt{P} \to \Psi \to \Sigma \to \Sigma \\
\texttt{run}\ p\ \psi\ \sigma & = & \texttt{run}'\ (\texttt{init}'\ p)\ \psi\ \sigma \\
 & & \\
\texttt{run}' & : & 2^{\texttt{J}} \to \Psi \to \Sigma \to \Sigma \\
\texttt{run}'\ \emptyset\ \psi\ \sigma & = & \sigma \\
\texttt{run}'\ js\ \psi\ \sigma & = & \texttt{run}'\ (\texttt{manage}\ \psi\ \sigma_t\ t\ js)\ \psi'\ (\texttt{exec}\ (js\ t)\ \sigma_t) \\
 & & \textbf{where}\ (t, \psi') = \texttt{choose}\ \psi\ \sigma\ js \\
 & & \\
\texttt{manage} & : & \Psi \to \Sigma \to \texttt{T} \to \texttt{J} \to \texttt{J} \\
\texttt{manage}\ \psi\ \sigma\ t\ js & = & js[t \mapsto \texttt{next}\ (js\ t)\ \sigma] \uplus_{\psi} \texttt{spawn}\ (js\ t)\ \sigma
\end{array}
\tag{2.9}
$$

The notable difference from $\texttt{run}_{seq}$ is, that it accepts a scheduler as an input. What actually happens is that the run function for each step of the execution maintains the current jobs of the execution, which the scheduler manages.

The first step of the execution model is the to find the initial jobs of the program ($\texttt{init}'$). This version of `init` is annotated because it returns a mapping, possible $[\texttt{T}_1 \mapsto \texttt{init}\ p]$.

The scheduler choose a thread to execute from the jobs, using the `choose` function. Then, statement is executed and the jobs of the next execution step is managed by the helper function `manage`. This execution model does not handle if `choose` returns the empty thread. That situation infers a deadlock and the program never ends.

The helper function replaces the executed threads statement with the new statement and adds all the spawned statement to the mapping, using new threads ($\uplus_{\psi}$).

Limitations and assumptions

This model has certain limitations:

1. This execution model requires that all events in some way can be serialized, i.e. executed after each other. This is not always the case. If the memory model of the computer is not sequential consistent, some executions might not be possible in our execution model [12].

2. The state cannot be changed throughout the execution of the program, this means that it is impossible to model true interactive programs with this model. It can on the other hand be simulated by modeling the user as a part of the program.

2.2 Schedules

So far we have worked with statements, which are static information, but to perform dynamic analysis we have to log data into a dynamic context. The event (`E`) solves

this problem. An event is an statement executed in a thread, in the context of a state:

$$\mathtt{E} \quad := \quad \mathtt{T} \times \mathtt{S} \times \Sigma \tag{2.10}$$

The event has the properties; a statement ($\mathtt{S}_e$), the thread ($\mathtt{T}_e$), and the state it where executed in (Σ_e):

$$\begin{array}{lll} \forall e \in \mathtt{E}: & \mathtt{S}_e = s & \textbf{where } e = (t, s, \sigma) \\ & \mathtt{T}_e = t & \\ & \Sigma_e = \sigma & \end{array} \tag{2.11}$$

A total ordering of events inside a thread is called a trace ($\mathcal{T}$) and a total ordering of events outside a thread is called a schedule ($\mathcal{H}$) (The $\mathcal{H}$ symbol is used of historic reasons. It were previously called history. It also avoids collisions with the state and statement, which also begin with "S"). The schedule is defined as a tuple containing the events and a total ordering of the same event, but is easiest represented as a list.

$$\mathcal{H} := 2^{\mathtt{E}} \times 2^{\mathtt{E} \times \mathtt{E}} \tag{2.12}$$

It is possible to work with the schedule as a tuple or as a list:

$$\forall h \in \mathcal{H} : h = (\mathtt{E}_h, <^h) \tag{2.13}$$

$$\forall h \in \mathcal{H} : |h| = |\mathtt{E}_h| \wedge e \in h \iff e \in \mathtt{E}_h \tag{2.14}$$

It is possible to define the event order ($\mathcal{O}^h_x$) and the trace total ordering ($<^{\mathcal{T}}_h$). The trace total ordering is the order of the events inside a single thread:

$$\begin{array}{rcl} \forall h \in \mathcal{H}, i \in 1..|h| & : & h_i = e \iff e \in h \wedge i = \left|\left\{e' \mid e' \in \mathtt{E}_h, e' <^h e\right\}\right| + 1 \\ \forall h \in \mathcal{H} & : & \mathcal{O}^h_{h_i} = i \\ e <^{\mathcal{T}}_h e' & \iff & e <_h e' \wedge \mathtt{T}_e = \mathtt{T}_{e'} \end{array} \tag{2.15}$$

A schedule can be generated from a program using an initial state (Σ) and a scheduler (Ψ):

$$\begin{array}{lcl} \mathtt{trace} & : & \mathtt{P} \to \Psi \to \Sigma \to \mathcal{H} \\ \mathtt{trace}\ p & = & \mathtt{trace}'\ (\mathtt{init}\ p) \\ \\ \mathtt{trace}' & : & \mathtt{J} \to \Psi \to \Sigma \to \mathcal{H} \\ \mathtt{trace}'\ \emptyset\ \psi\ \sigma & = & (\emptyset, \emptyset) \\ \mathtt{trace}'\ js\ \psi\ \sigma & = & (t, (js\ t), \sigma) \ll \mathtt{trace}'\ (\mathtt{manage}\ \psi\ \sigma_t\ t\ js)\ \psi'\ (\mathtt{exec}\ (js\ t)\ \sigma_t) \\ & & \textbf{where } (t, \psi') = \mathtt{choose}\ \psi\ \sigma\ js \end{array} \tag{2.16}$$

The equation for the calculation of the schedule is like the `run` function, see 2.9, but an event is added for each thread chosen by the `choose` function, using the list creator $\ll$. The list creator indicates that the first operand is smaller, or before, all elements in the second operand.

Using the `trace` function we can derive the program property ($\mathtt{P}_h$), indicating all programs that can produces the schedule h:

$$p \in \mathtt{P}_h \iff \exists \psi \in \Psi, \sigma \in \mathtt{S} : h \in (\mathtt{trace}\ p\ \psi\ \sigma) \tag{2.17}$$

The inverse construct is called the schedules of a program $\mathcal{H}_p$:

$$h \in \mathcal{H}_p \iff h \in \mathcal{H} \wedge p \in \mathtt{P}_h$$

2.3 The Language

Instructions are the building block of statements in most languages. They are stencils that we can add intend to and they can be combined to generate statements and finally programs. The mathematical model, presented in the last sections, is abstract. It allows various definitions of statements, and infinitely many different instructions. In real life languages this is not the case. In Java byte code there exist 198 different instructions. It is possible to generate Turing complete code with only one instruction (SUBLEQ; Subtract or Branch if less than or equal to zero) applied with different arguments. I propose a middle way, with a limited instruction set of 19 instructions, see Figure 2.1.

State model

The limited instruction set referees to a state model yet to be defined. There are two different contexts of the state. The shared state and the local state. The local state is the state only accessible by the thread. The values from this space are called local values, or locals for short.

The global state consists of instances, which are packs of variables accessible through keys. The variables also holds values, but the values are able to change. The combination of instance and variable is called a target.

The notion of internal and external, indicates whether the function is treated as a black box, or is calling code known to the model.

Definition

The goal of the language is to keep it as simple as possible, and to stay complete to the underlying model. Even though a simple language might loose some information, that could have been presented in a complex language like Java, it helps us keep the model simple help us stay faithful to it during the analyses.

Instruction	Description
`begin`	A thread has been started.
`end`	A thread has been ended.
`fork`	Creates a new thread.
`join`	Wait for thread.
`acq`	Acquire a lock.
`rel`	Releases a lock.
`let`	Sets a local value with a constant.
`new`	Create an new instance.
`symbol`	Creates a new symbol.
`read`	Reads a local from target.
`write`	Writes a local to target.
`branch`	Indicated that a choice of a branch were made.
`binopr`	Binary operation on local value.
`unopr`	Unary operation on local value.
`call`	Calls an external function.
`voidcall`	Calls an external function, with no return value.
`enter`	Enter an internal function.
`exit`	Exit an internal function with a value.
`voidexit`	Exit an internal function without a value.

Figure 2.1: Table of the limited instruction set, used in this thesis.

The model has 4 undefined functions for each statement; `next`, `exec`, `ready` and `spawn`. The default behaviour for the last three functions are:

$$\begin{aligned} \texttt{exec}\ \_\ \sigma &= \sigma \\ \texttt{ready}\ \_\ \psi\ \sigma &= \top \\ \texttt{spawn}\ \_\ \sigma &= \emptyset \end{aligned}$$

These functions will not be defined in this thesis, as their exact definitions are not important for an analysis of the schedules and traces. I have faith that it is a strait forward task for future publications.

Events

In the language there exist 19 different instructions, some of them, like `enter` and `exit` only exist for helping analyses. The language natively supports lookup tables as the only storages, but these can fully support simulation of higher order objects. Also statement for acquiring and releasing locks has been included.

It is impractical include the entire state in all events, partly because of the redundancy between each step, and because of the huge space required to capture the entire state. Because we have a more concrete model, is it also possible to make the events more concrete.

Events in this specialized model can be represented as a triple, thread, operation and annotation. The operation is a limited symbolic description of a combination of the statement and the state. The operations strips away the control flow data from the statement and concretes the accesses. An operation does therefore only make sense when ordered in a schedule.

Because we represent the state in a much simpler notation, annotations has been added to cover parts of these states, which we did not cover. The events representation is therefore not sound in respect to the entire state, but might be sound in respect to the analysis.

The operations

The names of the operations the are same as for the instructions. The instructions uses symbolic values to operate. The symbolic value is unique in the system and is only written to once, but can be read from many times.

- The begin and end operations contains no informations and exists to make concrete starts and ends of the traces.
- The `fork` and `join` starts and waits for another thread. This requires knowledge of that thread. In this thesis it will be referred to as the child of the event (e_{child}).

- The `read` and `write` events, they references a variable that the write to ($e_{variable}$) and they both store the concrete value read or written (e_{val}) and the symbolic value (e_{sym}).
- The last operation pair that I will introduce is the `acq` and `rel` operations. They both have knowledge about the lock which they are acquiring or releasing (e_{lock}), and they also have a version number, indicating in which order they were locked (e_v).

We are able to, using these operations alone, to find data races in a schedule. The other operations exit to track the symbolic state of the program, or to help to produce stack traces, showing the depth of the execution.

Notation

When talking about this format and this model we will use some simplifying notation. The operation name of an event can be found using a_{opr}.

$$e_{opr} = \texttt{begin}$$

The set of all events containing a specific operation in a schedule, is denoted as the schedule with the operation names as indices.

$$h_{\texttt{write},\texttt{read}} = \{e \mid e \in h, e_{opr} \in \{\texttt{write}, \texttt{read}\}\}$$

2.4 Symbol Classes

One of the key parts of this thesis is the combination of data races detectors and concolic engine. Schedule classes is a useful tool to understand the relationship between the coverage of the schedules and the coverage of the analysis is the schedule classes.

Symbolic Schedules

When doing analysis of schedules, the actual state of the program at certain point in the execution is often less important than where the data came from. Adding some degrees of freedom to the representation of the schedules helps some analyses get better solutions. Symbolic schedules are schedules with no state.

To get symbolic schedules we need a notion of symbolically similar events. Symbolically similar similar events are events which derives from same thread-statement pair.

$$e \sim_{\mathtt{E}} e' \iff (\mathtt{T}_e, \mathtt{S}_e) = (\mathtt{T}_{e'}, \mathtt{S}_{e'})$$

Two schedules are symbolically similar if and only if all their events are symbolically similar and in the same order.

$$h \sim h' \iff |h| = |h'| \wedge \forall_{i \in 1..|h|} : h_i \sim_{\mathtt{E}} h'_i$$

It is possible to evaluate a symbolic schedule, the states of in the schedule has been abstracted away:

$$\begin{aligned}
&\texttt{eval} && : \quad \mathcal{H} \to \Sigma \to \Sigma \cup \{\epsilon\} \\
&\texttt{eval}\ h\ \sigma_0 && = \quad \text{if } \exists \psi_0 \in \Psi, \forall i \in 1..|h| : \begin{cases} \sigma_i = \texttt{exec}\ \mathtt{S}_{h_i}\ \sigma_{i-1} \\ \mathtt{S}_{h_i} = js_{i-1}\ \mathtt{T}_{h_i} \\ js_i = \texttt{manage}\ \psi_i\ \sigma_{i-1}\ \mathtt{T}_{h_i}\ js_{i-1} \\ (\mathtt{T}_{h_i}, \psi_i) = \texttt{choose}\ \psi_{i-1}\ js_{i-1} \end{cases} \\
& && \quad \text{then } \sigma_{|h|} \text{ else } \epsilon
\end{aligned} \tag{2.18}$$

The evaluation of a schedule has been extracted from Equation (2.9). The function shows that besides some input to a symbolic schedule might not return a resulting state. This is either because the execution has been ended or that another branch in the program where followed due to the different state.

Theorem 2.1. *All symbolically similar events has similar executions.*

$$h \sim h' \implies \forall \sigma \in \Sigma : \texttt{eval}\ h\ \sigma = \texttt{eval}\ h'\ \sigma$$

We can now derive different abilities for schedules using only their symbolic counterparts. The symbolic executions of the schedule ($\vec{\Sigma}_h$) is the set of pairs of start, end states, which are possible, using a symbolic execution of schedule. The last derived property is symbolic traces ($\mathcal{T}^\star{}_h$), which is the set of pairs representing the symbolic traces of the schedule.

$$\begin{aligned}
(\sigma, \sigma') \in \vec{\Sigma}_h &\iff \texttt{eval}\ h\ \sigma = \sigma' \\
(t, <_e) \in \mathcal{T}^\star{}_h &\iff s <_e s' \Leftrightarrow (t, s, _) <^h (t, s', _)
\end{aligned}$$

Schedule Classes

Like the actual state of the executed event, the order of the events in the schedule does not always matter. Schedule classes are used to illustrate this.

Schedule classes ($[h]$) is the grouping of all schedules that are similar to h in some way. The most general schedule class is the *trace similar* class. All schedules in this class contains the same traces of h. This class covers all different combinations of the schedule, even those which is not runnable in a program.

$$h' \in [h] \iff \mathcal{T}^\star{}_{h'} = \mathcal{T}^\star{}_{h'}$$

The next class is all permutations of the schedule that are executable in the same programs. All schedules in $[h]^{\mathtt{P}}$ are *semantically similar*.

$$h' \in [h]^{\mathtt{P}} \iff h' \in [h] \wedge p \in \mathtt{P}_{h'} \wedge p \in \mathtt{P}_h$$

A subset to $[h]^{\mathtt{P}}$ are all schedules that have *symbolically similar executions* . This class is the equivalent to all the interleaving that could occur without changing the result of the execution.

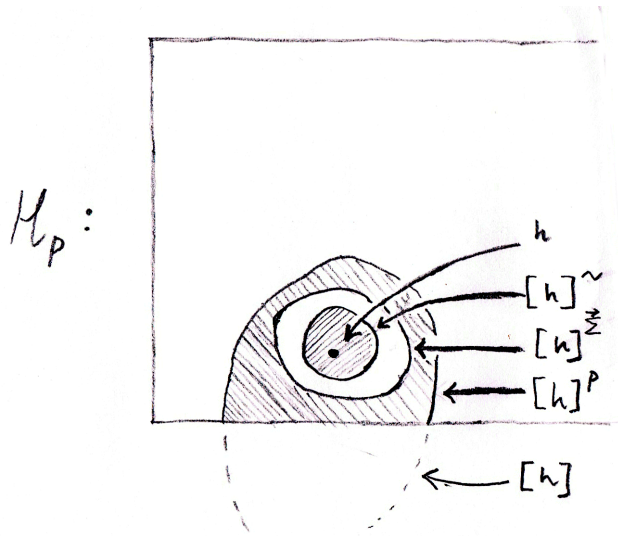


Figure 2.2: The arangement of schedule classes in relations to all schedules producable from a program ($\mathcal{H}_p$).

$$h' \in [h]^{\vec{\Sigma}} \iff h' \in [h]^{\mathrm{P}} \wedge \vec{\Sigma}_h = \vec{\Sigma}_{h'}$$

The most restrictive class is class of *symbolically similar schedules*, which where presented in the last section, not only is the result of the execution important but also the order of the scheduling.

$$h' \in [h]^{\sim} \iff h' \in [h]^{\mathrm{P}} \wedge h \sim h'$$

Properties of Classes

The most fundamental properties to notice are that all of these classes there is at least one schedule in each class and that if one schedule is the member of the others class is the first also the member of the second class.

$$h' \in [h]^{\star} \iff h \in [h']^{\star}$$

We can use these classes to determinate different abilities about the program that they occur in. The most essential is their internal relations.

