MODEL CHECKING OF DISTRIBUTED MULTITHREADED JAVA APPLICATIONS

NASTARAN SHAFIEI

A DISSERTATION SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
YORK UNIVERSITY
TORONTO, ONTARIO
MARCH 2014

Abstract

In this dissertation, we focus on the verification of distributed Java applications composed of communicating multithreaded processes. We choose model checking as the verification technique. We propose an instance of the so-called centralization approach which allows for model checking multiple communicating processes. The main challenge of applying centralization is keeping data separated between different processes. In our approach, this issue is addressed through a new class-loading model. As one of our contributions, we implement our approach within an existing model checker, Java PathFinder (JPF). To account for interactions between processes, our approach provides the model checker with a model of interprocess communication. Moreover, our model allows for systematically exploring potential exceptional control flows caused by network failures. We also apply a partial order reduction (POR) algorithm to reduce the state space of distributed applications, and we prove that our POR algorithm preserves deadlocks. Furthermore, we propose an automatic approach to capture interactions between the system being

verified and external resources, such as cloud computing services. The dissertation also discusses how our approach is superior to existing approaches. Our approach exhibits better performance which is mainly due to the POR technique. Furthermore, our approach allows for verifying a considerably larger class of applications without the need for any manual modeling, and it has been successfully used to detect bugs that cannot be found using previous work.

To my father, Mansour Shafiei, whose passion for knowledge, and love for his family are boundless.

Acknowledgements

I owe every bit of gratitude to my mother, Forough Azad, and my father, Mansour Shafiei, for their love and encouragement. This journey would not have been possible without their support. A very special thanks goes out to my grandmother, Azam Moein, who has been an inspiration throughout my life. I would also like to thank my sister, Niloufar, who has always been there for me, inspired me, and supported me with her unconditional love.

I would like to express my deepest appreciation for and my thanks to my advisor, Franck van Breugel, for his continuous support of my doctoral research; for his patience, motivation, enthusiasm, and immense knowledge. His guidance was of enormous help during the research for and the composition of this dissertation.

I would also like to express my special thanks to Peter Mehlitz for his tremendous support, continuous guidance, inspiring ideas, and for his invaluable friendship–he has been a true friend and a mentor.

I would like to thank Eric Ruppert for his valuable feedback on this research

project throughout my doctoral studies, and for teaching one of the most inspiring classes in which I have ever participated–his class of Theory of Distributed Computing, winter 2008, York University.

I wish to thank Dimitra Giannakopoulou whose feedback on the thesis has added much value and significant depth to this work. I would like to express my appreciation for Jonathon S. Ostroff for his much valued and thoughtful feedback. I would also like to thank Marin Litoiu and Suprakash Datta for their much appreciated time and their constructive feedback.

I wish to thank Cyrille Artho for our fruitful collaborations and his generous support. I would also like to thank Eric Mercer for enlightening discussions on the partial order reduction topic and his feedback on our reduction technique.

I also wish to thank Guillaume Brat for his advice and counsel, keeping me motivated and focused on this research, and David Hall and Joseph C. Coughlan for their support.

I would also like to thank my friends and colleagues who attended my defense rehearsal talk at NASA Ames Research Center and provided me with constructive feedback.

Table of Contents

List of Tables

List of Figures

1 Introduction

When it comes to concurrency, two levels of execution can be identified: *processes* and *threads*. Each process provides a self-contained execution environment, whereas, threads run within a process and share the process runtime resources such as memory space. In this dissertation, we focus on verifying a category of systems composed of multiple processes. We call these systems distributed applications.

We specifically focus on an automatic verification technique called model checking. In the last few decades, there has been an extensive amount of work towards model checking of single process multithreaded applications. Most of the existing model checkers can be applied only on single processes. Due to the concurrent nature of distributed systems, some existing techniques proposed for single process multithreaded applications can be used for distributed systems. In this work, we focus on model checking distributed applications composed of multiple multithreaded processes. We specifically target those distributed applications that are written in Java and use unbounded blocking buffers for interprocess communication.

There are four main research questions that we aim to answer through this dissertation. The first question is how we can capture multiple processes within a Java model checker. The model checking approach used in our work is based on checking all possible executions of the distributed system under test (SUT) using a scheduler. Therefore, the first question can be reduced to how a virtual machine that handles single process applications can be extended to execute distributed applications, where the execution semantics of individual processes is preserved. Another question is how communication channels, that exist at the operating system level, can be modeled within a Java model checker. Moreover, in this dissertation, we look into a selective search strategy to explore a part of the state space of distributed systems while still preserving properties of interest. Finally, the last question is how we can provide a mechanism that allows applications being model checked to use external resources which exist outside of the model checker environment.

This chapter discusses the factors that motivated us, our main objectives, and the contributions achieved in the dissertation. As one of the contributions, we implement our technique within the JPF model checker. We extend the scope of JPF, which can verify single process applications, to distributed applications. Chapter 2 describes model checking, compares it with other verification techniques, and discusses the features of the JPF model checker. After giving an overview of the current state of the art in Chapter 3, we discuss our approach (Chapter 4, 5,

and 7) and present results obtained from applying our work in Chapter 8.

1.1 Motivation

The direction of this research is motivated by several observations which we elaborate upon in this section. One such observation is an essential need for techniques that can help to detect errors in software systems. This section also discusses the importance of distributed systems and their specific verification requirements. Moreover, it explains why in our research we target applications written in Java by discussing the relevancy of Java as a platform for distributed applications.

1.1.1 Software Verification

In the last few decades, the involvement of software systems in human life has increased significantly. Nowadays, many aspects of our lives are affected by these systems and we use them on a daily basis. For example, every day we are confronted with software-driven devices like telephones, televisions, automobiles, elevators, automated teller machines, microwave ovens, or services like online banking and online shopping.

Even though the involvement of software systems is spreading through most aspects of our lives, these systems are not reliable. The major contributing factor to this problem is the high complexity of software. Unlike hardware, software has

a tendency to grow in size very fast, which can make software more vulnerable to errors.

One of the most notorious software errors of all times is the error of Therac-25, which is a computerized radiation therapy machine. Due to this error, known as a race condition, between June 1985 and January 1987, at least six patients received massive radiation overdoses which resulted in deaths and serious injuries. Another notorious software error is the one that caused the explosion of the Ariane 5 rocket. According to the report by the Ariane inquiry board, on June 4 1996, only 40 seconds after the launch, the rocket exploded.[1] They reported that the failure was due to a software exception. The exception occurred in a data conversion from a 64-bit floating point value to a 16-bit signed integer value in the Ada code. Since the converted result was too large to fit in a 16-bit signed integer, the data conversion instructions led to the exception that was not handled. There were some other data conversions in the code that were handled. The development of Ariane 5 cost the European Space Agency about $7 billion. The Northeast Blackout of 2003 which was a massive loss of electric power in parts of the northeastern United States and Ontario, Canada, also resulted from a software error known as a race condition. The cost of this outage is estimated to have been between $7 and $10 billion.[2]

[1] http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf

[2] https://reports.energy.gov/

As our lives are becoming more involved with software systems, their correctness is becoming a more serious issue. However, detecting and fixing a software error can be very hard and time consuming. According to an article[3] by the U.S. Defense Department and the Software Engineering Institute at Carnegie Mellon University, there are typically about 5 to 15 errors in every 1,000 lines of code. According to a five-year study by the Pentagon, it takes about 75 minutes to trace an error, and two to nine hours to fix it. It therefore takes about 150 hours to verify 1,000 lines, costing roughly $30,000.

As can be seen in Figure 1.1, the earlier the error is detected, the better [1]. The cost of detecting and repairing a software error during maintenance and operation is considerably higher compared to the early stages of development. In later stages, the malfunctioning of the system can severely damage the reputation of the company and can even be a threat to its survival. According to a study by the National Institute of Standards and Technology (NIST) in 2002, software errors cost the U.S. economy about $59.5 billion annually, which is about 0.6 percent of the gross domestic product.[4]

As pointed out earlier, software errors cannot be avoided and detecting them is a very difficult and tedious process. But, since errors in software can have extremely

[3] http://www.businessweek.com/1999/99_49/b3658015.htm

[4] http://www.nist.gov/director/planning/upload/report02-3.pdf

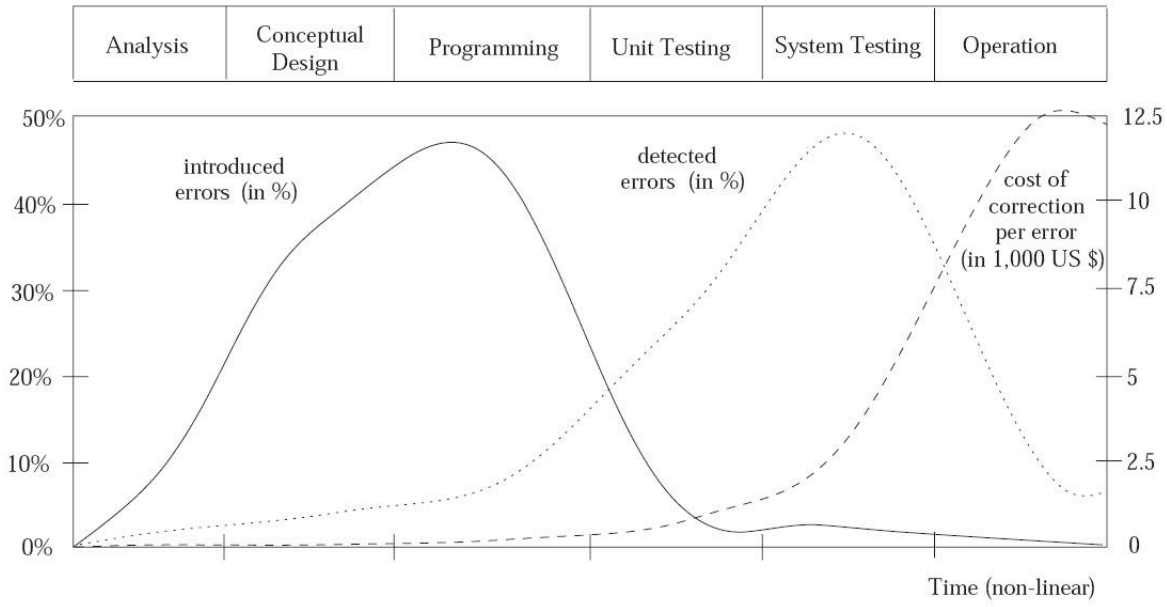

Figure 1.1: Software life cycle and error introduction, detection, and repair costs [2]

undesirable consequences, there is an essential need for techniques that can help to detect errors in software systems. These techniques are invaluable to developers and can cut the cost of software development significantly, especially if the error is detected in the early stages of software development.

1.1.2 Concurrency

It is becoming increasingly difficult to follow the traditional path to increase the performance of processors, by switching transistors at ever greater speeds. This is mainly the case because the amount of power used and the amount of cooling technology needed increase with the speed at which transistors switch [3]. Nowa-

days, there are many sequential systems that are being replaced with concurrent versions. That is done by putting multiple processors, or cores, on a single chip.

Compared to sequential programming, concurrent programming is more complex. Detecting errors in such systems is also more difficult. The way that concurrent systems behave depends on the relative speed of the executions of different components (including processes and threads) in the system, which cannot be predicted. Hence, the behavior of a concurrent system is non-deterministic. It is very hard for programmers to consider all possible interleavings of concurrently running components.

1.1.3 Distributed Applications

Distributed computing is becoming more and more important these days as most systems in use are distributed. For example, mobile applications, the popularity of which keeps rising, are mostly distributed [4]. Some examples of widely used, Java-based, mobile applications are Google maps mobile and Gmail mobile.

There are several key factors driving the development of distributed applications [5, 6]. Some services intrinsically require the use of a communication network to connect different components. Massively multiplayer online games are among such services which allow large numbers of people to play simultaneously, e.g.,

RuneScape[5], which is written in Java.

Distributed computing can also allow for developing fault tolerant applications where a failure in a process does not stop other processes from running, and the application can still complete its task. The Netflix API is an example of a system that uses distributed components to provide fault tolerance[6].

Moreover, distributed systems provide the use of the computational power of multiple machines to process tasks faster and handle larger problems. For example, Memcached[7] is a high-performance distributed memory caching system designed to speed up dynamic web applications. The Netflix EVCache[8] open-source project employs Memcached. Some other users of the Memcached caching system are Facebook, Twitter, Wikipedia, and YouTube. Distributed computing is also used in intensive scientific simulations to gain speed, e.g., CartaBlanca [9] is a physical system simulation package written in Java which uses MPJ Express[10] (a Java message passing library) to parallelize its computation.

