Executive Summaries in Software Model Checking

LASSE BERGLUND
Executive Summaries in Software Model Checking

LASSE BERGLUND
Abstract

Model checking is a technique used to verify whether a model meets a given specification by exhaustively and automatically checking each reachable state in the model. It is a well-developed technique, but it suffers from some issues, perhaps most importantly the state space explosion problem. Models may contain so many states that must be checked means that the model checking procedure may be intractable.

In this thesis we investigate whether procedure summaries can be used to improve the performance of model checking. Procedure summaries are concise representations of parts of a program, such as a function or method. We present a design and an implementation of dynamically generated summaries as an extension of Java PathFinder, a virtual machine executing Java bytecode that is able to model check programs written in Java by backtracking execution, to explore different schedulings etc.

We find that our summaries incur an overhead that outweighs the benefits in most cases, but the approach shows promise in certain cases, in particular when stateless model checking is used. We also provide some statistics related to cases when our summaries are applicable that could provide guidance for future work within this field.
Sammanfattning

Model checking, eller modellkontroll, är en välkänd teknik inom programverifikation som används för att verifiera att en modell, ofta av ett program, uppfyller en given specifikation genom att undersöka alla nåbara tillstånd i modellen. Det är en välutvecklad teknik som lider av några brister, en av de viktigaste är det så kallade state space explosion-problemet. Modellerna kan bestå av så många olika tillstånd att model checking inte går att använda.

I den här rapporten undersöker vi om vi kan tillämpa så kallade procedur-sammanfattningar för att förbättra prestandan av model checking. Procedur-sammanfattningar är representationer av delar av program, till exempel metoder eller funktioner. Vi presenterar en design och implementation av dynamiskt genererade sammanfattningar i form av ett tillägg till Java PathFinder, en virtuell maskin som exekverar Java bytecode som kan utföra model checking genom att backa körningar för att till exempel utforska olika schemaläggningar.

Våra procedur-sammanfattningar har i många fall en negativ effekt på körtid, men visar på lovande resultat i vissa fall, i synnerhet när så kallad stateless model checking används. Vi presenterar också resultat kopplat till fall när våra sammanfattningar är applicerbara som kan leda vägen för fortsatt arbete inom området.
Acknowledgements

First of all, I want to thank Cyrille Artho, whose guidance and advice has been invaluable. He was also very accommodating in supervising this thesis in spite of our mutual travel schedules.

I am also very grateful to Mads Dam, both for his role as examiner of this thesis, and his support and instruction more generally throughout my studies.

Thanks to Daniel Millevik and Karl Gylleus for their comments and feedback.

Thanks also to the JPF community for being a great source of help and information.

Finally, I want to thank Magnus Eriksson Thelin and Camille White, for bearing with me while I worked on this thesis.
Contents

1 Introduction 1
   1.1 Motivation ........................................ 1
   1.2 Problem Statement .............................. 2
   1.3 Contributions .................................. 2
   1.4 Delimitations .................................. 3
   1.5 Ethics and Sustainability ...................... 3

2 Background 4
   2.1 Procedure Summaries .......................... 4
   2.2 Multi-threaded Faults ......................... 5
   2.3 Model Checking ................................. 5
      2.3.1 State Representation ...................... 6
      2.3.2 Stateless Model Checking ................. 7
      2.3.3 Software Model Checking ................. 7
   2.4 Runtime Verification ........................... 8
   2.5 Java Pathfinder ................................ 8
      2.5.1 Internals .................................. 10
      2.5.2 Limitations ................................ 10
   2.6 Symbolic PathFinder ............................ 11
   2.7 Summary ....................................... 11

3 Related Work 13
   3.1 jpf-nhandler ................................. 13
   3.2 net-iocache .................................. 14
   3.3 Deterministic Blocks ......................... 15
   3.4 Summaries in Symbolic Execution ............ 16
   3.5 Summary ...................................... 18
## Contents

4 Design and Implementation 19  
  4.1 Design ........................................ 19  
  4.2 Construction in JPF .......................... 23  
  4.3 Maintaining Soundness ....................... 26  
    4.3.1 Concurrency Concerns ................ 26  
    4.3.2 Interrupting Recordings ............... 26  
  4.4 Nested Method Calls ........................ 28  
  4.5 Class and Object Initialisation .......... 29  
  4.6 String Arguments and Methods on Strings 29  
  4.7 Multiple Summaries per Method ........... 30  
  4.8 A Complete Example ....................... 30  
  4.9 Summary ...................................... 32  

5 Methods 33  
  5.1 Experiments .................................. 33  
    5.1.1 Experiment Programs .................. 33  
    5.1.2 Stateless Model Checking .............. 34  
    5.1.3 Centralised Applications .............. 34  
  5.2 Measuring Run Time ......................... 34  

6 Results 36  
  6.1 Performance .................................. 36  
    6.1.1 SIR Programs ............................ 36  
    6.1.2 Stateless Checking of SIR Programs .... 38  
    6.1.3 Centralised Applications .............. 39  
  6.2 Summary Statistics ......................... 40  
    6.2.1 Successful Recordings and Context Matches 41  
    6.2.2 Interrupted Recordings ................ 43  

7 Discussion 46  

8 Conclusion 48  

9 Future Work 49  

Bibliography 50  

A Java Bytecode 56
Chapter 1

Introduction

1.1 Motivation

As software grows steadily more integrated into our lives and at the same time becomes more and more complex, the need for verification grows as well.

Concurrency is an area that presents particular challenges, the introduction of shared memory can potentially introduce bugs that are notoriously hard to find [3]. One proposed method for verifying concurrent systems is model checking [14]. Model checking creates a model of a system and verifies properties by ensuring that they hold in all possible states of the model. The state space of the models may however grow so quickly as to make model checking intractable. Several techniques have been proposed to mitigate these issues, but the problem remains [23, 34, 32]. As such improving the speed of model checkers is still an active research area.

Another separate issue that arises in model checking software systems is the correspondence between the model and the actual software implementation [30]. If there are discrepancies between the two the verdict of the model checking procedure becomes unusable. In order to deal with this issue many modern model checking tools such as Java PathFinder (JPF) construct their model directly from the bytecode of a JVM application [52]. When model checking a concurrent program JPF has to ensure that the system is correct under all possible interleavings of the threads that make up the program. This is achieved through backtracking, where JPF will return to a previously visited state and explore different interleavings [52].
This thesis explores whether we can improve the performance of model checking by reducing the time spent on each specific trace through the use of procedure summaries. Procedure summaries are a concise representation of all the effects of a procedure. For this thesis we consider summaries to be on the level of Java methods. By applying the summary JPF can avoid re-executing each instruction in a method, given that the context in which it is called matches that of the summary.

The summaries are created dynamically as the program is executing, hence the pun in the title, by recording the dependencies of a method as its arguments, and all fields that it reads from during execution. The effects of the method are any field writes that the method performs, specifically the last write to any particular field. In order to maintain the correctness guarantees we only create summaries for methods that do not involve thread-switching.

1.2 Problem Statement

The major research questions evaluated in this thesis are the following:

1. When are procedure summaries applicable?
2. What effect do procedure summaries have on performance, specifically running time, in software model checking?

In order to answer this we present an implementation of method summaries in Java PathFinder. We also present results of our experimental evaluation of our implementation.

1.3 Contributions

The major contributions of this thesis are:

- A novel implementation of method summaries in Java PathFinder.
- Experimental results evaluating the applicability and performance of method summaries.
1.4 Delimitations

This thesis examines the effect of procedure summaries on the running time of a single model checker, Java PathFinder, on a selected suite of experiment programs used in previous research. These results may not generalise to all possible programs.

1.5 Ethics and Sustainability

This thesis aims to provide reliable and reproducible results. The implementation is available as open-source software, and most of the programs verified in our experiments were taken from the Software-artifact Infrastructure Repository [20], where programs used for evaluating many different program analysis techniques are shared between researchers ensuring a shared test-bed for controlled experiments. Code used for verification carries an additional burden of scrutiny as it is primarily used in applications where the correctness of the system is of high importance.

The impact of this thesis on sustainability seems negligible. Reduced running times could potentially be a minor contributor to reducing power consumption involved in verifying systems.
Chapter 2

Background

“At this point it should be emphasised that the verification problem and the model checking problem are mathematical problems. The specification is formulated in mathematical logic. The verification problem is distinct from the pleasantness problem [Di89] which concerns having a […] system that is truly needed and wanted.”

