Pós-Graduação em Ciência da Computação

"Rabbit: A novel approach to find data-races during state-space exploration"

By

João Paulo dos Santos Oliveira

M.Sc. Dissertation

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/-posgraduacao

Recife, August/2012

Federal University of Pernambuco
Center for Informatics
Graduate in Computer Science

João Paulo dos Santos Oliveira

"Rabbit: A novel approach to find data-races during state-space exploration"

A M.Sc. Dissertation presented to the Informatics Center of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Advisor: *Fernando José Castor de Lima Filho*
Co-Advisor: *Marcelo Bezerra d'Amorim*

Recife, August/2012

Catalogação na fonte
Bibliotecária Jane Souto Maior, CRB4-571

Oliveira, João Paulo dos Santos
Rabbit: a novel approach to find data-races during state-space exploration / João Paulo dos Santos Oliveira. - Recife: O Autor, 2012.
x, 48 f. : il., fig., tab.

Orientador: Fernando José Castor de Lima Filho.
Dissertação (mestrado) - Universidade Federal de Pernambuco. CIn, Ciência da Computação, 2012.

Inclui bibliografia.

1. Engenharia de software. 2. Teste de software. 3. Concorrência. I. Lima Filho, Fernando José Castor de (orientador). II. Título.

005.1 CDD (23. ed.) MEI2013 – 086

Dissertação de Mestrado apresentada por **João Paulo dos Santos Oliveira** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Rabbit: A novel approach to find data-races during state-space exploration**" orientada pelo Prof. **Fernando José Castor de Lima Filho** e aprovada pela Banca Examinadora formada pelos professores:

_____

Prof. Alexandre Cabral Mota
Centro de Informática / UFPE

_____

Profa. Roberta de Souza Coelho
Departamento de Informática e Matemática Aplicada/UFRN

_____

Prof. Marcelo Bezerra D'Amorim
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 30 de agosto de 2012

_____

Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Resumo

Condições de corrida são um importante tipo de erro em programação concorrentes. Software model checking(SMC) é um abordagem comum para encontrar condições de corrida. SMC explora todo o espaço de estado do programa em análise em busca de erros. Infelizmente, essa abordagem é impraticável computacionalmente em cenários que produzem um espaço de estado grande. Essa pesquisa apresenta Rabbit, uma nova abordagem para encontrar condições de corrida, complementando software model checking. Rabbit reporta eficientemente alertas de possíveis condiçções de corrida, durante a exploração do espaço de estado. Rabbit foi avaliado em 33 diferente cenários, e em 23 programas de tamanhos de diferentes. Os resultados mostraram que Rabbit encontra os erros muito rapidamente em relação ao software model checking, sendo que em 78% dos casos Rabbit encontrou a condição de corrida em menos de 5 segundos, uma fração do tempo levado pelo model checking. Também foi possível verificar que Rabbit é uma ferramenta útil para guiar a busca do model checking. Os experimentos mostraram que em 74.2% dos casos Rabbit ajudou o model cheking encontrar o erro em menos de < 20s.

Palavras-chaves: Concorrência, Verificação de Software, Model Checking, Condição de corridas

Abstract

Data-races are an important kind of error in concurrent shared-memory programs. Software model checking is a popular approach to find them. This research proposes a novel approach to find races that *complements* model-checking by efficiently reporting precise *warnings* during state-space exploration (SSE): Rabbit. It uses information obtained across different paths explored during SSE to *predict* likely racy memory accesses. We evaluated Rabbit on 33 different scenarios of race, involving a total of 21 distinct application subjects of various sources and sizes. Results indicate that Rabbit reports race warnings very soon compared to the time the model checker detects the race (for 84.8% of the cases it reports a true warning of race in <5s) and that the warnings it reports include very few false alarms. We also observed that the model checker finds the actual race quickly when it uses a guided-search that builds on Rabbit's output (for 74.2% of the cases it reports the race in <20s).

KeyWords: Concorrency, Software Verification, Model Checking, Race conditions

To my father and my mother.

Acknowledgements

This research would not have been possible without the support of many. Firstly, I would like to thank my coadvisor, Marcelo d'Amorim. I owe Marcelo for his patience and guidance. I started to work with Marcelo in January 2011 after taking class with him. Marcelo helped to focus on research that leads to this dissertation, beyond that, Marcelo motivated me in several times when I was discouraged during this work. I want to thank to my advisor Fernando Castor. Castor is an exceptional professor and with its enthusiasm I became interested in concurrent programming. I would also like to thank to all of my friends of the software engineering laboratory that helped me in my research, especially Benito, Weslley and Elton. Finally, I want to thank specially the members of my dissertation committee, the professor Roberta Coelho and Alexandre Mota for accepting the invitation and helping to improve my work.

Contents

List of Figures

List of Tables

1 Introduction

Concurrent programming is becoming more important with the increasing demand for software that runs on highly parallel multi-core machines, which are now more widespread. Many researchers believe that transitioning to multicore processor is a no back way, and software engineering should follow this way. Nowadays, all major processor manufacturers have run out of room with most of their traditional approaches to boosting CPU performance. Instead of driving clock speeds ever higher, they are moving processor architecture multicore architecture. Multicore architectures are used even in mobile phones, which supports the theory that concurrent programming will reach the mainstream programming soon. Herb Sutter, a researcher that is highly connected with the industry, says that Concurrency is the next major revolution in how developers write software [36]. Unfortunately, concurrency errors are very easy to introduce and difficult to find and reproduce, in particular, errors related to the undisciplined use of locks in programs that follow a shared-memory programming model. Even experienced programmers makes mistakes when developing concurrent programs and they agree that concurrent program is error prone and much harder than sequential one. Besides that, errors in concurrent programs are tough to discover reading code and, sometime, even more tough to discover executing the code! To prevent such errors, new languages and development methods have been recently proposed [23, 7, 39] and old approaches regained force [24]. Despite all these advances, it is still important to improve the existing support for finding concurrency errors in shared-memory programs, which remains the dominant programming model in practice. A common kind of concurrency error in shared-memory programs is a *data-race* (or simply race). The effect of a race is to modify the program state incorrectly. A race occurs when execution contains two accesses to the same memory location that are not ordered

by the happens-before relation and at least one of the accesses is a write [22]. Similar to other kinds of concurrency errors, a race typically manifests only in rare thread interleaving and these interleaving are difficult to reproduce. Programmers prevent races by protecting potentially conflicting accesses to shared data with at least one common lock. However, undisciplined use of locks can result in other problems such as inefficiency and deadlocks. Even though data-race free programs do not imply correctness (e.g., due to violations of atomicity requirements) and racy programs do not imply incorrectness (e.g., due to the need for while flags) [26], awareness of low-level data-races remains important to build correct concurrent programs.

This work proposes a novel approach to find races that *complements* model-checking by efficiently reporting precise *warnings* during state-space exploration (SSE): Rabbit. It uses information obtained across different paths explored during SSE to *predict* likely racy memory accesses. It builds on the observation that, during state space exploration, memory accesses are covered much sooner than the actual race occurs. We evaluated Rabbit on 33 different scenarios of race, involving a total of 21 distinct application subjects of various sources and sizes. Results indicate that Rabbit reports race warnings very soon compared to the time the model checker detects the race (for 84.8% of the cases it reports a true warning of race in <5s) and that the warnings it reports include very few false alarms. We also observed that the model checker finds the actual race quickly when it uses a guided-search that builds on Rabbit's output (for 74.2% of the cases it reports the race in <20s).

1.1 Contribution

The main contribution of this work is a novel approach to find data races during model checking state space exploration. This approach is called rabbit. It was inspired in the uncertain time that a developer needs to wait to receive an output during model checking. So, we think in a new approach that could give some feedback to the developer as soon as possible when him start to model checking his code. The focus of this work is not substitute model checking, instead of it we propose a technique that complements model checking and is built on top of it. Our general goal is to enable developers to take action in response to race warnings before a potentially long search for actual errors finishes. And it is exactly what our results suggest. We believe that rabbit idea is step in direction to bring model

checking usage to nonacademic world.

1.2 Dissertation Organization

This work is organized as follows:

- Chapter 2 provides an overview of the theoretical basis necessary for understanding this research, we explain important concepts like concurrency, data races and details about software model checking and Java Path finder
- Chapter 3 presents the Rabbit Idea and gives detailed information about its implementation.
- Chapter 4 introduce an evaluation of the results in a set of well know subject in verification community.
- Chapter 5 shows our final considerations, related and future works

2 Background

This chapter presents concepts and terminology to support the discussion of the remaining chapters. We first illustrate basic concepts in concurrency explaining its benefits and its hazards. Then we introduce software model checking and Java PathFinder giving details about how it works.

2.1 Concurrency

In software development, concurrency refers to a property of a system to perform computations simultaneously. Typically, concurrent computations are executed sharing time on the same processor or in different core in the same chip. When concurrent computations of the same system are executed in different processors, concurrency is called parallelism. But for purposes of this study, the differences between concurrency and parallelism are not relevant and the word will be used as synonyms.

2.1.1 Why Concurrency?

In the history of computer hardware, the number of transistors on integrated circuits double approximately every two years, while pricing keeps the same. It means that processing speed doubles every two years. So, in theory, a computer program could be executed twice faster just upgrading the hardware components in such time period. This observation became famous due to Gordon E. Moore, who described this trend in his 1965 paper [28]. This empirical observation became famous as Moore's law and the trend is approximately right until nowadays, with one important difference: In newest processor the increased transistor number

does not lead to increases speed of older programs. Instead of it, this addition of transistor gives to newer processor the capability to run several of the older programs at same time! And sometimes, each one of these several programs are executed slower than previous processors could do. It means that to a new program take advantage of modern processor, the program must be concurrent!

2.1.2 How Concurrency works?

In modern operating systems, the smallest concurrency unit is a thread[37], threads exist within a process , which are self-contained execution environment and complete abstraction of an execution program. Threads are sometimes called lightweight processes, because both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process. A process can own several threads, but each thread belongs just to one process. In the scope of this research, the term concurrency refers to a same process (program) running several threads.