Finally, using distributed applications allows for sharing resources in a networked system such as disks, printers, files, and databases. This can be seen in

[5] http://www.runescape.com

[6] http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html

[7] http://memcached.org

[8] https://github.com/Netflix/EVCache

[9] http://www.lanl.gov/projects/CartaBlanca

[10] http://mpj-express.org

systems based on cloud computing [7], which are distributed systems following the client-server model wherein one or more clients request information from a server. Cloud computing is one of the major focuses of leading companies in the computer industry, such as Apple, Amazon, IBM, and Google. The majority of Google services follow the cloud computing model. Some of those services, such as Google Docs, Google Calendar, and Gmail, are based on Java.

In general, distributed applications are hard to develop. These applications are inherently concurrent. Other than concurrency errors, developers of such applications need to deal with issues related to a distributed setting, such as the possibility of failures at different levels, for example, within the process initiating the communication, while the operating system (OS) writes data to the network, during the time that data is transmitted between processes, while the OS receives data and hands it over to the recipient process, and finally within the process receiving data. Another issue in programming distributed applications is gaining a consistent view of data across the system.

In general, testing distributed systems is hard. Different components of the system may have different software and hardware requirements, and therefore setting up an environment to test such applications can be difficult. Furthermore, due to the possibility of failures at different levels, testing such applications against potential defects requires injection or simulation of failures at several layers.

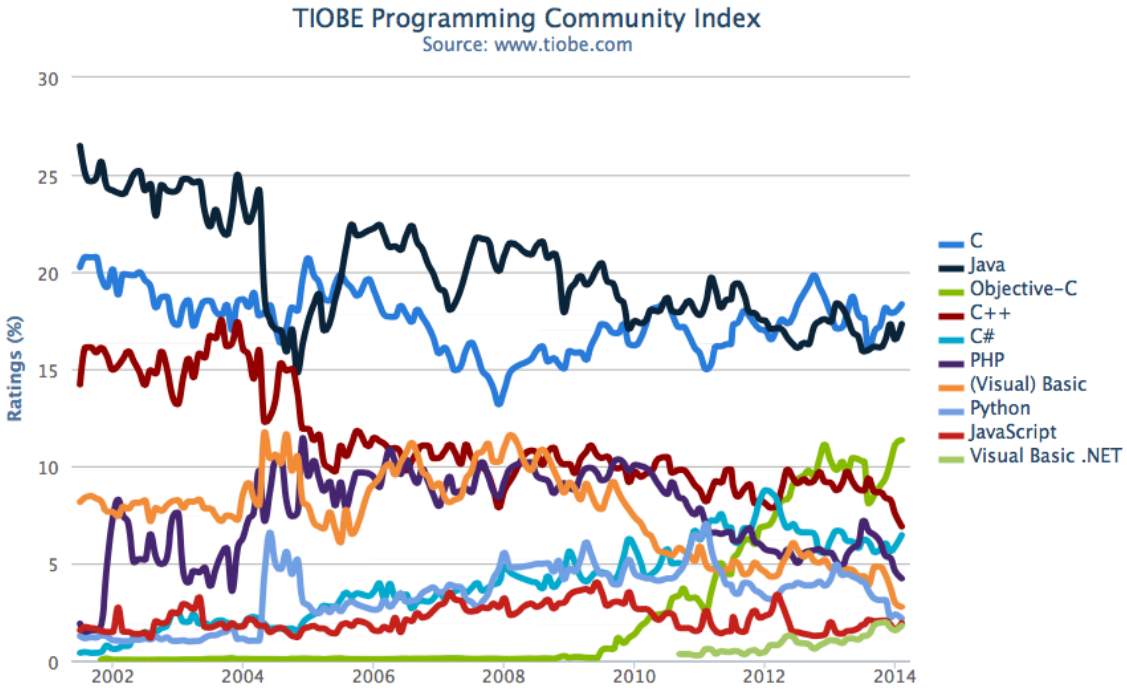

Figure 1.2: TIOBE Programming Community Index for February 2014

1.1.4 The Java Platform

Java is one of the most popular programming languages. That can be seen from the graph of Figure 1.2. This graph shows the TIOBE Programming Community Index in February 2014[11] which captures the popularity of programming languages. Java is considered a language of choice for many developers of distributed applications, e.g., the majority of the most watched Java projects on GitHub, which is a popular web-based hosting service for software systems, are distributed applications.

Java has several features that make it a powerful environment for developing

[11] http://www.tiobe.com/index.php/content/paperinfo/tpci/

such applications [5, 6]. Java is platform independent, that is, a single version of Java code can run on any platform with a Java virtual machine (JVM). It supports multithreaded programming and offers an exception handling mechanism which is useful for developing fault tolerant applications. It also provides multilevel support for network communication including basic networking support such as sockets used to establish a connection between processes, and data communication protocols such as TCP and UDP. At a higher level, it provides networking capabilities such as distributed objects, and communication with databases.

Finally, Java supports two aspects of security for distributed applications. Since in a distributed Java application, running processes (such as Java applets) can migrate across the network, Java provides ways to secure the runtime environment of recipient processes, for example, by restricting access to the local file system. It also allows for adding user authentication, and encryption of data sent across the network to establish secure network connections.

1.2 Objectives

Consider the distributed system presented in Figure 1.3. Processes are shown using cloud shapes. Threads running within a process are shown using rectangles surrounded by the cloud. Processes are running on different machines. They do not share memory and use distinct execution environments.

The client processes use the server process to translate phrases from English to some other language. Once they connect to the server, they notify the server of their language of choice. For each client the server creates a worker thread which handles all translation requests from the client in the requested language. For example, *Client 1* requests translations into French which are handled by the thread *worker 1*. To perform a translation, *worker 1* forwards the request from the client, and the language of choice to the Google translator which is a cloud computing service. Once *worker 1* receives the translation result from the Google translator, it sends it to the client. By creating a translator thread per client, the server can serve multiple clients simultaneously. In this example, the client processes also include more than one thread for processing their internal tasks. Google charges for each translation request ($20 per million characters of text) and there is also a daily usage limit. To avoid further charges, for each language, the server process uses a map to store translation results obtained from the Google translator. It only sends a request to the Google translator if the translation result is not in the map.

Our ultimate goal is to verify such a distributed system using the model checking technique. We are interested in verifying all possible states of the server and the two clients while capturing interactions with the Google translator. The server and clients are subjected to thread non-determinism, so, they can have more than one

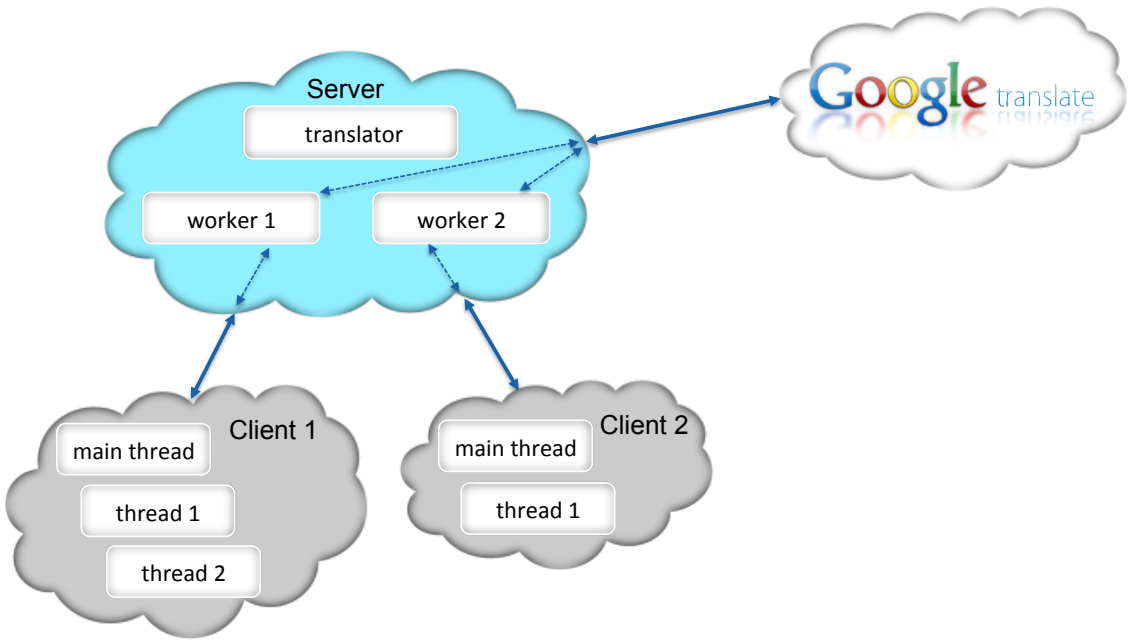

Figure 1.3: An example of a distributed multithreaded system

possible execution. Since these processes interact, non-determinism in one process can affect the state of the entire system. Besides, another source of non-determinism is the possibility of network failures. Therefore, to explore all possible executions of such a system, it is essential to feed the model checker with multiple processes and provide a communication model. Moreover, a mechanism is needed to allow interaction with external resources, e.g. the Google translator, during the model checking.

In this work, we propose a technique which allows for model checking multiple multithreaded processes that communicate with each other. Our technique is appli-

cable to those model checkers that drive the execution of the system using a runtime scheduler. In this work, we apply our approach to JPF which is a model checker for Java applications. However, the approach can be used for distributed systems written in some other programming languages which are dynamically typed, such as Android, Scala, and Smalltalk. Our goal can be broken down into the following main objectives.

- Applying a new instance of the so-called centralization approach on the model checker to support multiple processes.
- Modeling communication between processes running within the model checker.
- Applying a POR reduction technique to reduce the state space of the distributed SUT.
- Capturing communication with external resources.

1.3 Contributions

In this work, we focus on model checking of distributed multithreaded Java applications. One of the novelties of our work is a technique proposed to keep data separated between processes running within the model checker. This is one of the main challenges when capturing multiple processes within a model checker. We

address this by implementing a new class-loading model within the model checker. That also requires extending the type system of the model checker. Our approach to separate data between processes addresses several limitations of previous work to verify distributed applications.

This work also provides an accurate model of communication between Java processes running within the model checker. Our work can be used to detect errors which could not be detected by previous work. We also propose a new mechanism for injecting failures into the distributed system analyzed by the model checker. This enables systematic exploration of possible exceptional control flows which are due to network failures. Such a mechanism is essential since executions caused by network failures cannot be captured by exploring all the executions of the process code.

Furthermore, we apply a POR technique in this context and prove its correctness with respect to deadlocks. This technique is based on identifying those operations which are visible within only one process and do not impact the state of other processes (e.g., accessing a shared field) versus those operations which are globally visible (e.g., accessing communication objects).

Moreover, in this work, we propose a novel technique to connect the processes running within the model checker with external resources. The approach is automatic and generic as it is not specific to certain communication means. This

approach can be also used to handle native methods automatically, which can save a lot of modeling effort.

In this dissertation, we also propose a way to separate the execution environment for different processes, which requires implementing a new Java virtual machine. In addition, we propose a new model for Java finalizers in the model checker. Such a model is essential to capture all possible behaviors of distributed systems. We also provide a visualization of the search graph explored by the model checker when verifying a distributed application. Finally, comparing our work with existing work confirms that, overall, our approach has significantly better performance and scales better, which is due to our POR technique and our implementation choices.

2 Background

As we have seen in the previous chapter, it is very useful to develop techniques and tools that can be used to detect errors in software systems. In the last few decades, there has been a considerable amount of work on developing techniques and tools to verify code. In this chapter, we give an overview of different verification techniques and compare them with model checking. We also provide a general survey of existing tools used to model check Java applications. Finally, this chapter justifies the suitability of JPF to model check distributed applications, and it provides a broad overview of the JPF infrastructure and its features.

2.1 Software Verification Techniques

The goal of software verification techniques is to check that software behaves in a way that it is supposed to. The behavior of the software is outlined in the system's specification, which is a document that includes all the properties that the software should satisfy. Basically, the software is said to be correct if it is performing in a

way such that all of the required properties are satisfied.

It should be noted that code verification is undecidable. According to Rice's theorem, every non-trivial property of the language of Turing machines is undecidable. That implies that, given code and a non-trivial property, there may not exist an algorithm that decides whether the code satisfies the property [8]. When errors are detected and fixed, software systems can only get more reliable. Verification techniques are, in general, not able to prove the correctness of code. However, applying them can still have a significant impact. The main software verification techniques include testing, theorem proving, runtime verification, abstract interpretation, type systems, and model checking. In the following sections, we give a brief overview of each of these techniques.

2.1.1 Testing

Testing is one of the methods used to verify software systems [9]. The testing process includes providing the compiled code of the system under consideration with inputs and observing the outputs to find errors. The software should produce the outputs as expected. In other words, the outputs should be compatible with the system's specification. Software testing is a widely used method and usually about 30% to 50% of software engineering projects costs go to the testing process [2, page 4].