Theorem 2.2. *For any $h \in H$ it is the case, that all its classes is ordered like this:*

$$[h]^{\sim} \subseteq [h]^{\vec{\Sigma}} \subseteq [h]^{\mathrm{P}} \subseteq [h]$$

Proof. $[h]^{\mathsf{P}} \subseteq [h]$ and $[h]^{\vec{\Sigma}} \subseteq [h]^{\mathsf{P}}$ are deductable from the definition itself, and does not require a deeper argument. We use a proof by contradiction to show $[h]^{\sim} \subseteq [h]^{\vec{\Sigma}}$. Assume there exist a schedule that is symbolically similar to h, but does not have a symbolically similar execution.

$$h' \in [h]^{\sim} \wedge h' \notin [h]^{\vec{\Sigma}} \iff h \sim h' \wedge \vec{\Sigma}_h \neq \vec{\Sigma}_{h'}$$

Theorem 2.2:

$$\begin{aligned} h \sim h' &\implies \forall \sigma \in \Sigma : \texttt{eval}\ h\ \sigma = \texttt{eval}\ h'\ \sigma \\ &\implies \forall \sigma, \sigma' \in \Sigma : (\sigma, \sigma') \in \vec{\Sigma}_h = (\sigma, \sigma')\vec{\Sigma}_{h'} \\ &\implies \vec{\Sigma}_h = \vec{\Sigma}_{h'} \end{aligned}$$

We have shown a contradiction. $h' \in [h]^{\sim} \wedge h' \notin [h]^{\vec{\Sigma}}$ is imposible. This proves all parts of the theorem. □

Theorem 2.3. *If a program is **sequential**, means that at there is only one thread in the program, then*

$$[h]^{\sim} = [h]^{\vec{\Sigma}} = [h]^{\mathsf{P}} = [h]$$

Proof. If the program is **sequential**, then at no point will there exist mutiple jobs, which, means that the scheduler does not influence the generation of traces:

$$\forall \sigma \in \Sigma, \exists h \in \mathcal{H}, \forall \psi \in \Psi : \texttt{trace}\ p\ \psi\ \sigma = h$$

Since there is only one trace in the a sequential program, all schedules must contain only one trace. Since the trace is a total order and all classes derives from the class of similar traces. All schedule classes is eqaul to the class of similar traces. □

Theorem 2.4. *If a program is **deterministic**, meaning that at no point can the scheduler make a choice which changes the execution, then*

$$[h]^{\vec{\Sigma}} = [h]^{\mathsf{P}}$$

Proof. If the program is **deterministic**, the does the scheduling not influence the execution of the program, meaning that all changes in scheduling have no effect on the end result.

$$\forall \sigma \in \Sigma, \exists x \in \vec{\Sigma}, \forall \psi \in \Psi, \exists h \in \mathcal{H} : \texttt{trace}\ p\ \psi\ \sigma = h \wedge \vec{\Sigma}_h = x$$

$[h]^{\vec{\Sigma}} = [h]^{\mathsf{P}}$ is directly deriverable from the defintion of $[h]^{\vec{\Sigma}}$, which depends on the program, and that the execution is the same. □

Theorem 2.5. *If a program is **nondeterministic**, meaning that the execution depends on the choices made by the scheduler, then*

$$[h]^{\vec{\Sigma}} \subset [h]^{\mathsf{P}}$$

Proof. If follows from the definition that when the execution depends on the choices made by the scheudler, that there exist a scheduler which will give another execution. The schedule created by tracing the porgram with that scheudler, must therefore be in $[h]^{\mathtt{P}}$ but not in $[h]^{\vec{\Sigma}}$. $[h]^{\vec{\Sigma}}$ must be a strict subset of $[h]^{\mathtt{P}}$, because we know that $[h]^{\vec{\Sigma}} \subseteq [h]^{\mathtt{P}}$ from Theorem 2.2. □

Loosely based on the works of Netzer and Barton [17], we can define the data race, and the general race using the schedule classes.

Definition 2.1. *An event pair (e, e') is conflicting if changing the order they execute in would change the state.*

$$\exists \sigma \in \Sigma : (\mathtt{exec}\ \mathtt{S}_e \circ \mathtt{exec}\ \mathtt{S}_{e'})\ \sigma \neq (\mathtt{exec}\ \mathtt{S}_{e'} \circ \mathtt{exec}\ \mathtt{S}_e)\ \sigma$$

Definition 2.2. *A data race (e, e') exists in a program p, if for any schedule in $\mathcal{H}_h$ with e and e' ordered right after each other, and there exists a reordering in $[h]^{\mathtt{P}}$ where e and e' are executed in a different order but is still next to each other, and (e, e') are conflicting.*

$$\exists h \in \mathcal{H}_p : h' \in [h]^{\mathtt{P}} \wedge \mathcal{O}_e^h + 1 = \mathcal{O}_{e'}^h \wedge \mathcal{O}_{e'}^{h'} + 1 = \mathcal{O}_e^{h'}$$

Definition 2.3. *A general race (e, e') exists in a program p, if for any partial schedule in $\mathcal{H}_p$ exists a reordering, ordering the different, and (e, e') are conflicting.*

$$\exists h \in \mathcal{H}_p : h' \in [h]^{\mathtt{P}} \wedge e <_h e' \wedge e \not<_{h'} e'$$

The difference, besides their definition, between the general race and the data race is that the data race requires that the two events must not be ordered by a event in any way, where the general race just requires that the ordering is different. Both of these races causes nondeterminism. The nondeterminism caused by the general race is often by design, and data races is more often a problem [15].

Class Coverage

Some dynamic analyses needs a sample of each schedule from the program to guarantee completeness of the analysis. This quickly become unfeasible when exploring all combination of the state space, or when checking all interleavings introduced by parallel programs. This is where the schedule classes comes in handy. Some analyses only need a single schedule from each class to be complete. The completeness of these methods thereby greatly depends on the coverage of these classes.

The finest coverage of executions are *schedule coverage*, meaning that all concrete schedules, possible to find are found and analysed. This kind of coverage is impossible in any program where the state space is infinite.

A better notion of coverage is *symbolic schedule coverage*. This coverage is indifferent to concrete values in the schedule, and an only uses order of the events in the schedule. Covering schedules from all the symbolically similar classes ($[h]^{\sim}$), gives full *symbolic schedule coverage*.

Given the complexity of covering all interleaving of the schedules, an other notion of coverage is introduced, *path coverage*. In this coverage class a schedule from each program or semantically similar class ($[h]^{\mathsf{P}}$) is extracted. This filters out all one of the many interleaving and only covers those which can results in different paths through the system. One of these classes can be referred to as a path.

Theorem 2.6. *It is possible to get full symbolic schedule coverage, and therefore also full path coverage (by using Theorem 2.2), in all programs with a bounded schedule length.*

The proof builds on two assumptions, that programs have a finite set of set of statements, which is true except in cases of self modifying programs, and that the threads are inclemently numerated by the scheduler.

Proof. There is a limited number of possible statements in a program. The there is also atmost the length of the schedule different threads. Since the length of the schedule is bounded, there exists a limited number of choices when chosing the ordering or symbolic events making up the schedule. □

Limitations

Path coverage is much better than the two prior in terms of number of classes to cover, but can also give infinitely many classes in some cases. This program produces infinitely, or as many as we can represent in `i`, different classes:

```
i = getInteger()
while (i >= 0) {
    i = i - 1
}
```

When looking at the input we can see that different inputs would produce schedules of different lengths. Schedules containing different events cannot be in the same class. The example would therefore produces different classes for each input, which would cause an infinitely large class space.

Infinitely large class spaces, forces dynamic analyses to either run infinitely or terminate with an incomplete analysis. This raises some questions:

- Can we improve the class model to abstract away loops with determinable semantics?
- Is it possible to select the classes most likely to produce results when analysed?

I will postpone these questions future work, but a short description on a approach to searching through classes is presented in Section 4.3.

2.5 Related Work

Not a lot of work has been done in the area of schedule representations in the past. Most articles covering dynamic analysis does handle their schedules in an internal representation [20] where others do computations or proofs on theses schedules [8, 9, 3, 21, 10, 18]. Even though that these techniques have different goals, they work with the same core data. The dynamic generation techniques working with *concolic* execution is often only interested in maintaining the symbolic state of the program, while the data race detectors often work on the orders of events. Combining the data in one language enable the concolic engine to know about the ordering and the data race detectors to know about the symbolic state.

2.6 Summary

In this chapter I have introduced the execution model of the project, and how it can be used to create schedules. I have organized these schedules into classes, which I use in the coming chapters to prove properties about the analyses and the combination of the analyses and concolic execution. Another key point were path coverage which will become key in Section 4.1, about concolic execution.

The model where also made more concrete by adding an underlying language. From this language I defined some concrete events. The Events and the notations defined here will be used in the next chapter.

The key contribution of this chapter is the introduction of the symbolic schedule language that enable us to argue about complex relations between threads in an abstract machine.

CHAPTER 3

Dynamic Analyses

The data race detection analyses of the project is covered by this project. All of these analyses are based on state of the art techniques, but have been modified to fit using the symbolic schedule language, defined in Chapter 2. This is therefore both a test of the schedule language, and a recap of the techniques used to generate data races. I will mostly cover the happen-before approach, but will also mention the lock-set analysis.

3.1 Happens-Before Relations

This section is the theoretical background for detecting data races using happens-before analysis. The happens-before relations is a way of describing a schedule class using a partial ordering. As described in Equation (2.12), is a schedule a tuple of a set of events and a total ordering. Using a single partial ordering of events, it is possible to generate a set of schedules. The best coverage is achieved by being to be as close to the $[h]^{\mathsf{P}}$ as possible.

The $\prec$ is a partial ordering of events, known as a happens-before relation [13]. It represents the order that the events have to happen in to be correct in respect to the semantics of the program. The happens-before relations are *transitive*:

$$\forall e, e', e'' \in h : (e \prec e' \wedge e' \prec e'') \Rightarrow e \prec e''$$

In the literature is the relations often parted in three different kinds of relations [21, 8]; the must-happen-before relation ($\prec^{mhb}$), the lock relation ($\prec^{lock}$) and the read-write consistency relation ($\prec^{rw}$).The must-happen-before relations covers the happens-before relations that arises from the constructs of the language, a `fork` event should come before a `begin` event and that a thread in itself should be executed in serial. The lock relations ($\prec^{lock}$), requires that atomic sections are not overlapping. Lastly, the read-write consistency ($\prec^{rw}$) ensures that the order and values of the reads and writes, remain consistent of, ensuring that the schedule remain executable. Figure 3.1 illustrates two threads and the happens-before relation ships between them. These three happens-before relations can combined to a single happens-before relation:

$$\prec_{h,\omega} = \prec_{h}^{mhb} \cup \prec_{h,\omega}^{lock} \cup \prec_{h,\omega}^{rw} \tag{3.1}$$

The h in $\prec_{h,\omega}$, indicates that the $\prec$ relation a function of the information in h. This is what makes this a dynamic analysis. For some of these terms it is possible

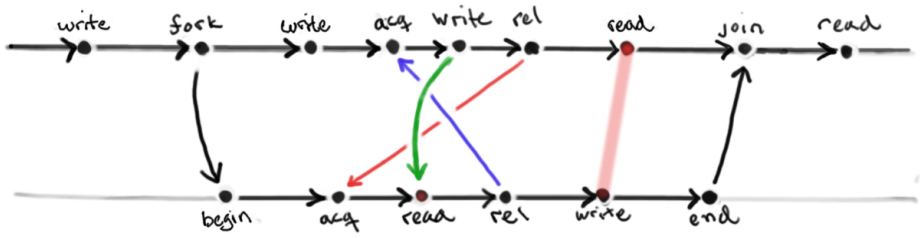


Figure 3.1: Happens-before relations between two threads. $\prec_{mhb}$ is black, $\prec_{lock}$ is blue or red, and $\prec_{rw}$ is green. A data race is illustrated with a red diffuse line.

to generate multiple different partial orders. ω represents an external total ordering which helps to resolve ambiguities. The total ordering has two parts, a lock total ordering (ω_{lock}) and a read-write consistency ordering (ω_{rw}). The ω is chosen among a set of allowed orders ($\Omega^\star$), a schedule class $[h]^\star$. Because the happens before relations can produce different classes, the schedule class is named after the ordering.

$$h' \in [h]^\star \iff \exists \omega \in \Omega^\star : \prec_{h,\omega} \subseteq <_{h'} \tag{3.2}$$

Must-happen-before

In my language, `fork` and `begin`, `end` and `join` and all events of a single trace have to be ordered, for the program to be semantically sound.

$$e \prec_h^{mhb} e' \iff \bigvee \begin{cases} e <_h^{\mathcal{T}} e' \\ e_{child} = \mathtt{T}_{e'} \wedge (e_{opr}, {e'}_{opr}) = (\texttt{fork}, \texttt{begin}) \\ \mathtt{T}_e = {e'}_{child} \wedge (e_{opr}, {e'}_{opr}) = (\texttt{end}, \texttt{join}) \end{cases}$$

The must-happen-before relation is completely extractable from partial information in the traces, and there is no need for any extra total orders of any kind.

Lock

The semantics of the atomic sections allows them to be ordered in any way. This means that they cannot be described by a single partial ordering. Given set of `acq-rel` pairs that generate atomic sections ($\mathbf{A}_h$), the naively $\prec^{lock}$ has to satisfy that:

$$\bigwedge_{(a,r),(a',r') \in \mathbf{A}_h} a_{lock} = {a'}_{lock} \Rightarrow (r \prec^{lock} a' \vee r' \prec^{lock} a)$$

There exist multiple ways to derive the atomic sections of the program. In the most general version we can gather $\mathbf{A}_h$ pairs from the schedule (h):

$$(a,r) \in \mathbf{A}_h \iff \bigwedge \begin{cases} a_{opr} \in h_{\texttt{acq}} \wedge r_{opr} \in h_{\texttt{rel}} \\ a_{lock} = r_{lock} \end{cases}$$

That definition of atomic sections is way to general in respect to Java programs. In these programs are the atomic sections always both released by the same thread. Using this knowledge we can define a much closer representation of the atomic sections.

$$(a,r) \in \mathbf{A}'_h \iff (a,r) \in \mathbf{A}_h \wedge \neg\exists r' \in h_{\texttt{rel}}, \tau \in \mathcal{T}_h : a <_\tau r' <_\tau r$$

The equation can be rewritten to an equation using a total ordering of the atomic sections with the same lock (ω_{lock}):

$$e \prec^{lock}_{h,\omega} e' \iff (e,e') \in \mathbf{A}'_h \vee \exists a,r : (a,e)\ \omega^{lock}\ (e',r) \tag{3.3}$$

There are three different ways to choose ω_{lock}; no lock ordering (Ω^{no}), free lock ordering (Ω^{free}) and strict ordering (Ω^{strict}). Ω^{free} requires that all atomic sections with the same lock are totally ordered. Ω^{strict} requires that the total ordering is the same as in the schedule.

$$\omega \in \Omega^{lock,no}_h \iff \bot \tag{3.4}$$

$$\omega \in \Omega^{lock,free}_h \iff A_{lock} = A'_{lock} \wedge A, A' \in \mathbf{A}'_h \iff A\omega A' \oplus A'\omega A \tag{3.5}$$

$$\omega \in \Omega^{lock,strict}_h \iff \omega \in \Omega^{lock,free} \wedge (a,r)\omega(a',r') \implies r <_h a' \tag{3.6}$$

Read-write Consistency

The purpose of the read-write consistency is to ensure the correct ordering of reads and writes. Depending on the semantics of the analysis can correct ordering mean multiple things, but in most cases should it guarantee that the execution follows the same path throughout the program. The articles, that I have read, present different encodings of this relations [21, 8, 25]. I will try to combine all into one encoding.

The fundamental of the read-write consistency is to ensure that all branches are executed with the same values. If we cannot ensure that, the happens-before order might represent schedules that does not exist. The values of branches are controlled by the local state only, which in turn only can change depending on the scheduling through the `read` events. The value read by the `read` events depends on the ordering of the `write` events, and that the `write` event writes the same values.

Not all reads and writes are problematic, only those known as *conflicting operation pairs*. Reordering these event would cause a change in the state of the execution.

$$(e, e') \in \mathbf{COP}_h \Rightarrow \bigwedge \begin{cases} e, e' \in h_{\texttt{read},\texttt{write}} \wedge e \neq e' \\ \texttt{T}_e \neq \texttt{T}_{e'} \wedge e_{variable} = {e'}_{variable} \\ \texttt{write} \in \{e_{opr}, {e'}_{opr}\} \end{cases} \tag{3.7}$$

Of this set there are two degrees, concrete conflicting or symbolic conflicting pairs. Concrete conflicting operation pairs, have read and written different concrete values, while the symbolic conflicting pairs, have read or written different symbolic values. Using symbolic values over concrete values is a stronger requirement, because symbolic values carry not only the concrete value but also the context of the writes.

$$(e, e') \in \mathbf{COP}_h^{con} \iff (e, e') \in \mathbf{COP}_h \wedge e_{val} \neq {e'}_{val} \tag{3.8}$$

$$(e, e') \in \mathbf{COP}_h^{sym} \iff (e, e') \in \mathbf{COP}_h \wedge e_{sym} \neq {e'}_{sym} \tag{3.9}$$

To combine data race detection with concolic execution, the context of the reads and writes is very important. I will therefore use $\mathbf{COP}_h^{sym}$ as the only model for calculating $\mathbf{COP}_h$. If loosing this requirement gives better results is left up to future work.