E. Allen Emerson,

The Beginning of Model Checking: A Personal Perspective [22]

This chapter presents the necessary background to understand the rest of the thesis. In particular we describe procedure summaries more formally, give some more background on issues raised by multi-threaded programs, and an overview of several different techniques related to model checking. We also give a brief description of Java PathFinder, the tool which our implementation of procedure summaries is built into.

2.1 Procedure Summaries

A procedure summary is defined as concise representation of a part of a program; usually a function or a method [59, 19]. They are heavily used in program analysis, as they allow for compositional verification, where previous results (the summaries) can be re-used throughout the verification process [48, 38, 33, 15]. We refer to the summaries created in this thesis as executive summaries as a pun on the term used
in business [16], while also referring to the fact that they are created dynamically during execution of code.

For the summaries we create in this thesis we consider summaries over methods in Java bytecode. We consider the dependencies of a summary to consist of the arguments to the method, and all the fields which the method reads from. Note that this does not include local variables, which are live only within the scope of the methods being analysed. Similarly we consider the effects of a method simply as the fields which the method writes to.

### 2.2 Multi-threaded Faults

Concurrent programming is a source of bugs that are notoriously difficult to resolve [3]. The introduction of several threads that are able to interact in many different ways depending on scheduling can lead to faults that only appear in rare cases. The following are some examples of common faults that arise due to concurrent programming using multiple threads:

- **Data races**, a type where a section of shared memory is concurrently accessed by two threads, where at least one of the accesses is a write. This usually results in undefined behaviour [51].

- **Incorrect initialisation** leading to uncaught exceptions. Threads may depend on resources being allocated by different threads. If this dependency is not made explicit, it is possible that certain schedulings lead to exceptions that are not handled [39].

- **Deadlocks** where a system becomes stuck and is unable to make progress. Deadlocks occur because of cyclic dependencies on locks, in the form of `synchronized` blocks in Java, or incorrect use of `wait-notify`, a separate synchronisation mechanism in Java [3].

### 2.3 Model Checking

Model checking is a verification technique based on having a model of a finite-state concurrent system and some properties, often described in temporal logic. Model checking works by exhaustively checking that the properties hold in all states of the model [13]. The primary
issue that this approach faces is the fact that the state space for systems grows exponentially with the number of processes. This is often referred to as the state space explosion problem [14]. Even so model checking has been shown to be a good way to identify the types of concurrency bugs described above, and many more [14, 57].

In most model checkers the states are explored in a systematic manner known as a search, this usually takes the form of traditional graph search algorithms, such as depth-first or breadth-first [32]. This requires that each state can be given some form of id, usually with a hash-function, so that each state is processed only once [26].

2.3.1 State Representation

While the primary problem of model checking is the size of the state space, the size of the states themselves can also pose a problem. Because of this, there have been several proposed approaches on how to represent the states of the models. An initial approach was to simply represent a state as the variables and their values. This is called explicit state model checking, and is still used in many model checkers [29]. However techniques have been introduced that can be used to compress the states leading to faster comparisons and lower memory usage [57].

Another way of reducing the size of states, and also the state space, was introduced by Burch et. al. [10]. They showed that it is often possible to gather several states within one by using a structure known as (Reduced Ordered) Binary Decision Diagrams (BDDs). This approach laid the groundwork for symbolic model checking where the checker processes sets of states rather than individual states [32]. Symbolic model checkers such as SMV [40] have proven very effective in hardware model checking [32]. BDDs are very useful when representing fixed size data structures, however, programming languages like Java make heavy use of dynamically allocated structures, which are difficult to represent symbolically [32, 47].

A separate but related technique is known as symbolic execution first introduced by King [34]. In symbolic execution concrete values are replaced by classes of values, related to their effects on the control flow of a program. For example, if a program contains one input integer variable that affects the flow $x$, and the control flow only changes if $x > y$ for some fixed $y$, the program can be analysed using a symbolic
value that represents \( x > y \) and another that represents \( x \leq y \), instead of having to consider all possible values of \( x \). This technique has also been applied in model checking [44, 46].

### 2.3.2 Stateless Model Checking

First introduced by Godefroid in his 1997 paper [26] stateless model checking is a slightly different approach where the model checking procedure does not store the states as they are explored and checked. Godefroid argued that in order to apply model checking to actual programs, one could not maintain the assumption that each state could be assigned a unique id, which is required if one performs stateful model checking. Stateless model checking cannot be applied to systems that contain cycles, as they will cause the model checking procedure to run indefinitely. Godefroid noted that a naive approach to stateless model checking will be infeasible due to the number of repeated executions of the same transitions. He proposed several methods for reducing these redundant computations, based on partial order techniques, where transitions are grouped together if they have the same effects on the system [26].

The key behind stateless search is that it reduces the amount of nondeterminism to a fixed set of schedulings [32]. The model checker then explores each schedule in turn, until a property violation occurs, or they have all been explored. While other techniques have been developed that have allowed stateful model checking to still be in use, stateless model checking is also a very active field [23, 1, 42].

### 2.3.3 Software Model Checking

Software model checking strives to change the application of model checking from models written in domain-specific languages to programs written in modern programming languages. The term was coined by Holzmann and Smith in their 2001 paper [30]. The idea was further expanded by Visser and his co-authors who presented several motivations in their landmark paper [57]. Creating the models automatically from program code solves the conformance problem; where a model is created and successfully verified, but bugs are introduced when translating this model into an actual executable program. The second major reason they presented was that it would allow the formal methods
community to take advantage of the many decades of research that has been done by the programming language community, which should make modern programming languages suitable for describing models and verification techniques. The final reason they proposed this direction for the formal methods community was that it would force the community to engage with difficult problems that would lead to new breakthroughs. In this paper they pointed to Java PathFinder as a proof of concept that this direction was feasible. A large amount of work has been done in this space since [5, 53, 23, 32].

2.4 Runtime Verification

A separate but related set of techniques for dealing with concurrency bugs is known as runtime verification. Runtime verification includes a large number of techniques that can be used for different purposes, but what connects them is that they extract information from a running system, known as traces, and use these as the basis for analysis. Unlike software model checking, runtime verification typically does not control or explore different schedulings, and is therefore imprecise [36].

Of particular interest to concurrency bugs is the Eraser algorithm [51] and similar techniques for data race and deadlock detection. They work by deriving testable properties that are stronger than data race freedom and testing these against the traces [27]. These properties are easier to test, but they may give false positives.

2.5 Java Pathfinder

Java Pathfinder (JPF) is a Virtual Machine that executes Java bytecode, with some additional features that separates it from a normal Java Virtual Machine [52]. Java PathFinder was open-sourced in 2005 [52] and has been used as a platform for many different verification techniques, not only model checking [45, 41, 6]. JPFs primary feature is the concept of execution choices. A choice point is a point where a program could proceed in at least two different ways. These usually either take the form of thread scheduling or a non-deterministic data access. JPF is able to backtrack to previously visited choice points, and explore different paths. In this way JPF executes all (necessary) different thread
While JPF does represent its states explicitly, it reduces the size of states by using state compression, where the state is reduced down to the values of a small number of memory locations. This allows for both efficient state comparison, and re-creation during backtracking [57].

JPF has a number of features to mitigate the state space explosion problem. One is state matching, where in JPF can abandon paths when it reaches a state it has already checked. Another integral part of this mitigation is Partial Order Reduction. Partial Order Reduction groups instructions that cannot have effects outside the thread they are executing in, greatly reducing the number of choice points that are created [18].

JPF also contains extensions based on runtime verification that allows it to dynamically identify data races, using an algorithm based on the Eraser algorithm [35].

One other important thing to note is the different execution levels involved in running JPF. JPF is a Virtual Machine executing an application (the System-Under-Test), JPF is running on top of a Host VM (the JVM), and that is in turn running on top of an Operating System. The diagram in Figure 2.1 illustrates how all these pieces fit together.

Figure 2.1: A diagram showing the different levels of execution involved in running JPF [18].
2.5.1 Internals

This section covers some of the JPF internals that are needed to explain our proposed design.