A main advantage of testing is that it can be used to verify all kinds of software. However, using testing, it is difficult to capture all potential executions of the code. It can be very hard to test the code against all possible input sequences. Since testers usually do not have control over the scheduling of the concurrently running components, it is very hard to use testing to detect concurrency errors. Even if a bug is detected, because the execution of the code is non-deterministic, the bug may be very difficult to reproduce. According to Edsger W. Dijkstra "Program testing can be used to show the presence of errors, but never to show their absence!" [10].

2.1.2 Theorem Proving

Another technique to verify software systems is theorem proving. This method uses axioms and inference rules to prove properties of the code. One of the main advantages of this technique is that it can be applied to code with infinite state spaces. Theorem proving may involve a high level of mathematical complexity and can only be used by an expert in the formulation of formal arguments and proof techniques.

Until the 1960s, this technique was completely performed by humans. Today, some interactive software tools, called theorem provers, have been developed which can be used to develop formal proofs. Some examples of such provers can be found in [11, 12]. These tools ensure the correct applications of axioms and proof

rules. At each stage of the proof, they can also suggest possible ways to make progress. It should be noted that there exist some theorem provers which are fully automatic [13, 14]. Compared with these fully automated theorem provers, in general the underlying logic of interactive theorem provers is more expressive. In general, theorem proving is a very time-consuming process and no bound can be set on the time and memory that is needed to verify code. In practice, that limits the use of this technique.

2.1.3 Runtime Verification

Runtime verification is also a software verification technique. Runtime verification tools analyze the behavior of the code at runtime. They go through a single execution of the code, and as it executes, they check if it is running correctly with respect to a system's specification. Some examples of tools performing runtime verification are Monitoring and Checking (MaC) [15] and Java PathExplorer (JPAX) [16].

In the MaC framework, during the execution of the code under consideration, information about the execution is extracted and it is checked against the system's specification. A prototype implementation of the MaC architecture has been developed for Java code, called Java-MaC [17]. The JPAX tool is very close to the MaC architecture. It extracts information about an execution. It checks the execution against the system's specification and it also applies error detection algorithms to

detect errors such as data races and deadlocks.

In general, this method scales very well, i.e., the time and the memory to verify an execution is a constant factor larger than the time and the memory to execute the code. Moreover, runtime verification can be completely automated. However, similar to testing, using the runtime verification technique, it is nearly impossible to capture all possible execution paths.

2.1.4 Abstract Interpretation

Another technique that can be used to verify software systems is abstract interpretation. The semantics of code can be defined as a mathematical formalization of all possible behaviors of the code. Abstract interpretation requires the precise definition of a concrete semantics which represents the actual executions of the code. This technique approximates the concrete semantics of the code to obtain an abstract semantics [18, 19]. The abstraction function is defined to assign each value in the concrete semantic domain to a value in the abstract semantic domain.

The goal of abstract interpretation is to obtain an abstract semantics as an approximation that gives reliable answers to questions about properties of the code, i.e., answers which are neither false positives nor false negatives. Otherwise, the analysis is not reliable. For example, consider an analyzer that uses approximations to check whether an index of an array is used out of its bounds. The analyzer

computes an approximation of the set of all the indices used to access the array. One possible case is that the analyzer approximates this set by computing a subset of the indices used to access the array. If the analyzer does not find any errors and indicates that no violation has been found, the answer could be a false positive, because the analyzer has not checked all the indices used in accessing the array. Another possible case is that the analyzer computes a superset of all the indices used to access the array. If the analyzer does not find any errors and gives an answer indicating that no violations have been found, the answer is reliable, because the analyzer has checked all the indices used in accessing the array. However, if the analyzer reports an error, it could be a false positive.

The tools based on abstract interpretation are considered as static analyzers since they determine runtime properties of the code statically, without executing it. There are many automated tools based on abstract interpretation that are used to verify code. Cibai [20] is an example of an abstract interpretation-based tool for the verification of code written in Java. The errors that can be detected by Cibai are division by zero, array indices out of bounds, and null dereferences. Cibai can also check for user-defined assertions.

Static analyzers implementing abstract interpretation are designed based on the properties of interest. They check the code against a set of properties that are hard coded into the tool and are not application specific [21]. Another limitation of these

techniques is that they do not produce counterexamples.

2.1.5 Type Checking

Another method to verify software systems is type checking. This approach is based on formal type systems. Basically, using the type system, properties can be expressed as types and the code verification is reduced to type checking. A type system consists of a set of types and a set of rules that associate types with (parts of) code. Type checking amounts to applying the rules with the objective of showing that the code can be typed. A type system checking a particular property is designed in such a way that code satisfies the property if it can be typed.

One example of a tool including a formal type system is the race detector discussed in [22]. This tool uses a type system to perform a static race detection analysis for code written in Java. The type system guarantees the absence of races in well-typed code. This race detector relies on code annotations. It is based on the lock-based synchronization discipline, i.e., each field is protected by a lock which is held while accessing the field. The race detector keeps track of the locks for each shared field and makes sure that the locks are held during every access of the field.

Another race detector based on a formal type system is presented in [23]. This race detector also relies on user annotations. The type system of this race detector is more expressive than the one in [22]. This type system allows the user to assign

different protection mechanisms to different objects of the same class. This is useful since it is not always necessary to protect an object by a lock, e.g., when the object is immutable no locking is needed. However, in [22], the protection mechanism has to be specified at declaration time and it is applied to all instances of the class. The type system of [23] is also extended to detect deadlocks in Java code as explained in [24]. Well-typed code in this type system is free of races and deadlocks.

The design of formal type systems is specific to certain properties. This implies that to check any other properties, a new formal type system has to be designed. But type checking is considered as an efficient technique for software verification, in terms of both time and memory consumption.

2.1.6 Model Checking

Finally, model checking is an automated verification technique that examines all possible system states in a systematic way to check if desired properties are satisfied. Figure 2.1 shows the three possible outcomes from model checking an application. One outcome is that the model checker fully explores the state space of the program without detecting errors. Another possible outcome is that a property is not satisfied. In this case the model checker provides a counterexample (i.e., an execution path that leads to the erroneous state) that can be used to help correct the error. The last possible outcome is that model checking leads to the *state space*

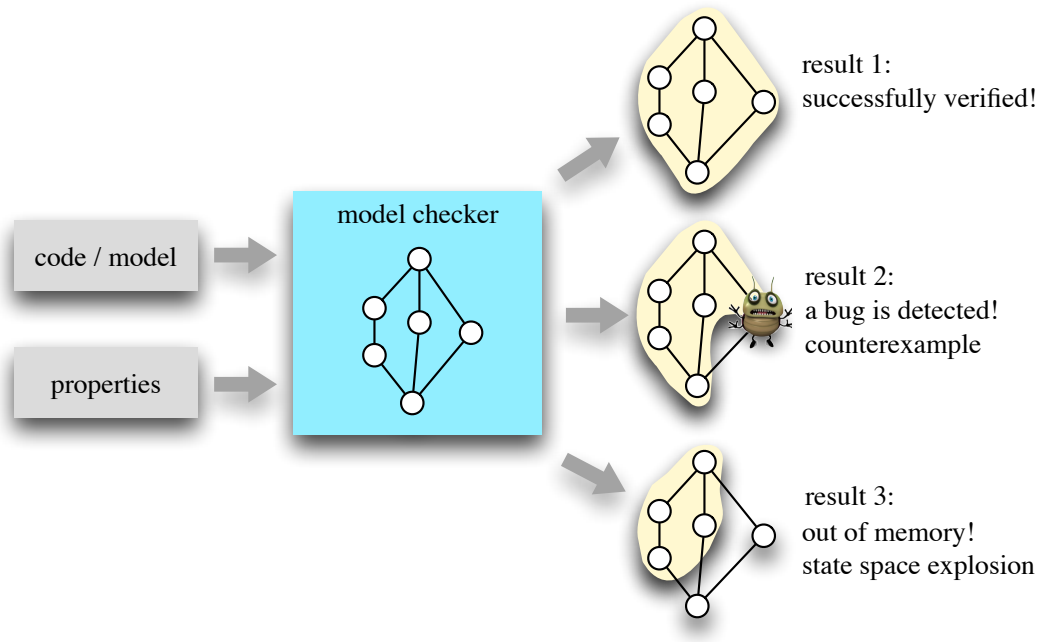

Figure 2.1: Applying model checking can lead to three different outcomes

explosion problem. In this case the model checking process cannot be completed and terminates by running out of memory. This is considered as one of the most challenging issues in model checking.

The state space explosion problem describes the exponential relation of the number of states of the system to the number of components that constitute the state [25]. These components include data variables, processes, and threads. Different values of data variables represent different states of the system. Moreover, in multiprocess applications and multithreaded applications, concurrent actions can be executed in any arbitrary order. Considering all possible interleavings of these

concurrent actions may lead to a very large state space. It can be shown that the number of states can increase exponentially with the number of interleaving components [2].

Although model checking is subject to the state space explosion problem, considering its advantages, in many cases, it can be preferable to the other methods. Model checking is naturally preferred to testing and runtime verification when concurrency comes into play since, unlike model checking, these techniques have no control over the scheduling of the concurrently running components, and therefore are not able to check all possible execution paths of the application. Moreover, counterexamples provided by model checkers make the process of fixing errors much easier. Since model checking is mostly automated, it is generally easier to use than theorem proving. Finally, model checkers allow for the specification of properties related to the functionality of the application, hence allowing for verifying a wide range of requirements, unlike the type checking technique and static analyzers based on abstract interpretation which are implemented specifically for certain properties.

It should be noted that the above comparison between model checking and other techniques is based on their basic definitions. In practice, these techniques can be closely related. For example, in some cases, model checking can be viewed as theorem proving [26]. Moreover, software verification tools may not rely on a single technique. For example, the systematic testers such as ExitBlock [27] and

CHESS [28] are designed to test concurrent code. The technique implemented by these tools is very close to model checking. They have control over the scheduling of the concurrently running components. Given a test scenario, these tools are able to execute the code repeatedly so that each execution has a distinct scheduling.

In the last two decades, the interest in using model checking has been considerably increased in industry. Many leading companies have started research on model checking and have developed their own model checkers, e.g., SLAM and ZING developed at Microsoft [29, 30], Spin developed at Bell Labs [31], and Rule-Base developed at IBM [32]. There are also many examples where model checking has been successfully applied in industry [33] and it has detected previously unknown errors [34]. In the next section, we provide a detailed description of the model checking technique.

2.2 Overview of Model Checking

Based on the way that states are represented, model checking algorithms can be classified into two main categories: explicit-state model checking versus symbolic model checking. In explicit-state model checking, graph algorithms are used to create and explore the state space. While exploring the state space of the system, the states are checked against the desired properties. To avoid regenerating states, the model checking algorithm keeps a record of visited states which are usually

stored in a hash table. That also allows the algorithm to backtrack to states which encapsulate non-deterministic choices in order to explore new paths. Spin [31] and JPF [35] are examples of explicit-state model checkers.

Symbolic model checking deals with sets of states instead of directly dealing with individual states. In symbolic model checking, states are usually represented using binary decision diagrams (BDDs) [36]. A BDD is a data structure for boolean expressions. It is a rooted, directed, acyclic graph that can be retrieved from a decision tree by identifying nodes. Due to BDD's features, there are efficient algorithms that can perform logical operations on them. One issue with using BDDs is choosing a suitable ordering of variables, which is tedious and affects the size of the BDD. Another method for symbolic representation of states is propositional logic formulas [37]. Microsoft's SLAM project [29] and SMV [38] are two examples of symbolic model checkers.

Symbolic model checking is mostly applied on hardware rather than software, because symbolic model checking deals well with static transition relations but is less suitable for systems including dynamic creation of threads and objects. Compared to explicit-state model checking, it can verify larger systems. Unlike explicit-state model checking, symbolic model checking can handle systems with infinite state spaces. Symbolic methods are more suitable for systems subjected to data non-determinism. Explicit-state model checking works better in finding concur-

rency errors [39]. It should be noted that the remainder of this thesis is restricted to explicit-state model checking. From here on, we use model checking to refer to explicit-state model checking.

Another criterion used to categorize model checkers is based on their input, which could be either a model of the system or the system itself. The first group includes model checkers that are applied on models that describe the possible behavior of the systems. The first step to use these model checkers is to create a model of the system using a modeling language understandable by the model checker. However, the verification results are just as good as the model of the system [2, Chapter 1]. These types of model checkers are more suited to verify the design of the system than its implementation. Due to the loss of precision, the model of the system might not reflect the exact behavior of the actual system. It should be noted, when applying model-based development, the gap between the model and its implementation becomes small. In this method, an executable model of the system is generated early in the lifecycle, and it is used to automatically generate code. Another drawback of using these model checkers is that the input languages for these model checkers are modeling languages which are usually too simple to capture all the features of the system. Spin [31], SMV [38], and SAL [40] are examples of model checkers applied on a model of the system.