Like with the lock, reads and writes can be ordered in different ways, while the program still remains semantically correct. Using a total ordering of the all the events in $\mathbf{COP}_h$ (ω_{rw}), the read-write consistency can be defined like this:

$$e \prec_{h,\omega}^{rw} e' \iff e \ \omega_{rw} \ e' \tag{3.10}$$

In this case I have found four different orderings; no ordering (Ω^{no}), *huang* ordering (Ω^{huang}) [8], the stricter *said* ordering (Ω^{said}) [21], and the strict ordering (Ω^{strict}).

$$\omega \in \Omega_h^{rw,no} \iff \forall e, e' : e \not\omega \ e' \tag{3.11}$$

$$\omega \in \Omega_h^{rw,huang} \implies (e, e') \in \mathbf{COP}_h \wedge e \in \Phi_h \vee e' \in \Phi_h \iff e \ \omega \ e' \oplus e' \ \omega \ e \tag{3.12}$$

$$\omega \in \Omega_h^{rw,said} \implies (e, e') \in \mathbf{COP}_h \iff e \ \omega \ e' \oplus e' \ \omega \ e \tag{3.13}$$

$$\omega \in \Omega_h^{rw,strict} \iff \omega \in \Omega^{rw,said} \wedge \forall w, w' \in h_{\texttt{write}} : w \ \omega \ w' \implies w <_h w' \tag{3.14}$$

For both $\Omega_h^{rw,huang}$ and $\Omega_h^{rw,said}$, it is also required that all of the reads should have dominating write of the same value than itself. In the equation is $r \in h_{\texttt{read}}$ and $w, w' \in h_{\texttt{write}}$ and w and r has the same value, concrete or symbolic depending on the method, and they all have the same variable:

$$\forall r \exists w : w \ \omega \ r \wedge \neg \exists w' : w \ \omega \ w' \ \omega \ r$$

Huang's order reduces the set of ordered events from all in $\mathbf{COP}$ to only be the events that can change the execution [8]. The order uses that that the state of schedules do not have to be consistent after the last branch in each trace for the events to still happen. A point in the execution can always be reached if the last

dominating branches of the point, where consistent, even if races occur between the branch and the point. A loosening filter can be defined Φ_h:

$$\begin{array}{lcl} \Phi_h & \supseteq & \left\{r \mid r \in h_{\texttt{read}}, r <_h^{\mathcal{T}} e, e \in \Phi_h^\star\right\} \\ \Phi_h^\star & \supseteq & \left\{w \mid w \in h_{\texttt{write}}, w_{sym} = r_{sym}, r \in \Phi_h\right\} \\ \Phi_h^\star & \supseteq & \left\{b \mid b \in h_{\texttt{branch}}, b <_h^{\mathcal{T}} e, e \in \Phi_h \cup \Phi_h^\star\right\} \cup \mathcal{B} \end{array}$$

The fixed point uses another fixed point ($\Phi_h^\star$). This fixed point covers all events that has to have a consistent local state to run. The first half of the set mentions all the writes that are required to write the same values. The second half mentions all the branches that has to have a consistent state. Main fixed point (Φ_h) is defined as all the read events that could possible change the local state required by the events in $\Phi_h^\star$. $\mathcal{B}$ is the last executed branch in each thread.

The schedules used in this report also supports symbolic execution, this would allow a tighter choice of $\mathcal{B}$, where the dominating branch do not have to be the first branch, but the first branch symbolically dependent on the data race. Even though it would give a better analysis coverage is the extra span not needed when using hyperconcolic, and has therefore been put off to future work.

Combining the Orders

So now we have a partial order function for the must-happen-before ($\prec_h^{mhb}$), the lock ($\prec_{h,\omega}^{lock}$) and the read-write consistency ($\prec_{h,\omega}^{rw}$) relations. In the program I define four different ordering, describing the different algorithms used in the different articles. The loose ordering (Ω^{loose}) is the simplest of the orderings, and works by not ordering anything.

$$\omega \in \Omega^{loose} \iff \omega_{rw} = \Omega^{rw,no} \wedge \omega_{lock} \in \Omega^{lock,no}$$

The strictest of the orderings orders both locks and read-write events after the order in the thread. The reads can still rearrange themselves.

$$\omega \in \Omega^{strict} \iff \omega_{rw} \in \Omega^{rw,strict} \wedge \omega_{lock} \in \Omega^{lock,strict}$$

After this we have the *said* ordering which is the closest representation of algorithm presented by Said [21], in this framework.

$$\omega \in \Omega^{said} \iff \omega_{rw} \in \Omega^{rw,said} \wedge \omega_{lock} \in \Omega^{lock,free}$$

The *huang* ordering is similar but uses the slightly more permissive read-write consistency ordering set, $\Omega^{rw,huang}$.

$$\omega \in \Omega^{huang} \iff \omega_{rw} \in \Omega^{rw,huang} \wedge \omega_{lock} \in \Omega^{lock,free}$$

Using the model from Equation (3.2), the orders can be presented as schedule classes:

$$\begin{array}{lll} h' \in [h]^{loose} & \iff & \exists \omega \in \Omega_h^{loose} : \prec_{h,\omega} \subset <_{h'} \\ h' \in [h]^{said} & \iff & \exists \omega \in \Omega_h^{said} : \prec_{h,\omega} \subset <_{h'} \\ h' \in [h]^{strict} & \iff & \exists \omega \in \Omega_h^{strict} : \prec_{h,\omega} \subset <_{h'} \\ h' \in [h]^{huang} & \iff & \exists \omega \in \Omega_h^{strict} : \prec_{h,\omega} \subset <_{h'} \end{array}$$

From these classes is would be interesting to set up relations to the general classes reported in the last chapter:

$$\begin{array}{llll} [h]^{\vec{\Sigma}} & \subseteq & [h]^{loose} & \subseteq [h] \\ [h]^{\vec{\Sigma}} & \subseteq & [h]^{said} & \subseteq [h]^{\vec{\Sigma}} \\ [h]^{\sim} & \subseteq & [h]^{strict} & \subseteq [h]^{\vec{\Sigma}} \end{array}$$

These relations are listed without a formal proof. Proving these relations is a requirement, if we are to prove full schedule coverage of the analysis, when used with hyperconcolic.

RVPredict, *huang*, claims to be maximal in respect to the information stored in their schedule format. Their proof is not directly transferable to my cases because the value calculation is different, and their schedule format contains less data than mine. Proving maximality in their format is therefore not the same as proving it in my format.

$$[h]^{\mathsf{P}} = [h]^{huang}$$

3.2 Data Race detection

This section is an overview about how to find data races using the happens-before relations. Also an short description of the lock-set analysis is made for completeness.

Happens-before Analysis

The happens-before analysis, is the finest, but also the slowest of the two analyses presented here. The process is to encode the data race definition, into constraints that a constraint solver can solve. The definition of the data race, see Definition 2.2, is the data race defined as two conflicting pairs with no synchronizing events. The conflicting pairs has already been defined in Equation (3.7), and to encode that no synchronizing events exits, it is often used that there then must exist a schedule were the two events are next to each other. If that is the case we can consider them racing.

$$(e, e') \in \mathbf{COP}_h \wedge \exists h' \in [h]^\star : \mathcal{O}_e^{h'} + 1 = \mathcal{O}_{e'}^{h'} \implies (e, e') \in \mathbf{DR}_h^{hb}$$

$[h]^\star$ is based on one of the happens-before classes defined in the previous section. If $[h]^\star \subseteq [h]^{\mathsf{P}}$, the analysis is considered complete and if $[h]^\star \supseteq [h]^{\mathsf{P}}$, the analysis is maximal in respect to the schedule.

Trying all orderings in an algorithm by brute force, is an almost impossible with almost $O(|h|!)$ different orderings. A constraint solver is therefore used. It is strait forward to transfer the partial orderings to constraint logic. The procedure is to expand all the equations using the schedule, thereby removing all the quantifiers. The happens-before relations is then encoded like $\mathcal{O}_e^{h'} < \mathcal{O}_{e'}^h$. If the constraint solver can give all $\mathcal{O}_e^{h'}$ an integer value, the schedule exists, and thereby also the data race. This solution also generates, witnesses which the user of the program can use to recreate the bug.

Lockset Analysis

Of the two analyses are lockset analysis the easiest. It is a reaction to the Happens-Before analysis which is a hard problem. The solution proposed by the lockset algorithm is to require a strict locking discipline from the programmers point of view [22]. The lockset algorithm takes every conflicting operation event pair and checks if they are locked by the same locks.

$$(e, e') \in \mathbf{DR}_h^{lock} \iff (e, e') \in \mathbf{COP}_h \wedge \mathbf{lock}_h(e) \cap \mathbf{lock}_h(e') = \emptyset$$

$\mathbf{lock}_h$ is defines as all lock which has been acquired and where no releasing event with the same lock is between the acquiring events and the event in question:

$$I \in \mathbf{lock}_h(e) \iff \exists e'' \in h_{t,\texttt{acq}} : \bigwedge \begin{cases} e''_{lock} = I \wedge \mathcal{O}_{e''}^h < \mathcal{O}_e^h \\ \neg \exists e' \in h_{\texttt{rel}} : e'_{lock} = I \wedge e'' <_h^{\mathcal{T}} e' <_h^{\mathcal{T}} e \end{cases}$$
where $t = e_{thread}$

The lockset analysis, contrary to the happens-before analysis which is **NP-complete** [17], runs linear time on the size of the schedule and the number of conflicting operation pairs. A small example of a linear time application can be seen in this algorithm written in python (Section 3.2).

```
1     rw_locks = dict()
2     for trace in schedule.traces:
3         locks = set()
4         for e in trace:
5             if e.opr == 'rel': locks = locks - {e.inst}
6             elseif e.opr == 'acq': locks = locks | {e.inst}
7             elseif e.opr == 'write': rw_locks[e] = locks
8             elseif e.opr == 'read': rw_locks[e] = locks
9
10    data_races = set()
11    for e1, e2 in COP:
12        if rw_locks[e1] | rw_locks[e2] = set():
13            data_races = data_races | {(e1, e2)}
```

The drawback with this analysis is that it produces a huge amount of false positives in real life cases. The method was also introduced as a code style detector which could help a programmer develop provable secure code. Sadly does over locking also produce an unintended overhead which might remove the advantage of multi threading in the first place and enhance the risk of deadlocks. Therefore there is multiple cases where the two methods has been used together. They use lockset analysis to reduce the number of conflicting operation pairs (**COP**) and then runs happens-before on them. This approach has proven to give a faster analysis [18].

3.3 Combining Traces

Besides finding data races can the happens-before analysis also be used for other things. Most prominent is combining traces of schedules recorded separately [9]. Logging traces separately from each other has shown to give up to 93% reduction in logging times, compared to logging schedules in one file, or single database [9]. The reason behind this speed up is that no synchronization is needed when writing the threads to different files. To ensure a sound schedule when exporting to a single file every event access has to be synchronized.

The unordered threads take a lot time to solve when the no synchronization is done between the variables. I choose a golden middle way, where I synchronize access to variables and locks, logging only what is necessary to generated the schedule. In essence is a order from Ω^{strict} recorded, while running the program. The solution is to reuse the happens-before relations, simply by requiring that the newly build schedule to be able to uphold a happens-before relation based of the equations of this chapter.

$$(\texttt{combine}\ \omega_\tau\ \tau_1 \dots \tau_n) = h \iff \bigwedge \begin{cases} T = \cup_{i=1\dots n} \tau_i \\ h \subseteq T \\ \forall e, e' \in h : e \prec_{T,\omega_\tau} e' \Rightarrow \mathcal{O}^h_e < \mathcal{O}^h_{e'} \end{cases}$$

3.4 Summary

In this chapter, I have looked at the happens-before relation. The happens-before relation is a partial order that can be used to define schedule classes. The different schedule classes is made using orderings chosen by different rules. I have looked at the works of Said and Huang, and showed how to encode their algorithms using the symbolic schedule language, and also looked at the span of their schedule classes. All of these happens-before schedule classes were defined using a notion of nondeterministic orders.

Two different dynamic data race detectors has also been introduced. The lock-set algorithm is able to find a surplus of data races, using a linear time algorithm. These results can be used as a filter for future analyses by the happens-before analysis. The

happens-before analysis is able to find all data races in one schedule class described by a happens-before relation.

The last section in this chapter is about how to use happens-before relations to combine traces that has been created by a less-synchronous `trace` function.

CHAPTER 4

Hyperconcolic Execution

This chapter is about how to gather schedules from a parallel program. This chapter will first introduce the concolic method, using the execution model. Then I will use this method as a base to explain the hyperconcolic engine and explain how it relates to data race detection.

In this section I will use the notion $[h]^\star$, which is the class of schedules that the target analysis can extract from h.

4.1 Concolic Execution

The goal of the concolic engine is to gain as large schedule schedule class coverage as possible. This is crucial for testing and dynamic analyses. Each unvisited class may result in undetected errors.

$$\mathcal{H}_p \approx \bigcup_{h \in \texttt{concolic}(p)} [h]^\star$$

Concolic is the contraction of the two words *concrete* and *symbolic*. The basic idea is to concretely run a program, while symbolically recording the execution. Concolic testing is a golden middle way between normal testing and complete symbolic executions (static analysis). Combining concrete and symbolic execution covers more paths than normal testing, which makes it easier for a developer to achieve full path coverage. While static analysis gets even better path coverage (all), concolic execution will only return real paths in the program. Static analyses also often have problem when it comes to real life cases where native libraries are used, or some other non-deterministic thing happens, which forces the analysis to be either over conservative or over liberal, producing false positives or fewer positives than the dynamic analysis would. Where the loss of the context for a symbolic engine is catastrophic, e.g. if native libraries are called, will the concrete part of the execution continues for the concolic engine, until it can regain a symbolic state.

The General Algorithm

The concolic execution take a program and produce a set of schedules from the program ($\mathcal{H}_p$), one from each analysis class $[h]^\star$. The algorithm start with the *results*

set to the empty set and the *inputs* to all possible inputs. As long that there are untested inputs the algorithm will extract an input (σ) from the untested inputs and run `trace`, of the program with the input. The resulting trace is analysed with the `classify` function. It finds a set of inputs which could be run on the schedule. These are then removed from the untested inputs. In this case is Σ_p a limited set of input states specified by the program.

Algorithm 1 Concolic execution

function `concolic`$(p : \mathtt{P})$ `P` $\rightarrow 2^{\mathcal{H}_p}$
  results : $2^{\mathcal{H}_p} \leftarrow \emptyset$
  inputs : $2^{\Sigma} \leftarrow 2^{\Sigma_p}$
  while $\exists\sigma \in_{\Sigma}$ inputs **do**
    $\exists\psi \in \Psi : h \leftarrow$ `trace` $p\ \psi\ \sigma$
    results $\leftarrow$ results $\cup \{(\sigma, h)\}$
    inputs $\leftarrow$ inputs $\setminus$ (`classify` h)
  end while
  return results
end function

The `trace` function is the execution part of the program, this part finds the symbolic schedules, which is can then be analysed by the `classify` function. This is often achieved by instrumenting the code [23, 24, 10] or even write a virtual machine. The trace function in this project has been abstracted away just assuming that the program returns a schedule from the schedules.

Even if it were possible to execute all possible inputs to the program that would be hugely inefficient. Many executions would lead to the same symbolic schedule. $\in_{\Sigma}$ finds a input state in the input states. The `classify` function extract the inputs from the schedule that would lead to the same symbolic execution, in a sense all symbolic executions ($\vec{\Sigma}$). Because all similar schedules have the same executions, it is safe to remove the inputs from the visited sets.

This leads to the problem that holding all possible inputs in a set is impossible in practice. We therefore need a smart structure to hold the inputs, which fast can find an untested input, and is easy to update with the output of `classify`.

Concolic requirements

The goal of concolic is to gain full analysis coverage. Full analysis coverage can be explained as graping at least one schedule from each analysis class ($[h]^{\star}$). Having one or more schedules from this class, gives us ability to fully analyse a program using an dynamic analysis, if the analysis of the schedule can analyse the class from the schedule.

The schedule classes in the concolic world ($[h]^{c}$) are created by the `classify` function. Since concolic has no notion of scheduling, It only knows that one classifier exists.

$$h' \in [h]^c \iff (\sigma \in \texttt{classify}\ h) \wedge (\exists \psi \in \Psi : \texttt{trace}\ p\ \psi\ \sigma = h')$$

The classes generated by the `classify` functions must uphold two properties for the concolic engine to produce full analysis coverage, the class must at least cover the schedule, and it should not cover more that the analysis can analyse.

$$\{h\} \subseteq [h]^c \subseteq [h]^\star$$

Taking one schedule for each concolic class will therefore result in taking at least one schedule from each scheduler class. If a finite number of schedules exist we can produce them all, see Figure 4.1.

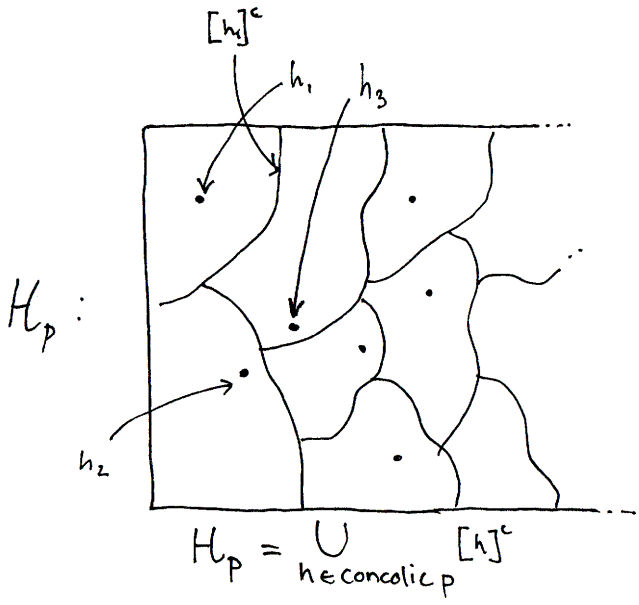


Figure 4.1: Concolic execution finds all schedule classes, if the schedule space is finite.

Using a Constraint Solver

There are multiple approaches to make `classify` function and the data structure for *inputs*. In the symbolic schedule language is the only place where the path of a schedule can change is in a `branch` event. These events can either be satisfied, allowing the schedule to continue, or not to force the program to take another path,

depending on the inputs. These branches can be used as constraints, which completely describes the path through the program.

A constraint solver can then solve these constraints, giving any input which will reproduce a schedule from the class. More interesting is it, that when the constraints are negated, a constraint solver will produce inputs to a schedule class that has not been visited. A constraint solver will continue to produce inputs to new schedule classes to no more exists, by maintaining a conjunction of negated constraints from visited scheduler classes.

After some iterations of this method, the conjunction of constraints becomes so big that it is hard to handle by any computer. This requires a even smarter solution.

The general approach to fix this problem is to use a stack [24, 10]. The branches in a sequential schedule are dominated by the preceding branches. A branch might not be executed if the last branch did not choose the same path as last time. It therefore makes sense to order the constraints gathered from the branches from oldest to newest. Negating the newest constraint, even though it negates the class as a hole, will not effect the validity of the other constraints. The stack can be ordered with the newest constraints on top, and the oldest on the bottom, automatically simulating the dependencies of the constraints.