States, transitions, and choices

In JPF a state is a snapshot of the System-Under-Test (SUT), and the execution history that lead to that specific state. A transition meanwhile is the instructions that lead to a new state. As mentioned above, a transition takes place within a single thread, and the instructions in the transitions have no side-effects outside the thread that the instructions are executed within. Finally, choices are the points where new transitions start, as a consequence of thread-switching or non-deterministic data-access.

Listeners

As JPF is a virtual machine executing on top of the Host JVM it is fully observable. The preferred way to monitor and intercept different events within JPF is through Listeners, classes that implement an Observer pattern [58]. These listeners allow for developers to extend the execution of JPF with their own classes [18]. The data race detection extensions mentioned above are implemented as a listener (gov.nasa.jpf.listener.PreciseRaceDetector).

2.5.2 Limitations

One of JPFs primary drawbacks is the handling of native methods. These are methods that are not able to execute fully within the context of the Java Virtual Machine, but require access to platform specific features such as I/O or network interfaces. As these methods are not executing Java bytecode exclusively, JPF is not able to analyze them completely. As such executing one of these methods, for example currentTimeMillis() in Java.lang.System, JPF must manage it in some way. The feature that exists in the core of JPF is called the Model Java Interface (MJI), this acts as an analogy to the Java Native Interface (JNI) that is used by the normal JVM to delegate native calls to the underlying OS. MJI will intercept an invocation of a native method, and replace it with a similar method known as a native peer.
This native peer will run in the Host VM and change this system in the same way as the original native method. If such a native peer does not exist, JPF will terminate with an UnsatisfiedLinkException.

2.6 Symbolic PathFinder

Symbolic PathFinder (SPF) is an extension of JPF, also known as jpf-symbc [44]. SPF extends the model checking of JPF with symbolic execution, where concrete values are replaced with symbolic values that can represent a range of values. In addition to these symbolic values SPF also extends the state of JPF with a so-called path condition, which is a constraint that needs to hold in order to reach a specific part of the program. These symbolic values usually consist of numeric constraints that are extracted from source code analysis. By using constraint solving procedures SPF can then generate test inputs that are guaranteed to reach specific parts of the code. While test case generation is the primary use case for SPF, it essentially turns JPF into a symbolic model checker. The addition of symbolic values to the state of JPF makes state matching undecidable, because states now contain path conditions on unbound data, this is resolved by bounding JPF’s search depth [49]. SPF has similar drawbacks to JPF with additional caveats, for example the introduction of constraint solvers adds an additional performance overhead [46], and while recent work has enabled SPF to handle arrays with symbolic values and lengths, more complex data structures remain outside of the capabilities of SPF [24]. The reason for this last drawback is that there has been a lack of a precise and succinct way to represent symbolic values over unbounded heaps. Recent work by Pham et. al. has used separation logic to model complex heap structures [47], but the work is ongoing.

2.7 Summary

This chapter presented the general background for the thesis. Software model checking, while being a powerful technique for verifying concurrent programs, still suffers from scaling problems related to the state space explosion problem. We will use procedure summaries to try to improve the performance of Java PathFinder’s model checking. Java PathFinder is able to create models directly from programs writ-
ten in Java bytecode, but has some limitations, including not being able to verify certain methods that call native code not written in Java bytecode.
Chapter 3

Related Work

If I have not seen as far as others, it is because giants were standing on my shoulders.

---

Hal Abelson (attributed) [28]

This chapter introduces related work that serves as the scientific context for the work done in this thesis.

Java PathFinder, as mentioned before, is a virtual machine in its own right, running on top of the Java Virtual Machine. Because of this, JPF cannot handle native methods directly, but require peer classes that can either model the behaviour or delegate the calls to the underlying JVM. In this chapter we first cover some related work that deals with these issues inside JPF, and later we cover some work that used procedure summaries in the context of symbolic execution, specifically Symbolic PathFinder.

3.1 jpf-nhandler

Not all native methods have implemented peer classes. These are time-consuming to implement, and require a good understanding of JPF’s internal object representation, which is significantly different from that of the normal JVM. One proposed way of dealing with this issue was presented by Nastaran Shafiei and Franck van Breugel in the form of an extension they call jpf-nhandler [53]. jpf-nhandler is able to generate native peer classes automatically, on-the-fly (OTF) as native methods are executed. The problematic native calls are then intercepted, and
replaced with calls to the OTF peers. The tool is also configurable in such a way that native calls can be skipped.

The authors showed jpf-nhandler to work well under certain limitations and assumptions. Delegating a method call carries an assumption that the execution of that method is atomic. Certain native methods can also have problematic side-effects to the environment outside the SUT, for example writing to a database or reading from a network socket. These methods cause changes that JPF is unable to backtrack, which might lead to inconsistent behaviour during model checking. Because of this the analysis performed when running JPF with jpf-nhandler is not sound, because it may give rise to false positives. It is not complete either, because the state during the delegated calls is not captured. Despite this, the authors showed that jpf-nhandler enabled them to verify programs that were previously dependent on hand-crafted native peer classes, as well as programs that could not be verified previously because the native peer classes did not exist.

In our work we do not handle native methods that do not have peer classes already implemented. For the ones that do, we need to consider if the native method modifies the state of the system. If it does, we can not safely summarise it, because we are not able to observe it. There are however several native method calls, such as \texttt{int java.lang.Math.max(int, int)} that do not modify the state, and are only dependent on their arguments. These calls can be summarised, which has the added benefit of avoiding the redirection to the native peer class.

### 3.2 net-iocache

Another limitation of JPF that was mentioned above is the fact that native methods can modify the external environment that lies outside the scope of the System-Under-Test, which cannot be verified by JPF. This problem is particularly important for distributed applications that use network connections. When backtracking JPF may for example cause the System-Under-Test to repeat reads from a network socket. But because the other processes is not under control of JPF, and therefore not backtracked, and the corresponding write has already been consumed in a previous transition, the SUT is unable to continue. Similar issues arise when threads send messages, where backtracking might cause
repeat writes that causes the system to behave in incorrect ways.

There are several approaches that exist to deal with this limitation. One way is called stubbing, which replaces network communication calls with pre-written hard-coded responses [43, 7]. Another is centralisation where the application is rewritten in such a way that the different processes are replaced by threads, so that the system can be executed within a single process [54].

Artho et. al. introduced another approach illustrated in a JPF-extension called net-iocache [5]. This extension places a cache-layer that intercepts network communication between the SUT, and the other processes that make up the distributed system. In this way net-iocache enables each part of the distributed system to be model checked in isolation, while the peers are executed normally. This means each process has to be verified separately, however this also brings some benefits. It allows the peers that are not executed under JPF to use features like databases that could not be used if they were being verified, and it also helps to keep the state space focused to a single (usually multi-threaded) process.

In our work we also introduce a cache-layer that is populated during verification, however our goals are strictly related to performance, rather than trying to expand the scope of JPF.

### 3.3 Deterministic Blocks

In their 2006 paper d’Amorin et. al. present several optimization techniques for software model checking [17]. In their work they refer to a section of code that does not involve any thread-scheduling or non-deterministic data choices as a deterministic block. In the context of JPF we can see this as all the code within a single transition. Their techniques are aimed at speeding up the execution of these deterministic blocks.

In order to facilitate backtracking JPF uses a special representation for its state, that differs from the normal JVM. While this representation enables JPF to efficiently perform all possible executions, it incurs an overhead for each of the single executions when compared to the normal JVM. Because JPF is executing on top of a Host JVM d’Amorin et. al. proposed mixed execution, where deterministic blocks are executed on the Host JVM.
This requires a translation from the JPF state representation to that of the Host VM, and then back again at the end of the block. This translation adds an overhead, but as d’Amorin et. al. show, the increased speed at which the Host JVM is able to execute the block can outweigh the overhead introduced by the translation. The authors also note that the idea of shifting execution between different state representations is not restricted to JPF, but is also applicable to different model checkers such as AsmLT [8], BogorVM[21], or Spec Explorer[55].

Our work with method summaries also tries to increase the performance of model checking by focusing on the execution of single blocks, rather than reducing the state space or changing the order of exploration. We also try to take advantage on the difference in execution speed between the Host JVM and JPF. The summaries are constructed and applied from the Host JVM, thereby decreasing the cost of these operations. We are also restricting ourselves to deterministic blocks, in that our summaries can only be created for methods that complete a full call-return cycle within one transition. One difference is that we identify these blocks dynamically rather than relying on manually added annotations. Our context matching is similar to the lazy translation in that it evaluates only the state that is relevant to the specific method execution, though we do not translate between the state representations.