The second group of model checkers are directly applied to the actual system.

Therefore, using them does not require creating a model of the system. That makes this group of model checkers much easier to use. These model checkers are able to verify both the design and the implementation of the system. However, in general, systems include more details compared to models of the systems. Therefore, directly model checking a system rather than a model of the system leads to a larger state space which intensifies the state space explosion problem. Some examples from this group are JPF [35] for Java code, VeriSoft [41] for C code, and CMC [42] for code in C and C++.

2.2.1 The Model Checking Process

The model checking process can be divided into four major steps explained below. Every model checker goes through some or all of these steps.

Modeling - modeling is the very first step of the model checking process. This step is performed by model checkers that are applied to a system model. In this step, the model of the system is generated from the system's specification using a model description language understandable by the model checker. The system is usually modeled using a finite-state automaton. For example, consider Java code. States of the Java code may contain the values of variables in the system and snapshots of the stacks and the heap. Transitions take the system from one state to another. In the Java code, transitions may be represented by bytecode instructions. The model

of the system should be validated to make sure that it specifies the behavior of the code accurately. Accurate modeling of the code can result in detecting mistakes such as incompleteness and ambiguities in the system's specification [2, Chapter 1].

There exist some model extractors that have been developed to automate the modeling process. They are usually combined with model checkers that accept a model of the system to make the whole model checking process automated. Basically, these tools receive the source code as input and extract a finite model of the system. The finite model is in the input language of an existing model checker. For example, Bandera [43] is applied to Java code and extracts a finite model of the code in three different modeling languages, namely the input languages for Spin, SMV, and SAL.

Properties Specification - in this step, the desired properties of the system under consideration are specified. This step precisely outlines what the code is supposed to do, and what it is not supposed to do. Properties of interest can be generic such as absence of deadlock and absence of race conditions. They can be also specific to the SUT, for example, the value of a counter does not reach a certain value. One usual way to state the properties at this stage is to specify them using temporal logics. Temporal logics are considered to be a useful way to formalize properties of concurrent code since they can be used to describe the behavior of code over time.

Running - in this step the model checker is initialized and, depending on the

type of model checker, it is run on either the model or the actual system. The model checker algorithm considers all possible interleavings of the code for (all) possible inputs. At each state it checks if the specified properties are hold. Usually, when a property does not hold, the model checker stops running and terminates to report the error. When the size of the state space gets too large, due to the state space explosion problem, the model checker terminates by giving an out of memory error.

Analysis - after the model checker terminates, it provides the user with different results. Different scenarios can be reported: the code satisfies the desired properties, the code does not satisfy a property, or the model checker runs out of memory and terminates due to the state space explosion problem. If the results demonstrate that the property is not valid, an error is detected.

The error can have different sources: a modeling error (for the group of model checkers which are applied on the system model), an error in the actual system, or an error in the property specification. Note that this is based on the assumption that the model checker behaves correctly. After all, the model checker is code as well and, hence, most likely contains bugs. However, by applying it over and over again, fewer and fewer bugs will remain in the model checker code.

The model checker typically provides a counterexample that helps the user detect the source of the error. A counterexample is a trace of the code that leads to

the error. For the group of model checkers that are applied on the source code, the trace of the error is directly mapped to the source code. Some model-based model checkers include embedded model extractors that exploit intermediate languages to map the model to the source code, e.g., Bandera [43].

2.2.2 Addressing The State Space Explosion Problem

As mentioned earlier, the state space explosion problem is an inherent limitation of model checking. There exist several approaches to mitigate the state space explosion problem. Symbolic execution treats input variables to a program as unknown quantities or symbols [44]. By symbolically representing values, this technique allows for reasoning about infinite domain. Abstraction which, based on the system specifications, abstracts away details from the system [45]. The abstract systems are usually much smaller than the actual systems, and therefore lead to smaller state spaces. In compositional verification [46, 47], the specification of a system, composed of multiple components running in parallel, is decomposed into properties specifying the behavior of the system components. One example of the compositional verification technique is assume-guarantee reasoning [48]. This technique first checks whether a component of a system guarantees local properties, where the system satisfies an assumption characterizing the behavior of the component with respect to other components. A compositional proof rule is set up to ensure

consistency among the assumptions made for different components. Proof rules are then used to infer system level properties from local verification steps.

Another commonly used approach to deal with the state space explosion problem is POR [49, 50, 51, 41, 52, 53]. In this work, we also apply a POR technique. Since this technique is well-suited for the type of the system we study in this work, we describe it in more detail. The aim of the POR technique is to reduce the number of possible orderings of concurrently executed actions to be analyzed by the model checker. This technique is based on identifying the execution path fragments that lead to the same state of the system regardless of the order of concurrently executed actions. These actions are called independent actions. For example, actions accessing different variables or accessing local variables of different processes, are considered as independent actions. However, two threads writing to the same variable or acquiring the same lock are considered as dependent actions. Independent actions occurring in any order give rise to the same behavior of the system. Consequently, instead of analyzing all possible orderings of this set of actions, only one representative ordering is analyzed.

This approach results in a state space which is a subset of the original state space. The resulting state space should be equivalent to the original state space with respect to the property of interest. The POR reduction technique can be applied either dynamically or statically [2, Chapter 8]. The dynamic POR reduction

is applied on-the-fly, that is, the reduced state space is generated during the model checking [53]. In contrast, in the static POR reduction, the reduced state space is generated before the model checking process starts [52]. POR reduction techniques can have a significant impact on the size of the state space and they are used by almost every model checker.

2.3 Race Detection

A (data) race is one of the most notorious concurrency errors. Roughly speaking, a race on a shared variable arises in concurrent code if two threads access that variable simultaneously and the accesses are conflicting, that is, at least one of them writes to the variable. Although some races are benign, races often represent errors in code. Hence, tools that detect them are invaluable to those writing concurrent code. For an example of a race consider the following Java code.

```
public class Account {
  private int balance;

  public Account() {
    this.balance = 0;
  }

  public void deposit(int amount) {
    this.balance += amount;
  }
}
```

```
public class Deposit extends Thread {
  private Account account;
  private int amount;

  public Deposit(Account account, int amount) {
    this.account = account;
    this.amount = amount;
  }

  public void run() {
    this.account.deposit(this.amount);
  }
}
```

```
public class Bank {
  public static void main(String[] args) {
    Account account = new Account();
    new Deposit(account, 100).start();
    new Deposit(account, 200).start();
  }
}
```

This code consists of the three classes `Account`, `Deposit`, and `Bank`. These classes represent a bank with one account, with an initial balance of zero. In the `main` method of the `Bank` class a new account is created and two threads deposit money into the account. By executing `new Deposit(account, 100).start()`, a new `Deposit` object is created, which is a thread, and its `run` method is invokdeposited. The execution of the `run` method should amount to deposit `100` into the `account`. Similarly, by executing `new Deposit(account, 200).start()`, another thread deposits `200` into the `account`.

Now consider the scenario where one of the threads first reads the value of the variable `balance`[12] and the other thread then reads the same value of the variable `balance`. In this scenario, the two invocations of the `deposit` method "overlap" and the final value of `balance` may be equal to 100 or 200 (depending which thread changes the value of `balance` last), rather than 300. This is caused by a race on the variable `balance`. This example demonstrates a race that makes the code behave in an unexpected way.

Many tools have been developed to detect races. These tools are based on the two types of race detection techniques: dynamic and static. In dynamic race detection, a single execution of concurrent code is checked for races. Dynamic race detection is NP-hard (see [54] and the references therein). The two key approaches to detect races dynamically are based on locksets and the happens-before relation. The former approach has been popularized by the Eraser tool [55]. During the execution, Eraser dynamically keeps track of the set of locks that have protected shared variables so far. Each time a thread accesses a shared variable, Eraser computes the intersection of the set of locks protecting the variable and the locks held by the thread. A race warning is issued if the intersection becomes empty. Another example of a dynamic race detector based on the lockset algorithm is Goldilocks [56].

[12]The Java statement `this.balance += amount` corresponds to a number of bytecode instructions including reading and writing the variable `balance`.

Another type of algorithm is based on the happens-before relation which is a relation between the actions of the code. In order to define this relation, first we define the program-order and the synchronizes with order. Consider multithreaded Java code. The actions are represented by bytecode instructions. The actions within a thread occur in the same order in which they show up in the code. This ordering of actions is called the program-order. Any action that releases a lock on an object synchronizes with all subsequent actions that acquire the lock on the same object. Moreover, writing to a *volatile* variable synchronizes with all subsequent reading of the same variable. A volatile variable in Java is declared using the keyword `volatile`. The value of such a variable is never cached in threads' local memory and is always written to and retrieved from the main memory. The happens-before relation can be defined as the transitive closure of the union of the synchronizes with order and the program-order [57]. For an example of a race detector based on the happens-before relation we refer the reader to [58].

Static race detection algorithms aim to consider all potential executions. Although this approach gives rise to tools that are usually sound (that is, the races that are reported by the tool are real races), the tools are generally not complete (that is, not all races are always reported). Examples of static race detectors can be found in [59, 60, 61, 62]. Several different approaches exist to statically detect races. One of these approaches is model checking. For example, in [63] model

checking is exploited to detect races in code written in an extension of C. The JPF model checker can also be used to detect races in Java code. The JPF race detector is discussed in Section 2.5.4.

2.4 Tools for Model Checking Java Code

In general, existing tools used for model checking Java code are developed based on two broad approaches. The first approach uses static analysis techniques to automatically extract a model out of code in the input language of an existing model-based model checker. Then, the model is model checked. The other approach systematically explores the state space of the actual system. This approach is based on executing the code. It uses a runtime scheduler to drive possible executions of the system. Next, we explain different tools that fall within one of the two above categories.

One of the tools that has been developed to automatically specify the system in the modeling language of an existing model checker is the first generation of Java PathFinder, JPF1 [64]. This tool was developed at NASA in 1999. It receives the Java source code as input and automatically translates it into the Promela language which is the input language of the Spin model checker. The motivation behind this project came from applying the Spin model checker on a multithreaded operating system written in LISP for NASA's Deep Space 1 mission. They manually

translated the LISP code into Promela and applied Spin, which led to the discovery of five previously unknown concurrency errors [34]. Some of the features of Java that cannot be translated using JPF1 include packages, method overloading and overriding, recursive methods, strings, and floating point numbers. Moreover, JPF1 does not map error traces produced by Spin back to their corresponding traces in the Java source code.

As described later in this section, besides JPF1, the Spin model checker serves as a back-end for other tools such as JCAT [65], and Bandera [43]. Spin is one of the most well-known software model checkers, which was developed by Holzmann [66]. It is an explicit-state model checker that can be used to verify models of concurrent code. Its development is started in 1980 at Bell Labs. As mentioned earlier, the input language of Spin is Promela, which is a high-level language to formalize the system model. Compared to other modeling languages, Promela has a rich modeling functionality. Spin relies on the fact that the behavior of asynchronous processes in distributed systems can be modeled by finite state automata. Concurrent code in Promela is described by a parallel composition of process templates which specify the behavior of the processes. Spin generates a finite automaton for each process template. Then it computes an asynchronous interleaving product of these automata to create the state space of the system.

To express behavior that should never happen, Spin provides the never claim.

Spin generates a Büchi automaton from a never claim. It computes the synchronous product of this automaton and the one representing the state space. The system does not violate the property if the language accepted by the resulting automaton is empty. Instead of hand-writing the never claim, the user can simply specify the undesirable behavior by a linear temporal formula which is translated to a never claim. Spin allows for dynamic allocation of processes. However, it does not support dynamic allocation of memory. An extension of the Spin model checker called dSPIN [67] has been developed. It provides support for dynamic memory allocation. However, both Spin and dSPIN impose a limit on the size of the state. Spin was originally developed for verification of communication protocols. Today, Spin is considered a mature tool and it is used for a wide variety of software systems.

JCAT [65] also receives Java source code as input and translates it into the input language of Spin. JCAT can be used to detect deadlocks in multithreaded Java code. Using the Java2Spin translator tool, JCAT creates an abstract formal model from the Java code. Spin verifies the model and if any error is detected, JCAT maps the model trace that leads to the error to a sequence of states in the actual Java code. One of the limitations of the tool is that it does not support polymorphism. In order to reduce the size of the state space, the Java2Spin translator applies static analysis on the Java code. It gets rid of the inherited class members that are not used and the object synchronization structures that are not used by any of the

threads. JCAT also combines some state sequences into a compound state in a way that does not influence the system properties. Further, in JCAT, the user can use annotations to remove variables that do not affect the concurrent behavior of the code.