When running concolic on the program it possible to only maintain a single stack with the constraints. Using a depth first strategy on the stack it is possible to transverse all possible paths in the program, see Algorithm 2.

Algorithm 2 Depth-first path search

function $\in_\Sigma(s) : [\mathcal{B} \times \mathcal{C}] \to \Sigma_p$
  $s, (visited, c) \leftarrow$ `pop` s
  while *visited* **do**
    $s, (visited, c) \leftarrow$ `pop` s ▷ Remove all visited end constraints.
  end while
  $s \leftarrow$ `push` s $(\top, \neg c)$ ▷ Add the unvisited negated constraint again, and mark it as visited.
  return `solve` c ▷ Solve and return the inputs.
end function

function `classify`$(h, s) : \mathcal{H} \to [\mathcal{B} \times \mathcal{C}] \to [\mathcal{B} \times \mathcal{C}]$
  $sc \leftarrow$ `fetch` s ▷ Find all constraints in order.
  $s' \leftarrow$ `merge` s sc ▷ Add all new constraints to the stack, marked as not visited.
  return s'
end function

The positive side of the depth-fist approach is that it is simple and uses almost no space. The negative sides are that there is no way to compute the schedules in parallel , it cannot guide the order of computation, and the worst; it requires the execution to be deterministic. Because the algorithm merges in the old stack with

the new constraints, they have to be equal up to the length of the stack. This is only guaranteed if the execution is deterministic.

It is often important enabling parallel computability when dealing with larger input spaces, and large amounts of code, which might make the run through cheaper in practice.

In practical cases with serial programs not handling nondeterminism is not a problem, as testers strive to keep the code under test as pure as possible. But when the programs become parallel they quickly become nondeterministic.

Limitations

The concolic execution has some limitations. The biggest problem is that it extracts an arbitrary schedule from an arbitrary scheduler, during the tracing. If concolic execution is run on a concurrent program, will the class not be guaranteed to be a subset of the program class $[h]^p$ which is the largest class of analyses can infer and still be sound.

This is not problematic for deterministic programs, where the schedule class tree collapses, see Theorem 2.3.

Other limitations that makes real life concolic engines incomplete is missing information. Calling libraries, IO functions, or changing the semantics of the program while running, all change the state of the execution in a unpredictable way. If such actions exist in the program it not certain that concolic will cover all schedule classes. Also is concolic execution only as powerful as the constraint solver underneath. Some actions on data are current symbolically unpredictable like floating point operations, or string operations.

4.2 Hyperconcolic Execution

Because the concolic execution cannot guarantee completeness on concurrent programs, it makes it ill suited for helping detecting data races. For the concolic engine to work on parallel executions, it has to have a deterministic scheduler.

Adding Determinism

We can add deterministic scheduling using a method called dynamic partial order reduction (DPOR) [6]. The method automatically finds general races and after executing one path perform falls back to the racing events and change the order in which they have been executed.

It is not easy to add deterministic scheduling to the concolic engine and still get path coverage. The problem lies in that the classifier methods are not orthogonal on each other. One single scheduler could be chosen by the executor, and then one could try execute all possible paths using that scheduler. This would be possible, but it would be impossible to choose the next scheduler. The scheduling depends on the input as well as the execution depends both on the input state and the scheduler. The

scheduler and the input states depends on each other and an analysis has to cover both to grantee full analysis coverage in a concurrent program.

The Algorithm

Hyperconcolic is a solution to this problem. Hyperconcolic adds a deterministic scheduler to the concolic engine to get theoretical full path coverage. This is possible by adding a scheduler to the possible inputs of the program. The scheduler and the input states combined is called the context.

Algorithm 3 Hyperconcolic execution

function `hyperconcolic`$(p : \mathtt{P})\ \mathtt{P} \to \mathcal{H}_p$
  results : $\mathcal{H}_p \leftarrow \emptyset$
  context : $2^{\Sigma\times\Psi} \leftarrow 2^{\Sigma\times\Psi}$
  while $\exists(\sigma,\psi) \in_{\Sigma,\Psi}$ context **do**
    $h \leftarrow \mathtt{trace}\ p\ \psi\ \sigma$
    results $\leftarrow$ results $\cup\ \{(\sigma, h)\}$
    inputs $\leftarrow$ context $\setminus$ (`classify`$*\ h$)
  end while
  return results
end function

The hyperconcolic differs from the concolic execution in that the scheduler is added to the input states. In the algorithm, this looks simple enough, but it is problematic to formalize. The naive solution to the *context* is run the program with all combinations of inputs and scheduling. While this solution would be sound for all analyses, as it would produce all schedules in the program, it is highly impractical in reality.

Partial Order Constrains

The scheduler in hyperconcolic are represented by a set of partial orders constraints ($\prec_\psi$), and a notion of past events ($\mathbf{PAST}_\psi$). The scheduler also contains a function, which names the jobs dependently after when they were encountered the scheduler (N^ψ). Using these notions the scheduler function can be created like this:

$$
\begin{aligned}
&\mathtt{choose} && :: && \Psi \to (\mathtt{T} \to \mathtt{E}) \to (\mathtt{T} \cup \{\epsilon\}) \times \Psi \\
&\mathtt{choose}\ \psi\ js && = && (t, \psi[\mathbf{PAST} \mapsto \mathbf{PAST}_\psi \cup \left\{N^\psi_j\right\}]) \\
& && && \mathbf{where}\ \exists t : s = js\ t \wedge j = (t, s) \wedge \\
& && && \qquad \forall j' \in \mathtt{T} \times \mathtt{S} : N^\psi_{j'} \prec_\psi N^\psi_j \Rightarrow C^\psi_{j'} \in \mathbf{PAST}_\psi
\end{aligned}
$$

Scheduler of the hyperconcolic engine is defined as a function, following the execution model defined in Section 2.1. The function simply finds a thread in the active jobs of the execution, and picks it, if all its pre requirements are in the **PAST** events.

Model

The partial order constraints are modeled as a tuple of two events. The first event should by choose by scheduler before the other. If this constraint has to work in stack, it is required that it is possible to negate it. The partial order $e \to e'$ can be modeled as that e' should not happen before e eventually happens, using linear temporal logic (LTL):

$$e \to e' \iff \neg e' \mathcal{U} e$$

To negate this we use that $\neg(a \mathcal{U} b) \equiv \neg a \mathcal{U} \neg b$ to realise that:

$$e \not\to e' \iff e' \mathcal{R} \neg e$$

Which can be translated into that e cannot happen before after e' might happen. The negation does therefore not depend on the existence of the event e' at some point, where the original does.

Extraction

The partial order constraints, or fall-backs, are, when using DPOR, found dynamically while running the program using the vector clock method [13, 6]. I reused the notion given in the happens-before constraints, which already has been developed to find data races.

The partial orders needed to represent each of analysis schedules classes, can be described like the orderings in the happens-before analysis, see Section 3.1. We can fully describe a schedule class by having a set of orderings. Ω^{strict} can be described by a single ordering strict to the schedule. The ordering is in essence a partial order. From ω_{lock} it is sufficient to extract the partial orders by ordering the `acq`s in the pairs. The ω_{rw} is extracted as they are. This produces a list of partial orderings, which would produce another schedule from the schedule class $[h]^{strict}$, if used in the hyperconcolic scheduler.

Stack Management

When that partial order constraints have been extracted they need to be added to the stack. This is problematic because negating the last constraint in the stack should not change the premisses for the others. They therefore have to be ordered by their dependencies. Furthermore, total ordering of constraints is needed if hyperconcolic should be able to run using the depth-first approach.

The first problem that occurs, when trying to totally order the constraints, is that inputs constraints, has one event, while partial order constraints has two events. This means that they are not directly comparable.

The most naive way of ordering the events is by ordering them by the dominating internal event. The dominating internal event (**dom**), for the input constraints are the branch. The partial order constraints are ordered by their second event when

not negated, because the race only happens when the second event is reached, the negated version of the race occurs when the first element is reached.

The dominating events is then totally ordered by the thread they were executed in, and if they come from the same trace, then the order in this traces. This ordering is not only always the same in a schedule but for all schedules in the most general class ($[h]$).

Example 4.1. *We can see see that* [`branch e1.3, e1.2` $\rightarrow$ `e2.3`] *is sorted in relation to the total ordering, but ordered wrong if we look to the dependencies.* `branch e1.3` *might depend on a value read in* `e1.2`.

The proposed total ordering does not take the dependencies into account, which is illustrated in Example 4.1. The solution is to add a partial order on top of the ordering of the stack.

It is not simple to describe the dependency between the partial orders. If an `branch` event is in the same thread as any of the two events of a partial order constraint, and it is after them in the execution, then it depends on the partial order. If the `branch` event is before the partial order constraint, then the partial order constraint would depend on the event. It is more complicated, with two partial order constraints. The problem is parted into four cases.

- The partial order constraints are completely disjoint, and share no threads. There are no dependencies between these orderings.
- The constraints are sharing one thread. The constraint which are last depends on the first.
- The constraints share both threads and one has both events before the others. In this case is the last of the constraints depending on the other,
- The last option is cross dependencies. It is possible that two partial orders are crossing. This makes them dependent on each other.

It is obvious that there can exist dependency loops in this approach. Not just not between a two threads, but in a chain of multiple threads. Negating any constraints in such a dependencies chain would possible make another partial order constraint in the chain not happen. The solution is to remove all chains and replace them with a total ordering, based of of all events touched by the chains. These orders are guaranteed to be without dependency loops, and can therefore be used in the stack.

With all partial orders normalized, it is possible to order them with the branches using the following total order:

$$s <^{stack} s' \iff \vee \begin{cases} \exists e \in \mathtt{E}_s, e \in \mathtt{E}_{s'} : e <^{\mathcal{T}} e' \\ \neg\exists e \in \mathtt{E}_s, e \in \mathtt{E}_{s'} : e' <^{\mathcal{T}} e \\ \mathbf{dom}^{s}_{thread} < \mathbf{dom}^{s'}_{thread} \end{cases}$$

Here $\mathtt{E}_s$ refers to the events in the constraint; the branch in the input constraint and the two events in the partial order constraints.

Soundness of Hyperconcolic

The hyperconcolic class ($[h]^{hc}$) is different from the concolic class, because Hyperconcolic chooses its scheduler:

$$h' \in [h]^{hc} \iff (\sigma, \psi \in \texttt{classify} * \ h) \wedge (\texttt{trace}\ p\ \psi\ \sigma = h')$$

Depending on how the ψ is found does $[h]^{hc}$ span different classes. Choosing ψ as a total ordering of the schedule will set $[h]^{hc} = [h]^{\sim}$.

The total dependencies inferred in the stack, will try every combination of constraints that do not depend on each other. Also does the algorithm also order all locks, even if they do not conflict. This makes the classes found by hyperconcolic using a stack smaller than the actual executions ($[h]^{hc} \subset [h]^{\vec{\Sigma}}$).

Ideally would a structure with a more loose dependencies semantics be able to extract classes equivalent to the $[h]^{\vec{\Sigma}}$, or even $[h]^{\mathtt{P}}$. This would require hyperconcolic to run a lot fever times while still upholding the requirements of the analysis.

But given that that analysis can infer the rest of the schedule class, and that the program analysed contains a finite number schedules, it can make the analysis cover the entire program. This is both the case for `said` and for `huang`:

$$\mathcal{H}_p = \bigcup_{h \in \texttt{hyperconcolic}(p)} [h]^{huang} \tag{4.1}$$

$$\mathcal{H}_p = \bigcup_{h \in \texttt{hyperconcolic}(p)} [h]^{said} \tag{4.2}$$

4.3 Search Strategies

In real life programs is the number of execution classes huge if not if not infinite. This makes a full class coverage unfeasible. It is possible to look at the program as a graph, and hyperconcolic as the depth-first search through the graph. It is therefore possible to search in different ways Smart search techniques has to be used, to find errors as fast as possible.

In the case of data race detection, it is ideal to efficient to search for a set of possible data races, fetched from a complete static analysis.

Iterative Deepening Depth-First Search

The most obvious approach is to limit the length of the schedules reported. Within a limited length program there is fixed number of symbolic schedules. This means that it is possible to extract all classes in these classes.

Limiting the length of schedules directly might result in different results depending on the schedule, so a better approach is to limit the length of the traces instead. Which would give the same events each time.

It is still possible to gain total coverage of these programs, by using a iterative deepening depth-first search [11]. The search works by making a complete search to on one depth. If the goal is not reached its expands the depth and do a total search again. This is at least asymptoticly as in space, time and length of the solution.

Heuristics

It is possible to use the same idea but with cost heuristics [11]. The heuristics, in this case, estimates minimal number of events in a schedule to a target. The approach is build on the A* algorithm [7], where the minimal cost path to a target can be found using *admissible* heuristics. *Admissible* heuristics never overestimates the cost of reaching the goal but is always being optimistic.

The iterative deepening depth-first search can be extended to use the cost heuristics [11], instead of the depth. The algorithm followers the idea of the depth-first search but instead of cutting of the schedules at a certain depth it cuts of the schedules when the total cost, schedule cost plus the minimal cost to finish, is larger than a certain threshold. This approach reduces the space of the algorithm to the length of the stack and schedule.

The heuristic is a bit tricky to derive when using program graphs. One way of determining the heuristics of the distance of a data race, is to determine the minimal distance assuming all branches chooses in the correct direction. This distance to the each of the statements in the pairs can easily be calculated for each statement in the program using a static analysis. These distances can be used to improve the solution.

4.4 Related Work

There has been some attempt at adding a deterministic scheduler to concolic, most prominent *LIME Concolic Tester* (LCT) [10] and *Java Concolic Unit Testing Engine* (JCUTE) [23]. Both JCUTE and LCT uses the *dynamic partial ordering* (DPOR) to ensure a complete running of the program [6].

Model checkers is similar to concolic execution in many ways, but is often interested which states exist and how to move between them. Java PathFinder is a model checking is a good example[28]. It has been shown that concolic execution where super our to Java PathFinder on larger applications[3].

4.5 Summary

Hyperconcolic is a new way of integrating partial orders into the concolic execution. Partial orders are treated as first class citizens in hyperconcolic and are used directly in the execution stack.

The hyperconcolic schedule span is more concrete than concolic because it has direct control over its scheduler. Using hyperconcolic with a dynamic analysis with larger span than the hyperconcolic class, results in sound and complete detection in

all programs with bounded schedules. The `huang` and `said` happens-before schedule classes upholds this property and all constraints based detectors using these, when the schedule class coverage is created by hyperconcolic, will be in sound and complete.

It is possible to integrate searches with the hyperconcolic engine. It is also possible to use the iterative deepening depth-first algorithm to inclemently expand the coverage of hyperconcolic, even in programs with non bounded schedules.

CHAPTER 5

Implementation

This chapter whil both cover challenges and decisions made while implementing the project. Figure 5.1 illustrates the entire project as a flow diagram containing the modules that work on the outputs on each other. SVM is the Symbolic Virtual Machine, and is the program that produces the schedules from Java. The output format, or the format that all the rest of the analysis work on is the STF and SSF formats. The STF format is the intermediate trace format that allows SVM to output the runs in traces. HCC, short for hyperconcolic client, is a multi-tool for working with schedules and traces. It can both combine traces but also find data races in the schedules.

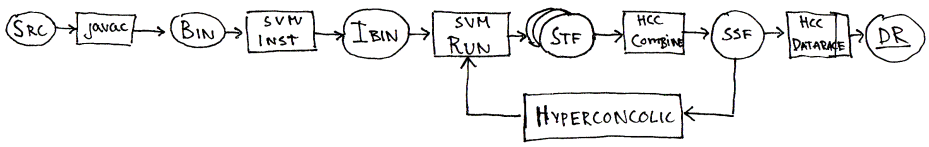


Figure 5.1: The project describped like a flow chart .

5.1 Symbolic Schedule Format (SSF)

I use two different formats, an incomplete format called symbolic trace format (STF) and a complete format called symbolic schedule format (SSF). Both the STF and the SSF contains orderings of events.

The STF format handles traces and is only used to asynchronous log different threads while running an engine. The order of events in the STF represent the order in which they where recorded in the trace. STF files are named accordingly with the thread that they where produced from. The trace of thread 12 is, for example, named `t0012.stf`.

The SSF format handles schedules and the order of the events represent a order in which the events could have been run. All events are named with a thread and a number indicating the order in which it where executed in thread.

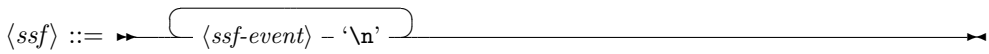
The following listing is an example of a schedule written in SSF. The example is a extracted from the example program in Listing 6.1, and is the beginning, middle and end of a trace.

```
t0.0 begin
t0.1 let a2 0r
t0.2 enter main(java.lang.String[])@HyperConcolicExample <>
t0.3 new i3 1r
t0.4 enter <init>()@HyperConcolicExample <>
t0.5 let i4 2r

:

@location main(java.lang.String[])@HyperConcolicExample 27
@depend <19v,4r>
t0.39 read a5.1=0r 21r i8
@depend <21r>
t0.40 fork t2
t2.0 begin
t0.41 let s4.00000001 22v
t0.42 binopr add 19v 22v 23v

:

t0.69 voidexit
t0.70 voidexit
t0.71 end
```

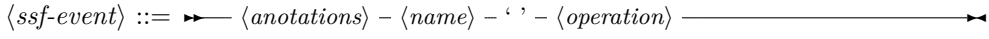
The Grammar

I have designed the text format of the symbolic schedule to be easy to parse, able to be parsed in a stream and be compact enough to write real time. SSF and STF has been designed to be an LL(1) grammar. LL(k) grammars are interesting as the computer only have to read the next token to correctly identify the path of the parsing. This makes LL(k) language easy parse and also allows the computer to parse streams of data. This comes in handy when parsing large schedules or traces, where it would be smart to throw away events after they have been processed.

The SSF is a line separated format:



An event is parsed in an schedule by first parsing a number of annotations, preceded by a `@` and then extract the name of the event and then the event itself.