Concurrency can be used in several scenarios to gain performance, to improve user experience, to provide separation of concerns(as design pattern), to handler multiples requisitions and so on, but in all cases that concurrency is useful is must coordinate and/or communicate among the threads. There are two mainstream communication strategies in concurrent programming. The first one is sharing memory, in this approach the concurrent components(threads) communicates each other through writes and reads in a common memory location. This approach is used in object oriented language as Java and C#. The other communication mechanism is message passing. In this approach the components communicate by exchanging messages, functional languages like Scala, Erlang and Haskell use this concurrent communication mechanism. The exchange of messages may be carried out asynchronously, or may use a rendezvous style in which the sender blocks until the message is received. Message-passing concurrency tends to be far easier to reason about than shared-memory concurrency, and is typically considered a more robust form of concurrent programming, however the shared memory approach is used in mainstream programming language, because is faster than message-passing. This research is focused in shared memory concurrent programs.

```
1  class SymmetricThread {
2          public static void main(String[] args) {
3                  for (int i = 0; i < 50; i++) {
4                          MyThread mt = new MyThread();
5                          mt.start();
6                  }
7          }
8  }
9
10 class MyThread extends Thread {
11         public void run() {
12                 // perform computations
13         }
14 }
```

Figure 2.1 Symmetric threads

Symmetric threads

In concurrency, a common scenario occurs when different threads are running similar(Symmetric) computations. In Java concurrent programming an usual approach that create symmetric threads is creating several instances of a same class which extends Thread. Figure 2.1 shows a Java code that creates 50 threads of `MyThread` class. All of these thread often will have the same behavior(or at least similar behavior). That is the reason they are called symmetric threads.

2.1.3 Concurrency hazards

Concurrent programming is notorious difficult and harder than sequential programming. This hardness is mostly due to communication and synchronization among different threads or due to nondeterministic. In practice, Nondeterminism means that the programmer does not know how exactly its code will execute. The operating system can nondetermistically choose which and when each thread should run. So a same program does different threads interleaving in different programs execution, and it could leave to unexpected output results. The programmer can not reproduce a program execution exactly equal to a previous one, and, sometimes, can not repeat a previous program output. Obviously, it is a barrier to debug a program that has a unknown behavior. These are some reasons why is so challenging produce a proper concurrent code.

Even though, concurrent programs are nondeterminisc, in some scenarios is

(a)

```
1  totalRequests = totalRequests + 1;
```

(b)

```
1  synchronized(totalRequestsLock) {
2      totalRequests = totalRequests + 1;
3  }
```

Figure 2.2 Lock example in Java, the keyword synchronized defines a region that is protected

important have guarantees that program's external behavior appears equal to sequential one. If a program obeys this condition, it is sequential consistent program as defined by Leslie Lamport [21]:

> A multiprocessor is Sequentially Consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

A common built-in mechanism in modern programming languages to allow writing of sequential consistent code are locks. Using locks is possible to assure that a set of locked instruction will not be executed by a thread that does not held that lock. The most common kind of lock goes by many different names. It is sometimes called a monitor, a mutex, or a binary semaphore, but regardless of the name, it provides the same basic functionality. The lock provides an scope(or an enter and exit methods) and once a thread enter in the scope its held a lock, all attempts by other threads to held this same lock will cause the other threads to block (wait) until the thread releases the lock.

The Figure 2.2(b) shows a lock usage in the java programming language, observe that keyword `synchronized` defines a region protected by a lock. When a thread holds the lock `totalRequestsLock` another thread can not get it, until it be released for the first thread. Two threads can't access the region inside lock bracket simultaneously, so it is guaranteed the mutual region [10] in this critical region.

Data Races

Suppose that a program processes requests and has a global counter, `totalRequests`, that is incremented after every request is completed. As you can see, the code that does this for a sequential program is simple as show in Figure 2.2(a)

To how understand reason that a race occurs, suppose the machine code in Figure 2.3. For sure, there is no problem in this code if is accessed only for one thread, however if this code can be reached by multiple threads, when those threads are updating `totalRequests` could lead to an unexpected value! Suppose that the compiler generate the operations to the intermediate code depicted in Figure 2.2(b)

```
1 MOV EAX, [totalRequests]   // load memory for totalRequests into register
2 INC EAX                    // update register
3 MOV [totalRequests], EAX   // store updated value back to memory
```

Figure 2.3 Intermediate code generated from a Figure 2.2(a)

Consider what would happen if two threads ran this code simultaneously. Modern operating systems don't guarantee threads scheduling order, so it is impossible to know when a interleaving will happen! So it is possible that a certain thread A loads the value in the the memory `totalRequest` to the EAX register after the thread execute the operation INC EAX and store back the updated value in the memory, after that an thread B start executing the same operations, so to this thread interleaving, the final value of `totalRequest` will be increased by two. However, suppose a different thread interleaving, as thread A load the value `totalRequest` to a processor register, after that the same thread A increases the value of the register, so the operating system interleaves to thread B and thread B loads the value from memory position pointed to `totalRequest` and load to register EAX doing the register losses the increased value and thread B executes the INC operation, store the new value the memory and the operating system interleaves back to thread A, which executes the last store operation. In this second case, the final value of `totalRequest` is different from the first interleaving! This is a low level explanation of a Data Race.

The examples shown could leave to understand that an anatomy of a Race may seem trivial, but it is not. However, the general structure for the problem is the same as for more complicated real-life races. There are four conditions needed for a race to be possible. The first condition is that there are memory locations that are accessible from more than one thread. Typically, locations are global/static variables or are heap memory reachable from global/static variables. The second condition is that there is a property associated with these shared memory locations that is needed for the program to function correctly. In this case, the property is that `totalRequests` accurately represents the total number of times any thread has

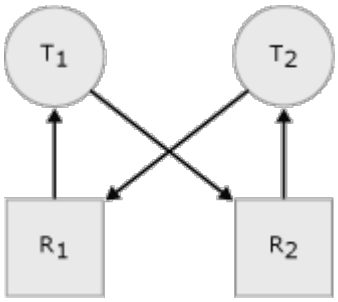


Figure 2.4 two threads in deadlock, thread T1 waits resource R1 and owns resource R2 while thread T2 waits resource T2 and owns resource R1

executed any part of the increment statement. Typically, the property needs to hold true (that is, `totalRequests` must hold an accurate count) before an update occurs for the update to be correct. The third condition is that the property does not hold during some part of the actual update. In this particular case, from the time `totalRequests` is fetched until the time it is stored, `totalRequests` does not satisfy the invariant. The fourth and final condition that must occur for a race to happen is that another thread accesses the memory when the invariant is broken, thereby causing incorrect behavior.

Deadlocks

Deadlock is concurrency hazard caused by a wrong lock usage. It is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does. This situation occurs when a thread enters a waiting state because a resource requested by it is being held by another waiting thread, which in turn is waiting for another resource. If a thread is unable to change its state indefinitely because the resources requested by it are being used by other waiting thread, then the system is said to be in a deadlock

Figure 2.4 A deadlock occurs when two or more tasks permanently block each other by each task having a lock on a resource which the other tasks are trying to lock. The following graph presents a high level view of a deadlock state where:

- Thread T1 has a lock on resource R1 (indicated by the arrow from R1 to T1) and has requested a lock on resource R2 (indicated by the arrow from T1 to R2)

- Thread T2 has a lock on resource R2 (indicated by the arrow from R2 to T2) and has requested a lock on resource R1 (indicated by the arrow from T2 to

R1).

Because neither thread can continue until a resource is available and neither resource can be released until a task continues, a deadlock state exists.

2.2 Software Model Checking

Software verification is a broader discipline of software engineering whose goal is to assure that software fully satisfies all the expected requirements. For purpose of this work, Software testing is subset of software verification and in some scenarios can be simply defined as a controlled execution of a program.

The main difference between verification and testing is that testing can only detect presence of errors, but almost never detect the lack of them, ie, proof that the program is correct. This is an even bigger problem in concurrent program due to its nondeterminisc properties, thus, classical testing approaches like unit test is not useful to find concurrency. In other hands, software verification is sound and complete, i.e., given an property(absence of an error) an verification approach can proof that the property exists or not, in other words, it can show that a program does not have an error. However, verification approach has typically high computational or human resources costs. Actually, every test/verification approach has limitations, the focus of this research is in test based approach that aims in reduce tests incompleteness and approximates it to model checking approach. So this section introduces an important verification technique: software model checking(SMC). SMC inspects if there is a property violation of a given specification(model), if has a property violation in a model the model checking will find it. Model checking is supposed to be a rigorous method that exhaustively explores all possible behaviors of given input program.

2.2.1 Java Path Finder

The JPF is a Virtual Machine (VM) for Java bytecode, which means it is a program which you give Java programs to execute. It is used to verify some program properties, so the programmer can specify property to check on a given input program. JPF virtual machine is implemented in java itself, so it is slower than typical java virtual machine, but the main purpose of JPF is to instrument and verify JPF code. JPF supports listener interface that allow instrumentation of

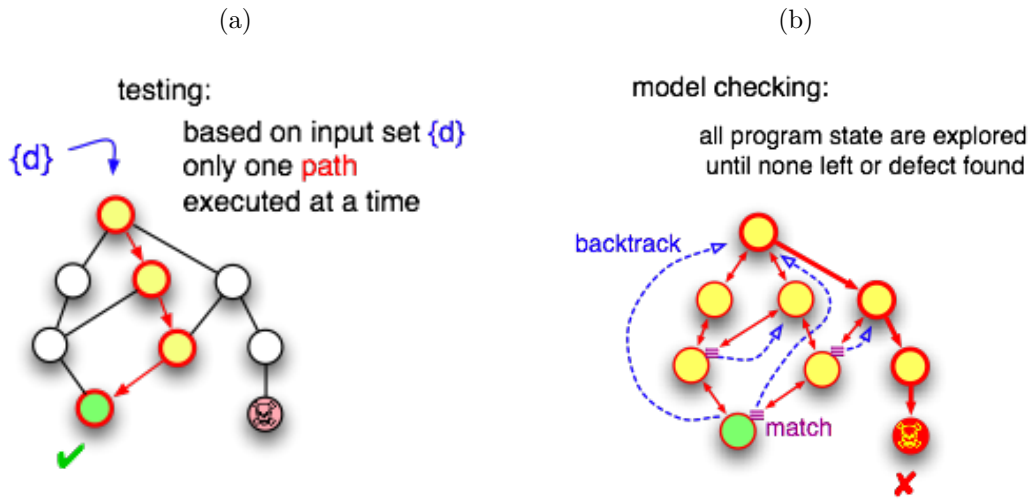


Figure 2.5 Testing VS Model Checking

bytecodes, in other words, jpf listeners are little "plugins" that let you closely monitor all actions taken by JPF, like executing single instructions, creating objects, reaching a new program state and many more. A typical example of such a listener-implemented property is a race detector, which identifies nonsynchronized access to shared variables in concurrent programs.

Model Checking with Java PathFinder

The Figure 2.5(a) depicts how a classical test behaves in subject under test(SUT). Each test can only cover a single execution path and can't detect errors in other execution path, however, in theory, model checking the same subject under test will explore all possible programs path, as depicted in Figure 2.5(b). In the scope of this research the model checking refers to program model checking as the technique described above.

The random example in Figure 2.6 illustrate a java program that does arithmetic operation in runtime generated random numbers. However, depending on the generated number the program can throws an ArithmeticException due to a division by zero. The Figure 2.7 depicts this in state space graph and illustrate the generate values to program variable `a` and `b` that leaves to a program crash.