Another tool that converts Java code into the Promela modeling language is Bandera [43]. Bandera automatically extracts a finite-state model from the source code and creates a model in the input language of one of the supported verification tools, which are Spin, SMV, and SAL. Bandera is based on the Soot compiler [68], a framework for optimizing Java bytecode. To optimize Java bytecode, Soot transforms bytecode to multiple intermediate languages. One of these intermediate languages is Jimple. In Bandera, the Java code is translated to Jimple. In order to simplify the code and to reduce the state space of the model, Bandera applies three main techniques on the Jimple representation of the code. The first one is a slicing technique that removes from the code the statements that are irrelevant for checking the desired property. Another reduction technique used by Bandera is a data abstraction technique that reduces the unnecessary details associated with variables. The last technique is component restriction, which decreases the number of components involved in the code or limits the ranges of variables. Applying the component restriction technique, the model does not exhibit all the behavior of the code. After applying the techniques described above, Bandera generates

the Bandera Intermediate Representation (BIR), which is a low-level intermediate representation of guarded commands, from the reduced code. Then a finite model of the code is created in the input language of a verification tool chosen by the user. After running the chosen verification tool, Bandera interprets the output of the tool. If the model does not satisfy a desired property, Bandera maps the trace back to the Java source code.

The SAL model checker has also been adapted to work with Java code [40]. The input can be either Java source code or Java bytecode which is automatically translated into the SAL intermediate language, which supports dynamic data structures. As in Bandera, in SAL, Java code is first translated into the Jimple intermediate language. Basically, SAL is a framework to merge different tools for model checking, abstraction, theorem proving, etc. SAL uses an intermediate language for specifying concurrent code. The SAL model checker is written in C++. The idea of the SAL model checker is based on the VeriSoft [41] model checker. VeriSoft is a model checker for concurrent C code which uses a scheduler to systematically execute the code in all possible ways. However, since VeriSoft does not store the visited states, termination of the model checking process is not guaranteed. In order to terminate, VeriSoft imposes a limit on the depth of the search.

Bogor is a software model checking framework that can also be adapted to verify Java code [69]. It extracts a model out of Java code in an extended version of BIR

which is the input language of the Bogor model checker component. Bogor uses a scheduler to systematically execute the model in all possible ways. The three main modules included in Bogor are search, the scheduler, and the state manager module, which are used to explore the state space. The goal of the Bogor project is to provide flexibility in the choice of input language, search algorithm, reduction techniques, and state-space representation. Since the Bogor framework is not tied to a particular setting, it can be adapted to efficiently model check any code.

The second generation of Java PathFinder, JPF, is an explicit-state model checker which directly model checks Java bytecode by driving all possible executions of the code using a scheduler. As was mentioned earlier, the first generation of JPF was simply a translator from Java to Promela. But this version of JPF worked on the source code of the Java application, which is not always available. Moreover, some features of the Java language cannot be easily represented in Promela, e.g. Promela does not support floating point numbers. Therefore, in 2000, JPF was refactored as a JVM which can execute all Java bytecode instructions generated by a Java compiler.

JPF provides support for dynamic features, such as class loading, dynamic data structures, and method invocation. But, unlike Spin and dSPIN, it does not impose any limit on the size of the state space. Compared to the other model checkers, JPF is very flexible. It offers a highly configurable structure, and introduces numerous

extension mechanisms which make it a suitable engine for many existing tools, e.g., tools that automatically create test cases by performing symbolic execution of Java applications [70, 71]. JPF can also be configured to use different functionalities, such as different search algorithms. Furthermore, JPF is applied directly on the code and does not require the modeling process. Since it directly handles bytecode, it can be used to verify any code that is compiled into bytecode such as Scala and Android applications. In 2005, JPF became an open-source project on the source code repository SourceForge.net. In 2009, it was moved to the NASA server, babelfish. Today, JPF is a mature tool with hundreds of active users and more than one hundred downloads every month. In Section 2.5, the structure of JPF and some of its features are explained in more detail.

2.5 Java PathFinder

In this section, we focus on the design and the structure of JPF. We explain JPF's main components, which are JVM and Search. We also explain some of its extension mechanisms. Finally, we discuss the POR technique implemented by JPF.

As mentioned earlier, one of the most interesting features of JPF is that it provides several extension mechanisms. To capture non-deterministic choices, JPF introduces *choice generators*, which keep the list of all enabled transitions at states. As one of its extension mechanisms, JPF allows the user to define choices captured

by choice generators. For example, through choice generators, one can make JPF consider certain values of a set of random integers. Another important extension mechanism of JPF is *listeners*. Listeners are runtime plugins that can interact with JPF as it runs the SUT. They can be used to modify the runtime environment of JPF and retrieve information from the execution of the SUT.

JPF also includes the *model Java interface* (MJI) to delegate code execution from the JPF level to the underlying JVM that runs JPF. This mechanism is mainly used to handle native calls. JPF also provides *model classes* which are used as alternatives to the actual Java classes. Moreover, the JPF *attribute system* allows for associating verification specific information to values of fields, local variables and stackframe operands which are manipulated and passed along the execution paths. This feature is suitable to implement data-flow related properties. Finally, JPF allows for changing the semantics of the execution of bytecode instructions.

2.5.1 JVM Component

As its core, JPF implements a JVM which executes the SUT. The JVM of JPF is able to handle all of the bytecode instructions that are created by a standard Java compiler. JPF itself is written in Java. That means that it is running on top of another JVM which we refer to as the host JVM. Figure 2.2 demonstrates the different layers that are involved in model checking a SUT using JPF.

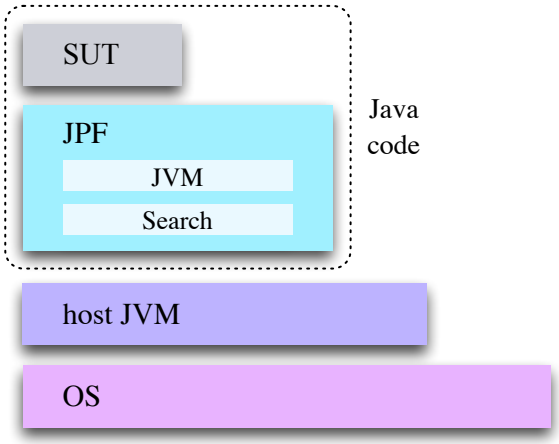

Figure 2.2: Different layers involved in running JPF on a target application

Similar to a standard JVM, the JVM of JPF implements a garbage collection mechanism. Java does not provide any explicit statement to deallocate the memory for an object that is no longer referenced by the code. Instead, it is provided with a garbage collection mechanism which automatically removes such objects from the heap. The garbage collection algorithm implemented in JPF is called mark and sweep [72]. In the marking phase, JPF marks all the objects that can be referenced by the code. Next, in the sweeping phase, it checks all the objects stored in the Java heap, and it removes all the objects that have not been marked in the previous phase.

JPF is a special JVM. It explores all potential executions in a systematic way. Each execution is a sequence of transitions and each transition is a sequence of bytecode instructions that takes the system from one state to another. In con-

trast, an ordinary JVM executes the code in only one possible way. Moreover, as JPF executes the target application, it checks for certain properties. Some of the properties checked by JPF are unhandled exceptions, deadlocks, and user-defined assertions which are used to test properties of the code's behavior.

While executing the code, JPF also generates the state space. Each state is composed of three elements: 1. a snapshot of the current execution status of the code, 2. the path that leads to this state, 3. possible choices in the current state where each choice value is associated with a transition. In order to represent choices at states, JPF defines the `gov.nasa.jpf.vm.ChoiceGenerator` class. The way that choices are encapsulated by JPF is explained in Section 2.5.3.

2.5.2 Search Component

As JPF explores the state space, in order to move from one state to another, it has to choose from the transitions leading out of the state. The way that JPF chooses the next transition is determined by its Search component. In other words, the Search component of JPF works as a driver for its JVM. The JVM component provides key operations which are used by Search to make it traverse the state space of the SUT. These operations are implemented by methods (namely, `forward()`, `backtrack()`, and `restoreState()`) within the class `gov.nasa.jpf.vm.VM`, which encapsulates the JVM component. `forward()` makes JPF move from the current

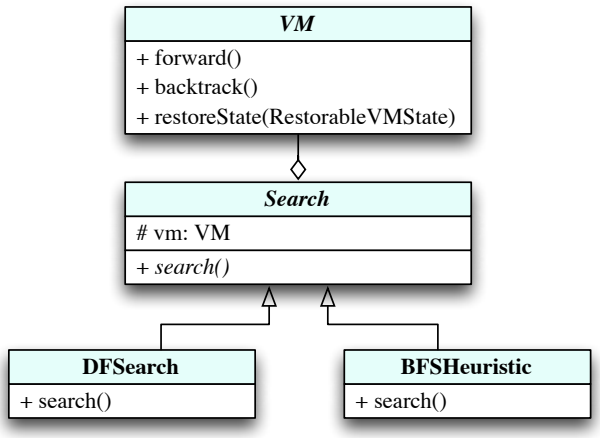

Figure 2.3: UML diagram representing the relationships between some of the main classes of JPF

state to a new state. `backtrack()` makes JPF to move back to the previous state. Finally, `restoreState()` is used to move to the state given as an argument.

The Search component can be configured to use different algorithms to traverse the state space, such as depth-first search (DFS) and breadth-first search (BFS). This component is encapsulated by the abstract class `gov.nasa.jpf.search.Search`, which includes an abstract method called `search()`. Java classes such as `gov.nasa.jpf.search.DFSearch` and `gov.nasa.jpf.search.heuristic.BFSHeuristic` implement different search algorithms by extending the `Search` class and overriding its `search` method. Using the JPF configuration mechanism, one can specify which `Search` subclass to use to encapsulate the Search component.

The UML diagram in Figure 2.3 depicts the relationship between the classes `Search` and `JVM`.

2.5.3 Encapsulating Choices

The goal of every model checker is to check if certain properties hold in states of the SUT. Since choices that a model checker makes at each state determine which states are explored next, the way that choices are computed is a fundamental part of every model checker.

Whenever there is non-determinism in code, the model checker needs to compute the possible choices. Non-determinism can be either thread non-determinism or data non-determinism. For code subject to thread non-determinism, the concurrent actions can be executed in any interleaving. Since different interleavings may lead to different states, scheduling choices are required to capture all possible interleavings of these actions. For code subject to data non-determinism (for example code including `java.util.Random.nextInt()` which returns a uniformly distributed `int` value), a variable can assume different values where each value may lead to different execution paths. Therefore, data choices are required to capture (some of) the values of the corresponding data type.

As mentioned earlier, JPF uses the `ChoiceGenerator` type to capture choices. Different subtypes of `ChoiceGenerator` capture different types of choices. For example, `gov.nasa.jpf.vm.ThreadChoiceGenerator` represents scheduling choices, and `gov.nasa.jpf.vm.IntChoiceGenerator` represents integer data choices.

In JPF, the interface `gov.nasa.jpf.vm.SchedulerFactory` encapsulates scheduling strategies, and the creation of all `ThreadChoiceGenerator` instances is delegated to an instance of this type. By default, JPF uses an instance of the subtype of `SchedulerFactory`, `gov.nasa.jpf.vm.DefaultSchedulerFactory`, which creates scheduling choices for operations that are subject to thread non-determinism such as `Object.wait()`. As one of its extension mechanisms, JPF can be configured to use a different subtype of `SchedulerFactory` to impose different scheduling policies.

2.5.4 Listeners

Listeners are considered the most important extension mechanism of JPF. Listeners interact with the JPF components, JVM and Search, and they retrieve useful information about the execution of JPF as it model checks the target application.

As is shown in Figure 2.4, listeners run at the same level as JPF, on top of the host JVM. They register themselves with the `Search` and/or `JVM` components in order to receive notifications on the occurrence of certain events. When any of those events occur, the component notifies the registered listener and it invokes a method of the listener which corresponds to the event. The notifications can be issued on a wide variety of events, from low level events like `instructionExecuted`, indicating that an instruction was executed, to high level events like `searchFinished`,

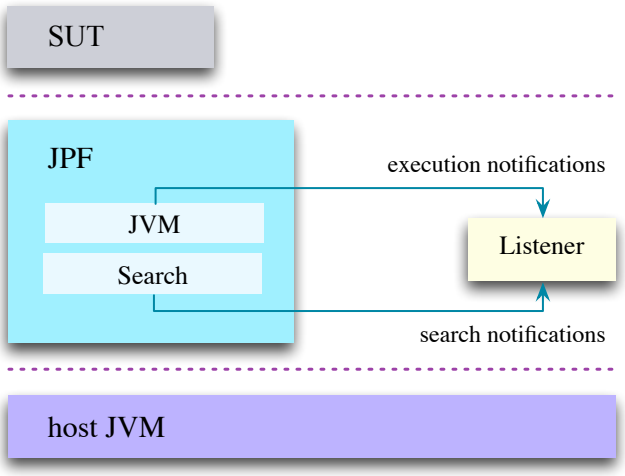

Figure 2.4: Listeners and JPF running on top of the underlying host JVM

indicating that JPF is done exploring the state space.