The annotations gives the ability to add extra information to the event, like line numbers, the CPU it was executed on, cache misses or extra dependencies. The

annotations are not a native part of the language and they can be ignored if not used. All annotations start with an identifier. I have used three annotation kinds in the project:

⟨*annotation*⟩ ::= ‘`location`’ – ‘ ’ – ⟨*method-name*⟩ – ‘ ’ – ⟨*lineno*⟩
‘`depends`’ – ‘ ’ – ⟨*local-array*⟩
‘`invents`’ – ‘ ’ – ⟨*local*⟩ – ‘ ’ – ⟨*concrete*⟩
...

⟨*anonotations*⟩ ::= ‘`@`’ – ⟨*annotation*⟩ – ‘`\n`’

The `name` is the trace identifier, describing which thread the event were from and where in the trace is was located. ⟨*thread*⟩ is seperated from the ⟨*order*⟩ by a ‘`.`’. The ⟨*order*⟩ is simply an integer and the ⟨*thread*⟩ is a ‘`t`’ followed by an integer indicating the thread id.

⟨*name*⟩ ::= ⟨*thread*⟩ – ‘`.`’ – ⟨*order*⟩

⟨*order*⟩ ::= ⟨*int*⟩

The thread is simply preceded by a `t`.

⟨*thread*⟩ ::= ‘`t`’ – ⟨*int*⟩

The STF is like the SSF but because the ordering and the thread is implicit to the trace, the event can be described with the ⟨*stf*⟩ and ⟨*stf-event*⟩ format:

⟨*stf*⟩ ::= ⟨*stf-event*⟩ – ‘`\n`’

⟨*stf-event*⟩ ::= ⟨*anotations*⟩ – ⟨*operation*⟩

Which brings us to the interesting part of the format, which is the 19 different operation types. The event duplicates the ones described in the section about the symbolic schedules.

⟨*event*⟩ ::=

- ‘`begin`’
- ‘`end`’
- ‘`fork`’ – ‘ ’ – ⟨*thread*⟩
- ‘`join`’ – ‘ ’ – ⟨*thread*⟩
- ‘`acquire`’ – ‘ ’ – ⟨*lock*⟩
- ‘`release`’ – ‘ ’ – ⟨*lock*⟩
- ‘`let`’ – ‘ ’ – ⟨*concrete-value*⟩ – ‘ ’ – ⟨*local*⟩
- ‘`new`’ – ‘ ’ – ⟨*instance*⟩ – ‘ ’ – ⟨*local*⟩
- ‘`symbol`’ – ‘ ’ – ⟨*local*⟩ – ‘ ’ – ⟨*local*⟩ – ‘ ’ – ⟨*concrete-value*⟩ – ‘ ’ – ⟨*local*⟩
- ‘`binopr`’ – ‘ ’ – ⟨*binary-opr*⟩ – ‘ ’ – ⟨*local*⟩ – ‘ ’ – ⟨*local*⟩ – ‘ ’ – ⟨*local*⟩
- ‘`unopr`’ – ‘ ’ – ⟨*unary-opr*⟩ – ‘ ’ – ⟨*local*⟩ – ‘ ’ – ⟨*local*⟩
- ‘`branch`’ – ‘ ’ – ⟨*int*⟩ – ‘ ’ – ⟨*local*⟩ – ‘ ’ – ⟨*bool*⟩
- ‘`write`’ – ‘ ’ – ⟨*local*⟩ – ‘ ’ – ⟨*global*⟩ – ‘ ’ – ⟨*concrete*⟩
- ‘`read`’ – ‘ ’ – ⟨*global*⟩ – ‘ ’ – ⟨*local*⟩ – ‘ ’ – ⟨*concrete*⟩
- ‘`enter`’ – ‘ ’ – ⟨*method-name*⟩ – ‘ ’ – ⟨*locals-array*⟩
- ‘`exit`’ – ‘ ’ – ⟨*local*⟩
- ‘`voidexit`’
- ‘`call`’ – ‘ ’ – ⟨*method-name*⟩ – ‘ ’ – ⟨*local-array*⟩ – ‘ ’ – ⟨*concrete*⟩ – ‘ ’ – ⟨*local*⟩
- ‘`voidcall`’ – ‘ ’ – ⟨*method-name*⟩ – ‘ ’ – ⟨*local-array*⟩

First look at the ⟨*local*⟩ and ⟨*global*⟩ types, they denote the position of data and hints the type. Symbolically there exists two types of data, references (‘`r`’) and values (‘`v`’). ⟨*local*⟩ are just a symbolic value in the context of the thread. The information about which thread the local is in is inherited from the event.

⟨*local*⟩ ::= ⟨*symbolic-value*⟩

⟨*symbolic-value*⟩ := ⟨*int*⟩ – ⟨*data-type*⟩

⟨*data-type*⟩ := ‘`v`’ | ‘`r`’

When using locals in an method call it is useful to have them in an array:

⟨*local-array*⟩ ::= ‘`<`’ (⟨*local*⟩ (‘`,`’ ⟨*local*⟩)*) ‘`>`’

The ⟨*global*⟩ type is different from the ⟨*local*⟩, because there is two different global memory accesses; instance and constant. Instance memory accesses uses a concrete instance, a variable name (integer for arrays) and a variable version. Static accesses are assumed to use the fictive global instance `i0`. Constant accesses (used in array lengths, exclusively) are modeled like locals, but can be differentiated by being in the slot of an global argument.

⟨*global*⟩ ::= ⟨*concrete-instance*⟩ – '`.`' – ⟨*variable-name*⟩ – '`=`' – ⟨*symbolic-value*⟩
⟨*symbolic-value*⟩

There are two different kinds of ⟨*concrete*⟩ types, ⟨*concrete-instance*⟩ and ⟨*concrete-value*⟩.

⟨*concrete*⟩ ::= ⟨*concrete-instance*⟩
⟨*concrete-value*⟩

Instances are either real instances ('`i`') or arrays ('`a`'), and are identified by integers.

⟨*concrete-instance*⟩ ::= '`i`' | '`a`' ⟨*int*⟩

The value are harder to represent because the computer works on data differently depending on the type of the data, and how many bits they use. Therefore is the designed with 3 different data types with 4 different sizes. There are floating points ('`f`'), signed integers ('`s`'), unsigned integers ('`u`'), which can be 1, 2, 4 or 8 bytes long. The data itself is represented in hex.

⟨*concrete-value*⟩ ::= '`f`' | '`s`' | '`u`' → '`1`' – '`.`' → ⟨*hex*⟩ (2) | '`2`' – '`.`' → ⟨*hex*⟩ (4) | '`4`' – '`.`' → ⟨*hex*⟩ (8) | '`8`' – '`.`' → ⟨*hex*⟩ (16)

The lock contains a locking instance, and the version of the locking on that variable. This data has been added to accommodate for fast solving.

⟨*lock*⟩ ::= ⟨*concrete-instance*⟩ – '`.`' – ⟨*version*⟩

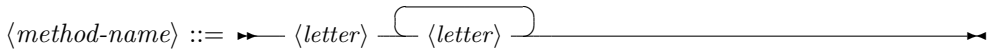
⟨*version*⟩ ::= ⟨*int*⟩

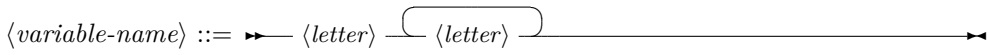
The binary and unary operator names are not verified during the parsing. They are just accepted as strings. The language aims at being as language independent as possible, and different language might have different operations. The only requirement is that it is a combinations of number and letters.

⟨*binary-opt*⟩ ::= ⟨*alphaNum*⟩ ⟨*alphaNum*⟩

⟨*unary-opt*⟩ ::= ⟨*alphaNum*⟩ ⟨*alphaNum*⟩

There are no rules for ⟨*variable-name*⟩ and ⟨*method-name*⟩, because the names are often system dependent, except that they cannot contain whitespace characters, or the "=" sign, and should be at least one character long.





⟨*letter*⟩ ::= not '`\n`', '`\t`', ' ' or '`=`'

The Parsers

I have build parsers implemented parsers in both `python` and in `haskell` to support analyses written cross platform by multiple languages. The parser implementation is strait forward as any LL(1) parsers, but to improve speeds, and memory overhead the parser can read the schedules and traces a line at a time.

5.2 Java Engine

The target language to find data races in where Java. I have therefore implemented a asynchronous tracer to collect traces in real Java programs. The implementation is targeting 1.7 Java bytecode, but might work on other versions.

The Java engines works by injecting commands into the bytecode, in processes called instrumenting. These commands performs calls to library called the Symbolic Virtual Machine. This virtual machine is symbolic copy of the Java virtual machine (JVM) and is maintained throughout the entire execution. The **soot** library is used to do the instrumentation [27].

The SVM is able to control some of the inputs to the program through instructions added manually by the user and the scheduling of the program.

Symbolic Virtual Machine (SVM)

I have implemented a Symbolic Virtual Machine (SVM) to correctly symbolically copy the Java virtual machine (JVM). SVM mimics every action that JVM performs by simulating the underlying stack and memory.

The Stack

The JVM is a stack-based virtual machine, this means that it maintains a stack of frames. Each frame contains a *local variable array* and an *operand stack*. When the machine enters a method a new frame is added on the top of the frame stack containing new local variables and a fresh *operand stack*.

The operand stack is then maintained though the execution by commands. I have reduced the 198 Java bytecode instructions to 62 SVM instructions. This was mostly possible do to type abstractions (partially borrowed from **soot**). This helped on two fronts. First of all did the abstraction remove all the specialized instructions for adding integers, longs, doubles and so on. Secondly in the symbolic operand stack all data has the same size, which removed the requirement for special movement commands.

On the other hand did I need to add instructions to handle the beginning and forking of threads. These type of events is normally handled through the Java standard library, but because the events is important to some analysis, explicit instructions where created for them. Some synchronization instructions where also added to ensure that reading from and writing to the heap, where done in the same order as they did in the SVM.

The operand stack and local variable array is filled with `Local`'s. A `Local` is `Value` which is local to the thread. Opposite does `Global`'s exists which holds values global to world. All these values can be symbolic values which means that they represent a concrete value, or symbolic references, in which case the values contains a reference to an `Instance` in the heap. `Value` should be considered immutable, but `Local` has a temporary state, so that a reference, can be assigned after the creation.

The Heap

When simulating the heap I use a decentralised structure. When a new object is registered in the system, a corresponding `Instance` or `Array` will be created to store all the symbolic variable that they hold. As in the JVM the only way to access these variables is though references in the stack. This has two advantages; less synchronization, because accesses is not handled through a centralized structure and automatic garbage collection by the JVM as soon as the instance is dereferenced.

`Instance` is modeled as a dictionary from fields to variables. A `Variable` is first synchronized parts in the heap. They maintain current symbolic value, called a `View`. A `View` is an immutable reference to the version of the variable, and can like all other values also hold a reference to another instance.

`Array`s are modeled as instances, and differ only in that the keys are assumed to be integers.

Special Cases

Making this adaption of the JVM to the SVM is hard work , but is pretty straight forward, except in some special cases.

Handling Traps Traps in the JVM is hard to manage, because an exception is not necessarily caught at every frame in the frame stack. This leads to the problem that SVM might not know in which frame that JVM will be in when the exception is caught. To handle this case, a local variable is added to the JVM frame indicating

the depth of the symbolic stack when entering the stack. The depth is then read when an error is caught, which allows SVM to throw away the frames deeper than the depth.

Handling the `synchronized` Modifier Java has two ways of creating an atomic section, either with the `entermonitor-exitmonitor` command pair or with the `synchronized` function modifier. The bytecode compiler handles that a `entermonitor` call always is superseded by a `exitmonitor`, but SVM have to handle the `synchronized` modifier manually in each frame shift. This requires that the frame knows that it is currently `synchronized`. In SVM this is simulated by locking the first local variable, which is always the object.

Sadly this does not work for `synchronized` and `static` methods. In these methods is SVM supposed to lock the class object of the object. Using an quick hack is the class object loaded using reflection on the method name, which is saved in the frame, and then create an symbolic instance of this object, which we then lock.

Unknown State Sometimes the SVM can find itself in an unknown state, when an instrumented method is called through an method which has not been instrumented. This happens when using the `Thread` class, in which the `run` method is started by standard library in an new thread. The problem is minimal with values, as SVM just assume that they depend on the values in the library call. Objects on the other hand is a big problem, because the variable has to trace its variables.

To fix this SVM maintain an map from objects to symbolic instances. When SVM find itself in an unknown state, it can load all the symbolic instances from the objects.

This approach has a big flaw because the hanging references of the object will cause a memory leak. When the objects do not get de-referenced then the garbage collector cannot collect it. But no better solution has presented itself and impact is limited on the limited executions that I have run.

Object Creation Object creation is a problem, because from the point where the `new` command is send to the JVM, and to the `Object` constructor is called then the object is only a pointer, and can therefore not be used in SVM. Therefore, the registration of the object is postponed to when an standard library constructor is called. There, the new object is registered in the SVM with an `svm_init` command.

The `wait`, `notify` and `notifyAll` Because the `wait`, `notify` and `notifyAll` methods does effect the access to an atomic section. Control the access would be hard if we had to model an ordering of all `notify`s and `wait`s. This is luckily not the case, because the semantics of wait does not guard for spurious wake ups. This means that notify and waits are not necessarily ordered. I can therefore model the `wait` command as consequent `release` and `acquire` events, and do not need to do anything with the `notify` or `notifyAll` methods.

Instrumentation

Even though making my own implementation of the JVM, would be the most efficient method. I decided to use an instrumentation technique. Instrumentation has some advantages to rebuilding the JVM. The first is that the tools for instrumenting code does already exist and have been tested for some time. This means that even though the underlying bytecode might change, the API for the instrumented stays more or less the same. The second, if then implementation isn't complete we might still get some data out of the run, where as if reimplementing the JVM, a bug would cause a critical error ending the execution.

Instrumenting a Method

When using **soot** each method is instrumented one by one.

I start by allowing all the arguments to the method to be read by the JVM. If the arguments are not read as the first thing in the JVM, the computation fails. First SVM is informed that JVM have entered a new frame. The name of the method, the number of locals in the method and a bit array describing the type of method is send to SVM. The two types that a method can have is static, describing if the first local should be recognized as `this` and synchronized which describes if the method is a part of a monitor.

Because that the JVM already have loaded the parameters to the local space, the instrumentation simulates it by consequent calls to `svm_param`, with the local id and concrete value. The instrumentation also have to send over the concrete values to SVM, because it might not know the symbolic state of the called values, see Section 5.2.

After the parametrization, the instrumentation informs the SVM that the method has been entered, with the `svm_enter` command and save the current stack level to the `level` local, and the SVM stack to the `stack` local.

Handling Heap Accesses

I order heap accesses in a way so that we can ensure that the values are read, or written in are equivalent to the symbolic values. This ordering has been done like this:

```
+ svm_get*($var_name) -+ - Symbolicly lock the variable
  get $var_name        |
+ dup                  |
+ svm_getdone         -+ - Release the variable,
                         - and log the concrete read value.
```

The same thing is be done with puts.

```
+ svm_put*($var_name) -+ - Symbolicly lock the variable
  put $var_name        |
+ dup                  |
+ svm_puttdone        -+ - Release the variable & log
```

There are then different puts and gets for `static`, `field` and arrays, but the procedure is the same. When building a data race detection tool, it is important to be able to present information about the data race, instead of just reporting it. Therefore, is a decorate instruction added before all accesses which sends the method name and line number to the SVM. It will then log it together with the events.

Handling Locking

Locks become tricky, because there are two ways that they can happen, either by entering a synchronized method, or by executing a `monitorenter` command. The order of the locking is important if we want deterministic executions. This means that we want to be able to decide which lock is accessed first. To do this SVM has to have a chance to order the acquiring of locks before JVM does.

```
+ svm_monitoracquire     -+ - Symbolicly order the lock
  monitorenter            |
+ svm_monitorenter       -+ Log that the montor has been entered
```

Since methods also can be synchronized we have to order them too.

```
+ svm_invokeinternal     -+ - Symbolicly lock
  *invoke %somemethod     |
                          |
  somemethod:             |
  // READ PARAMETERS      |
+ svm_enter              -+ - Log that the monitor has been entered
```

Instrumenting the Instructions

I used the **soot** library to implement the instrumentation engine. Using their **baf** api, the instructions where put into six categories.

`DupInst` All the instructions that duplicated operands on the stack. They exist in different flavours, but behaves similar.

`NoArgInst` These instructions has no arguments in itself, but might pop something from operand stack. `nop`, `throw` and `entermonitor` is among these instructions.

TargetArgInst TargetArg instructions, are instructions that allow for jumping in the code. Most of these instructions are variations of comparisons and jumps like `ifcmpeq`. These instructions are simulated by sending the corresponding compare command followed by a branch command. If the branching fails, ei that we do not jump to a new location that is logged. As we cannot log if the branch succeeds.

```
+ load stack
+ dup1
+ call svm_eq
+ call svm_branch $branch_number
ifcmpeq %jump1
+ load stack
+ call svm_branch_failed
```

Some of the instructions is without the `cmp` infix, they are to be compared to zero. A simple fix is by adding a zero to the symbolic stack:

```
+ dup1
+ push 0
+ call SVM.convert
+ call svm_push
```

Gotos are not registered at all, as they do not change our analysis. `JSRInst` is a jump to subroutine instruction which has greatly been replaced by instruction call, therefore if a `JSRInst` is encountered, the code simply throws an `UnsupportedOperationError`. Since none of our test has actually thrown this error, is it safe to assume that it is not used in practice.

OpTypeArgInst These instructions only works on the operands on the stack, like `add` and `return`, and does therefore not need instrumentation. With most of the instructions, the instrumentation only have to pass the corresponding function to SVM. There are some exceptions, namely there are three different versions of `cmp`, and `aload` and `astore` needs extra instrumentation.