3.4 Summaries in Symbolic Execution

Symbolic execution is an analysis technique with multiple applications including test-case generation [56] and error-detection [9]. It works by replacing unspecified input values of a program with symbolic representations. These symbolic values are then used to compute path conditions for each of the paths in the program as they are executed, based on branching conditions. At each branching point, the satisfiability of the path condition is checked using external constraint solvers. Because the potentially large number of branching conditions and the computational cost of constraint solvers symbolic execution suffers from similar scalability issues as model checking, and there has been work that has used procedure summaries to manage these.

Rojas and Pasareanu presented a technique based on partial evaluation, also known as program specialization [50]. They defined a method
summary as a set of summary cases, covering all symbolic paths through a method. Each summary case represents specific path and heap constraints, and contains a specialised version of the method code for those constraints. When a method is re-executed its summary is applied, which in this case means its specialised code is executed. The specialised code of the summaries is guaranteed to contain no branch conditions, meaning that when it is applied there will be no additional solver calls. The authors showed that their summaries were able to improve the running time at somewhat high memory costs [50].

Qiu et. al. presented another technique for symbolic execution in their 2015 paper [48]. Their summaries are based on memoization trees, the major difference from the previous approach is that they do not encode the effects of the methods, but instead only captures information about the feasible paths of the methods. This means that the summaries are not applied, but instead they are enable efficient replaying. By using the additional information about the feasible paths of a method, the symbolic re-execution of a method only has to examine which of the paths are still feasible in the new calling context. After that point the method can be explored without any more calls to the constraint solver. The authors showed that by using these memoization tree-based summaries, the number of calls to constraint solvers can be greatly reduced. They showed that this increased performance of symbolic execution by as much as 88%, and reduced the number of calls to constraint solver procedures by a factor of 22 [48].

Both of these approaches improved on previous approaches that were based on logical formulas. In particular they both enable analysis of programs that modify the heap. They are also compositional as several other successful techniques [11, 12, 31] meaning they analyse a program by first analysing parts of a program, such as methods, to create summaries and then composing and using these summaries. Compositionality offers several benefits; results can be re-used, they increase scalability, and are parallelizeable if the methods do not have a caller-callee relationship [19].

Our work is similar to that of Rojas and Pasareanu, but we are looking at summaries over concrete values rather than symbolic ones. This means that we cannot create summaries for all possible feasible paths. The memoization trees in [48] are able to be stored on disk, and then loaded and re-used in a later symbolic execution, given that the methods have not been changed. Adding similar capabilities to our sum-
maries is one possible extension to the work done in this thesis.

3.5 Summary

In this chapter we have presented previous work done related to JPF and with procedure summaries. Much of the previous work has been focused on increasing the capabilities of JPF in terms of what programs it can verify, particularly by handling native methods. This has lead to performance improvements as well in certain cases [5]. We make use of the work of d’Amorin et. al. to motivate our design, in the sense that our summaries avoid executing code on the JPF VM. The work in symbolic execution showed that procedure summaries can be usefully applied in a similar scenario, though the overhead costs of constraint solvers may be larger than that of re-executing a method during model checking.
Chapter 4

Design and Implementation

This chapter describes our proposed design for procedure summaries as well as our implementation as a JPF extension, including how they deal with complex objects, and some limitations on which kinds of methods they can successfully capture.

4.1 Design

We define a summary of a Java method as $S = (P, I, O, R)$. $P$ is a an ordered list, representing the arguments of the method, by the values of the parameters. For methods that are not static, $P$ also includes a reference to the callee object (this). $I$ is a set of tuples $(id, value)$ and contains all fields that are read by the method, and the values that were read. We refer to $P$ and $I$ as the context of a method. The reason why $P$ does not contain identifiers is that the parameters of a method do not have identifiers in bytecode, they are distinguished only by ordering. $O$ is a set of tuples $(id, values)$ containing all fields that are written by the method, capturing the effects of the method. $R$ represents the return value of the method, for methods with return type void it is simply $\emptyset$.

We create a new summary through recording. We say that a method is recorded for a given context if we have a summary for that context. Recording starts at method invocation, and finishes successfully when the method returns. When we perform the modifications contained in the summary we say that the summary is applied. A summary can only be safely applied if the context in which the method is called matches that of the context stored in the summary.
Our summaries are able to capture the effects of methods that read and write to fields of complex objects. However, there are several cases where recording must be aborted, see Figure 4.1. A summary cannot capture the effects of a method that initialises new objects or classes. Recording is also aborted if the method accesses a field that is shared or static. Most native methods also interrupt recording, though some may be manually whitelisted. Interruptions may also come from different sources, separate from the method itself, if an exception is thrown, or if execution is interrupted by a thread-switch. The reasoning behind each of these interruptions is described later in the chapter.

![Figure 4.1: A diagram showing the process of recording, and possible interruptions.](image)

When the method is invoked, $P$ is set to the specific values of the arguments of the method (including `this` for non-static methods). While we are recording the method, each field instruction modifies the sum-
mary. When the method reads a field, a tuple \((\text{name}, \text{value})\) is added to \(I\). Because the input is concerned only with the pre-conditions of the method, a field is only added the first time that the method reads from it. When the method writes to a field, a tuple is instead added to \(O\). Because our summaries model the state after the method has completed, we only record the last write to a particular field. When the method returns the return value, if applicable, is stored in \(R\).

As an example, consider the method in Listing 4.1: if we call \(\text{inner}(37)\) we will create the following summary:

\[
S = ([r, 37], \{\text{this.value}, 5\}, \{(\text{this.flag}, \text{true})\}, 37).
\]

\(P\) contains the reference to the callee object, \(r\), and the argument \(x\) with the value 37. The method reads the value of the field \text{value} at the condition in the loop-statement, so it is added to \(I\). \(O\) contains the new value written the field \text{flag}. The return value is simply stored as 37, if the method returned a non-primitive type we would store a reference to the object.

Looking at the bytecode for \(\text{inner}\) in Listing 4.2, the important instructions are 4: \text{getfield}, which adds \((\text{this.value}, 5)\) to the context, and 27: \text{putfield}, which adds \((\text{this.flag}, \text{true})\) to the modifications. All the instructions related to the loop are essentially discarded by the summary, as they are not necessary to describe the effects of the method.
Listing 4.1: A small example class.

```java
public class Example {
    private boolean flag;
    private int value = 5;

    public int inner(int x) {
        for(int i=0; i<value; i++) {
            x++;
        }
        if (x == 42) {
            flag = true;
        } else {
            flag = false;
        }
        return x-5;
    }

    public void outer(int a, int b) {
        int c = inner(a);
        if (b > c) {
            System.out.println(b);
        } else {
            System.out.println(c);
        }
    }
}
```
Listing 4.2: Bytecode for the example method inner(...) in Listing 4.1. For an explanation of the bytecode instructions, see Appendix A.

```java
public int inner(int);
Code:
0: iconst_0 // line 6 put 0 on stack
1: istore_2 // line 6 i = 0
2: iload_2 // line 6 put i on stack
3:aload_0 // line 6 put this on stack
4: getfield #2 // line 6 Field value: I
7: if_icmple 19 // line 6 i < value
10: iinc 1, 1 // line 6 i++
13: iinc 2, 1 // line 7 x++
16: goto 2 // line 6 loop
19: iload_1 // line 10 put x on stack
20: bipush 42 // line 10 put 42 on stack
22: if_icmpne 33 // line 10 if x == 42
25: aload_0 // line 11 put this on stack
26: iconst_1 // line 11 put 1 (true) on stack
27: putfield #3 // line 11 Field flag: Z
30: goto 38 // line 11 exit if-statement
33: aload_0 // line 13 put this on stack
34: iconst_0 // line 13 put 0 (false) on stack
35: putfield #3 // line 13 Field flag: Z
38: iload_1 // line 16 put x on stack
39: iconst_5 // line 16 put 5 on stack
40: isub // line 16 put x-5 on stack
41: ireturn // line 16 return
```

### 4.2 Construction in JPF

In order to create the summaries inside JPF, we make use of a Listener, a class implementing the Observer pattern [58]. This Listener primarily listens to notifications related to instruction execution. When a JPF executes a method invocation, our Listener checks to see if the method has been recorded; if that is the case, the Listener compares the context of that/those recording(s) to the current state of the system. The process is illustrated in Figure 4.2. The context may contain non-primitive types. In particular, non-static methods include the callee object `this`,
for these we compare the object references, essentially the address at which the object is stored in the VM.