The JPF implementation includes two basic interfaces for listeners, which are `VMListener` and `SearchListener`. Listeners are represented by classes that implement one or both of these interfaces. Listeners that implement the interface `gov.nasa.jpf.vm.VMListener` register themselves with the `JVM` component. Similarly, listeners that implement the interface `gov.nasa.jpf.search.SearchListener` register themselves with the `Search` component. The race detector of JPF is an example of the listeners included in the implementation of JPF.

JPF Race Detector

As was mentioned earlier, JPF can be exploited to detect races in Java code. The JPF race detector is based on static race detection, and it is implemented in the form of a listener [73]. The idea behind this race detector is fairly simple. In every state that JPF visits, it checks all actions that can be performed next. If this collection of actions contains at least two conflicting accesses of a shared variable v, then a race on v is reported. A similar approach in a considerably simpler setting has been proposed in [74]. The race detector of JPF is sound, that is, the races that are reported by the tool are real races.

2.5.5 Handling Native Calls

A method is called native if it is implemented in a language other than Java but it is invoked from a Java application. Many of the classes of the Java standard library include invocations of native methods. Therefore, it is essential for Java model checkers to provide a way to handle native calls. The JPF model checker provides different ways to handle native methods. But before explaining how JPF handles native calls, we describe how an ordinary Java virtual machine handles native methods.

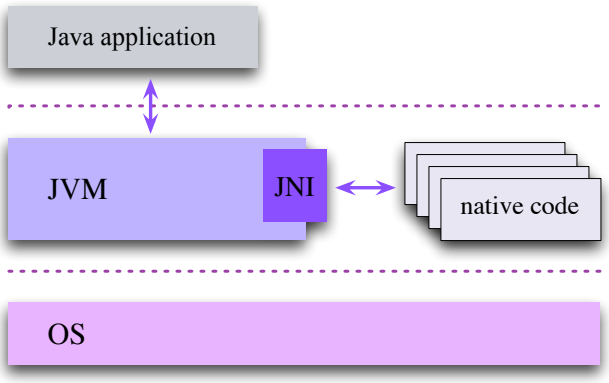

Figure 2.5: Java native interface included in JVM

Java Native Interface

Every JVM includes a Java native interface (JNI) [75] that allows Java applications to call or to be called by functions which are written in other languages such as C, C++, and assembly. Basically, JNI is used to transfer the execution from the Java level to the native level, as shown in Figure 2.5. Whenever the JVM comes across a bytecode instruction that invokes a method defined as native, the execution is transferred to the native level. After the native method returns, the execution is transferred back to the JVM. JNI allows us to use functions that have already been implemented in other languages. Moreover, in some cases, accessing code written in, for example, C and C++ from applications written in Java can improve the performance. Furthermore, JNI can be used when Java does not support certain platform-dependent features.

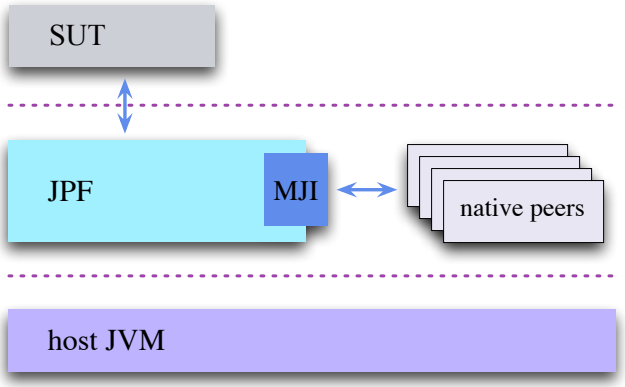

Figure 2.6: Model Java interface included in JPF

Model Java Interface

One of the important features of JPF is its model Java interface (MJI). In analogy to JNI, which is used to transfer the execution from the Java level to the native level, MJI is used to transfer the execution from the JPF level to the underlying JVM level (see Figure 2.6). The so called *native peer* classes play a key role in MJI. Native peers run on top of the underlying host JVM. Therefore, these classes are completely unknown to JPF and are not model checked by JPF at all.

JPF uses a specific name pattern to establish the correspondence between a native peer class and a class included in the SUT. It also relates the methods of these corresponding classes using a name pattern and an annotation of type `gov.nasa.jpf.annotation.MJI`. For example, Figure 2.7 shows how the correspondence is established between the `java.lang.StrictMath` class and its native peer.

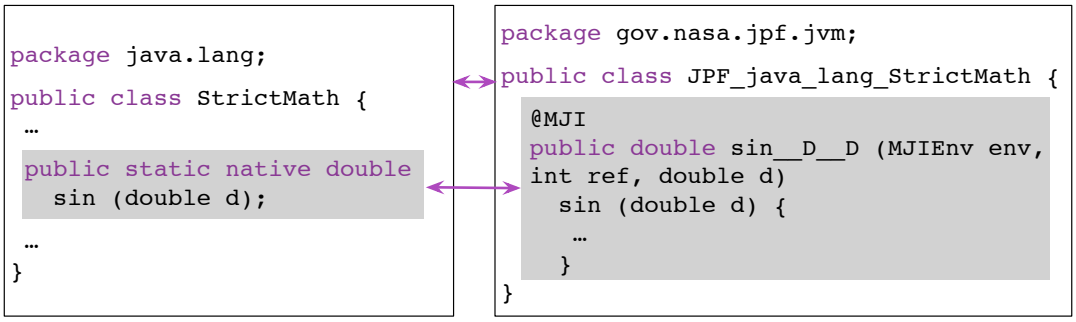

Figure 2.7: The correspondence between `java.lang.StrictMath` and its native peer

In this figure, the class on the left is part of the Java standard library and the one on the right is its native peer. When JPF gets to the bytecode invoking the method `sin` of `java.lang.StrictMath`, since `sin` is linked to a method in the native peer, it does not model check this method. Instead, the execution is transferred to the underlying JVM, and the host JVM executes the method `sin__D__D` of the class `JPF_java_lang_StrictMath`. MJI provides native methods with at least two arguments. The first argument of native methods is of type `MJIEnv`. `MJIEnv` is an interface that allows native peers to access all internal features of JPF. The second argument of native methods is an integer representing the JPF object or the JPF class that has invoked the native method (note that, in JPF, objects and classes are represented by a unique integer).

2.5.6 Model Classes

JPF has special classes called model classes which are used by JPF as alternatives to the actual Java classes. The model classes are considered to be part of the SUT. They are model checked by JPF and are invisible to the host JVM. These classes model the behaviour of actual classes and often they abstract from particular details of the actual classes. A model class has the same name as the class it models. For example, JPF contains a model class named `java.lang.String` which replaces the class `java.lang.String` in the standard Java library.

Before executing the SUT, JPF starts loading classes. To load classes, JPF starts from the directories that include model classes. Once a class is loaded, JPF does not load the other versions of the same class. In other words, by implementing model classes, we force JPF to use alternative versions of certain Java classes.

2.5.7 Reduction Technique

JPF uses a reduction technique to cut down the state space which is similar to the reduction technique by Godefroid [41]. The reduction of JPF is applied on-the-fly, and it is based on combining a sequence of bytecode instructions in a thread, that do not have any effects outside the thread, into a single transition. This approach is precisely described in Chapter 6.

3 Related Work

Conventional model checking techniques implemented by various Java model checkers [35, 43, 40, 69] are only applicable to single-process applications, and they cannot handle distributed systems. In general, applying the model checking technique on distributed Java applications is not trivial.

The techniques that have been proposed to model check distributed Java applications can be divided into two main categories: (1) cache-based and (2) centralization. In the *cache-based* approach, the model checker verifies only one process, and the rest of the processes run outside of the model checker. This approach uses a cache layer to keep the external processes in synchronization with the SUT. In the *centralization* approach, the distributed application is captured within a single process, and the model checker is able to verify all the communicating processes. The main distinction between these two techniques can be seen in Figure 3.1.

The cache-based approach runs only one process (which could be either a server process or a client process), as a SUT, within the model checker, and the rest of the

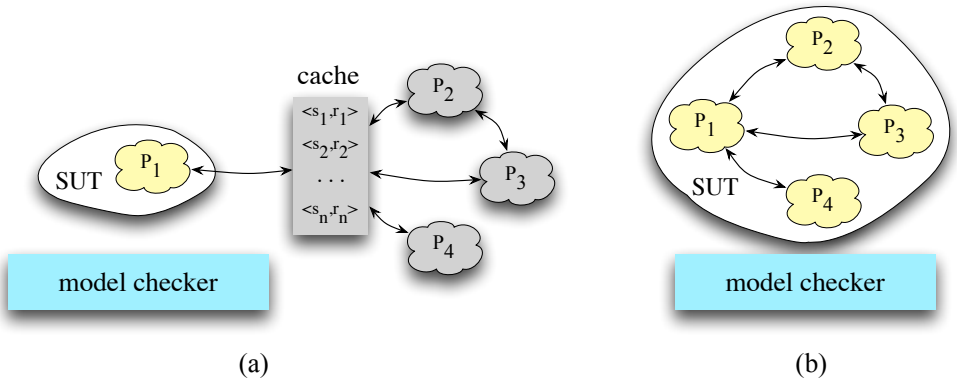

Figure 3.1: Different main approaches used to verify distributed systems, (a) cache-based, (b) centralization

processes, henceforth called *peers*, run outside of the model checker either within their native environment or a virtualization thereof [76, 77, 78].

The main challenge of this approach is to keep the SUT in synchronization with its peers since the model checker does not have any control over the execution of peers, and their execution is not subject to backtracking. After the model checker backtracks, the SUT may resend data which might interrupt the correct behavior of the peers. Moreover, after backtracking, peers do not resend previously sent data to the SUT. Existing cache-based techniques [76, 77, 78] address this problem by introducing a cache layer between the SUT and its peers. Section 3.1 explains these techniques in more detail.

An alternative to the cache-based approach is centralization [79, 80, 81, 82]. The existing centralization techniques can be applied at either the SUT level or the OS

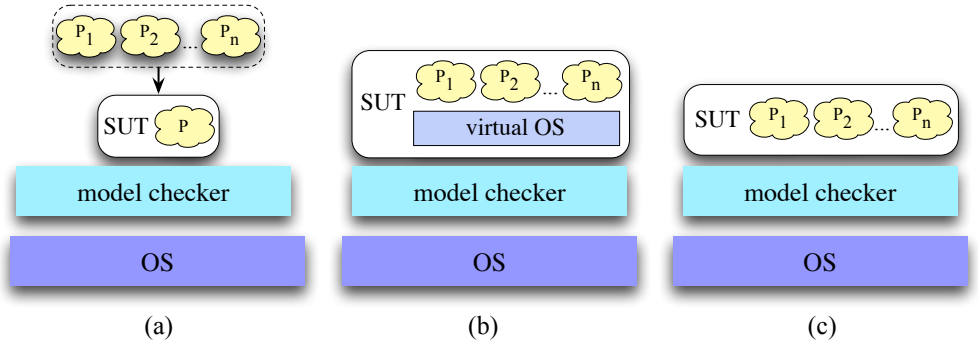

Figure 3.2: Different centralization techniques applied at different levels, (a) the SUT level, (b) the OS level, (c) the model checker level

level. In centralization at the SUT level (see Figure 3.2 (a)), the distributed application is transformed into a single-process application which is then fed to a model checker as a SUT [79, 80]. In this technique, distributed processes are mapped onto communicating threads within a single process application. This requires a model of the inter-process communication (IPC) mechanism that is used for communication. Centralization at the SUT level requires dealing with several issues. How are processes represented? How is exclusive access to static attributes provided for different processes? How are static synchronized methods handled? How is the shutdown semantics specified? Since the technique proposed in this research is also subject to similar issues, we provide a detailed discussion of techniques that apply centralization at the SUT level in Section 3.2.

In centralization at the OS level (see Figure 3.2 (b)), the distributed application

is run on a virtualization tool. This approach does not require transforming the SUT. The model checker's scope is extended to capture the state of the virtualization tool running all communicating processes. Nakagawa et al. [82] develop a model checking framework based on this approach. Their model checking environment is very close to the actual execution environment. This framework combines the user-mode Linux, which is a Linux virtual machine generated by porting the Linux OS to Linux system calls [82], and the GNU debugger (GDB) [83]. It can save and restore the entire Linux state. In this approach, each process is systematically run by a GBD process. Once non-determinism is detected within a process, the state of the OS and the possible execution paths leading out of it are computed and are used to explore the state space. GBD can support several programming languages including Java.

One of the drawbacks of this approach is that it requires manual user intervention, e.g., the user needs to specify non-deterministic points within processes. Moreover, in this approach, the OS along with the running processes form the SUT, and therefore states include redundant information if one is only interested in the behavior of the distributed application, and not the OS. It consumes large amounts of time and memory resources, and aggravates the state space explosion problem.