`cmpg` and `cmpl` compares two doubles returns +1 if the first is larger than the other 0 if they are equal and -1 otherwise. They only differ in the case were either of the operands are NaN, where `cmpg` return 1 `cmpl` return -1. `cmp` does the same thing for longs. Since the concolic engine do not support doubles, just parse the `cmp` on to SVM, not caring about the type.

When accessing the `aload` and `astore`, we have to break the symbolic execution to get the correct variable. Instrumented code does this by extracting the concrete index from JVM Stack, and using it to access the symbolic variable in the array. The `aload` and `astore` is then ordered like other

FieldArgInst `FieldArgInst` are instructions that takes a field argument, and either read from it or writes to it. The instrumentation of theses fields follow the ordering presented in the previous section, Section 5.2.

`MethodArgInst` This category hold all the method operations; static, interface, virtual and special invocations. There is a lot to handle in these categories. Calls to instrumented methods are handled by calling an `invokeinteral` command onto the SVM. Al other calls are put into three categories: Good, Bad and Ugly. The Good are method calls to the standard library, which we know the semantics of;`Thread.start` instrumented by a `fork`, `Thread.join` by a `join` and the `Object.wait` instrumented with a `release` and an `acquire` (see the SVM section). The Ugly are the rest of the standard library calls, they are ugly as we do not have their semantics so we have to report them with the `voidinvoke` (for void methods) or `invoke` (for returning methods), and hope that preceding analysis know the semantics. The Bad are calls to libraries which could have been instrumented but where not in the class paths, if one of is found then the program will throw an exception requiring the user to fix it.

The Rest The last instructions has been stoved away in a single category. Most of them are stack operations like `swap` or `pop`, and is simulated accordingly. Because assume that the JVM grantees type safety, there is no reason to register casting of objects.

SVM Scheduler

Hyperconcolic requires some control over the scheduler to guarantee coverage. It would be possible to replay a schedule step by step, but because hyperconcolic engine produces partial orders, it is easier to use partial orderings in the logger too, instead of full schedules. The partial orderings also improves the run time when running parallel programs because the events does not have to happen in the exact same order.

I have tried to build the scheduler as non intrusive as possible so that it does not slow down non conflicting parallel execution. This is done by pushing the synchronization out to the Event class, which in essence is an inverse semaphore. When calling the `waitForDepenendecies`, the thread will wait until the event has been released enough times.

```
public class Event {
    int released;
    int depends;

    synchorized void release() {
        released += 1
        if (released > depends) notifyAll();
    }

    synchorized void waitForDepenendecies() {
        while (released > depends) wait();
    }
}
```

This class is used to order schedules using an partial order. The data structure to hold the events requires two data structures, a structure to find the ordered events from the names, and a structure to find the actual ordering.

```
Map<EventName, Event>  orderedEvents;
Map<Event, List<Event>> order;
```

Given a list of partial ordering SVM can populate the structures:

```
void init (List<Pair<EventName, EventName>> n) ) {
    orderedEvents = emptyMap();
    orders = emptyMap();
    for (Pair<EventName, EventName> p : n) {
        if (!orderedEvents.contains(a)) orderedEvents.put(a, Event(0, 0));
        if (!orderedEvents.contains(b)) orderedEvents.put(b, Event(0, 0));
        e_a = orderedEvents.get(a)
        e_b = orderedEvents.get(b)
        e_b.depends += 1
        if (!order.contains(e_b)) orders.put(e_b, emptyList());
        orders.get(e_b).append(e_a);
    }
}
```

Notice that the SVM scheduler only keep ordered events, this can give considerate speedup, because all unordered events would not be required to grab the locks, also SVM do not have to create events to them.

```
void waitUntilAllowed(n :: EventName){
    if (orderedEvents.contains(n))
        orderedEvents.get(n).waitForDepenendecies();
}

void allowSuccesor(n :: EventName) {
    if (orderedEvents.contains(n)) {
        e = orderedEvents.get(n);
        if ( orders.constains(e) ) {
            for (Event succ : orders.get(e)) succ.release();
        }
    }
}
```

All events that can race, `read`, `write` and `lock` calls the `waitUntilAllowed` before allowing the actual JVM statement to execute. After the execution SVM allow preceding events to run using the `allowSuccesors`. This model efficiently handles partial orderings at runtime.

5.3 Data Race Detection

Using the theory derived from RVPredict in the previous section, I have implemented a data race detector using my intermediate language. The implementation is based

on python, which made it easy and fast to develop prototypes. This was possible because I used an external constraint solver, most of the heavy lifting was removed from the scripting language.

The name of the data race detector is the *hyperconcolic client* (HCC), and its usage is threefold. It serves as a data race detector on the SSF schedules, a crude STF combiner, and the backbone for the partial order reduction using in the hyperconcolic engine.

Constraint Solving

SMT solvers are the black boxes of constraint solving techniques used to solve data races. They specialises on solving NP-hard problems using logic. Most of these solvers accept the SMT-LIB standard format [1], which is why HCC output constraints in that language.

I choose the Yices2 constraint solver as the primary solver. I did this because it is fast, support the SMT-LIB, and is free for non-commercial use [2]. It is possible to use the c api, but to stay compatible with other solvers it where easier to use the textual interface.

The problem of finding data races was parted into four parts; The three from the happen-before relations; the must-happen-before, lock order and read-write order, and then the data race conditions. To have efficient solving some of the clean mathematical equations where broken up in smaller bits.

All events with importance is represented as a symbol which should be giving an integer. The total ordering of these integer represents the ordering of the events in the schedule. The objective of the solver is then to assign values to these symbols, and if it is possible without breaking any of the partial orders, a schedule must exist.

Like with the orders from the happens-before-analysis, can the user choose between four different algorithms when running the program; “strict”, “said”, “huang” and “loose”. They are each implemented after the descriptions in Section 3.2.

HCC is also uses as a combine tool. It does this by using the strict ordering on the unsorted traces. The solver then returns a total ordering of a part of the schedule. This order is then topological sorted with the must-happens-before relations, which then creates an entire schedule.

Optimizations

Some optimizations where required to make the constraint solver perform the best. Since the $\prec_{rw}$ allows a certain granularity, by calculating Φ differently, it in a way so that we can test the different approaches.

The program employs multiple small tricks to speedup the solving, some of them are sound while others are not.

Unsound Optimizations

The most prominent problem when working with the constraint solving is that the number of pairs to check for data races is squared in the length of the schedule. This quickly becomes a serious problem, when solving for one race candidate can take up to a second. Data races are normally close to each other in respect to synchronization. I use this by removing all data races candidates which are more than a certain number of write events from each other. This optimization is not sound but speed benefits makes it possible to find the real races in a realistic time frame.

Algorithmic Optimizations

The `huang` algorithm takes a lot of time per candidate because for every data race it has to recalculate the entire read-write constraint. To reduce this impact, pre-solving is used. Pre-solving uses that the `loose` algorithm will find all the data races possible found by the `huang` algorithm. By first solving with the `loose` algorithm can the program reduce thousands of candidates to only a few. These few candidates can then be quickly be verified by the `huang` algorithm.

Solver Optimizations

Solver optimization is optimizations directed towards making it easier for the solver, to find the data races. The two used in the program is bunching and interactive solving.

Interactive solving uses that some of the constraints remains static for all candidates. Instead of restarting the solver for each candidate, the candidates is added the solver in a retractable way. The candidate is then checked. After the operation, can the candidate be retracted, and another one checked. This keeps the memory and pre-optimizations in the solver, and can be reused for each candidate.

The interactive technique can easily be combined with bunching, which uses that HCC only report race candidates for locations in the code. It is therefore possible to bunch all candidates from the same location in smaller bunches. The disjunction of constraints of the entire bunch is then send to the solver. Because the constraints is pretty small the solver performs better when able to work on more input. If one of the bunches are solvable, then the location contains at least one data race.

5.4 Hyperconcolic

The implementation of hyperconcolic is pretty strait forward. Most of the way it reuses the components described in the other sections. It uses the partial orders from the data race detection tool, the scheduler from SVM, and the parser of SSF.

Input Constraints

Novel to the hyperconcolic execution is the exaction and calculation of the input constraints. Hyperconcolic extracts input constraints by rerunning the schedules. For each events it then updates a symbolic state. The symbolic state is a couple of dictionaries; one with locals as key and one with variables as key. For each of theses keys the dictionaries is populated by symbolic expressions. The symbol event add a symbol to the local symbolic expressions. The read and write events moves data between the local dictionary and the variable dictionary. Each unary or binary operation adds a symbolic expression that takes the symbolic expressions of the locals as its arguments.

This way, the symbolic expressions are step by step created. A branch event can then simply fetch the expression related to the local representing the condition from the dictionaries, and use it as a constraint.

I use bit vector logic to formulate the constraints. Bit vector logic looks at every integer operation as a number of logical operations on bits. The constraint solver tries to solve the value of every bit. Bit vector logic is great at solving expressions in a concolic execution, because the constraints can contain binary logic that cannot be modeled in the other logics. Also, overflow errors can be taken into consideration. Using bit vectors in concolic engines is not something new, and has proven to give good result in prior cases [10].

Stack management

Currently, the hyperconcolic engine use a stack heap queue instead of a single stack. This enables me to test different run through and get larger path coverage than when running depth-first on the schedules. To run depth-first efficiently requires that a limit can be put on the length of the executions, which is yet to be implemented.

Each stack is represented using a linked list. When a new stack is extracted from the execution it is negated and added to the heap queue and so are all the sub stacks in the stack. By making it a linked list, the computer can save some memory by only saving each event ones.

Output

Hyperconcolic has two different outputs; either it can output schedules into a folder which can then be analysed later or it can check the schedules for data races without any extra analysis. It is also possible to disable the hyper in hyperconcolic. This produces fewer schedules, but the schedules are not guaranteed to cover all execution classes.

5.5 Summary

Three different modules, five programs and two output formats where needed to implement the hyperconcolic engine. The project has used four different languages and consists of more than 7000 lines of code.

The symbolic schedule format has been developed to cover all the needs of both the hyperconcolic engine and data races, and has proven through tests that it is very usable. The symbolic virtual machine is a complete copy of the JVM. Every operation that the JVM performs, SVM performs too. An instrumentation of Java code has been made, which uses the symbolic virtual machine.

A data race detector has been implemented. It has been optimized using various techniques. A prototype of the hyperconcolic engine has also been implemented, using the principles described in Chapter 4.

CHAPTER 6

Results and Benchmarks

This chapter holds comparisons between the programs developed under the project and state of the art tools. The chapter consists of an example run showing the entire project in actions, a section about the speed of logging schedules, a sections on speed and abilities of the implemented data races detectors and lastly a walk through of the state of the hyperconcolic engine.

6.1 Example Run

I have developed a small example use of the tool chain to illustrate the implemented capabilities. The example where extracted from Mahdi Eslamimehr's article about data races [3] and implemented in java, which is listed in Listing 6.1. The key part of the example is the methods `a` and `b`. The two methods are run in parallel and they share the variables `x`, `y` and `out`. `out` is the variable which describes if a branch were executed. `y` is an arbitrary integer between 0 and 10, see line 17.

The program races on variable `out` in line 12 and 18, but that requires that both thread `A` and thread `B` both have the correct values in the branches in line 11 and 17. The example has been designed so that the only way both branches are satisfied is if the order of events is `.. B.16, A.10, B.17 ..` and the value of `y` is in the range $[0, 4[$.

The first step in the progress is to compile the code using the java compiler. Because we use the `svm.Runtime` methods to generate the inputs, we have to include the `svm.jar` file to the class path.

```
>$ java -cp svm/build/svm.jar:examples/bin HyperConcolicExample
```

The program can now be run as normal, with no slow-down. `y` is simply chosen at random. In this case `y` where below 4.

```
>$ java -cp svm/build/svm.jar:examples/bin HyperConcolicExample
Ended up in branch: a
```

To get more information about the program, we have to instrument it. We put the instrumented bytecode into the `examples/inst` folder.

```
import svm.Runtime;

/** Example of Hyperconcolic, as thought up by Mahdi Eslamimehr.  */
public class HyperConcolicExample {

    public int x, y;
    public String out = "-";

    public void a() { // runs parallel with b.
        x = 6;
        if (y < 4)
            out = "a";
    }

    public void b() { // runs parallel with a.
        x = 2;
        if (y*y + 5 < x*x)
            out = "b";
    }

    public static void main(String [] args) throws InterruptedException {
        HyperConcolicExample hc = new HyperConcolicExample();
        Thread [] ts = { hc.new A(), hc.new B() };

        hc.y = Runtime.getInteger(0, 10);

        for (Thread t : ts) t.start();
        for (Thread t : ts) t.join();

        System.out.println("Ended up in branch: " + hc.out);
    }

    class A extends Thread { public void run() { a();} }
    class B extends Thread { public void run() { b();} }

}
```

Listing 6.1: Example application with dataraces.

```
>$ svm/svm-inst -cp examples/bin -d examples/inst HyperConcolicExample
Classpath=examples/bin Output=examples/inst App=HyperConcolicExample
Soot started on Sun Jan 11 13:01:52 CET 2015
Transforming HyperConcolicExample...
Transforming HyperConcolicExample$A...
Transforming HyperConcolicExample$B...
Writing to examples/inst/HyperConcolicExample.class
Writing to examples/inst/HyperConcolicExample$A.class
Writing to examples/inst/HyperConcolicExample$B.class
Soot finished on Sun Jan 11 13:01:53 CET 2015
Soot has run for 0 min. 0 sec.
```

The newly instrumented code can now be executed with the logger, which runs the instrumented code and output the traces into the folder `HyperConcolicExample.log`. This run through did not enter any of the branches.

```
>$ svm/svm-run -cp examples/inst HyperConcolicExample
Ended up in branch: -
```

The traces can then be analysed, and combined, by HCC. We can see that using the `huang` algorithm on this schedule is there only 2 data races, one in line 10 and 16 and one on line 10 and 17.

```
>$ hcc/hcc datarace --algorithm huang --traces HyperConcolicExample.log
x@HyperConcolicExample a()@HyperConcolicExample:10 b()@HyperConcolicExample:17
x@HyperConcolicExample a()@HyperConcolicExample:10 b()@HyperConcolicExample:16
```

To find the last datarace we have to run `hyperconcolic` on the program. This method runs the `HyperConcolicExample` multible times, generating a schedule from each hyperconcolic schedule class:

```
>$ ./jhyper -cp examples/inst HyperConcolicExample
INFO:classifier: Found 16 schedules
/tmp/tmpfno4pd8u_hcc/schedule_0.ssf
/tmp/tmpfno4pd8u_hcc/schedule_1.ssf
/tmp/tmpfno4pd8u_hcc/schedule_2.ssf
/tmp/tmpfno4pd8u_hcc/schedule_3.ssf
/tmp/tmpfno4pd8u_hcc/schedule_4.ssf
/tmp/tmpfno4pd8u_hcc/schedule_5.ssf
/tmp/tmpfno4pd8u_hcc/schedule_6.ssf
/tmp/tmpfno4pd8u_hcc/schedule_7.ssf
/tmp/tmpfno4pd8u_hcc/schedule_8.ssf
/tmp/tmpfno4pd8u_hcc/schedule_9.ssf
/tmp/tmpfno4pd8u_hcc/schedule_10.ssf
/tmp/tmpfno4pd8u_hcc/schedule_11.ssf
/tmp/tmpfno4pd8u_hcc/schedule_12.ssf
/tmp/tmpfno4pd8u_hcc/schedule_13.ssf
/tmp/tmpfno4pd8u_hcc/schedule_14.ssf
/tmp/tmpfno4pd8u_hcc/schedule_15.ssf
```

We could run the data race detector on each of these schedules manually, but it is a lot faster to make data race detection on the fly and then throw away the schedules, after we are done with them.

```
>$ ./jhyper -cp examples/inst --datarace huang HyperConcolicExample
INFO:classifier: Finding Dataraces
x@HyperConcolicExample a()@HyperConcolicExample:10 b()@HyperConcolicExample:16
x@HyperConcolicExample a()@HyperConcolicExample:10 b()@HyperConcolicExample:17
out@HyperConcolicExample a()@HyperConcolicExample:12 b()@HyperConcolicExample:18
```

Running hyperconcolic on the execution finds the last data race, and all data races in the program has been found.

6.2 Schedule Logging

This section will analyse the speed and memory usage of the schedule logging, and compares to RVPredict.

I want to be able to compare our logging performance compared to other data race detectors and concolic engines. I have settled on showing comparing with RVPredict [8] as the only comparison. This choice were made because, event though RVPredict is not logging the same amount of data, it is separated in instrumentation, logging and data-race detection. To make the comparison with other concolic engines is not interesting as they do not have the same separation of stages.

The interesting parts to test is the instrumentation time, the logging slowdown and the logging memory usage. I have parted to test sets into two categories, real life and constructed. The real life cases are based of versions of the Java Grande Suite, examples modified by RVPredict team. The constructed test cases, contains a single program tailored to stress test the logging.

Stress Test

The stress test was devised so that it was computing intensive, and with no shared memory accesses. The test ran with two inputs, the number of threads to run an how many iterations they each should run a dummy computation. The computations where not dependent on each other in any way. The resulting graph can be seen in Figure 6.1. There is no comparison to the running time of the code with out instrumentations, this is because the JVM is able to run our example in almost constant time, in the range of the tests. Even though that RVPredict is faster than SVM on the large number of iterations, which is expected since SVM logs more data, we can see that there almost only a constant overhead of having more threads, if they do not share data.