If the Listener gets a notification that an instruction is about to be executed, and that the instruction is an invocation and finally the Listener checks if the current state matches the recorded context. If there is a match, the invocation is skipped, and the summary is applied instead. If the method has not been recorded in that context, the Listener will instead start recording, and continue to do so until the corresponding return instruction is executed, or recording has to be stopped. There are a few reasons why we might stop recording prematurely: the method might be interrupted by a transition break, or the method might call a native peer that we are unable to summarise, or the method calls a method that has been previously blacklisted.
In order to increase the number of methods we can successfully summarise, we manually configure a so-called white-list of native methods. This list contains the names of native methods that are known to have no side-effects that we cannot summarise. Without this list we would have to abort recording unconditionally when a method calls a native method, even if the method in question does not have side-effects. Some examples of methods that we add to the white-list are `print` and `println`, because their side-effects are generally not of concern to the verification process, and `desiredAssertionStatus`, which is the native method called when evaluating an `assert`-statement in Java, a failed assertion will cause JPF to halt, while a successful assertion has no effect.

The application of a summary involves a few more steps than described above. In particular, the Listener has to propagate context information to other methods that may be recording, because the read instructions will not be executed. After that the Listener has to get the instruction that follows directly after the method invocation, remove any arguments from the current stack frame, get the return value from the summary’s `modifications` and place it on the stack, and finally set the program counter of JPF to the instruction after the method invocation.

If a method throws an exception, we stop recording. Exceptions in Java are objects that inherit from the `Exception` class. In order to throw an exception, a method will typically refer to a specific instance of an object that has a sub-type of `Exception`. These instances are often created at the point where they are thrown, and we would have to extend our summaries to store these instances across backtracking. While we could possibly store a reference to the exception, and mark this as a special return value, we chose to leave this as future work.

As the Listener is executing on the Host VM, rather than inside the JPF VM, we should see a similar performance benefit to the one described by d’Amorin et. al. in [17]. Our summaries can also be significantly smaller than the method themselves. Let us consider the method `inner` in Listing 4.1 once again: if we look at the bytecode presented in Listing 4.2, we can see how our summary represents the critical parts of the method; the `getfield`, `putfield`, and `ireturn` instructions, while disregarding the instructions that operate on local variables and stack management.
4.3 Maintaining Soundness

4.3.1 Concurrency Concerns

When a method invocation is replaced by a summary, it is possible that we hide some errors: in particular, care is needed to avoid hiding data races. We demonstrate that our summaries will maintain the soundness of the JPF model checking procedure in a case based manner.

- We create separate summaries when only one thread is alive, and when multiple threads are alive. This avoids a potential issue where a method is recorded in a single-threaded scenario, and is later called again when multiple threads are running. If we were to apply the single-thread summary in this new scenario, we might hide data races.

- Methods that use locks always create transition breaks. Regardless of whether the lock is needed to ensure safe access to some resource, JPF will still create a transition break whenever a lock is obtained or released. Because we stop recording when a transition break occurs, we will never create a summary for a method that obtains a lock, regardless of whether the lock is strictly needed or not.

- Methods that access shared fields without locks. These methods are obviously at risk for containing data races, as they do not enforce mutual exclusion, and so instead rely on the programmer to ensure that no incorrect concurrent access occurs. In this case it would not be sufficient to simply note that multiple threads are running, because the potential data race only occurs when two threads are accessing the same data, where at least one of them is a write. As such we stop recording at any point when shared data is accessed.

4.3.2 Interrupting Recordings

The following sections explains the reasons why a recording may be interrupted, as in 4.1. Some of these reasons are fundamental restrictions, such as lock access. Other, like initialisation, could potentially be included in summaries with additional engineering effort.
1. **Blacklisted**: If the method has been previously blacklisted, recording is interrupted. Whenever the recording of a method is interrupted, we add the method to a blacklist, so that we do not try to record a method that will always be interrupted. Some methods may be recordable in different contexts, but for now we blacklist methods by name.

2. **Array type**: If the method returns an array, of any type, the summaries cannot capture that method. This is due to complications introduced by how JPF represents arrays, that differs from other types. This limitation could be addressed with additional work.

3. **Initialisation**: Methods that create new objects, or classes, have to manipulate the heap of JPF. This is particularly complex in the presence of backtracking, and has therefore been left out of our summaries. As such, we interrupt recording if either of the following instructions are executed:
   - `<clinit>` — class initialisation instruction.
   - `<init>` — object initialisation instruction.

4. **Native method**: If the method calls a non-whitelisted native method, recording is stopped. Native methods pose a particular problem for our summaries, as they are executed outside of JPF, which means the effects of the method are not observable. Therefore, recording is interrupted when a native method is called. However, there are some native methods that are known to not have any side-effects such as `java.lang.Math.max`. These invocations can safely be summarised, as long as the context matches. In order to increase the number of methods we can record, our system includes a manually constructed whitelist of native methods that are known to not have side-effects.

5. **Access to shared or static fields**
   - **Shared fields**: If a method reads from or writes to a shared field, a summary could potentially hide a data-race. Therefore, we interrupt recording. This would impact the soundness of the model checking procedure, and can therefore not be safely summarised.
• **Static fields**: If a method reads from or writes to a static field we also interrupt recording. This is due to an implementation issue, where some of the internal structures of JPF were not initialised in rare cases. This could most likely be resolved with additional engineering effort. However, it was not prioritised, because our experiments showed that this interruption was rare.

6. **Transition break**: As described above, if summaries were created across different transitions, we could impact the soundness of JPF’s model checking. So, any using synchronisation mechanism, such as lock access or calling wait, will interrupt recording.

7. **Exception**: If a method throws an exception we interrupt recording. In order to summarise these methods, we would have to keep a reference to the exception object instance, and ensure that it was available when the summary should be applied. This additional work was not well motivated, given that exceptions generally are created at the point where they are thrown, meaning that recording would already be interrupted by an `<init>` instruction.

### 4.4 Nested Method Calls

Special care is required in order to ensure that we handle nested method calls in a safe manner. Consider a scenario where the program is executing `A.x()`, and during the execution of `x()` the program calls `B.y()`. In this case the summary of `x()` must be a super-set of the summary of `y()`. Our approach allows several methods to be recorded simultaneously, so that any additions that are made to the summary of `y()` are also made to the summary of `x()`. Similarly, if a method performs an operation we cannot record, such as a non-whitelisted native method call, recording is stopped for all methods that are currently recording.

As an example, consider the methods in Listing 4.1, let us assume that we have constructed the summary of `inner(37)` as before, \( S_i = ([r, 37], \{this.value, 5\}, \{(this.flag, true)\}, 37) \).

If we now call `outer(37, 13)` on the same object, the summary for `outer` will initially be \( S_o = \{(this, r), (a, 37), (b, 13)\}, \emptyset, \emptyset, \emptyset \). When `inner` is called, this expands the summary context to
\( S_0 = ([r, 37, 13], \{(this.value, 5)\}, \{(this.flag, true)\}, \emptyset) \). Note that we do not add the argument of \texttt{inner}; this is implicitly guaranteed, given that we were able to apply the summary. We also do not propagate return values to outer functions. The print statement that follows does not modify the summary.

In the case where we called \texttt{outer(37,13)} without having already summarised \texttt{inner}, we would still generate the same summary, but we would add \((this.value, 5)\) to both contexts at the same time; when the \texttt{getfield}-instruction is executed at PC offset 4 in Listing 4.2.

### 4.5 Class and Object Initialisation

In the current construction of procedure summaries in JPF, we do not summarise methods that modify the heap. In particular, we are not able to capture the effects of methods that invoke \texttt{<init>} or \texttt{<clinit>}. These bytecode instructions correspond to initialising objects and classes, respectively. These involve modifying the internal structure of the JPF heap, and operate differently when the method is being executed the first time, and when it is being re-executed after backtracking. As such, if we are recording a method and it executes either \texttt{<init>} or \texttt{<clinit>}; the Listener stops the recording, blacklists the method, and no summary is created for the method. One potential solution to this issue would be to do something similar to Qiu et. al. and replay that part of the method [48].