For sure, the code in Figure 2.6 is trivial and it space state graph contains few elements, however in real world programs, the space state graph can be really huge. Model checking a real world program still intractable problem due to time restriction even using the best machines, this challenge is the state-explosion problem.

```
1  public class Rand {
2       public static void main (String[] args) {
3            Random random = new Random(42);          // (1)
4
5            int a = random.nextInt(2);               // (2)
6            System.out.println("a=" + a);
7
8            //... lots of code here
9
10           int b = random.nextInt(3);               // (3)
11           System.out.println("␣␣b=" + b);
12
13           int c = a/(b+a -2);                      // (4)
14           System.out.println("␣␣␣␣c=" + c);
15      }
16 }
```

Figure 2.6 Model Checking example

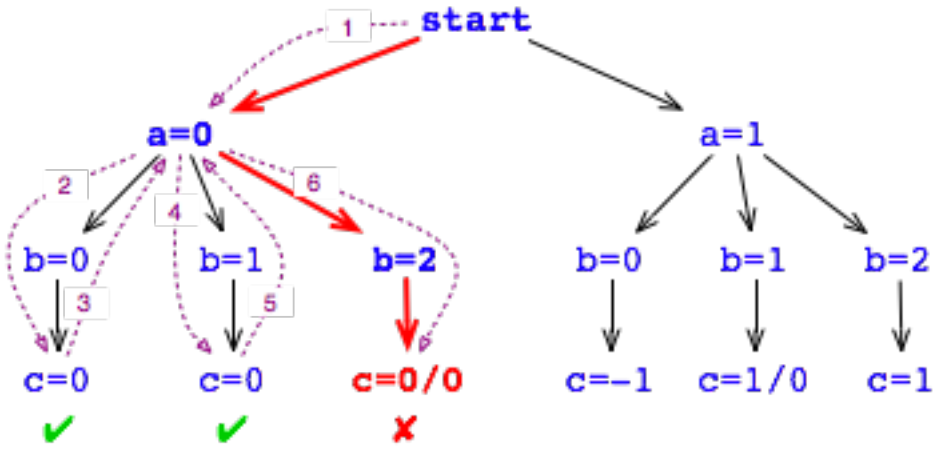


Figure 2.7 states of random program

State-Explosion Problem

With the random example, it is possible at least see the choices in the program. Assume a concurrent programming example - do you know where the operating system switches between threads? All we know is that different scheduling sequences can lead to different program behavior, but there is little we can do in our tests to force scheduling variation. There are program/test spec combinations which are "untestable". Being a virtual machine, JPF does not suffer the same fate - it has complete control over all threads of our program, and can execute all scheduling combinations.

That is the theory. In reality "all possible" can be a pretty large number - too large for existing computing resources or our patience. Just assume the number of different scheduling sequences of a program consisting of N threads P_1 .. P_N that each has n_i atomic instruction sequences. Figure 2.8 illustrates this idea, the left side of the Figure 2.8 shows the threads and its atomic instructions, which are the minor unit that can be interleaved. The Right side of the Figure shows possible thread scheduling with the atomic instructions. The number of thread interleavings is given in Equation 2.1, where M is the total number of interleavings, N is the number of threads and n_i is the number of atomic instructions in a thread

$$M = \frac{(\sum_{i=1}^{N} n_i)!}{\prod_{i=1}^{N} (n!)} \tag{2.1}$$

Using Equation 2.1, it can be realized that for 2 threads with 2 atomic sections each this gives us 6 different scheduling combinations. For 8 sections the result is 12870, 16 sections yield 601080390 - that is why it is called state-explosion.

Partial Order Reduction

State-explosion is a key problem in model checking approach, the huge number of states in typical a nontrivial program does not allows all possible orderings. The partial order reduction is aimed at reducing the size of the state space that needs to be searched. Fortunately, for most practical purposes it is not necessary to explore all possible instruction interleavings for all threads. The number of scheduling induced states can be significantly reduced by grouping all instruction sequences in

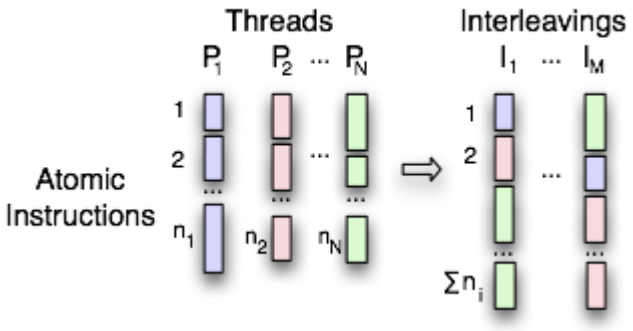


Figure 2.8 depicts threads interleavings in program with N threads

a thread that cannot have effects outside this thread itself, collapsing them into a single transition. According with JPF documentation, the JPF on-the-fly partial order reduction can typically results in more than 70% reduction of state spaces.

JPF employs an on-the-fly partial order reduction that does not rely on user instrumentation or static analysis. It automatically determines at runtime which instructions have to be treated as state transition boundaries. Explain details about partial order reduction strategies used is beyond the scope of this work, however the simplest reduction strategy adopted by JPF is instruction based, it benefits from the fact, due to the stack based nature of the JVM, that only about 10% of the Java bytecode instructions are scheduling relevant, i.e. can have effects across thread boundaries. The interesting instructions include direct synchronization (monitorEnter, monitorexit, invokeX on synchronized methods), field access (putX, getX), array element access (Xaload, Xastore), and invoke calls of certain Thread (start(), sleep(), yield(), join()) and Object methods (wait(), notify()). Based on this approach, JPF states matches scheduling that leaves to equivalent states.

Choice Generation

Software model checking is mainly about doing the right choices during states exploration, to reach the interesting system states within the resource constraints of the tool and execution environment. We refer to the mechanism used by JPF to systematically explore the state space as ChoiceGenerators JPF's ChoiceGenerator mechanism guarantees consistently use the same mechanism for scheduling choices. To understand how choice generator works, it's important to comprehend three JPF's key concepts: state, transition and choice.

State is a snapshot of the current execution status of the application (thread and heap states), plus the execution history (path) that lead to this state.

Transition is the sequence of instructions that leads from one state to the next. There is no context switch within a transition, it's all in the same thread. There can be multiple transitions leading out of one state (but not necessarily to a new state).

Choice is what starts a new transition. This can be a different thread (i.e. scheduling choice), or different "random" data value or any thing that could generate different program states.

The flexibility of JPF allow to change and configure the way that new transitions are chosen. It is possible to configure the tool to choose or prioritize a given a

thread sequence. Furthermore, it is possible to configure JPF to guide the state exploration with a given criteria. JPF has default search strategies like deep first search(DFS) and bread first search(BFS), these search defines the order that states and transitions are chosen by JPF and reflect directly on the state exploration.

Listeners

Listener are a important extension JPF mechanism. They allow to observe, interact with and extend JPF execution. They are dynamically configured at runtime, and do not requires modification in JPF internal mechanism, they work as a plug-in. There are two basic type of JPF listeners `SearchListener` and `VMListener`, both are interfaces that extend a JPF interface called `JPFListener`. The first one listen events related with JPF search during state space exploration, so the `SearchListener` allow monitors information about states. Figure 2.9 show all methods of a `SearchListener` interface

```
1  package gov.nasa.jpf.search;
2  public interface SearchListener extends JPFListener {
3    void searchStarted (Search search);
4    void stateAdvanced (Search search);       // got next state
5    void stateProcessed (Search search);
   // state is fully explored
6    void stateBacktracked (Search search);
   // state was backtracked one step (same path)
7    void stateStored (Search search);
   // somebody stored the state
8    void stateRestored (Search search);
   // previously generated state was restored (any path)
9    void propertyViolated (Search search);
   // JPF encountered a property violation
10   void searchConstraintHit (Search search); // e.g. max search depth
11   void searchFinished (Search search);
12 }}
```

Figure 2.9 SearchListener Methods.

The `VMListener` monitors events related with virtual machine itself, so with `VMListener` is possible inspect carefully all action in virtual machine. Figure 2.10 show all methods of a `VMListener` interface

A important and useful application of JPF listener mechanism is to find race condition during exploration. JPF has a native class to do it, the `PreciceRaceDetector`

class. This class is based on the idea that every time it encounter a new scheduling point that is due to a field access on a shared object, so the listener check if any of the other runnable threads is currently accessing the same field on the same object. If at least one operation is a write, the listener found race. A simplified version of this listener is presented on Figure 2.11 At every scheduling point(always that can happens this kind of choice) JPF invoke the listener method `choiceGeneratorSet` passing an instance of virtual machine to the method. the code inside refers to some basic details about JPF internals and java bytecodes, but essentially it get all threads(`ThreadInfo` class) that could be chosen in the last choice scheduling(`vm.getLastChoiceGenerator()` call) and monitor if two of them could access a same field and one of these access is write.

```
1  package gov.nasa.jpf.jvm;
2  public interface VMListener extends JPFListener {
3    //--- basic bytecode execution
4    void executeInstruction (JVM vm);  // JVM is about execute instruction
5    void instructionExecuted (JVM vm); // JVM has executed an instruction
6
7    //--- thread operations (scheduling)
8    void threadStarted (JVM vm);       // new Thread entered run()
9    void threadBlocked (JVM vm);       // thread waits to acquire a lock
10   void threadWaiting (JVM vm);       // thread is waiting for signal
11   void threadNotified (JVM vm);      // thread got notified
12   void threadInterrupted (JVM vm);   // thread got interrupted
13   void threadTerminated (JVM vm);    // Thread exited run()
14   void threadScheduled (JVM vm);     // new thread was scheduled by JVM
15
16   //--- class management
17   void classLoaded (JVM vm);         // new class was loaded
18
19   //--- object operations
20   void objectCreated (JVM vm);       // new object was created
21   void objectReleased (JVM vm);      // object was garbage collected
22   void objectLocked (JVM vm);        // object lock acquired
23   void objectUnlocked (JVM vm);      // object lock released
24   void objectWait (JVM vm);          // somebody waits for object lock
25   void objectNotify (JVM vm);        // notify single waiter for object lock
26   void objectNotifyAll (JVM vm);     // notify all waiters for object lock
27
28   void gcBegin (JVM vm);             // start garbage collection
29   void gcEnd (JVM vm);               // garbage collection finished
30
31   void exceptionThrown (JVM vm);     // exception was thrown
32
33   //--- ChoiceGenerator operations
34   void choiceGeneratorSet (JVM vm);  // new ChoiceGenerator registered
35 //new choice from current ChoiceGenerator
36   void choiceGeneratorAdvanced (JVM vm);
37 //current ChoiceGenerator processed all choices
38   void choiceGeneratorProcessed (JVM vm);
39 }
```