From this can we see that SVM has around a factor 2 worse running time than RVPredict on large cases, but is still linear on the size of the schedule. It is also possible to see that RVPredict handles the logging in a way that reduces the effect of

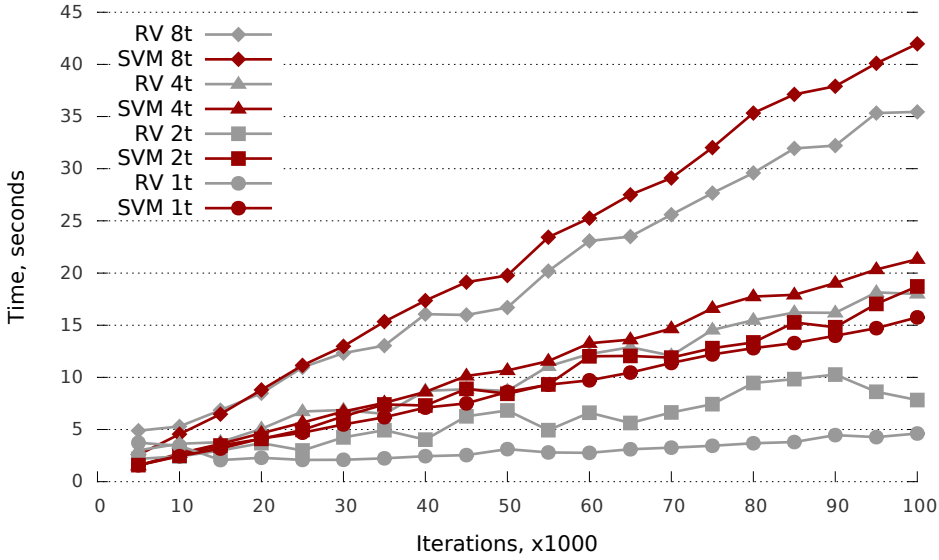


Figure 6.1: Stress test of RVPredict and SVM.

running the program in parallel. In the last case with 8 threads, it is very possible that the sudden performance drop from SVM, is due to a cap on the disk speed. Improving the format to a compressed binary format, might help a lot on the performance.

This one to one comparison is not completely fair as we do not take into account that the traces has to be combined in the other end. Also does SVM log a lot more data than RVPredict, which gives SVM a disadvantage.

Real Life Test

The real life set consists of three modified cases from the Java Grande Benchmark Suite [26, 8], `MolDyn`, `MonteCarlo` and `RayTracer`. `MolDyn` (JGFMDBSA) is a N-Body particle simulation program, `MonteCarlo` (JGFMCBSA) is a finacial simulation program and RayTracer is a 3D RayTracer program. The modifications has tried to dramatically reduce the computation times, while the same parts of the code is reached. The RVPredict team has done this to find data races in the programs. All of the applications are parallel, and CPU bound.

I ran the programs 5 times, with 4 threads, averaging the measurements. The logging times has been depicted in Figure 6.2, and the memory in Figure 6.3. Notice that all of the measurements are presented on a logarithmic scale, this has been done to make all the data fit on one graph.

SVM uses more memory than RVPredict in all the tests, but the memory usage does not seam to be proportional with the used of memory of the application. It is

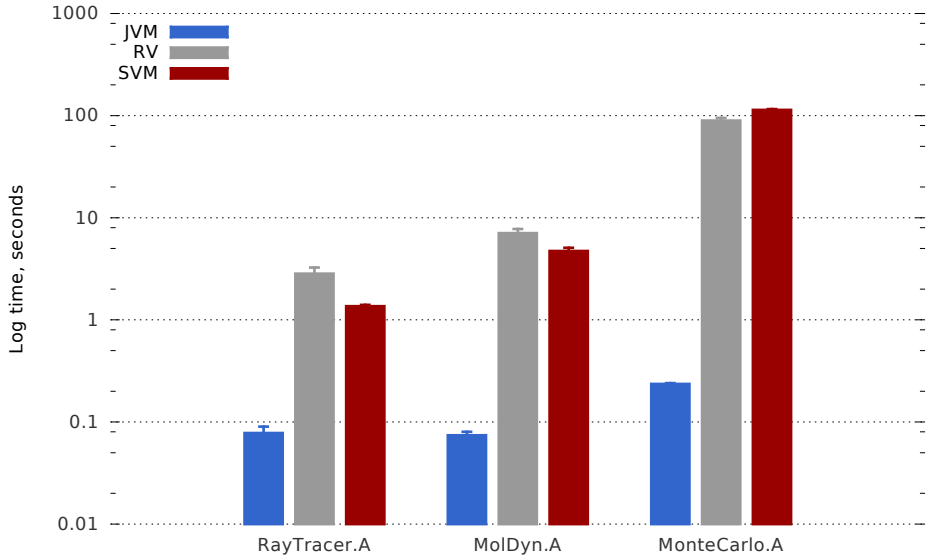


Figure 6.2: The logging times for RVPredict and SVM .

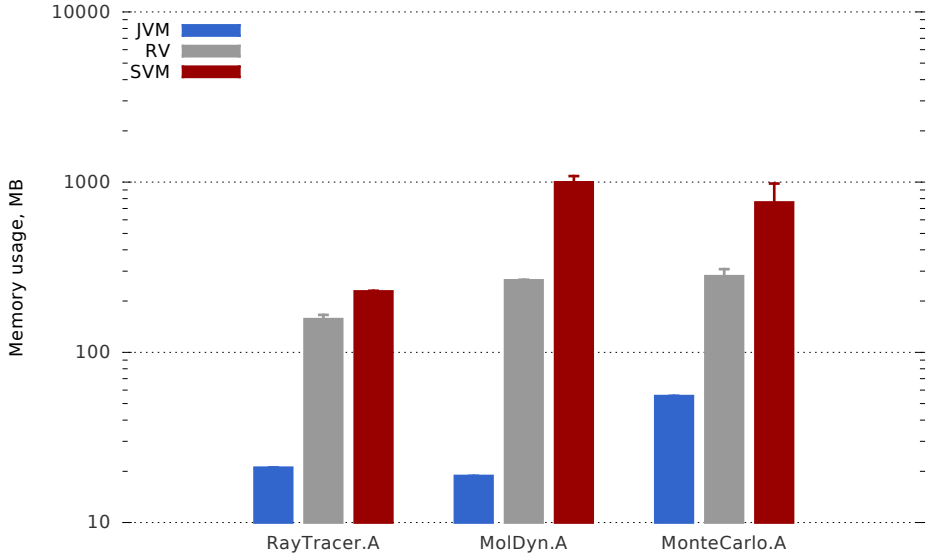


Figure 6.3: The memory usage for RVPredict and SVM .

understandable that SVM uses more memory, because of the memory leak problem explained in the section about SVM. In the logging time category is SVM actually faster than RVPredict in the short cases. This is compatible with knowledge gained from the stress test.

6.3 Data Race Detection

I test the data race detection in two different ways. First I check how HCC compares with other data race detectors, and then I compare the different data race algorithms against each other.

Comparative Test

This section covers the number of data races found by HCC, and the speed at which it does it. These numbers are compared to RVPredict and Chord, two state of the art data race detectors.

I have run the data race detectors on benchmarks created by the RVPredict team and by Mahdi Eslamimehr. For most of the test cases was the data race detectors run multiple times of different inputs, giving them a larger coverage. I did some modifications to the test set because HCC where unable to run them. One test case where removed because it exited during the run, which is something HCC does not handle well and the `airlinetickets`, had the biggest test case removed. Also `boublesort` and the Java Grande test where removed as the solving times and memory used where too big. RVPredict can therefore handle bigger test cases, which is mostly do to the fact that they employ a windowing technique, where they take pieces of the schedule and analysing them separate. This does in practice give linear solving times for large test cases, but worse results.

RVPredict and HCC claims to be sound, and JChord claims to be complete. It would therefore be interesting to put this to the test. I have looked on each data race and compared them to each other, which enables me to create first three categories. `S&C` stands for sound and complete, these data races has been confirmed by at least one Sound data race detector, and is in all the sets of the complete data race detectors, in this case only JChord. The `C-S` are all the possible data races, which has jet to be confirmed by a sound data race detector. `S-C` are errors, either is the complete data races not complete or a sound data race detector has reported a problem.

Table 6.4: Test cases run with the data race detectors. The first row is the name. The preceding three are sound and complete dataraces, complete but not sound dataraces, and sound but not complete dataraces. The number of dataraces detected, and the time it took for each datarace detector..

name	S&C	C-S	S-C	hcc	rvp	jchord	hcc	rvp	jchord
account	4	1	0	4	4	5	6.60	14.41	133.07
airlinetickets	9	1	0	6	9	10	12.21	18.74	134.70
array	0	1	0	0	0	1	4.84	10.75	131.39
boundedbuffer	11	2	1	8	11	13	2170.29	66.30	136.18
bufwriter	2	10	0	2	2	12	15.32	23.91	134.50
critical	8	10	0	5	8	18	5.76	12.92	136.35
mergesort	2	44	3	5	3	46	168.01	24.86	210.94
pingpong	2	5	1	1	3	7	6.53	14.70	136.58
tsp	3	103	0	ERR	3	106	-	53.47	225.45

Even though that the timings and results are comparable, can different schedulers have occurred while performing the analyses. This might give different results. Even though HCC and RVPredict has been based on the same algorithm do their detections vary according to the scheduling.

In all cases, except `mergesort` did HCC score worse than RVPredict. This is mostly due to that RVPredict is a more mature piece of software, so that small bugs in the logger or in the detector, has been found. In `airlinetickets` does RVPredict find all data races where HHC only finds 6. This is due to the different scheduling of the two runs. HCC did not go down the same path as RVPredict. Another run showed the exact opposite result HCC finding 10 data race and RVPredict finding only 6. The `boundedbuffer` example shows that the windowing algorithm is working, so they find more races at a shorter time. The conflicting data race `S-C` is generated by HCC. After examining the race it turns out that is a race, and that JChord is not complete in that sense, and the race where missed by RVPredict.

It is interesting to see that RVPredict, HCC and JChord conflicts in `mergesort`, in this case is JChord was not complete, and did not find the data races.

All the timings include multiple runs and instrumentation to stay as fair to JChord as possible. We can see that HCC is faster than the other analyses on short cases with few threads. This ideally as we want to find races as fast as possible, in small traces generated by the hyperconcolic engine.

Algorithm Test

As explained in the implementation have I implemented 4 different data race algorithms; `strict`, `said`, `huang` and `loose`. The three first algorithms are sound, while the last is maximal in respect to the schedule.

I have based the testing on the schedules generated in the comparative tests, but I removed the tests which produced errors. Figure 6.5 shows the data race detection abilities of the different algorithms on different test sets. Keeping in mind that the data sets was still created by the RVPredict team, does it verify that the new algorithm, `huang` can find more data races, than `said`. It also shows that in most of the cases is there a very little space for improvements upwards to the potential data races found by `loose`. The `strict` algorithm falls behind, but only a data race or two.

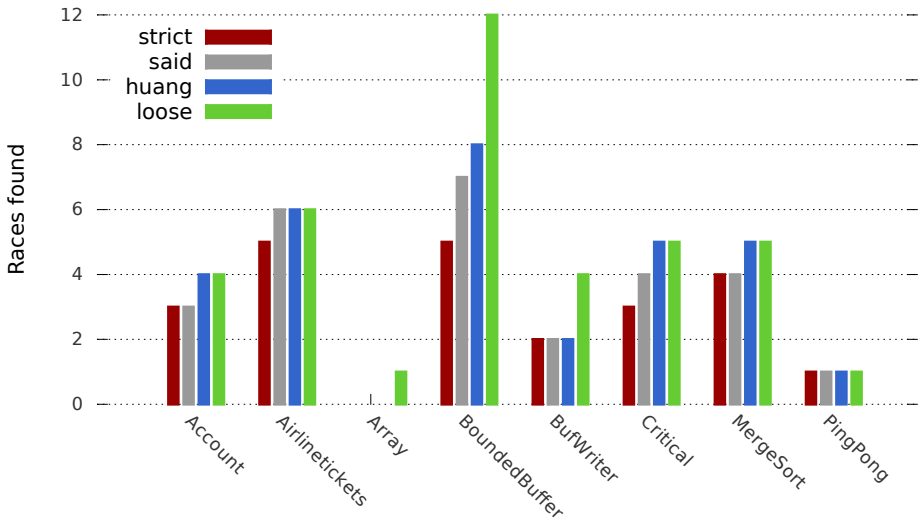


Figure 6.5: The capabilities of the algorithms .

As we can see in Figure 6.6 is the `strict` data race detector a lot faster than the other algorithms in the larger cases, especially in the bounded buffer, where it is around 20 times faster. The running times of `huang` and `loose`, is almost the same, even though `huang` uses a more complicated algorithm. This is due to the pre solving done with `loose` which cuts downs the possible candidates to 12, see Figure 6.5. Solving those gives almost no overhead, and a more precise result.

The memory usage of the algorithms is almost the same for the smaller cases, as we can see in Figure 6.7, but the difference becomes substantial in the larger. Both `loose` and `strict` uses less memory that the other algorithms. `strict` saves on only having one ordering of the locks, while `loose` does not have to describe the read-write order at all. The two algorithms `huang` and `said` are almost equivalent in used of data, but `huang` saves a little on only having to the read-write orderings up the data race.

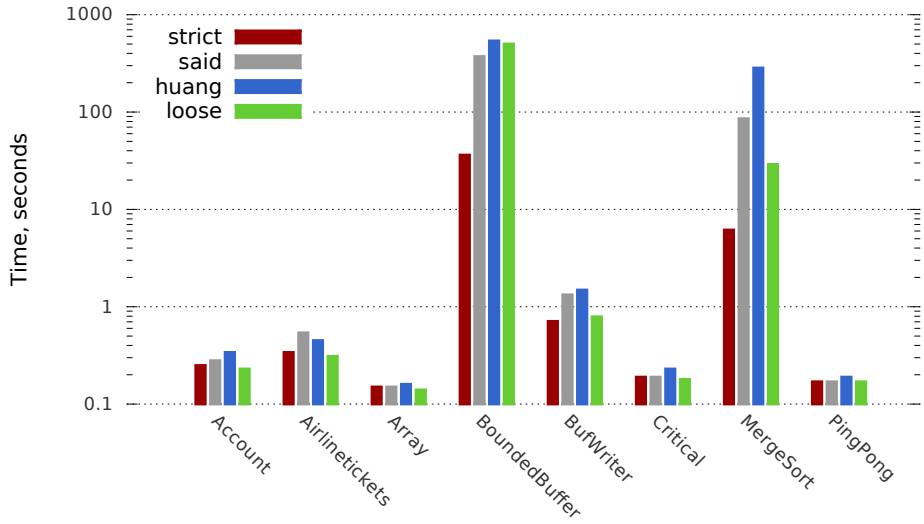


Figure 6.6: The solving times for HCC, using different algorithms .

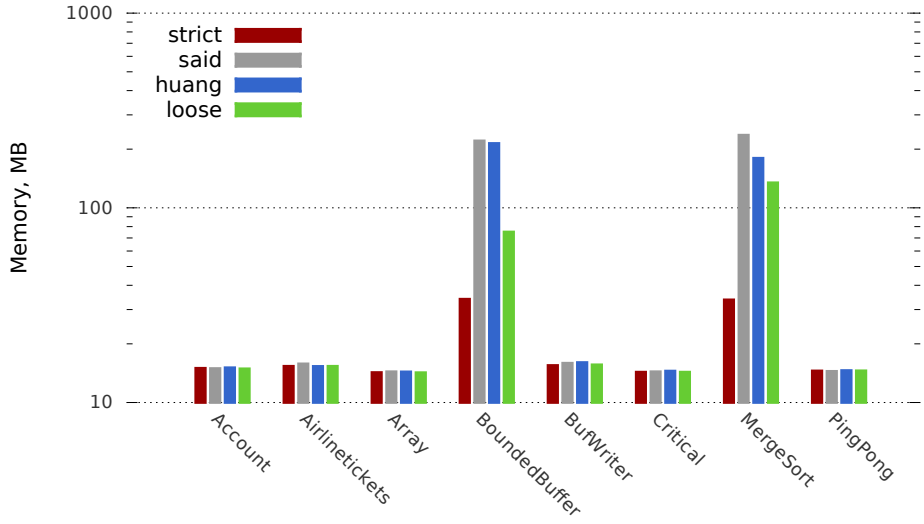


Figure 6.7: The memory usage for HCC, using different algorithms .

6.4 Hyperconcolic

The code of hyperconcolic is in the early state of its development. Therefore it is limited in the number of real life cases that it can run on. The usage of hyperconcolic is two fold, it is capable to run as a concolic engine and it can run as a partial ordering executer when no inputs has been given. It is possible to define injection points where the state should be controlled. It is therefore possible to run a test case without any inputs. In this configurations can hyperconcolic be used to extend a dynamic analysis to cover all cases.

Hyperconcolic does merge the input and partial order constrains, while it is not jet optimal. The next listing is taken from the outputs of hyperconcolic, it shows that the stacks has been added to the heap queue, and the last one with the minus is the extracted stack.

```
+ t2.12.read -> t1.5.write;
  t1.9 (bvsge s0 #x00000004);
  t2.15 (not (bvsge (bvadd (bvmul s0 s0) #x00000005)
                    (bvmul #x00000002 #x00000002)))

+ t2.12.read -> t1.5.write;
  t1.9 (not (bvsge s0 #x00000004))

+ t2.12.read -/ t1.5.write

- t2.12.read -> t1.5.write;
  t1.9 (bvsge s0 #x00000004);
  t2.15 (not (bvsge (bvadd (bvmul s0 s0) #x00000005)
                    (bvmul #x00000002 #x00000002)))
```

I have not run hyperconcolic on cases with inputs, besides the example in Section 6.1, because there were problematic enough to get hyperconcolic to work at the test cases without.

Test Cases

I ran hyperconcolic on all the test cases from the test set, that had had an acceptable running time.

`account` In `account` test case, an unexpected dependency exited between the call `isAlive` and if the threads where alive or not. These call where in fact racing with the `end` event of the threads, which caused indeterminism. This made the events come in a different order and the program went into a deadlock waiting for the events that newer came.

`airlinetickets` In the `airlinetickets` case, hyperconcolic find an extra data race compared to a single run through. The seventh is found after 34 rounds. This still lacks behind the data 9 data races found by RVPredict. The execution also deadlocks from time to time, when an exception is thrown, which make some the traces stop abruptly without releasing the event.