### 4.6 String Arguments and Methods on Strings

Methods that operate on strings are very common in Java programs [49]. However the fact that they are immutable in Java can cause our approach of comparing object references to give some false negatives when comparing contexts. For example, consider a method that calls \texttt{str.length()} on some String object \texttt{str}. Even if the string literal in \texttt{str} remains the same, the actual instance that \texttt{str} points to might be different due to backtracking, or if the method is called in a different thread. However the summary would still be valid, as the value returned by \texttt{str.length()} depends only on the string literal.

As such, we treat arguments of type String and methods on string objects slightly different from other reference objects. When a method
is called on a string, or a string is passed as an argument, we store the literal value in the context, rather than the reference. When the method is called later, we compare the literal value of the argument or caller-object, instead of its address.

### 4.7 Multiple Summaries per Method

A summary for a method in our current construction is only valid for a single context. This means that if a single argument or variable differs from the previously recorded context at the time of invocation, the summary is not valid. In order to capture several contexts we create a cache that maps method names to a list of summaries, where the list has a fixed capacity. When a method is later invoked it is checked against all the existing summaries for that method, until a point where a match is found, or until all have been compared. If none of the summaries match the current calling context and there is still capacity remaining in the cache recording starts again, otherwise the method is simply executed normally. This design is also illustrated in Figure 4.2. Our experiments did not motivate a more complicated cache invalidation strategy, where older summaries would be replaced by newer ones. But if one would extend our summaries, this may be something worth investigating.

### 4.8 A Complete Example

Listing 4.3: A short example program illustrating several aspects of our summaries.

```java
public static void main(String[] args) {
    Example ex = new Example();
    ex.inner(37);
    ex.inner(37);
    ex.inner(17);
}
```

Consider the program in Listing 4.3, using the Example class from Listing 4.1.

At the point of the first invocation of `inner` the Listener will find that there is no summaries for this method in the cache. So it will start
recording \texttt{inner}, and create a summary: 
\[ S_{i1} = ([r_{ex}, 37], I_{i1}, O_{i1}, R_{i1}); \]
where \( I_{i1} = \emptyset, O_{i1} = \emptyset, \) and \( R_{i1} = \emptyset. \) JPF will continue to execute \texttt{inner} instruction by instruction, see Listing 4.2, but the Listener will not do anything until the \texttt{getfield}-instruction has executed at PC offset 4. At this point the Listener will expand \( I_{i1} \) to \( \{(this.value, 5)\}. \) Execution will then continue, and though the \texttt{getfield} instruction is executed four more times, the Listener only adds the field to \( I \) the first time it is read.

JPF continues to execute the method normally, and the Listener does not do anything until JPF has executed the \texttt{putfield} instruction at PC offset 27. This instruction takes a reference to the \texttt{flag} field and the value 1, representing true, from the stack and updates the field. At this point the Listener adds \((this.flag, true)\) to \( O_{i1}. \)

JPF then continues to execute the method, branching to PC=38, and loads the return value on the stack, and executes the \texttt{ ireturn} instruction. At this point the Listener sets \( R_{i1} = 37, \) and adds the finished summary 
\[ S_{i1} = ([r_{ex}, 37], \{(this.value, 5)\}, \{(this.flag, true)\}, 37) \] to the cache.

At this point control returns to \texttt{main}, and the next instruction to be executed is a method invocation. The Listener interrupts JPF at this point and finds a summary in the cache for \texttt{inner}. The listener then compares the context of \( S_{i1} \) to the calling context: it finds that the callee object has the same reference, that the value of the argument \( x \) is the same in the summary and in the new invocation, and that the field \texttt{value} of the callee object is equal to 5.

Having found a summary with a matching context, the Listener removes the arguments, the reference to the callee object and 5, from JPFs stack. It then sets the field \texttt{flag} of the callee object to \texttt{true}. Finally it pushes the return value from the summary, 37, on JPFs stack, and sets the PC to the next line of \texttt{main}.

This time the Listener again sees that a method invocation is the next instruction that JPF will execute, so it looks in the cache, and finds that there is a summary for the method. While comparing the calling context to the summary \( S_{i1} \) this time, it will find that the argument \( x \) is 17, and not 37 as the summary requires. So instead of applying the summary as before, the Listener will check if there is space remaining in the cache. Given that the cache would be useless with a limit of one summary per method, we will assume that there is space remaining.

So the Listener will create a new summary \( S_{i2} = ([r_{ex}, 17], \emptyset, \emptyset, \emptyset), \)
and record as before. The method will now take the other branch of the
if-statement, but the interaction between JPF and the Listener is very
much the same. When this final call of \textit{inner} returns, the summary
will be
\[
S_{i2} = ([r_{ex}, 17], \{(this.value, 5)\}, \{(this.flag, false)\}, 17).
\]
At the end of the program the cache will contain \((Example.inner \mapsto
[S_{i1}, S_{i2}])\).

4.9 Summary

In this chapter, we have presented our implementation of procedure
summaries as a JPF extension. Each summary is valid for a single con-
text, which consists of the arguments of the method and any potential
the values of any fields that the method may read. The summaries
are able to capture field updates, and handle certain native methods
that do not have side-effects. Our summaries do not affect the sound-
ness of JPFs model checking, as they only concern events that can be
considered atomic, and do not access shared data.
Chapter 5

Methods

This chapter describes the steps taken to validate our proposed summaries, through a number of experiments. It also describes the statistical methods used to validate the results from these experiments.

5.1 Experiments

In order to evaluate the efficiency of procedure summaries we have ran a number of benchmarks. We compared the performance of running JPF with our summary extension adapted to the standard version of JPF8. All experiments were run on an Intel Core i5-4200M CPU 2.50GHz with 8GB of RAM, running Ubuntu 16.04 LTS, and Oracle’s VM version 1.8.0_151 with a memory limit of 1GB.

For each experiment (SUT) we run JPF 10 times and present the mean running times. Because the setup phase of JPF involves a large amount of variance, we only start recording the running time when the search procedure actually starts.

5.1.1 Experiment Programs

The primary source for experiments was the Software-artifact Infrastructure Repository [20], an initiative that aims to make Software Engineering research reproducible and comparable by providing a number of programs that can be used to evaluate testing and analysis techniques.

Each of the experiment targets contains some bug that can be found with JPF, some of the targets also contain a version without the bug,
denoted by -fixed; these typically take much longer to run, as JPF has to explore the entire state space.

The size and complexity of the targets varies a lot, from the very small and simple deadlock which is only 24 lines of code, to examples from real bugs from open source projects such as pool6 which involves 2043 lines of code [20]. While the larger targets are more interesting, the smaller are included for the sake of completeness.

5.1.2 Stateless Model Checking

In addition to looking at JPF’s standard model checking procedure, we also ran our experiments with state matching turned off. This effectively causes JPF to perform stateless model checking. As state matching is one of the corner-stones for JPF’s performance removing it decreases performance significantly, as such many of our experiment programs caused JPF to run out of memory, or not terminate in several hours.

5.1.3 Centralised Applications

In order to investigate the performance of our summaries on programs with particularly large state spaces, we also ran some experiments on a number of centralised applications. Centralisation is a technique that allows for model checking of distributed applications, wherein separate processes are modelled as threads [54]. This transformation typically makes the state space infeasibly large [5]. What this meant for our experiments meant that when model checking these programs JPF did not terminate within several hours. So instead of looking at the effects of summaries on running time, we observe at how many states JPF is able to explore within a fixed time with or without summaries. We ran our experiments on two applications, a simple chatserver that broadcasts messages to all clients, and a server that returns the $n$th character of the alphabet to a client. Both of these applications have been used in previous studies evaluating JPF extensions [5, 4].