Figure 2.10 VMListener Methods.

```
public class PreciseRaceDetector extends PropertyListenerAdapter {
  FieldInfo raceField;
  ...
  public boolean check(Search search, JVM vm) {
    return (raceField == null);
  }
  //--- the VMListener part
  public void choiceGeneratorSet(JVM vm) {
    ChoiceGenerator<?> cg = vm.getLastChoiceGenerator();

    if (cg instanceof ThreadChoiceFromSet) {
      ThreadInfo[] threads = ((ThreadChoiceFromSet)cg).getAllThreadChoices();
      ElementInfo[] eiCandidates = new ElementInfo[threads.length];
      FieldInfo[] fiCandidates = new FieldInfo[threads.length];

      for (int i=0; i<threads.length; i++) {
        ThreadInfo ti = threads[i]; Instruction insn = ti.getPC();

        if (insn instanceof FieldInstruction) { // Ok, its a get/putfield
          FieldInstruction finsn = (FieldInstruction)insn;
          FieldInfo fi = finsn.getFieldInfo();

          if (StringSetMatcher.isMatch(fi.getFullName(), includes, excludes)) {
            ElementInfo ei = finsn.peekElementInfo(ti);

            // check if we have seen it before from another thread
            int idx=-1;
            for (int j=0; j<i; j++) {
              if ((ei == eiCandidates[j]) && (fi == fiCandidates[j])) {
                idx = j;   break;
              }
            }
            if (idx >= 0){ //we have multiple accesses on the same object/field
              Instruction otherInsn = threads[idx].getPC();
              if (isPutInsn(otherInsn) || isPutInsn(insn)) {
                raceField = ((FieldInstruction)insn).getFieldInfo();
                .. return;
              }
            } else {
              eiCandidates[i] = ei; fiCandidates[i] = fi;
            }
          }
        }
      }
    }
  }
  public void executeInstruction (JVM jvm) {
    if (raceField != null) {  // we're done, report as quickly as possible
      ThreadInfo ti = jvm.getLastThreadInfo();
      ti.breakTransition();
    }
  }
}
```

Figure 2.11 JPF native listener to find races

3

Rabbit Approach

This chapter explain Rabbit's idea to find race conditions, firstly we introduce Rabbit with an illustrative example and after that presents more details about Rabbit algoritm.

3.1 Illustrative Example

Figure 3.1(a) shows a fragment of a Java class implementing a bank account that contains a data race. We illustrate how the JPF model checker in standard mode performs to find this error and then how JPF in Rabbit mode performs. The class `Account` has one field to store the amount available in the account and a method to transfer money between accounts. The method `transfer` withdraws money from the target account object (denoted by reference `this`) and deposits money into the account object passed as parameter. Note the use of the modifier `synchronized` in the declaration of `transfer`. It has the effect of setting the target account object (i.e., `this`) as the *monitor* for the method body; hence blocking any thread that attempts to execute code that is also protected by this monitor [6]. Unfortunately, the protection used in this example is not sufficient to prevent race. The figure shows a scenario where two distinct threads make calls to method `transfer` on account objects `acc1` and `acc2`. One thread transfers money from `acc1` to `acc2` and the other does the opposite. The problem in this example is that each thread *acquires a different monitor* to protect its critical region. The first thread acquires monitor `acc1` and the second acquires monitor `acc2`. As such no synchronization is in effect; any interleaving is possible.

Figure 3.1(b) shows one particular scenario of race. To facilitate illustration, we use a thread-local artificial variable `tmp` to store values read from object fields. The

second thread reads the values of `acc2.amount` first, then the first thread executes to completion, and finally the second thread resumes using the value stored in `tmp`, which is incorrect as it differs from `acc2.amount`'s current value. Assuming that both accounts store 100 in field `amount` at the beginning of execution, the improper synchronization will result in an incorrect modification of `acc2.amount` to 97 instead of 107. This violates the state assertion in the main method as the sum of the account balances resulted in 190 instead of 200, as expected.

Java PathFinder (JPF). Software model checkers have been proposed to systematically explore the space of thread interleavings to find errors like this. Java PathFinder (JPF) is one explicit-state model checker for Java programs. It implements a Java Virtual Machine (VM), equipped with state storage and backtracking capabilities, different search strategies, and listeners for monitoring and influencing the search. For this example, JPF explores all possible thread interleavings (modulo its space reduction strategies) reachable from `Account`'s main method and takes only ∼1s to find the error above. Unfortunately, the state-space reachable from the tests of arbitrary concurrent programs can be significantly larger than this one [14, 19]. To simulate this, we considered a modified version of this main method involving 7 threads and other methods from `Account` such as `deposit` and `withdraw`; for such modified configuration, JPF took 3m39s to report the same error (see Figure 4.2).

Rabbit. Rabbit reports the corresponding warning in less than 5 seconds for this larger configuration, involving 7 threads. It reports a total of 3 different but positive warnings all of which are related to the same problem: the improper synchronization in method `transfer`. Rabbit reports 2 additional conflicts involving accesses made within transfer and other methods of `Account` that also access the field `amount`, namely the `withdraw` and `deposit` methods.

Figure 3.2(a) illustrates the Rabbit approach at the conceptual level. The figure shows with vertical lines two executing threads. The small rectangles over the line segments denote context switches and the line segments denote straight-line execution within the individual threads. Rabbit signals the presence of potential conflicts during the search by relating previously observed and current memory accesses. This is possible because Rabbit is able to relate identical objects across the state-space exploration; this is key because there is no guarantee that the id assigned by the model checker to one new object will be preserved across different exploration paths (see Section 3.2.2). The note inside the box from Figure 3.2(a) characterizes the scenario of a potential race. It is important to note that even

though this scenario shows only one schedule of the program under test, Rabbit observes and relates memory accesses across multiple schedules. It is also important to note that reported warnings are not necessarily conflicting as events r_1 and w_1 from Figure 3.2(a) could be causally related. For example, the semantics of the program could denote that the event w_1 always follows from r_1; therefore, it would not be possible to construct a scenario where these events compete for the same resources. However, we conjecture that Rabbit is precise in practice even ignoring the possibility of general causalities. The *precision* of Rabbit originates from its ability to: (i) dynamically relate object ids from potentially distinct schedules, (ii) detect simple causal relationships, and (iii) detect (and remove) similar warning reports.

The *efficiency* of Rabbit originates from the observation that memory accesses are typically covered much sooner than the actual race. For example, Figure 3.2(b) illustrates a safe scheduling of two threads that a model checker may explore. For this particular case, Rabbit needs only one schedule to report a positive warning. Note that this schedule does *not* actually manifest a race: when the second thread reads from `acc1.amount` the first thread already left its critical region. In this case, Rabbit predicts that a racy scheduling could be created as the same field from the same object was accessed (to read and to write) from different threads using disjoint lock sets {acc1} and {acc2}.

Rabbit's output. Figure 3.1(c) shows a fragment of the output of Rabbit for the main method from Figure 3.1(a). It indicates that there is a potential race in the access of field `amount` through one instance of `Account`. Note that the output of Rabbit is per object and per field and appears in tree format to indicate to which object and field an access is associated. Each line in leaf position shows if the access was a read or write, the thread that originated the access, the method and line of the access, and a list of locksets. Each distinct lockset observed across field accesses appears inside parentheses. For example, the report from Figure 3.1(c) shows that two different threads access field `amount` on `Account` object 415 using inconsistent locksets (i.e., disjoint sets of locks). While thread "Thread-4" holds locks 408 and 415 when reading field `amount`, thread "Thread-3" writes to the same field using the lock 401. This warning is in fact positive and looks similar to the (actual) race report from JPF.

```
class Account {
  double amount;
  synchronized void
    transfer(Account ac, double mn){
      this.amount = this.amount - mn;
      ac.amount = ac.amount + mn;
    }
  ...}
class TranT() extends Thread {
  Account a1, a2;  double val;
  void run() {
    a1.transfer(a2, val);
  }}
void main(String[] args) {
  Account acc1, acc2;...
  double total = acc1.amount + acc2.amount;
  TranT t1 = new TranT(acc1, acc2, 10);
  TranT t2 = new TranT(acc2, acc1, 3);
  t1.start(); t2.start();
  Assert.assertEqual(total,
    acc1.amount + acc2.amount);
}
```

(a) Fragment of `Account` subject.

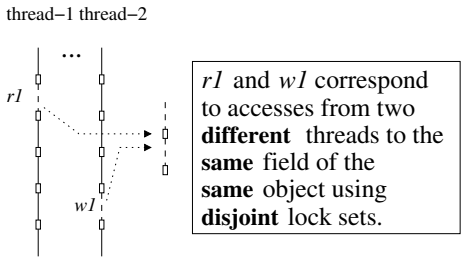
acc1.transfer(acc2, 10)	acc2.transfer(acc1, 3)
	tmp = acc2.amount;
tmp = acc1.amount;	
acc1.amount = tmp – 10;	
tmp = acc2.amount;	
acc2.amount = tmp + 10;	
	acc2.amount = tmp – 3;
	tmp = acc1.amount;
	acc1.amount = tmp + 3;

(b) Racy scheduling.

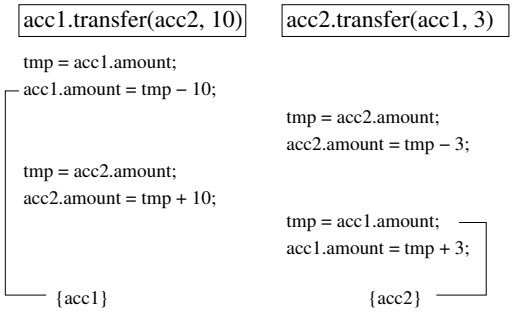
```
======== summary ========
 object Account 415
   field double amount
     write [Thread-3] Account.transfer(Account.java:27)
       [(401)]
     read  [Thread-4] Account.transfer(Account.java:31)
       [(408, 415)]  ...
======== monitors =======
 Account objects [401, 408, 415, ...]
```

(c) Rabbit warning (running on top of JPF).

Figure 3.1 Example of data race due to improper synchronization.



(a) Conceptual illustration of Rabbit.



(b) Non-racy scheduling for `Account` example, but Rabbit produces a correct warning reported.

Figure 3.2 Illustration using *one* schedule.

3.2 Approach

Rabbit relates memory accesses observed during state-space exploration to infer likely races. At conceptual level, Figures 3.2(a) and 3.2(b) illustrate Rabbit's approach.

Overview. Rabbit works by monitoring the execution of instructions during state-space exploration. Let us assume that the tuple $\langle R, o, f, t, ls \rangle$ denotes a read access (R for read and W for write) on field $o.f$ from thread t using lockset ls. Consider, for example, that Rabbit receives notification of such an event during exploration and that previously it observed, possibly along a different path, a write access of the form $\langle W, o, f, t', ls' \rangle$, where $t \neq t'$ and $ls \cap ls' = \emptyset$. Rabbit considers that this pair of accesses reveals a potential conflict: the memory location was the same, the read and write accesses were realized by different threads, and the locksets used to protect the data were disjoint. Rabbit reports this threat to the user as a warning only if it is distinct from previously reported warnings and if it fails a test that indicates a particular (simple) case of causality (see Section 3.2.1). Even though the cost of this approach is quadratic on the number of memory accesses, we only compare accesses to the same memory locations.