In this research, we propose a novel centralization technique which is applied at the model checker level (see Figure 3.2 (c)). Chapter 4 explains the proposed

technique and the ways it addresses the limitations of existing approaches outlined in Table 3.1.

technique		weaknesses	strengths
cache-based [76, 77, 78]		• can check only one process and its communication with the peers	• scales better than centralization • does not require transforming SUT • allows for processes in different formats
centralization	SUT level [79, 80, 84]	• does not keep types separated for Java (standard) libraries • does not address the class version conflict problem for Java standard libraries • cannot handle static initializers with side effects • cannot support custom class loaders • cannot support Java reflection API • requires manual user intervention to capture correct shutdown semantics • does not address starting shutdown hooks upon a normal shutdown • requires all the processes to be in the same format	• scales better than centralization at the OS level • can check all communicating processes
	OS level [82]	• aggravates the state space explosion problem • requires manual user intervention • includes redundant information in states for just analyzing the distributed application	• can check all communicating processes • does not require transforming SUT • allows for processes in different formats

Table 3.1: The comparison of different existing techniques for model checking distributed applications

3.1 Cache-based Approach

In the cache-based approach, the model checker only captures one process, as a SUT, and its communication with its peers [76, 77, 78]. The main effort in applying this technique is to synchronize the execution of the SUT with the peers' execution which is achieved through a cache-layer. Existing techniques associate a cache to each communication object (e.g., a socket) between the SUT and the peers, which captures communication data by storing interactions in the cache.

Initial work by Artho et al. [76] based on the cache-based technique uses a cache to capture the communication between the SUT and the peers. We refer to such a cache as a *linear-time cache*, since it can only handle deterministic communication traces between the SUT and the peers regardless of thread scheduling within the processes. It assumes that all execution paths of the SUT lead to communication traces which are consistent with the first observed sequence of communication data. This requires that in all execution paths, for each communication object, the SUT sends the same sequence of requests and receives the same sequence of responses.

This is illustrated in example (a) in Figure 3.3. The graph on the left represents the search graph generated from model checking a SUT. The graph on the right side represents the communication data between the SUT and the peers. After exploring the state s_3, the model checker backtracks to the previous state and

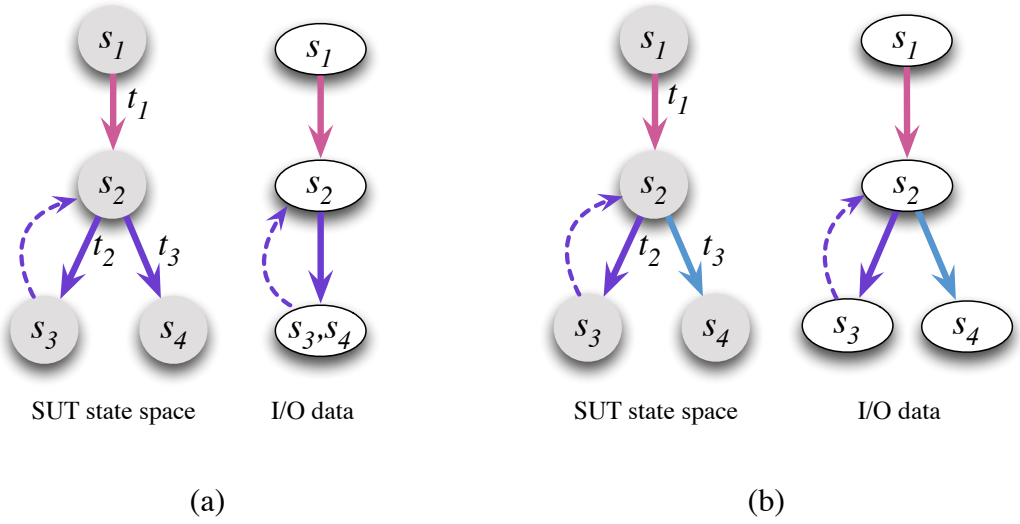

Figure 3.3: Two different cache-based techniques for model checking distributed applications, (a) based on linear-time cache [76] (b) based on branching cache [77]

takes a new transition, t_3. When executing t_3, this approach requires observing the same communication data as it observed during t_2. In this approach, when the SUT requests data that was previously received in an earlier run, the request is not resent, and instead the corresponding response is obtained from the cache. It only transmits data over the network if communication proceeds beyond the cached traces, and then the cache information is extended with the newly received data.

In the approach presented in [77], Artho et al. replace the linear-time cache with a cache which is applicable to a wider range of applications (see Figure 3.3). We refer to such a cache as a *branching-time cache*, since it allows for non-deterministic communication traces between the SUT and the peers. However, it still does not

consider non-determinism within the peers. In this approach, communication data sent by the SUT can deviate from previously observed data. However, for a given sequence of requests, the peers are still required to issue the same sequence of responses. After backtracking to an earlier state, if the SUT sends data to a peer which is different from previously cached information, the cache implementation spawns a new peer, and communicates with the peer using the cached trace up to the current state. Then it transmits the SUT's new request over the network, and it updates the cache correspondingly. This is shown in example (b) in Figure 3.3.

To take into account non-determinism in peers, a technique has been proposed which combines the use of a cache layer with checkpointing [78]. Checkpointing is an approach to capture the state of a group of processes. The checkpointing environment [85] can run the peers, store their states, and backtrack to previously visited states at any point. As the SUT is model checked, a strategy is used to decide at which points the SUT needs to synchronize with the peers. At those points, the checkpointing environment stores the state of the peers. When the SUT backtracks to a state that requires synchronization with a peer, the checkpointing environment restores the peer's state accordingly, and makes it re-execute from the restored state.

Compared to other cache-based techniques, this approach covers a wider range of applications, since every time the model checker backtracks to a state, it does

not require the peer to send the same sequence of requests to each communication object up to that state. This approach guarantees that, in every execution path, the SUT receives the expected responses from non-deterministic peers. However, it still does not have control over the scheduling of peers' threads, and it thus cannot model check the SUT against all the possible inputs from peers. Moreover, checkpointing is a very expensive procedure in terms of time and memory, since it needs to store and restore the state of the peers and the underlying OS.

In general, cache-based techniques scale better than the centralization techniques, since the model checker only captures one single process. However, a major drawback of these techniques is that they are not able to check all possible behaviors of the distributed application, and they may miss errors caused by certain responses of peers.

3.2 SUT Level Centralization

Centralization at the SUT level was initially proposed by Stoller and Liu [79]. It transforms distributed Java applications by merging all processes into a single process, and it replaces remote method invocations (RMIs) with local ones that simulate RMIs. Using the Java RMI mechanism, Java processes can communicate with one another remotely. This mechanism allows an object in one JVM to call methods on an object running within another JVM.

Every Java process provides a self-contained execution environment which includes an exclusive set of basic runtime resources, for example a memory space. A Java process includes at least one thread. Each thread also provides an execution environment, and it is represented by an instance of the `java.lang.Thread` class. Threads that exist in a process share the same runtime resources. Centralizing distributed applications requires a way to associate each process to the group of threads that exist within that process.

To address this, Stoller and Liu replace all the occurrences of the `Thread` class with the class `CentralizedThread` that *extends* `Thread` and declares an instance field of type `integer` which represents the process id. By extending the class `Thread`, `CentralizedThread` becomes its subclass and it thus inherits all the `Thread` members.

One of the main efforts in applying centralization at the SUT level is to provide different parts of the application representing different processes with exclusive access to *Java types*. Java types are represented by instances of the class `java.lang.Class` which are also referred to as *class descriptors*. As an example, consider the class `java.lang.String`, the instances of which represent string objects. The type `String` itself is encapsulated by an instance of `Class` in the JVM runtime environment. In Java, *class loaders* are responsible for *defining* types by accessing the class files, parsing them, and generating the instances of `Class`. To

run a Java application, the JVM uses system-defined class loaders to define standard Java library classes, platform specific frameworks, and application classes. In a standard JVM, a class with a given name can be loaded only once by a certain class loader (see Section 4.1 for a more detailed description of Java types and class loaders).

Each attribute and method of a Java class is declared as either instance or static (i.e., by using the keyword `static` in the declaration). Every object has its own copy of instance members. The static members belong to classes instead of objects, and there is thus only one fixed memory location associated with a static attribute.

In a distributed system, since Java processes run on different JVMs, every process has its own system-defined class loaders, and therefore processes do not share types. For example, consider a distributed system composed of two Java processes, P_1 and P_2, that declare the following class.

```
public class C1 {
  public static int i;
}
```

Since P_1 and P_2 are two different processes, each of them has a separate copy of the class `C1`, and it thus has an exclusive memory location for the value of `i`.

By combining processes into one, different parts of the transformed system that represent different processes now share the same class loaders. Consequently, they share the same classes and access the same static attributes and methods. By

centralizing the example above, P_1 and P_2 access the same memory location for `i`. That leads to a model which is not sound, that is, it includes executions which are not consistent with the correct behavior of the system. To capture the correct behavior, it is essential to provide a mechanism to avoid processes from sharing static members of classes.

To address this, Stoller and Liu use arrays to capture the effects of multiple copies of a class, i.e., they replace the declaration of a static field of type `T` with an array of type `T`, and transform the static methods and class initialization accordingly. For example, by applying their centralization approach on the system composed of P_1 and P_2, the class `C1` is transformed to the following.

```
public class C1 {
  public static int[] i = new int[2];
}
```

In the transformed application, each part that represents a process is provided with exclusive access to an element of the array `i`.

Another problem to deal with in centralization at the SUT level is handling static synchronized methods. In Java, every object is provided with a monitor which can be locked or unlocked by threads. A lock on an object monitor can be held by only one thread at the time. Executing a synchronized method requires acquiring a monitor lock. For an instance method, the monitor of the object to which the method belongs is used. For a static method, the monitor of the class

descriptor is used which, as mentioned earlier, is a unique instance of `Class`. In other words, executions of static synchronized methods of a class are synchronized on the class descriptor.

Since in the transformed application, a single class represents multiple classes, the class descriptor cannot be used as a lock anymore. If it were, different parts of the application representing different processes would share the same lock to execute static synchronized methods of the class. Stoller and Liu include an array of objects in each transformed class where each element of the array is used as a lock associated with a process. This lock is held by all the threads belonging to that process in the entire body of the static synchronized methods of the class. Consider the following class declared by the processes P_1 and P_2.

```
public class C2 {
  public static synchronized void m() {
    ...
  }
}
```

By applying this centralization technique, the body of `C2` becomes as follows where `prcId` refers to the id of the current process.

```
public class C2 {
  private Object[] locks = new object[2];
  public static void m() {
     synchronized(locks[prcId]) {
       ...
     }
   }
```

```
}
```

In the transformed class, the `synchronized static` method `m()` is marked as not synchronized, and each simulated process has exclusive access to an element of the array `locks` and holds that element as a lock to execute the body of `m()`.

Finally, centralization at the SUT level needs to address the shutdown semantics of the centralized application. Shutting down a process requires terminating threads that exist within the process. Shutdown semantics also refers to actions performed by the JVM upon the application termination which can be either a normal shutdown or an abnormal shutdown. Normal shutdown occurs when the method `java.lang.Runtime.exit(int)` is invoked or the last non-daemon thread terminates. Upon a normal shutdown, the JVM goes through two phases. In the first phase, the JVM initiates all *shutdown hooks*. A shutdown hook is a thread that has not been started yet. Shutdown hooks are explicitly set through the method `java.lang.Runtime.addShutdownHook(Thread)`. Upon normal termination, the JVM starts the threads representing shutdown hooks in some unspecified order. This way, the JVM allows each process to dispose of resources in use and do other clean up actions.

In the second phase, upon the Java process request, the JVM executes *finalizers* of objects that have not been invoked yet. Any object whose class overrides `java.lang.Object.finalize()` is called a finalizable object, and the method

`finalize()` is referred to as its finalizer. The JVM invokes a finalizer when a finalizable object is no longer referenced, and it is about to be garbage collected by the JVM. Using finalizers allows for performing clean up actions before the object is reclaimed by the garbage collector.

Upon an abnormal shutdown, the running JVM is forced to terminate without initiating the shutdown sequence. The abnormal shutdown occurs when the method `java.lang.Runtime.halt()` is executed or the JVM is interrupted externally, for example by shutting down the machine.

To deal with the shutdown semantics, the approach proposed by Stoller and Liu replaces invocations of the method `System.exit(int)` (i.e., it terminates the execution of the Java process by invoking `java.lang.Runtime.exit(int)`) with throwing the exception `java.lang.ThreadDeath`. Since throwing this exception only terminates the calling thread, their approach can only deal with distributed applications composed of single-thread processes. Moreover, their approach does not include any mechanism to free resources after processes terminate, and it does not deal with shutdown hooks in the case of normal shutdown.