5.2 Measuring Run Time

Because we only run each experiment ten times, our sample size is small, we cannot assume that the sample variance is a good approxi-
mation of the true variance. We instead apply the Student’s t-distribution with \( n - 1 \) degrees of freedom to provide a confidence interval for our measurements [25]. The Student’s t-distribution with \( n - 1 \) degrees of freedom gives a confidence interval for a given significance level \( \alpha \) from the following equations:

\[
\begin{align*}
    c_1 &= \bar{x} - t_{1 - \alpha/2, n-1} \frac{s}{\sqrt{n}} \\
    c_2 &= \bar{x} + t_{1 - \alpha/2, n-1} \frac{s}{\sqrt{n}}
\end{align*}
\] (5.1) (5.2)

where \( \bar{x} \) is the arithmetic mean of the measurements, \( s \) is the sample standard deviation, and the value of \( t_{1 - \alpha/2, n-1} \) is usually given from precomputed tables [25]. The use of this method was inspired by [2].

For this thesis we use a significance level of \( \alpha = 0.1 \).
Chapter 6

Results

This chapter presents our results, in particular the impact of our summaries on the running time or state exploration of several experiment targets using both stateful and stateless model checking. We also present some results related to the summaries themselves, including ratios of how many methods could be successfully summarised and a breakdown of reasons why methods failed to be summarised in each experiment.

6.1 Performance

This section contains the run times for the experiment programs with and without summaries, and the number of states in the case of the centralised applications.

6.1.1 SIR Programs

This section presents the results from running the programs from the Software-artifact Infrastructure Repository.

The graphs in Figures 6.1 and 6.2 present the run times with and without the summary extension enabled. We see that for many of the smaller experiment targets the differences in running time are not statistically significant, because the confidence intervals overlap at our significance level. This changes as the size and complexity of the targets increase.
Figure 6.1: Run times for SIR experiments with and without summaries measured in milliseconds. The whiskers show the confidence interval at $\alpha = 0.1$. 
Figure 6.2: Run times for SIR experiments with and without summaries measured in milliseconds. The whiskers show the confidence interval at $\alpha = 0.1$.

Table 6.1 shows the relative slowdown or speed-up when summaries are active, for those experiment targets where there was a statistically significant difference. A number greater than 1 indicates a slowdown when summaries are active, and a number less than 1 indicates a speed-up.

### 6.1.2 Stateless Checking of SIR Programs

The following section presents the run times of those programs that could be checked with JPF’s stateless model checking.

As Figure 6.3 shows doing stateless checking increases the time needed to verify the programs significantly. The programs from the previous section that are not present either caused JPF to run out of memory, or did not terminate in several hours when state matching was turned off.

Table 6.2 shows the relative slowdown or speed-up when summaries are active. A number greater than 1 indicates a slowdown, and a number less than 1 indicates a speed-up.
<table>
<thead>
<tr>
<th>Experiment</th>
<th>JPF-core (ms)</th>
<th>JPF-summary (ms)</th>
<th>Relative change</th>
</tr>
</thead>
<tbody>
<tr>
<td>boundedBuffer-orig</td>
<td>145.7</td>
<td>155.3</td>
<td>1.07</td>
</tr>
<tr>
<td>log4j1-orig</td>
<td>237.1</td>
<td>261.7</td>
<td>1.10</td>
</tr>
<tr>
<td>log4j2-orig</td>
<td>285.8</td>
<td>299.6</td>
<td>1.05</td>
</tr>
<tr>
<td>groovy-orig</td>
<td>304.7</td>
<td>340.9</td>
<td>1.12</td>
</tr>
<tr>
<td>groovy-fixed</td>
<td>639.5</td>
<td>715.0</td>
<td>1.12</td>
</tr>
<tr>
<td>pool3-orig</td>
<td>868.4</td>
<td>1218.9</td>
<td>1.40</td>
</tr>
<tr>
<td>pool6-orig</td>
<td>1081.6</td>
<td>1268.5</td>
<td>1.17</td>
</tr>
<tr>
<td>linkedlist-orig</td>
<td>1086.7</td>
<td>1410.0</td>
<td>1.30</td>
</tr>
<tr>
<td>pool4-orig</td>
<td>1272.8</td>
<td>1451.4</td>
<td>1.14</td>
</tr>
<tr>
<td>log4j3-orig</td>
<td>1474.0</td>
<td>1758.4</td>
<td>1.19</td>
</tr>
<tr>
<td>pool2-orig</td>
<td>3014.2</td>
<td>3913.7</td>
<td>1.30</td>
</tr>
<tr>
<td>pool1-orig</td>
<td>3061.6</td>
<td>3646.0</td>
<td>1.19</td>
</tr>
<tr>
<td>log4j3-fixed</td>
<td>11298.4</td>
<td>12301.4</td>
<td>1.09</td>
</tr>
<tr>
<td>log4j1-fixed</td>
<td>22358.4</td>
<td>24751.6</td>
<td>1.11</td>
</tr>
<tr>
<td>pool5-orig</td>
<td>28818.2</td>
<td>29633.9</td>
<td>1.03</td>
</tr>
<tr>
<td>pool1-fixed</td>
<td>259168.7</td>
<td>322284.8</td>
<td>1.24</td>
</tr>
<tr>
<td>pool2-fixed</td>
<td>261422.8</td>
<td>324724.4</td>
<td>1.24</td>
</tr>
<tr>
<td>pool3-fixed</td>
<td>271131.1</td>
<td>343094.4</td>
<td>1.27</td>
</tr>
<tr>
<td>pool6-fixed</td>
<td>966102.5</td>
<td>1075927.3</td>
<td>1.11</td>
</tr>
</tbody>
</table>

Table 6.1: Table summarising the results of the SIR experiments, where there was a statistically significant difference.

### 6.1.3 Centralised Applications

This section shows our results for two centralised applications, `chatserver` and `alphabetserver`. These implementations are correct, but because they are centralised they cannot be verified even after several hours. We ran each application for a fixed amount of time, and recorded how many states JPF were able to explore within this time limit.

For `alphabetserver` we do not see a statistically significant difference in the number of states that JPF is able to explore, as the confidence intervals are overlapping in Figure 6.4. So we cannot say anything about the summaries impact on JPFs performance in this case.

For `chatserver` we see a slight decrease in the number of states that JPF can explore, shown in Figure 6.5. Unlike the case of `alphabetserver` we are able to summarise several methods that are called very frequently in different contexts, meaning that the number of failed context matches increases significantly.
Figure 6.3: Run times for the stateless experiments measured in milliseconds. The whiskers show the confidence interval at $\alpha = 0.1$.

### 6.2 Summary Statistics

This section presents summary statistics; ratios of successful and failed recordings, and context matches.
<table>
<thead>
<tr>
<th>Experiment</th>
<th>JPF-core (ms)</th>
<th>JPF-summary (ms)</th>
<th>Relative change</th>
</tr>
</thead>
<tbody>
<tr>
<td>loseNotify-orig</td>
<td>85.8</td>
<td>96.8</td>
<td>1.13</td>
</tr>
<tr>
<td>twoStage-orig</td>
<td>425.6</td>
<td>464.3</td>
<td>1.09</td>
</tr>
<tr>
<td>alarmclock-orig</td>
<td>20950.4</td>
<td>22196.2</td>
<td>1.06</td>
</tr>
<tr>
<td>groovy-orig</td>
<td>31786.2</td>
<td>34923.6</td>
<td>1.10</td>
</tr>
<tr>
<td>lang-orig</td>
<td>116648.6</td>
<td>125244.6</td>
<td>1.07</td>
</tr>
<tr>
<td>pool3-orig</td>
<td>386734.6</td>
<td>320813.3</td>
<td>0.83</td>
</tr>
</tbody>
</table>

Table 6.2: Table summarising the results of the stateless experiments, where there was a statistically significant difference.

Figure 6.4: Visited states of alphabetsever in a set number of minutes with and without summaries, averaged across 10 runs. The whiskers show the confidence interval at $\alpha = 0.1$.

## 6.2.1 Successful Recordings and Context Matches

We observe a large amount of variance in the number of methods that we are able to summarise. Figure 6.6a shows the ratio of methods that are summarised for each experiment program. The lowest is pool4 where we summarise 5% and the highest is in pool3-fixed where summarise 27%. On average we are able to summarise 15% of the methods.
that are called while verifying the program.

We also note how often we are able to apply a summary, that is how often we call a method in a context such that we have a summary that has been recorded in a matching context. Figures 6.6b shows the percentage of calls to summarised methods where a summary is applicable. The ratio varies between 0% in clean, log4j1, and loseNotify, this is because a very small number of methods are recorded, and they are called very infrequently, and 87% in log4j3. On average we are able to apply summaries 29% of the times we call summarised methods.
6.2.2 Interrupted Recordings

There are several reasons why recordings might fail. To recap from Chapter 4, Section 3, these are the reasons why summary creation is interrupted:

1. **Blacklisted**: If the method has been previously blacklisted.