3.2.1 Pseudo-code

We implemented Rabbit with a listener of Java bytecodes. The listener receives notifications prior to the execution of instructions instructions that modify the heap or the static area. The listener searches for potential races that can be activated subject to those notifications. Figure 3.3 presents the pseudo-code of Rabbit. It uses two supporting data-structures to encapsulate analysis data. The class `MemAccess` represents a memory access and stores related information such as the instruction that generated the access (`insn`), the id of the target object (`objref`), metadata about which field of that object was the target of the access (`fi`), the thread that generated the access (`ti`), the locks held by that thread at that moment (`ls`), and the set of live threads at that moment (`live`). The class `WarningInfo` represents a potential race and stores a pair of `MemAccess` objects (`ma1` and `ma2`). The state of the listener consists of a history of memory accesses (`rwset`) and a set of warnings (`warnings`) that Rabbit has already reported on output. We use the historical data to relate current with previous memory accesses and use the set of stored warnings to avoid reporting to the user similar/duplicate warnings that often happens, specially

```
Map<Integer, Map<FieldInfo, Set<MemoryAccess>>> rwset;

class MemAccess {
  Instruction insn; ThreadInfo ti; LockSet ls;
  boolean isSymetric(MemAccess another){
    if(insn.equals(another.insn)
      && this.ls.equals(another.ls) && !ti.equals(another.ti)){
      return true;
    }
  return false;
  }
}
class PotRaceInfo{
  Object objRef; FieldInfo fi; MemAccess ma1, ma2; }

void executeInstruction(Instruction insn){

  if(!insn.isFieldInstruction()) return;

  ThreadInfo ti = getCurrentThread();
  // peek the object reference from stack
  int objRef = ti.peek();
  FieldInfo fi = insn.getFieldInfo();
  LockSet ls = ti.getLockSet()

  MemAccess memAccess = new MemAccess(insn, ti, ls);

  checkRace(objRef, fi, memAccess);

  updateRWSet(rwset, objRef, fi, memAccess);

}

checkRace(Integer objRef, FieldInfo fi, MemAccess ma1){

  Set<PotRaceInfo> potentialRaces;

  // pass 1: find conflicts
  foreach(ma2 in rwset.get(objRef).get(fi)){
    if (isRead(currAccess.insn) && isRead(memAccess.insn) {
      continue;
    }
    if (intersect(currAccess.ls, memAccess.ls).isEmpty()) {
      PotRaceInfo potRace =
        new PotRaceInfo(objRef, fi, currAccess, ma2);
      potentialRaces.add(potRace);
    }
  }


  // pass 2: filter symmetric
   Set<PotRaceInfo> filteredPotencialRaces;
   foreach(potRace in potencialRaces){
     if(!potRace.ma1.isSymetric(potRace.ma2)){
       filteredPotencialRaces.add(potRace);
     }
   }

  // pass 3: find simple causalities
}
```

Figure 3.3 Rabbit's pseudo-code.

for systems with symetric threads. We organized the memory access history as a nested map so to efficiently retrieve the set of observed accesses on a given field of a given object at the moment the listener observes a new access.

The method `executeInstruction` represents Rabbit's listener. It is called before the execution of each instruction and returns immediately if the instruction argument is not an instruction that manipulates a (static or instance) field (`PUTFIELD`, `GETFIELD`, `PUTSTATIC`, and `GETSTATIC`). This method accesses the state of the program to obtain all relevant information necessary to build a memory access object (e.g., object id, thread id, etc.). It then calls method `checkRace` passing this object as argument.

This call may result in the addition of a warning to the set of warnings (`warnings`). Finally, execution calls the method `updateRWSet` to add the memory access object to the access history (`rwset`). For simplicity, we omit the definition of this method and methods that access the state of the program under test (`get*`).

The method `checkRace` plays a key role in Rabbit. It receives a memory access as argument and checks, by comparing this access with previous accesses, if it can lead to a potential race. The method iterates through all previously observed memory accesses (`ma2`) that target the same object and field as the current access. If both are read accesses, then that pair cannot lead to a race and execution continues to the next access. Otherwise, if the locksets held on each access are disjoint then execution builds a `WarningInfo` object denoting a likely race. In the sequence, execution checks if it is worth reporting this warning to the user. We elaborate these additional checks in the following.

Optimization 1: Live threads check. Conceptually, we would like Rabbit to identify whether or not the two events from a warning are causally related. However, it is not only expensive to obtain such information in the context of model checking but also challenging to do it precisely when events originate from multiple exploration paths. We propose a simple heuristic to partially deal with this problem. We only report a warning if both threads that generated the access (i.e., `ma1.ti` and `ma2.ti`) are alive at the moment of each other access (see step 2). This simple check defined with method `WarningInfo.areThreadsSpawned` can capture a simple form of causal relationship. For example, without this check Rabbit reported additional false warnings for the subjects `airline`, `raxExt`, `twostages`, and `wronglock` (see Table 4.1). The scenario is manifested when, for example, one thread t_1 constructs an object o_1, which is not accessed by t_1 later, and passes that object to the constructor of thread t_2, which is spawned by t_1. Thread t_2 later accesses some field of o_1. Note that when t_1 accesses o_1, t_2 is not spawned and when t_2 is spawned t_1 will never access o_1 again.

Optimization 2: Handling Symmetric threads. Conceptually, we do not want to show repeated warnings for the case where the race can be manifested through different combinations of symmetric threads. To deal with that we check if the warning is similar to another that has been observed in the past (see step 3). The method `MemAccessNo.eqNoThread` defines this check. We observed that two race warnings that involve the same program locations and objects is sometimes manifested by different pairs of threads. This happens for the cases where the

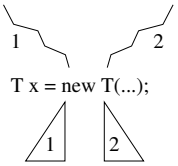


Figure 3.4 Two problems: (a) same object with different ids across distinct paths, and (b) different objects with same ids across distinct paths.

application contains symmetric threads [9]. All subjects from group 1 (see Table 4.1) with the exception of `weblench` contain symmetric threads.

3.2.2 Object Ids

Rabbit is applicable to a popular class of model checkers that manipulate the state of the program explicitly. Java PathFinder (JPF), for example, uses one array of integers to represent one object and uses one array of integer arrays to represent the entire heap. When JPF needs to explore a different path, it needs to backtrack state and that results in updating such arrays.

The identifiers that the model checker assigns to the objects created during the state-space exploration, including monitors and threads, are important to Rabbit: memory accesses are only comparable if they refer to the same memory location. We do not want Rabbit (a) to report spurious warnings because the model checker assigned the same id to semantically distinct objects across different exploration paths. Likewise, we do not want Rabbit (b) to miss a race because the model checker assigned different ids to the same object across different exploration paths. Figure 3.4 highlights these problems. A broken line denotes one execution path to assignment `T x = new T(...)` and a triangle denotes all subpaths from one of such paths. Note that the id assigned to one object will remain fixed on all subpaths (which the triangles denote) from one given execution path, however, the id for the same object will not necessarily remain fixed across differen paths. For example, a different number of objects could have been created along the execution path 2 compared to execution path 1. That would result in the assignment of a different id to the same object stored in variable `x`. Similarly, the id assigned in path 1 at `T x = ...` could be assigned to a different object in path 2.

Figure 3.5 shows the pseudo-code for the algorithm that Rabbit uses to approximately compute search-global object ids. Conceptually, it builds object ids by observing the context of allocation expressed on the state of the stack at the moment

```
1Set<Integer> visited;
2executeInstruction(Instrucion insn) {
3  if (!insn.isInit()) return;
4  int oref = getObjectRef();
5  if (visited.contains(oref)) return;
6  Stack stack = getCurrentThread().getStack();
7  Heap heap = getHeap();
8  HashData hdata = new HashData();
9  foreach (StackFrame sf : stack) {
10   // ignores local variables on the stack
11   for (int i = sf.getBase(); i < sf.top(); i++) {
12     int data = sf.slot(i);
13     int tmp = data;
14     // not considering arrays and box classes
15     if (isRef(data)) {
16       ElementInfo ei = heap.get(data);
17       tmp = ei.getOID();
18     }
19     hdata.add(tmp);
20   }
21 }
22 heap.get(oref).setOID(hdata.getValue());
23}
```

Figure 3.5 Approximate computation of search-global object Ids.

of object initialization (i.e., constructor invocation). The algorithm traverses the program stack adding integer elements denoting program values (both reference and primitive-type) to a `HashData` data-structure. This data-structure provides a `getValue()` method to compute a hash value from the integer sequence. The algorithm processes the operand stack of each stack frame it encounters during the traversal, leaving out local variables which can create noise in the characterization of objects. Finally, at line 22, the algorithm uses the method `setOID` to assign the computed hash to the abstraction of the object of interest that the model checker tracks in memory, which the expression `heap.get(oref)` denotes. Note that the object ids considered during the stack traversal are computed with this same algorithm and obtained with method `getOID`, as indicated at line 17.

3.2.3 JPF implementation

We implemented Rabbit using the infrastructure of JPF. We implented the code from Figures 3.3 and 3.5 as listeners of bytecode execution. Important to note that JPF implements a (precise) race *detector* based on the Eraser algorithm [34], but Rabbit does *not* build on JPF's race detector. While race detectors consider only

one execution path, Rabbit considers multiple paths using the history of memory accesses.

4 Evaluation

This section presents the evaluation of Rabbit.

Subjects. Figure 4.1 describes the 21 subjects we used to evaluate our approach. The subjects have different sizes and complexity and have been used in different related studies [11, 4, 2, 5, 18]. We organized the subjects in 2 groups. Group 1 includes 16 unmodified subjects containing documented races. Group 2 includes 5 subjects with no races documented. We asked 5 volunteers to introduce changes that could lead to races; we asked them *not* to use JPF or other tools to guide what to change. Experience with concurrent programming varied across volunteers. Column "group" shows the subject group, column "subject" shows subject name, column "source" indicates the source from where we obtained that subject, column "#loc" shows the number of lines of code (we used JavaNCSS [20]), and column "description" explains purpose.