`array` In `array`, hyperconcolic manages to get full path coverage, with 3 schedules. And no data races existed in the test case. The test case requires that hyperconcolic can put locks in the right order without deadlocking, which it manages too.

`bufwriter` Running the buffer resulted in hyperconcolic finding 2 data races. The solution is found in the first run through, and after 18 runs where the execution halted, because the solving times exceeded.

`citical` Finds 8 data races in the first round, after 75 more rounds nothing more is found. The program has to be terminated, after some time because the testcase contains an possible infinte loop.

`pingpong` A good example of where hyperconcolic, performs well is in the `pingpong` case. Where 4 data races where detected in the first 10 runs. The program then precedes to error when the program where abruptly ended.

Evaluation and Experiences

To gain full path coverage without limiting the length of the execution is almost impossible in real life cases. Hyperconcolic is therefore best suited for testing parts of a program where the complexity stays relative low.

If hyperconcolic is to be used with larger applications, it has to be able to search after the errors as described in Section 4.3. Also trying to get full path coverage in a parallel programs is problematic because in these edge cases are very bad defined. When exit is called in one thread of the program, when do the other threads end. Better handling of these cases is also needed to gain full path coverage, in the complicated cases.

Hyperconcolic, in its present state the other hand, works very well on small programs with no unexpected actions. Here is Listing 6.1 a good example. In the simple program, hyperconcolic produces 16 different schedules in average, and finds the last data race in round 3.

Hyperconcolic shows real potential, but some of work has to be put in fixing all the special cases, that an executions might expect, because hyperconcolic will find them.

6.5 Summary

The implementation of hyperconcolic was successful and a vertical prototype exists. The prototype can take a simple Java program and check it, and report all data races.

The SVM engine is almost complete, and can handle most test cases. Logging the schedules in traces is a very successful strategy, when working with a few threads. In 45 seconds SVM stores more that 1.6 GB of traces. Even though SVM stores more data than RVPredict, it is still faster on some real life cases.

HCC is faster than RVPredict and JChord when working on small schedules. HCC and RVPredict were almost equally good at finding data races, but Huang did a little better job, probably because of the little looser definition of the conflicting operation pairs (See Equation (3.8) vs Equation (3.9)).

A working prototype of hyperconcolic has been developed, which works on a limited Java set. Running a data race detector with hyperconcolic, finds more data races than when running the data race detector alone.

CHAPTER 7

Conclusion

Hyperconcolic is a new concolic engine that tackles the problem of running automatic tests on parallel programs. Hyperconcolic can deterministically run one schedule from each schedule class ($[h]^{hc}$) by interleaving input constraints with partial order constraint.

Four different data race detection algorithms where implemented and compared. The `huang` algorithm was the most precises, but the `loose` data race detector proved to be relative precise, but where magnitudes faster. This made it an excellent filter. My implementation seams to be equivalent to the state of the art on small test-cases.

A mathematical framework for describing the relation between schedule creators and dynamic analyses described. The notion schedule class has been introduced, which describes the span of dynamic analyses. This concept is then used to evaluate the coverage of these classes by schedule creators. This framework could be used to optimize the relationship, so only one schedule from each schedule class spanned by the dynamic analysis is produced.

I have made probable that classes spanned by the `huang` algorithm ($[h]^{huang}$) and the `said` algorithm ($[h]^{said}$) both cover the hyperconcolic schedule class ($[h]^{hc}$). Using hyperconcolic with any of these techniques ensures both sound and complete data race detection in all programs which has a bounded length schedules for all inputs. Given time to improve the efficiency and to make the Java logger more robust, hyperconcolic could be an essential part of a testing environment where ensuring data race free code is key.

In total around 7000 lines of code where put into this project. This thesis also introduced the SSF and the STF formats to language independent store informations about the schedules of the program. One logger called SVM has been created which logs schedules from JVM. The SVM is a complete symbolic copy of the JVM, and handle most cases of JAVA programs. Similar loggers could be build for languages like C++11 or C#, and the data race detectors and the hyperconcolic engine would work for them too.

7.1 Future work

A good thing about this project is that there is still so many improvement, and subjects left to explore:

- Using the same techniques, it is also possible to detect dead-locks. Detecting possible deadlocks would not only be a feature, but is also important in the

sense of full analysis coverage.

- Adding search capabilities to hyperconcolic, would greatly increases its value when analysing large real life examples.
- Producing a general proof, using the execution model and the limited instruction set, that hyperconcolic allows for sound and complete analysis when combined with the `huang` or `said` schedule classes.
- Slacking the stack constraints so that the hyperconcolic class is equal to the program class. This would produce a lot fewer schedules, but there would still be enough to guarantee full schedule coverage with `huang`.
- Develop another data structure than stacks to handle the input and partial order constraints. The linear stack fits bad with the partial dependencies of the parallel constraints.
- A binary format to SSF and STF would improve the logging times of hyperconcolic. Also logging a complete ordering besides the traces, would enable HCC to build the schedules from trace streams.

APPENDIX A

Example Output Schedule

This is the output from running logging a run on HyperConcolicExample (See Listing 6.1).

```
t0.0 begin
t0.1 let a2 0r
t0.2 enter main(java.lang.String[])@HyperConcolicExample <>
t0.3 new i3 1r
t0.4 enter <init>()@HyperConcolicExample <>
t0.5 let i4 2r
@depend <1r>
@location <init>()@HyperConcolicExample 7
t0.6 write 2r i3.out@HyperConcolicExample=0r i4
t0.7 voidexit
t0.8 let s4.00000002 3v
@depend <3v>
t0.9 new a5 4r
t0.10 write 3v 1v s4.00000000
t0.11 let s4.00000000 5v
t0.12 new i6 6r
t0.13 call getClass()@java.lang.Object <1r> i7 7r
t0.14 enter <init>(HyperConcolicExample)@HyperConcolicExample$A <>
@depend <6r>
@location <init>(HyperConcolicExample)@HyperConcolicExample$A 33
t0.15 write 1r i6.this$0@HyperConcolicExample$A=0r i3
t0.16 voidexit
@depend <5v,4r>
@location main(java.lang.String[])@HyperConcolicExample 23
t0.17 write 6r a5.0=0r i6
t0.18 let s4.00000001 8v
t0.19 new i8 9r
t0.20 call getClass()@java.lang.Object <1r> i7 10r
t0.21 enter <init>(HyperConcolicExample)@HyperConcolicExample$B <>
@depend <9r>
@location <init>(HyperConcolicExample)@HyperConcolicExample$B 34
t0.22 write 1r i8.this$0@HyperConcolicExample$B=0r i3
t0.23 voidexit
@depend <8v,4r>
@location main(java.lang.String[])@HyperConcolicExample 23
t0.24 write 9r a5.1=0r i8
t0.25 let s4.00000000 11v
t0.26 let s4.0000000a 12v
```

```
t0.27 symbol 0 11v 12v s4.00000002 13v
@depend <1r>
@location main(java.lang.String[])@HyperConcolicExample 25
t0.28 write 13v i3.y@HyperConcolicExample=0v s4.00000002
t0.29 read 1v 14v s4.00000002
t0.30 let s4.00000000 15v
t0.31 binopr ge 15v 14v 16v
@location main(java.lang.String[])@HyperConcolicExample 27
t0.32 branch 2 16v false
@depend <15v,4r>
@location main(java.lang.String[])@HyperConcolicExample 27
t0.33 read a5.0=0r 17r i6
@depend <17r>
t0.34 fork t1
t0.35 let s4.00000001 18v
t0.36 binopr add 15v 18v 19v
t0.37 binopr ge 19v 14v 20v
@location main(java.lang.String[])@HyperConcolicExample 27
t0.38 branch 2 20v false
t1.0 begin
@invents 0r i6
t1.1 enter run()@HyperConcolicExample$A <>
@depend <0r>
@location run()@HyperConcolicExample$A 33
t1.2 read i6.this$0@HyperConcolicExample$A=0r 1r i3
@depend <19v,4r>
@location main(java.lang.String[])@HyperConcolicExample 27
t0.39 read a5.1=0r 21r i8
@depend <21r>
t0.40 fork t2
t2.0 begin
t0.41 let s4.00000001 22v
t0.42 binopr add 19v 22v 23v
t0.43 binopr ge 23v 14v 24v
@location main(java.lang.String[])@HyperConcolicExample 27
t0.44 branch 2 24v true
@invents 0r i8
t2.1 enter run()@HyperConcolicExample$B <>
@depend <0r>
@location run()@HyperConcolicExample$B 34
t2.2 read i8.this$0@HyperConcolicExample$B=0r 1r i3
t2.3 enter b()@HyperConcolicExample <>
t2.4 let s4.00000002 2v
@depend <1r>
@location b()@HyperConcolicExample 16
t2.5 write 2v i3.x@HyperConcolicExample=0v s4.00000002
@depend <1r>
@location b()@HyperConcolicExample 17
t2.6 read i3.y@HyperConcolicExample=0v 3v s4.00000002
t1.3 enter a()@HyperConcolicExample <>
t1.4 let s4.00000006 2v
@depend <1r>
@location a()@HyperConcolicExample 10
t1.5 write 2v i3.x@HyperConcolicExample=1v s4.00000006
@depend <1r>
```

```
@location a()@HyperConcolicExample 11
t1.6 read i3.y@HyperConcolicExample=0v 3v s4.00000002
@depend <1r>
@location b()@HyperConcolicExample 17
t2.7 read i3.y@HyperConcolicExample=0v 4v s4.00000002
t0.45 read 1v 25v s4.00000002
t1.7 let s4.00000004 4v
t1.8 binopr ge 3v 4v 5v
@location a()@HyperConcolicExample 11
t1.9 branch 0 5v false
t2.8 binopr mul 3v 4v 5v
t2.9 let s4.00000005 6v
t2.10 binopr add 5v 6v 7v
@depend <1r>
@location b()@HyperConcolicExample 17
t2.11 read i3.x@HyperConcolicExample=1v 8v s4.00000006
@depend <1r>
@location b()@HyperConcolicExample 17
t2.12 read i3.x@HyperConcolicExample=1v 9v s4.00000006
t1.10 let i9 6r
@depend <1r>
@location a()@HyperConcolicExample 12
t1.11 write 6r i3.out@HyperConcolicExample=1r i9
t0.46 let s4.00000000 26v
t0.47 binopr ge 26v 25v 27v
@location main(java.lang.String[])@HyperConcolicExample 28
t0.48 branch 3 27v false
@depend <26v,4r>
@location main(java.lang.String[])@HyperConcolicExample 28
t0.49 read a5.0=0r 28r i6
t1.12 voidexit
t1.13 voidexit
t1.14 end
t2.13 binopr mul 8v 9v 10v
t2.14 binopr ge 7v 10v 11v
@location b()@HyperConcolicExample 17
t2.15 branch 1 11v false
@depend <28r>
t0.50 join t1
t2.16 let i10 12r
@depend <1r>
@location b()@HyperConcolicExample 18
t2.17 write 12r i3.out@HyperConcolicExample=2r i10
t0.51 let s4.00000001 29v
t0.52 binopr add 26v 29v 30v
t0.53 binopr ge 30v 25v 31v
@location main(java.lang.String[])@HyperConcolicExample 28
t0.54 branch 3 31v false
t2.18 voidexit
t2.19 voidexit
t2.20 end
@depend <30v,4r>
@location main(java.lang.String[])@HyperConcolicExample 28
t0.55 read a5.1=0r 32r i8
@depend <32r>
```

```
149 t0.56 join t2
150 t0.57 let s4.00000001 33v
151 t0.58 binopr add 30v 33v 34v
152 t0.59 binopr ge 34v 25v 35v
153 @location main(java.lang.String[])@HyperConcolicExample 28
154 t0.60 branch 3 35v true
155 @location main(java.lang.String[])@HyperConcolicExample 30
156 t0.61 read i0.out@java.lang.System=0r 36r i11
157 t0.62 new i12 37r
158 t0.63 let i13 38r
159 t0.64 call append(java.lang.String)@java.lang.StringBuilder <37r,38r> i12 39
        r
160 @depend <1r>
161 @location main(java.lang.String[])@HyperConcolicExample 30
162 t0.65 read i3.out@HyperConcolicExample=2r 40r i10
163 t0.66 call append(java.lang.String)@java.lang.StringBuilder <39r,40r> i12 41
        r
164 t0.67 call toString()@java.lang.StringBuilder <41r> i14 42r
165 t0.68 voidcall println(java.lang.String)@java.io.PrintStream <36r,42r>
166 t0.69 voidexit
167 t0.70 voidexit
168 t0.71 end
```

Bibliography

[1] Clark Barrett, Aaron Stump, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.0*. Technical report. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2010.

[2] Bruno Dutertre. "Yices 2.2". In: *Computer-Aided Verification (CAV'2014)*. Edited by Armin Biere and Roderick Bloem. Volume 8559. Lecture Notes in Computer Science. Springer, July 2014, pages 737–744.

[3] Mahdi Eslamimehr and Jens Palsberg. "Race Directed Scheduling of Concurrent Programs". In: *PPoPP* (2014).

[4] Mahdi Eslamimehr and Jens Palsberg. "Sherlock: Scalable Deadlock Detection for Concurrent Programs". In: *UNPUBLISHED* (2014).

[5] Cormac Flanagan and Stephen N Freund. "Type-based race detection for Java". In: *ACM SIGPLAN Notices*. Volume 35. 5. ACM. 2000, pages 219–232.

[6] Cormac Flanagan and Patrice Godefroid. "Dynamic Partial-order Reduction for Model Checking Software". In: *SIGPLAN Not.* 40.1 (January 2005), pages 110–121. ISSN: 0362-1340. DOI: 10.1145/1047659.1040315. URL: http://doi.acm.org/10.1145/1047659.1040315.

[7] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *Systems Science and Cybernetics, IEEE Transactions on* 4.2 (1968), pages 100–107.

[8] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. "Maximal Sound Predictive Race Detection with Control Flow Abstraction". In: *PLDI* (2014).

[9] Jeff Huang, Charles Zhang, and Julian Dolby. "CLAP: recording local executions to reproduce concurrency failures". In: *ACM SIGPLAN Notices*. Volume 48. 6. ACM. 2013, pages 141–152.

[10] Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. "LCT: A parallel distributed testing tool for multithreaded Java programs". In: *Electronic Notes in Theoretical Computer Science* 296 (2013), pages 253–259.

[11] Richard E. Korf. "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search." English. In: *Artificial Intelligence* 27.1 (1985), pages 97–109. ISSN: 03742539, 00043702, 18727921.

[12] Leslie Lamport. "Proving the correctness of multiprocess programs". In: *Software Engineering, IEEE Transactions on* 2 (1977), pages 125–143.

[13] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Communications of the ACM* 21.7 (1978), pages 558–565.

[14] Shan Lu et al. “AVIO: detecting atomicity violations via access interleaving invariants”. In: *ACM SIGOPS Operating Systems Review*. Volume 40. 5. ACM. 2006, pages 37–48.

[15] Shan Lu et al. “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics”. In: *ACM Sigplan Notices*. Volume 43. 3. ACM. 2008, pages 329–339.

[16] Mayur Naik, Alex Aiken, and John Whaley. “Effective Static Race Detection for Java”. In: *PLDI* (2006), pages 308–319.

[17] Robert H. B. Netzer and Barton P. Miller. “What Are Race Conditions?: Some Issues and Formalizations”. In: *ACM Lett. Program. Lang. Syst.* 1.1 (March 1992), pages 74–88. ISSN: 1057-4514. DOI: 10.1145/130616.130623. URL: http://doi.acm.org.globalproxy.cvt.dk/10.1145/130616.130623.

[18] Robert O’Callahan and Jong-Deok Choi. “Hybrid Dynamic Data Race Detection”. In: *PPoPP* (2003), pages 167–178.

[19] Shaz Qadeer and Dinghao Wu. “KISS: keep it simple and sequential”. In: *ACM SIGPLAN Notices*. Volume 39. 6. ACM. 2004, pages 14–24.

[20] Olli Saarikivi, Kari Kahkonen, and Keijo Heljanko. “Improving dynamic partial order reductions for concolic testing”. In: *Application of Concurrency to System Design (ACSD), 2012 12th International Conference on*. IEEE. 2012, pages 132–141.

[21] Mahmoud Said et al. “Generating Data Race Witnesses by an SMT-Based Analysis”. In: *LNCS* 6617 (2014), pages 313–327.

[22] Stefan Savage et al. “Eraser: A dynamic data race detector for multithreaded programs”. In: *ACM Transactions on Computer Systems (TOCS)* 15.4 (1997), pages 391–411.

[23] Koushik Sen and Gul Agha. “A race-detection and flipping algorithm for automated testing of multi-threaded programs”. eng. In: *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Lect. Notes Comput. Sci* 4383 (2007), pages 166–182. ISSN: 16113349, 03029743.

[24] Koushik Sen, Darko Marinov, and Gul Agha. *CUTE: a concolic unit testing engine for C*. Volume 30. 5. ACM, 2005.

[25] Yannis Smaragdakis et al. “Sound predictive race detection in polynomial time”. In: *ACM SIGPLAN Notices* 47.1 (2012), pages 387–400.

[26] Lorna A Smith, J Mark Bull, and J Obdrizalek. “A parallel java grande benchmark suite”. In: *Supercomputing, ACM/IEEE 2001 Conference*. IEEE. 2001, pages 6–6.

[27] Raja Vallée-Rai et al. “Soot - a Java Bytecode Optimization Framework”. In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON ’99. Mississauga, Ontario, Canada: IBM Press, 1999, pages 13–. URL: http://dl.acm.org/citation.cfm?id=781995.782008.

[28] Willem Visser, Corina S Păsăreanu, and Sarfraz Khurshid. “Test input generation with Java PathFinder”. In: *ACM SIGSOFT Software Engineering Notes* 29.4 (2004), pages 97–107.

{2.7182818284}ϑφερτυθιοπσδφγηξκλ