2. **Array type**: If the method returns an array, of any type, the summaries cannot capture that method.

3. **Initialisation**: Methods that create new objects, or classes, have to manipulate the heap of JPF. If either of `<init>` or `<clinit>` is executed, recording is aborted.

4. **Native method**: If the method calls a non-whitelisted native method, recording is stopped. Whitelisted native methods are manually added methods that are known to not have side-effects.

5. **Access to shared or static fields**
   - **Shared fields**: If a method reads from or writes to a shared field, recording is aborted.
   - **Static fields**: If a method reads from or writes to a static field we also interrupt recording.

6. **Transition break**: If a transition break occurs during recording, recording must be aborted.

7. **Exception**: If a method throws an exception we interrupt recording. This does not show up in Figure 6.7, as recording was always interrupted by `<init>` before exceptions were thrown, in all experiments.

While the causes vary between each experiment, as seen in Figure 6.7, we see that the primary cause is methods calling `init`, the bytecode instruction corresponding to calling `new` in Java, that is the methods are creating new objects. The two other most common reasons are native methods that we cannot safely summarise, and that the execution of a method is interrupted by a transition break, which also can have effects that the summaries cannot capture.
(a) Percentage of methods that are summarised.

(b) Percentage of calls to recorded methods in matching contexts.
Figure 6.7: Ratios of interruptions for each experiment.
Chapter 7
Discussion

In almost all of the experiments we see either an unchanged or a reduced performance in terms of run time. This is because the overhead incurred from creating the summaries and evaluating contexts is larger than the benefits from the replaced method calls.

One small consolation is that the overhead, while larger than the benefits, does not seem to be prohibitively large in most cases. However in certain cases, like in pool3-orig where the largest slowdown was seen, the run time was increased by 40%. This is clearly unacceptable, and implies that the summaries need more optimising if they are to be useful.

The only case where the summaries could be said to provide a clear benefit was when pool3-orig was checked using stateless model checking. In this case the summaries reduced the time it took for JPF to find the bug by 18%. For the other programs that were checked without state matching the relative impact of the summaries were also improved, in the sense that the slow down was smaller than when the same target was checked with state matching. This seems to imply that state matching already removes some cases where procedure summaries can be applied. It is possible that procedure summaries would be more useful in stateless model checkers such as VeriSoft [26].

The fact that most of the failed recordings were due to methods initialising new objects, suggests that enabling summaries to capture these methods could increase the applicability and the impact of the summaries. Some of the interruptions related to native methods could possibly be resolved by white-listing these methods, however care has to be taken so that they do not have any side-effects. The methods
that were interrupted by transitions or locks cannot be safely summarised, given that other threads could have essentially arbitrary effects from our perspective. While these interruptions made up between 11% (deadlock) and 39% (alarmclock) of all interruptions this still leaves a large number of methods that we could potentially summarise.

In centralised applications it seemed that our summaries caused the search to slow down somewhat in chatserver, but had no significant effect in alphabetserver. The state spaces in centralised applications are typically very large, and our summaries do not make verifying centralised applications any more feasible.

There may be some cases where summaries perform well that we were not able to find in our experiments. In these cases the programs contain a large portion of deterministic computation that does not allocate new memory and does not access shared data. An argument could be made that these methods could be replaced with stubs, but this would involve additional developer effort, particularly when the methods need to respond differently to different parameters.

While we make use of a number of benchmark programs that have been used in previous studies, it is important to note that this does not mean that our results can be generalised to all programs. It may be the case that summaries would have greater impact on more complex programs, that may be difficult to run with JPF, because of issues related to native methods.
Chapter 8

Conclusion

This thesis sought to investigate whether procedure summaries could be useful in improving the performance of software model checking. To this end we have created a proof of concept implementation of procedure summaries in the form of an extension to JPF. Looking back to our research questions, it may be the case that procedure summaries could have positive impact on the performance of software model checking, if implemented in a different way. Our results suggest that procedure summaries as designed in this thesis would be most promising when used with stateless model checking. We believe that our results suggest that the most important step to make procedure summaries more viable in software model checking would be to increase summary capabilities to capture methods that create new objects. One possible way of doing that would be to redesign the summaries to re-execute specialised versions of methods, similar to the work of Qiu et. al. [48].
Chapter 9

Future Work

We still feel like there are multiple routes left to explore that may increase the potential of procedure summaries within software model checking.

We would suggest to start by enabling summarising methods that modify the heap by allocating memory. Memory allocation is the primary reason for interrupted recording in almost all of our experiments. One approach that seems promising would be to memoize these parts of the methods, as in [48].

Another approach for making summaries more versatile would be to enable them to capture native methods with side-effects. This would be similar to writing a native peer class, in that the additions to the context and modifications would have to be supplied by an external class that specifies the behaviour of the native method.

Another direction would be to represent the summaries more effectively. In the current solution, even if only a single variable is different, this leads to a whole new summary that is potentially the same size as the previous summary, even if context or modification information could be shared between the two. This could potentially be solved by defining summaries over symbolic values, as in [46].
Bibliography


Appendix A

Java Bytecode

This appendix explains a subset of Java bytecode instructions that are necessary to understand the example program shown in Chapter 4. For more detail, see [37].

<table>
<thead>
<tr>
<th>Name</th>
<th>Other Bytes</th>
<th>Stack [Before→After]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aload_&lt;n&gt;</td>
<td></td>
<td>_ → objectref</td>
<td>load a reference onto the stack from local variable n, where $0 \leq n \leq 3$.</td>
</tr>
<tr>
<td>bipush</td>
<td>1: byte</td>
<td>_ → value</td>
<td>push a byte onto the stack as an integer value.</td>
</tr>
<tr>
<td>icont_&lt;i&gt;</td>
<td></td>
<td>_ → 0</td>
<td>push the int value i onto the stack, where $0 \leq i \leq 5$.</td>
</tr>
<tr>
<td>if_icmpne</td>
<td>2: branchbyte1, branchbyte2</td>
<td>value1, value2 → _</td>
<td>if value1 and value2 are not equal, branch to instruction at branchoffset (given by branchbyte1 × 8 + branchbyte2).</td>
</tr>
<tr>
<td>if_icmpge</td>
<td>2: branchbyte1, branchbyte2</td>
<td>value1, value2 → _</td>
<td>if value1 is greater than or equal to value2, branch to instruction at branchoffset (given by branchbyte1 × 8 + branchbyte2).</td>
</tr>
<tr>
<td>iinc</td>
<td>2: index, const</td>
<td>[no change]</td>
<td>increment local variable index by signed byte const</td>
</tr>
<tr>
<td>iload_&lt;n&gt;</td>
<td></td>
<td>_ → value</td>
<td>load an int value from local variable n, where $0 \leq n \leq 3$.</td>
</tr>
<tr>
<td>ireturn</td>
<td></td>
<td>value → [empty]</td>
<td>return an integer, value, from a method.</td>
</tr>
<tr>
<td>istore_&lt;i&gt;</td>
<td></td>
<td>value → _</td>
<td>set variable n to the integer value on top of the stack, where $0 \leq n \leq 3$.</td>
</tr>
<tr>
<td>isub</td>
<td></td>
<td>value1, value2 → result</td>
<td>pops two integer values from the stack, and puts $value1 - value2$ on top of the stack.</td>
</tr>
<tr>
<td>getfield</td>
<td>2: indexbyte1, objectref → value</td>
<td>indexbyte2</td>
<td>get a field value of an object objectref, the field is identified in the constant pool of the current class, at offset given by (indexbyte1 × 8 + indexbyte2).</td>
</tr>
<tr>
<td>goto</td>
<td>2: branchbyte1, [no change]</td>
<td>branchbyte2</td>
<td>sets Program Counter to the offset given by branchbyte1 × 8 + branchbyte2.</td>
</tr>
<tr>
<td>putfield</td>
<td>2: indexbyte1, objectref, value → _</td>
<td>indexbyte2</td>
<td>set field to value in an object objectref, the field is identified in the constant pool of the current class (indexbyte1 × 8 + indexbyte2).</td>
</tr>
</tbody>
</table>