Setup. We evaluated Rabbit by considering two modes in which it can be used: to report warnings about potential data races and to guide JPF in heuristic-search. For the first one, we evaluated the quality of the reported warnings and the time it took for Rabbit to produce them *(Mode 1)*. For the second one, we evaluated the capability of finding the actual race fast *(Mode 2)*. We present the results of the evaluation in terms of these two modes of use. **Test drivers.** Each subject contains a test driver (i.e., a main function) that sets up the execution environment. We changed the test drivers of some subjects from group 1 to make the number of threads a parameter. **JPF.** We ran each given test driver on JPF until it finds the error or runs out of memory. We set JPF to use its precise race detector implementation based on the Eraser algorithm [34] (`+listener=sa.jpf.listener.PreciseRaceDetec tor`) and used default values for all parameters, but disabled logging information as it

group	subject	source	#loc	description
1	account	[11]	66	Bank account
	account-pecan	[18]	148	Bank account
	airline	[11]	31	Competing passengers
	alarmclock	[11]	125	Wathdog thread
	cache4j-pecan	[18]	3,897	Cache library
	jpapa	[4]	3,903	Aerial vehicle
	lang	[11]	990	Hashcode operations
	log4j	[11]	15,744	log framework
	montecarlo	[2]	3,619	Monte Carlo Simulation
	pool	[11]	1,693	Thread pool library
	raxext	[11]	128	Calendar scheduler
	raytracer	[18]	1,924	3D raytracing
	shop	[18]	280	Shopping simulation
	twostages	[11]	52	2-stage race pattern
	weblech	[5]	1,309	Web crawler
	wronglock	[11]	38	Race sample
2	cache4j	[5]	1,096	Cache library
	cocome	[32]	2,449	Supermarket
	daisy2	[32]	880	File system
	moldyn	[2]	807	Chemical simulation
	tsp	[5]	465	Travelling Salesman

Table 4.1 Experimental subjects

can significantly affect exploration time (`+report.console.property_violation=`). We used an Intel I5 machine, with Ubuntu 11.04, and ran JPF with a maximum of 1GB of memory (default).

4.1 Results for Mode 1

This section elaborates on several experiments with the goal of quantifying the quality of the warnings reported by Rabbit during state-space exploration.

4.1.1 Comparison with JPF

We evaluated Rabbit's effectiveness by measuring (a) how *fast* it reports *useful* warnings compared to JPF and (b) how precise are the reported warnings. Figure 4.2 shows the results for this comparison. Column "subject" shows the subject name, column "err#" shows the identifier of the error, column "#thrs." shows the number of threads involved, column "JPF" shows the time required for JPF to find the error under different search-heuristics: depth-first (DFS), breadth-first

group	subject	err#	#thrs.	JPF (default=DFS)				Rabbit (DFS)	
				?	DFS	BFS	MaxPreem	5s	next
1	account	1	7		3m24s	10s	6s	3-3	-
		2	8		4h09m41s	22s	7s	3-3	-
	account-pecan	1	2		14s	14s	14s	0-0	9s 1-1
		2	10		41s	44s	48s	1-1	-
	alarmclock	1	8	*	7m01s	OME 1m51s	OME 5m48s	1-0	5m05s 2-1
		2	9	*	31m25s	OME 2m01s	OME 3m15s	1-0	24m39s 2-1
	airline	1	19		8m05s	1s	1s	2-2	-
		2	20		17m48s	1s	1s	2-2	-
	cache4j-pecan	1	2		1s	OME 8m16s	OME 3m33s	2-1	-
		2	2	*	OME 2h38m16s	OME 6m38s	OME 3m30s	2-1	-
	jpapa	1	15	*	OME 2h27m00s	OME 1m46s	OME 2m06s	0-0	1m31s 1-1
	lang	1	10		OME 20m07s	1m20	1m08s	1-1	-
	log4j	1	2		54s	27s	23s	1-1	-
	montecarlo	1	5		20s	OME 1m51s	OME 1m51s	0-0	8s 1-1
	pool	1	3	*	OME 3h59m01s	OME 5m42s	OME 2m48s	2-1	-
		2			2m31s	OME 3m24s	OME 2m58s	2-1	-
	raytracer	1	5	*	12m02	OME 4m23s	OME 5m45s	0-0	15s 1-1
	raxext	1	13		8m16s	1s	1s	2-1	-
		2	15		38m23s	1s	1s	2-1	-
	shop	1	4		OME 21h05m00s	2s	1s	1-1	-
	twostages	1	17		27m22s	2s	2s	3-1	-
		2	18		33m18s	2s	2s	3-1	-
	weblech	1	5	*	17m54s	OME 2m09s	OME 2m40s	2-2	-
	wronglock	1	18	*	6m07s	OME 2m36s	OME 2m01s	3-1	-
		2	19	*	12m39s	OME 2m01s	OME 2m01s	3-1	-
2	cache4j	1	3		15m01s	20s	18s	3-2	-
	cocome	1	4		OME 6h32m00s	5s	5s	30-30	-
		2			1s	OME 55s	OME 3m15s	5-5	-
	daisy2	1	2	*	19m24s	OME 3m55s	OME 3m34s	1-1	-
		2			4s	OME 1m43s	OME 3m08s	1-1	-
	moldyn	1	5		OME 4m26s	1s	1s	1-1	-
		2			OME 4m30s	1s	1s	0-0	1m01s 1-1
	tsp	1	3	*	NOERR 9m16s	OME 33m07s	OME 23m02s	4-4	-

Table 4.2 Results for JPF with different search strategies (DFS, BFS, and MaxPreem), Rabbit (on top of JPF with DFS search), and JPF using Rabbit-driven search heuristic. The label "*" identifies cases where all search modes in standard JPF either took more than 5m to find the error or ran out of memory.

(BFS), and maximize preemption (MaxPreem). The MaxPreem heuristic prioritizes the exploration of states that make more context-switches. We also evaluated the minimize preemption heuristic (analogous to MaxPreem and similar to CHESS [30]), but did not report results as they were poorer for all cases (i.e., rows). Column "?" highlights, with a star, cases where all of the searches either ran out of memory (OME) or took more than 10m to find the race. Column "Rabbit" shows the results of our approach, subordinate columns "5s" and "next" show respectively the warnings found in less than 5 seconds and the time required to find the first *positive* warning in case one is not found up to 5 seconds. The notation $x - y$ means that of x warnings, y are positive: the difference indicates the number of false alarms.

- We inspected all cases for which Rabbit reported false alarms and found that their presence was result of Rabbit's inability to detect causal dependencies. Note, however, that the ratios of false alarms was very low. The difference of x and y (on "x-y" cells) under column "5s" indicates the number of false alarms.

- The effectiveness of JPF varied a lot with the search strategy used. BFS and MaxPreeem search performed similarly: they raised OME in similar cases and found the error fast when did not raise OME. In constrast, the DFS search raised OME less frequently but more often took longer to find the race.

- For 11 of the 33 cases (33.3%) we analyzed from Figure 4.2, *all* JPF search-heuristics considered take more than 5m to find the error, runs out of memory, or miss the error (see rows marked with "*"). When considering each search in separate and the time-memory criteria above, DFS search falls short in 24 cases (73%) and BFS and MaxPreem search both fall short in the same ratio (48%). As expected, the effectiveness of any particular search-heuristics depends on a number of factors and subsumption should not be expected in general [15]. Rabbit reports true warnings in less than 5s for 28 out of the 33 cases (84.8%); thus improving predictability as it reports the warning consistently and improving responsiveness as it reports quality warnings to the user fast.

- For `tsp`, Rabbit reported a positive warning of error that JPF missed. Across the exploration, distinct threads access the shared variable denoting the current best path along the solving of the TSP problem. However, for any particular execution schedule explored, only one thread writes to that variable. That occurs because, for the given input and test driver, the first spawned worker thread always finds

the best path. JPF misses the error for the given input and test driver. Rabbit reports a warning as it realizes that threads with different ids access the variable without the proper protection; it ignores the fact that such accesses do not coexist in the same run. This scenario is indeed possible for this subject (i.e., multiple worker threads modifying the best path variable) and could be manifested in JPF with different inputs to the test driver.

- For `alarmclock`, Rabbit took longer to produce the first positive warning than in any of the other cases. Also the time difference between Rabbit reporting the first positive warning and JPF with DFS search finding the error was small. That happened because it took more time to cover an important memory access that enabled Rabbit to emit a positive report and JPF quickly find the race after that.

4.1.2 Impact of number of threads

We evaluated the impact of input size, more specifically the impact of the number of threads involved, in the number and precision of warning reports. For that, we used some of the subjects from group 1 which are parametric in the number of threads. For all subjects in that group we varied the number of threads from 15 to 20. For all cases, except `alarmclock`, the number and precision of the report remained the same. This results shows that Rabbit scales better overall compared to JPF, making it specially well suited to analyze parametric concurrent systems. As discussed before, for the `alarmclock` subject, Rabbit only detects the race after the model-checker covers one specific statement of the program and it takes long to cover that statement as the size of the state-space grows.

4.1.3 Impact of optimizations

We evaluated the use of each optimization in separate. Recall that optimization 1 works to reduce the number of false alarms that stem from the detection of a particular (but common) kind of causal relationship. We realized that when disabling this optimization the absolute number of false alarms raised from a total of 7 to 75, for subjects on group 1. In fact, we observed additional false alarms in *all* subjects from this group with this optimization disabled. Subjects `raxext` and `cocome` were the ones that exhibited the largest increase in number of false alarms; the first one with 20 additional alarms and the other one with 12. The goal of optimization 2 is to reduce the number of duplicate warnings (false or not).

The number of reports increased significantly when disabling this optimization: the absolute number of alarms increased from a total of 42 to 931, considering both groups. The subjects `raxext` and `airline` were responsible for the largest part of this amount (73.6%=686/931). As expected, the increase was more localized on subject with large number of symmetric threads. We observed that this optimization was effective on all subjects, i.e., Rabbit reported additional warnings without it. For some cases, the increase was not very significant. For example, for `account` Rabbit reported 7 warnings without the optimization and 3 with the optimization activated. For the case of `raxext`, Rabbit reported 496 reports with this optimization disabled.

4.1.4 Number of warnings

We evaluated how fast the number of reported warnings grow with the progress of the state-space exploration. We observed that Rabbit finds most of the errors very fast and then rarely reports new findings. For example, note from Table 4.2 the low increase in the number of reports for the cases where Rabbit cannot find the positive warning in 5s. Also, for the reported cases that Rabbit finds the races in less than 5s, no other distinct warning is reported after that.

4.1.5 Impact of search strategy

We also evaluated Rabbit with breadth-first search (BFS). As expected, the search strategy can influence results of Rabbit the same way it influences time and memory requirements during state-space exploration. We observed that of the 7 cases from Table 4.2 where Rabbit with depth-first search (DFS) could *not* find positive warning in 5s (see column "next"), Rabbit with BFS reported positive warning within 5s in 4 cases: `alarmclock` (both races), `jpapa`, and `moldyn` (one of the races). Of these 4 cases the BFS search could find the actual race quickly only for `moldyn`. Considering all the cases we considered, Rabbit with DFS performed better overall compared to Rabbit with BFS. For that reason, Table 4.2 only shows results of Rabbit running with DFS search. But we will use Rabbit with BFS later.

4.1.6 Subjects with errors removed

In a typical scenario of use of a model-checker the user is not aware about the errors the application may have. It is possible, for example, that the application does not contain races, other kinds of concurrent errors, or even errors that the

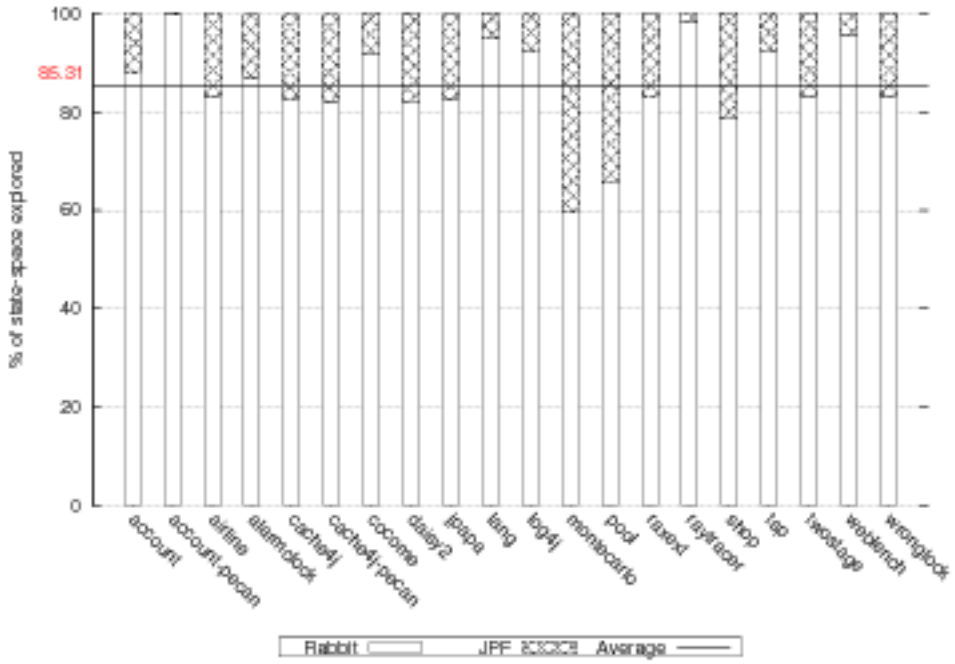


Figure 4.1 Overhead of Rabbit on state-space exploration.

test driver can reach. This experiment evaluated the magnitude of the noise from Rabbit's warnings during state-space exploration when the application does not contain known errors. To evaluate the impact of such scenario, we ran Rabbit on subjects with errors removed and found that all false positives reported persisted and no others were added. Note that considering results from Table 4.2 the ratio of false alarms is already very low.

4.1.7 Impact of search-global object ids

In principle, the same object could receive different ids across different exploration paths (and conversely) and that could influence our results. To avoid that we proposed an identity function that approximates a canonical (i.e., search-global) object identity, valid across different exploration paths (see Section 3.2.2). This section evaluates the impact of using our proposed search-global object identity function [29**?**] as opposed to the one JPF provides. It is important to note that JPF ids are preserved across portions of the search space. For example, considering Figure 3.4, JPF preserves the object ids in all sub-paths of the sub-spaces denoted by triangles 1 and 2. Considering all experimental subjects, the use of Rabbit's-provided ids found one true alarm that Rabbit with JPF-provided ids could *not* find (cocome-1). Our experimental data also reveals that Rabbit with JPF-provided

ids in fact reported relatively more true alarms (but not new) as the number of distinct JPF ids per unique object is higher. However, optimization 2 removed such duplicates accurately. To our surprise, the number of false positives and negatives remained the same overall. One reason for not missing alarms is that JPF could indeed find the races inside the sub-spaces mentioned above (where JPF ids are preserved). As for not reporting false positives one key reason was that distinct objects with identical JPF ids did not expose races because our detection operates at field-level.

4.1.8 Time Overhead

Figure 4.1 uses stacked bars to show the overhead of using Rabbit using our search-global ids. We sorted the subjects by alphabetical order of names. We disabled the seeded errors and ran each experiment for 5 minutes. The smaller bar indicates the percentage of instructions visited with Rabbit; this measure approximates the amount of the state-space explored. The bigger bar indicates the number of instructions that JPF without Rabbit visits for the allotted time. The shaded region highlights the difference. We ran each experiment for 5 times taking averages. The horizontal line marks the average time overhead across all cases: 14.69%. This overhead decreases to ∼10% when not using our proposed ids. There is no apparent correlation between the runtime overhead and the size of the application or the size of the state-space associated with the subject. Cost is proportional to the number of memory accesses made throughout the exploration (see Section 3.2.1). Note that `montecarlo` is the case with highest overhead but it is not among the largest applications. The number of field accesses involved in the numerical simulation is indeed very high, which justifies this behavior. It is very important to not that, in principle, it is *not* necessary to run Rabbit during the entire state-space exploration. As pointed out in Section 4.1.4, running Rabbit during the beginning of the search is already very beneficial. Sampling field accesses [27] sporadically or based on some other criteria is also a possible future alternative.

4.1.9 Comparison with FindBugs and JLint

We also compared Rabbit with popular pattern-based bug-finding tools, namely FindBugs [1] and JLint [3]. These tools look for particular bad practices using syntactic pattern-matching (not aware of dynamic data-relationships); we focused

on the race anti-patterns these tools support. We evaluated precision of reported warnings on our experimental subjects. It is important to note that these tools require different inputs from the user. They are typically very efficient and they do not require test drivers on input as JPF and Rabbit. We observed from the results that FindBugs reported a true alarm in only 2 cases (`account` and `shop`) and JLint in only 4 cases (`airline`, `tsp`, `moldyn`, `cache4j`). We also noted that FindBugs reported a high number of false alarms for `tsp` and `daisy2`. Considering the apparent ineffectiveness of these tools in this context and lack of space, we did not elaborate results.

4.2 Results for Mode 2

This section describes the results for an alternative mode of use of Rabbit that looks for actual errors. We first elaborate on the necessary modification to enable this mode and its results (Section 4.2.1) and then show the result of using a swarm of short-lived instances of Rabbit in this mode using different parameters (e.g., search-mode, and target warnings) (Section 4.2.2).

4.2.1 Guided-search

Rabbit observes events and infers potential problematic future memory accesses from them. This section discusses the effect of guiding the JPF search towards likely racy schedules so that warnings can be confirmed quickly. To enable this guidance, we configured JPF to use an existing search strategy class that we customized for our purposes (parameter set in JPF: `+search.class=gov.nasa.jpf.search.heuristic.PreferThreads`). This search takes as input a set of informed threads and has the effect of making JPF give higher priority to the schedules involving these threads. In our case, the threads involved in Rabbit's warnings.

Note that, with the exception of the case of `daisy2`, error number 1, JPF with our custom heuristic-search either finds the actual race in a few seconds (median time of 2s and a worst case of 18s) or runs out of memory. For `daisy2`, error number 1, even though it takes longer to find the race, it is still much faster than the DFS (both the BFS and MaxPreem produce Out of Memory errors). It runs out of memory for the same number of times as DFS search but under a different set of subject configurations. One reason for reaching memory limits with this search heuristic is that it can potentially postpone the scheduling of threads that

group	subject	err#	#thrs.	JPF (default=DFS)	Rabbit Swarm
1	account	1	7	3m24s	32s
		2	8	4h09m41s	32s
	account-pecan	1	2	14s	25s
		2	10	41s	55s
	alarmclock	1	8	7m01s	46s
		2	9	31m25s	46s
	airline	1	19	8m05s	0s
		2	20	17m48s	0s
	cache4j-pecan	1	2	1s	40s
		2	2	OME 2h38m16s	-
	jpapa	1	15	OME 2h27m00s	-
	lang	1	10	OME 20m07s	36s
	log4j	1	2	54s	31s
	montecarlo	1	5	20s	29s
	pool	1	3	OME 3h59m01s	59
		2		2m31s	-
	raytracer	1	5	12m02	-
	raxext	1	13	8m16s	7s
		2	15	38m23s	7s
	shop	1	4	OME 21h05m00s	15s
	twostages	1	17	27m22s	9s
		2	18	33m18s	9s
	weblech	1	5	17m54s	-
	wronglock	1	18	6m07s	29s
		2	19	12m39s	29s
2	cache4j	1	3	15m01s	50s
	cocome	1	4	OME 6h32m00s	-
		2		1s	13s
	daisy2	1	2	19m24s	4m27s
		2		4s	22
	moldyn	1	5	OME 4m26s	-
		2		OME 4m30s	7
	tsp	1	3	NOERR 9m16s	-

Table 4.3 Rabbit swarm results

contribute to creating intermediate states from where races becomes reachable. Note that `account-pecan` was the only case for which the search ran out of memory when JPF in standard mode did not in at least one search. Overall, this experiment shows that the Rabbit-guided heuristic-search finds the error very fast for several cases where JPF takes long to find the race or runs out of memory (see rows with "*").

4.2.2 Swarm

Overall, results show that Rabbit often reports warnings quickly and most of them are positive. Also, JPF often finds races very fast when guided by Rabbit's output. These observations suggest that it is beneficial to spawn short-lived instances of Rabbit (see Section 4.1.4) using different search strategies with the goal of finding race *warnings* quickly (see Section 4.1.5) and to spawn short-lived instances of JPF in heuristic-search mode, using the feedback from these multiple runs of Rabbit, with the goal of finding/confirming the data-races quickly (see Section 4.2.1). Based on these observations, we developed a tool that integrates these two goals: finding and confirming warnings. Conceptually, the idea is to improve model-checking by partitioning the state-space and using the multiple cores available in modern machines of these days; idea inspired by Verification Swarming [17, 16]. The tool follows a producer-consumer design, where the elements on the shared buffer are warnings. The producers of warnings are instances of Rabbit (running different search strategies) and the consumers are short-lived instances of a model-checker (JPF) that consume the generated warnings. These instances run in heuristic-search mode (Section 4.2.1); each one focuses the search on one warning it selects from the buffer. We implemented this design with 2 instances of Rabbit as producers; one running DFS and the other BFS. We spawn consumer instances on-demand based on the influx of warnings and the termination of previously spawned consumers. We limit the number of simultaneously running consumer processes running our guided-search to the number of cores in the machine, in our case 4. For each new warning, we spawn 2 new threads one with a 30s timeout and the other with a 4m timeout. The search finishes when each thread executes a new guided search(as explained 4.2.1) until timeout. Table **??** shows results of Rabbit swarm. The last column(Rabbit Swarm) shows the maximum time that the swarm approach takes to find a real race. The symbol '-' in column means that to that subject Rabbit swarm could not confirm the race in maximum of 4m after the race was reported

by Rabbit. The column JPF (default=DFS) presents the time the JPF takes to find the same error. The JPF results was also show in 4.2 and they are here just to be easy compared with new results. One stack denotes the best-case time for the JPF search (with BFS or DFS) and the other stack indicates the search time for swarming. The absence of on stack indicates it ran out of memory. We limit the y-axis to 800s for better visualization of low values (Table 4.2 shows all values). The swarming strategy was able to find the error quickly in 6 (in the worst-case in 4m30s) out of the 11 of the marked cases; it misses where the execution already ran out of memory (see column JPF(mod)+Rabbit on Table 4.2), confirming our expectation that it is one way to automate the process of generating and confirming data-race warnings.

4.2.3 Discussion

The list below summarizes the reasons *in favor* of Rabbit:

- For 84.8% (28/33) of the cases Rabbit responds very fast with positive warnings. See column "<5s" from Table 4.2. Rabbit is particularly applicable when the search produces large state-spaces. For these cases, a model-checker can take long to find the race;
- Results indicate that the amount of warnings reported does not grow unbounded during state-space exploration. For the cases we considered, the number of distinct warnings is, on average, low and stabilizes fast;
- In one particular case (`tsp`), Rabbit inferred a conflict that JPF missed. This was unexpected and showed utility beyond increased responsiveness;
- Results indicate that Rabbit performs differently with depth-first and breadth-first search. However, we observed that often when one strategy could not find a positive warning quickly the other could;
- Results indicate that Rabbit's output was very effective to guide JPF's heuristic-search, enabling JPF to find the race in a few seconds for cases where it would run out of memory or report the error in hours.

The list below summarizes the reasons *against* Rabbit:

- It can report false alarms. Rabbit is not aware of general causal relationships across the concurrent events of the application under test;

- It adds overhead to the state-space exploration time.

4.2.4 Threats to validity

One important threat to validity is the possibility of always existing one search-heuristic that could find the race very fast; hence, to compete with Rabbit, one could run all of the search-heuristics in parallel for a short period of time. We understand that even though improved diversity in search-heuristic can be helpful, it may not be always effective: one particular heuristic invariably makes hard commitments on which paths to follow [**?**], while Rabbit is a path-independent observer. Another threat is the possibility of another, not considered, race-finding tool detecting the race more efficiently and precisely than Rabbit. It is important to note that Rabbit makes progress on improving software model checking (SMC) for the particular purpose of finding races. It is possible that the subject under test contains errors that race-finding tools are unable to find but for which software model checking is still applicable. Shortly put, SMC complements these race-finding tools and Rabbit complements SMC. Other threats include our selection of subjects, our selection of search-heuristics, and lack of willingness of model checker users in inspecting warnings that may not lead to actual errors.

5 Conclusions

This research presents Rabbit, an approach to assist model checker users in finding data race errors. The general goal of our approach is to increase responsiveness of model checkers, enabling users to take action before a potentially long search for actual errors finishes. Rabbit looks for *potential* races *during* state-space exploration that can be inferred from the memory accesses it monitors. Rabbit uses a search-global identity for objects (and threads) that enables it to relate objects created across different exploration paths that the software model checker takes. We analyzed 33 cases involving 21 subjects of various sources and sizes previously used in the analysis of concurrent systems. Results indicate that the overhead in runtime compared to a regular state-space exploration is low on average, the number of false positives is low, and the reports are given most often very fast to the user. The approach of Rabbit is lightweight as to handle the high volume of data associated with observed memory accesses and the requirement to not severely affect overall exploration time (to find actual errors). Even though Rabbit's primary goal is to report true warnings quickly, we observed that our approach can be used effectively to guide a customized heuristic search and confirm the race warnings that Rabbit reported in a previous stage. To the best of our knowledge, this is the first work that exploits the synergy between predictive analysis and program model checking. Our implementation and the subjects we used in our experiments are available from the link `http://pan.cin.ufpe.br/rabbit`.

5.1 Related Work

Space reduction techniques. To alleviate the high cost associated with state-space exploration program model checkers use, often lossy, space reduction tech-

niques [25, 38, 30]. The opportunity of improvement of Rabbit is directly proportional to such high cost. Musuvathi and Qadeer [30] recently proposed CHESS to constrain the number of context switches that the model checker performs during the state-space exploration. They show substantial gain in space reduction without practical loss in the capability of finding errors. We plan to evaluate Rabbit with CHESS-like search in the future.

Model checking. A software model checker takes as input a test driver and explores the state-space of a program, reachable from the test driver, in seek of errors like races and deadlocks [13, 24, 22]. The approach is sound (i.e., it reports no false positives), however, the exploration of large state-spaces can be very expensive. In this context, time efficiency becomes an important requirement: the errors that a model checker can find are subject to one manually-written test driver and the runtime cost associated with the exploration of the state-space from that test driver often can be very high. Heuristic model checking has been investigated under different contexts in the past [15, 33]. Rungta and Mercer [33], for example, use the warnings produced by tools such as FindBugs [1] or JLint [3] to drive state-space exploration. One important difference to our approach is that we do not try to severely constrain the search space as that could prevent the search from exploring necessary states to provoke race. More precisely, our heuristic only increases the priority of selected threads for the purpose of scheduling as opposed to guiding the search towards specific program statements.

Even though it is possible to build on similar ideas to guide exploration with the output of Rabbit, this is orthogonal to our current goal. Note that the use of Rabbit does *not* interfere in search order and the capability of the model checker in finding different kinds of errors.

Predictive analysis. Predictive analysis has gained force recently as a dynamic technique to find concurrency errors [8, 35, 13, 18]. The typical approach uses a representative schedule of the program (containing, for instance, reads and writes to memory, lock acquires and releases, etc.) and, from that, infers new schedules based on some criteria. Different techniques vary in what they use to infer new schedules (e.g., causal dependencies). Considering that not all of the schedules inferred are feasible in the program, some techniques, like Penelope [35], execute the schedule to confirm (or not) the fault. The analysis involves the construction of a model to represent the space of possible schedules, the analysis of that model to produce schedules some of which may not be feasible, the instrumentation of

the program to make execution follow a particular schedule, and the execution of inferred schedules. In contrast to predictive analysis tools that produce and execute different thread schedules from some approximate model, our approach observes the feasible schedules that a model checker explores so as to infer potential races. Our approach is orthogonal to existing predictive analysis tools (even though it was inspired by them) and complementary to stateful model checking.

Static tools. Several static and dynamic race detection tools have been previously proposed. Static tools, such as RacerX [12] and Chord [31], are typically fast but can report many false alarms due the conservative assumptions that accumulate during the analysis. Static tools based on pattern matching such as FindBugs [1] or JLint [3] can in addition miss errors due to an incomplete set of supported patterns. Compared to static tools, dynamic based analyses are often more precises and reports less false positives. The warnings Rabbit reports are based on dynamic information and therefore do not suffer from the same sources of imprecision. Our approach is complementary to static tools; it builds on model checkers that require different inputs and provide different guarantees.

Language support. To facilitate development of multithreaded software, new methods [39] and language support [7] have been recently proposed and old languages regained force [24]. The approach of Rabbit complements these initiatives which aim at making concurrent software safer. It focuses on improving the support to the dominant model of concurrent programming to date, which is based on the shared-memory model.

5.2 Future work

To improve even more this work, theses are some of the next steps that will be done in the near future,

- evaluate Rabbit on other subjects, to test causality of warnings offline so as to further improve precision
- evaluate swarming with different parameters(e.g., adding and calibrating weights of preferred threads)
- investigate the use of the Rabbit approach to find deadlocks.

Bibliography

[1] FindBugs webpage. http://findbugs.sourceforge.net/.

[2] Java Grande Forum Multi-Threaded Benchmarks webpage. http://www2.epcc.ed.ac.uk/computing/research_activities.

[3] JLint webpage. http://jlint.sourceforge.net/.

[4] JPapa webpage. http://code.google.com/p/jpapabench/.

[5] Parallel Java Benchmarks webpage. http://code.google.com/p/pjbench/.

[6] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[7] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA*, pages 691–707, 2010.

[8] Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. jPredictor: a predictive runtime analysis tool for java. In *ICSE*, pages 221–230, 2008.

[9] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[10] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[11] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[12] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

[13] Malay K. Ganai. Scalable and precise symbolic analysis for atomicity violations. In *ASE*, pages 123 –132, 2011.

[14] Milos Gligoric, Vilas Jagannath, and Darko Marinov. Mutmut: Efficient exploration for mutation testing of multithreaded code. In *ICST*, pages 55 –64, 2010.

[15] Alex Groce and Willem Visser. Model checking Java programs using structural heuristics. In *ISSTA*, pages 12–21, 2002.

[16] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm Testing. "To Appear in ISSTA 2012".

[17] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification techniques. *IEEE Trans. Software Eng.*, 37(6):845–857, 2011.

[18] Jeff Huang and Charles Zhang. Persuasive prediction of concurrency access anomalies. In *ISSTA*, pages 144–154, 2011.

[19] Vilas Jagannath, Qingzhou Luo, and Darko Marinov. Change-aware preemption prioritization. In *ISSTA*, pages 133–143, 2011.

[20] JavaNCSS website. *JavaNCSS - A Source Measurement Suite for Java.* `http://www.kclee.de/clemens/java/javancss/`.

[21] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, C-28(9):690 –691, sept. 1979.

[22] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978.

[23] James Larus and Christos Kozyrakis. Transactional memory. *CACM*, 51:80–88, July 2008.

[24] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A Framework for State-Space Exploration of Java-Based Actor Programs. In *ASE*, pages 468–479, 2009.

[25] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *SPIN*, pages 80–102, 2001.

[26] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, pages 37–48, 2006.

[27] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *PLDI*, pages 134–143, 2009.

[28] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *Solid-State Circuits Newsletter, IEEE*, 11(5):33 –35, sept. 2006.

[29] Madanlal Musuvathi and David L. Dill. An incremental heap canonicalization algorithm. In *Proceedings of the International SPIN Workshop on Model Checking of Software (SPIN)*, pages 28–42, 2005.

[30] Madanlal Musuvathi and Shaz Qadeerk. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.

[31] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI*, pages 308–319, 2006.

[32] Pavel Parizek and Ondrej Lhoták. Identifying future field accesses in exhaustive state space traversal. In *ASE*, pages 93–102, 2011.

[33] Neha Rungta and Eric G. Mercer. A meta heuristic for effectively detecting concurrency errors. In *HVC*, pages 23–37, 2009.

[34] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transaction on Computer Systems*, 15:391–411, 1997.

[35] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. Penelope: weaving threads to expose atomicity violations. In *FSE*, pages 37–46, 2010.

[36] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[37] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[38] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test input generation for red-black trees using abstraction. In *ASE*, pages 414–417, 2005.

[39] Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. Cooperative reasoning for preemptive execution. In *PPoPP*, pages 147–156, 2011.