Artho and Garoche [80] also apply centralization to transform the distributed SUT to a single-process system. Their work provides a more accurate transformation of distributed Java applications, and addresses some of the limitations of previous work by Stoller and Liu. In contrast to the former centralization approach,

which works with Java source code, their work performs bytecode instrumentation. During the last decade, the Java language has been considerably extended, but changes to Java bytecode instructions have been very minor. Unlike the former approach, Artho and Garoche's work is applicable to systems compatible with newer versions of Java.

As mentioned earlier, work by Stoller and Liu replaces each class descriptor with an array of the objects used as locks to synchronize the execution of static synchronized methods. However, that can lead to incorrect behavior of the SUT if a part of the SUT code, other than its static synchronized methods, relies on synchronization on a class descriptor, e.g., a thread holds the object `C.class` as lock, which is the instance of `java.lang.Class` representing the type `C`.. Consider the following example.

```
public class C3 {
  public static synchronized void m1() {
    ...
  }

  public void m2() {
    synchronized(C3.class) {
      ...
  }
}
```

In this example, the method `m1()` cannot run concurrently with the `synchronized` block in the method `m2`, since executing any of them requires a thread to hold the

class descriptor `C3.class` as a lock. Since the Stoller and Liu technique marks `m1()` as not `synchronized`, in the transformed system the body of `m1()` and the `synchronized` block within `m2()` can execute concurrently. To address this issue, the Artho and Garoche approach performs additional checks at runtime to see if the lock being used is a descriptor of a transformed class. If so, it is replaced by the corresponding element of the array of locks.

To address the shutdown semantics, similar to work by Stoller and Liu, the Artho and Garoche approach also replaces invocations of methods that terminate the process, such as `System.exit(int)`, with throwing the exception `java.lang.ThreadDeath`. Their approach also does not provide any mechanism to kill other threads within the terminating process. But for the cases that all other remaining threads are daemon, their solution suggests identifying and ignoring the failures in a dead process. They also provide a mechanism to free resources after processes terminate. However, starting shutdown hooks requires manual instrumentation of the centralized program.

The RMI model proposed in work by Stoller and Liu cannot be extended to support arbitrary communication based on sockets. The centralization technique by Artho and Garoche is applicable to applications that use sockets for communication.

Ma et al. [84] also use centralization at the SUT level. Their approach extends the Artho and Garoche approach and addresses some of its limitations. Their

approach addresses the class version conflict problem which occurs when different processes use classes with the same name but different bytecode. Unlike the other proposed approaches, this approach does not force processes to use the exact same version of Java classes. Before applying their centralization algorithm, similar to the one proposed in [80], they resolve class version conflicts between processes by renaming classes that have identical names but different bytecode. However, their approach cannot address the class version conflict problem for Java system libraries, since it is not applied on native methods and code relying on the reflection API.

The approach by Ma et al. provides a way to terminate processes by killing all of their threads. This has been achieved through the Java thread interruption mechanism, and it requires code instrumentation. This approach also does not address starting shutdown hooks upon process termination.

Another approach to model check distributed Java applications based on centralization at the SUT level has been proposed by Barlas and Bultan [81]. They mainly focus on the environment generation problem for network communication, and present a framework, NetStub, that models the Java networking packages `java.net` and `java.nio`. They do not address the problem of merging processes into a single process. To check communicating processes, their approach requires the user to manually implement a driver that centralizes the SUT. Moreover, this approach requires a manual integration of the NetStub framework into the SUT.

The resulting system is a standalone, centralized, Java application which uses the NetStub framework for communication. NetStub also allows for model checking just one process within the distributed application through an event generator component. The user needs to write an event generator component using the NetStub API, and a driver that starts the process. This component generates network events which are perceived by the SUT.

3.2.1 Limitations

There are several issues that existing centralization techniques at the SUT level do not address. None of these techniques are able to apply the transformation on classes from the Java standard libraries, and they only transform the user-defined classes. The processes thus end up sharing the same classes and therefore accessing the same static attributes and static methods. One of the main reasons is that there are standard classes that provide access to global resources where transformation is not desirable. For example, consider the class `java.lang.System`. This class has the static field `out` which represents the standard output stream corresponding to display output. Another reason is that transformation is not applied on native methods. However, there are numerous classes in the Java standard libraries that include native methods. Sharing standard classes is a major drawback of these techniques, since every Java application relies on classes from standard libraries.

Making processes share the same standard classes may affect the soundness of the verification. To validate our claim, we developed a distributed system composed of a server process and a client process. The server process uses the value of `java.lang.System.out` as a lock for a `synchronized` block in which it blocks until it connects to a client. The client process uses the value of `java.lang.System.out` as a lock for a `synchronized` block in which it waits until its connection request is received by the server. When executing this distributed system, the connection is successfully established between the processes. However, after applying the SUT level centralization, the connection cannot be established between the processes. In the centralized system, two processes share the field `java.lang.System.out`. After one of the processes, p_1, holds the monitor on the value of `java.lang.System.out`, the other process, p_2 waits until p_1 releases the monitor. However, p_1 waits for p_2 in the synchronized block. Therefore, the processes never connect to one another, and this execution is not consistent with the behavior of the actual system.

In Java, every class is initialized by its *static initializer* method which is invoked at class load time. The SUT level centralization techniques are not able to handle applications where static initializers rely on certain data or include side-effects. The static initializer of a class is only executed once after the class is loaded. Centralizing the application transforms each static initializer code in a way that it executes the original code once per process. However, the static initializer code may have certain

side-effects where executing it more than once in the transformed version affects the correct behavior of the application.

To validate our claim, we developed a Java process, p, with a class static initializer that accesses a file, named f, where its execution depends on the f content. If f is empty the static initializer enters the text `processed` into the file, and p continues and prints `processed` on the console. If f already includes the text `processed`, the static initializer terminates the execution. Consider a distributed system composed of two processes, p_1 and p_2, that behave similar to p. When executing p_1 and p_2 on two separate machines where the processes access empty files, each process prints the text `processed` on the console. However, by centralizing the distributed application, both processes access the same file and thus the second process always accesses a non-empty file. Therefore, the centralized version leads to an execution in which only one process prints the text `processed`, which is not an admissible execution in the original system. As a consequence, the SUT level centralization is not sound.

Another limitation of centralization at the SUT level is missing support for the Java reflection API. Java reflection is a commonly used API by Java applications that allows for examining classes at runtime, e.g., using this API one can retrieve an object that represents an attribute declared in a class. In general, the use of Java reflection calls is dependent on the class structure. However, by transforming the

SUT, the structure of classes changes, and the behavior of code relying on reflection may become undefined. That affects the soundness of the approach.

To validate our claim, we develop a distributed system composed of two processes that manipulate a static field, `f`, of type `int`. Each process uses the reflection API to retrieve the value of `f`, i.e., processes invoke the method `getInt(Object)` on the object of type `java.lang.reflect.Field` which represents the field `f`. By applying the SUT level centralization, the type of the field `f` is changed from `int` to an array of `int`. However, the call `Field.getInt(Object)` remains unchanged, and executing the centralized system gives rise to the exception `java.lang.IllegalArgumentException`. Since this execution is not an admissible execution in the original system, it can be concluded that the approach based on the SUT level centralization is not sound.

Moreover, previous work is not able to address the class version conflict problem for Java standard libraries, and different processes are forced to use the exact same version of Java standard classes, e.g., they cannot capture a system composed of two processes where one process uses Java 1.5 and the other one uses 1.6. Centralization at the SUT level requires all the processes to be in the same format, e.g., in the approach by Stoller and Liu all processes must be written in Java, and in the approaches explained in [80] and [84] all processes must be compiled to Java bytecode. Furthermore, the existing approaches require the process implementa-

tion to be independent of host information such as host name. Finally, processing shutdown hooks, which is addressed by only one of the techniques, requires manual intervention of the user.

In the next chapter, we present a new centralization technique which is applied at the model checker level and addresses the limitations of existing techniques. The proposed approach does not require transforming the SUT. Our centralization technique can keep the Java standard libraries separated between different processes, and it addresses the class version conflict problem for such libraries. Moreover, it can handle code using the Java reflection API. Finally, it can handle static initializers that have side-effects.

4 Centralization

In the previous chapter, we described the existing centralization techniques used to model check multiple processes and discussed their limitations. This chapter describes our centralization approach and the way it addresses the limitations of the existing techniques. Applying centralization is the first step of our work towards model checking distributed applications. We apply our technique to JPF. Our centralization approach extends a JVM (which is the core of JPF) that executes single process applications to distributed systems, while preserving the execution semantics of each process.

Using centralization allows for analyzing the behavior of the entire distributed application rather than cache-based techniques that can only check the behavior of one process. Unlike the existing centralization techniques, which are applied at either the SUT or the OS level, the proposed approach is applied at the model checker level (see Figure 3.2 (c)), that is, it expands the scope of the model checker to accept multiple processes.

The proposed centralization approach is part of JPF 7. This approach does not require instrumenting the code of the distributed SUT, and it allows the model checker to accept multiple processes in their original form as input.

Providing support for distributed applications in JPF requires addressing several issues. The first issue is to separate types between processes. As mentioned earlier, each Java process has its own separate memory space. To keep data separated between Java processes, it suffices to separate types. To provide processes with exclusive access to types, we propose a new class loading mechanism.

Another issue is representing processes. Since each process represents a distinct execution environment, a mechanism is needed to partition the execution environment of the model checker into different parts where each is associated with one process. Our approach models each process as a group of threads. That is achieved by mapping threads that belong to the process to an object that uniquely identifies the process.

Finally, to execute the centralized SUT, a specialized JVM, which can handle multiple processes, is required. As part of our work, we implement a new JVM within JPF which we refer to as the *multiprocess JVM*. The multiprocess JVM is able to execute distributed applications. The thread scheduling policies used by this JVM are different from the ones used by the existing JVM of JPF. To impose new policies, our approach introduces a new scheduler factory (Section 2.5.3) which

is used by the multiprocess JVM to define scheduling choices. The remainder of this chapter explains the three steps of our centralization approach mentioned above.

4.1 Separating Types

In JPF, each type is identified by only the name of the class. Therefore, by merging multiple processes into one process in JPF, processes share the same types. Subsequently, merged processes access the same static attributes and execute the same static methods. Keeping types separated between different processes is the main challenge in applying centralization at the model checker level, which requires extending the JPF type system.

The class loading mechanism plays a central role in the Java type system. It is one of the key concepts defined in the JVM specification [86, 87]. It provides on-demand lookup and generation of class files, and it transforms such data into VM specific constructs, which are encapsulated by `Class` instances known as Java types. This process is performed by class loaders.

In Java, a type is identified by a pair $\langle N, L \rangle$ where N is the fully qualified name of the class and L is the class loader that defines the type. Consider, for example, two different class loaders L_1 and L_2 that access the same class file, `x.y.Z.class`, to define a type. The JVM identifies the types defined by L_1 and L_2 as pairs $\langle x.y.Z, L_1 \rangle$ and $\langle x.y.Z, L_2 \rangle$, respectively. They are treated as distinct types, and

the static fields of their associated class objects are kept apart and, hence, may have different values. Setting an instance of type $\langle x.y.Z, L_1 \rangle$ to an instance of the type $\langle x.y.Z, L_2 \rangle$ gives rise to a type mismatch error in the JVM.

In this work, we propose a class-loading mechanism which is based on the class-loading model of Java. We implement our model within JPF. Such a model allows for capturing multiple processes within the model checker, but it still preserves the execution semantics of each individual process. Prior to this work, JPF used a basic mechanism to define types without using entities that represent class loaders. Our approach also required extending the JPF type system by taking class loaders into account to identify types. Therefore, for a given fully qualified name of a class, there can exist multiple types in JPF. This functionality provides a basis for giving each part of the SUT that represents a process exclusive access to a set of classes used within the process. This is essential to support distributed applications.

Next, we elaborate on the class-loading model of Java, and then, we describe how we extend JPF's class-loading model to separate types between different processes.

4.1.1 The Java Class-Loading Model

This section gives a high level description of Java's class-loading model. For a precise description, the reader is referred to previous work [86, 88, 89, 90] which provides formal specifications of the model.

Java distinguishes between two fundamental categories of class loaders: (1) bootstrap class loaders and (2) `java.lang.ClassLoader` instances. A standard JVM has exactly one bootstrap class loader. This class loader is integrated into the JVM implementation and cannot be accessed by Java applications. It is used to define standard Java library classes, e.g., classes in the `java.*` and `javax.*` packages. The JVM assumes that every class defined by this class loader is trusted, and is exempt from verifying its binary representation [87].

`ClassLoader` instances are ordinary Java objects. Any application can control how its own classes are loaded by overriding `ClassLoader` methods in the subclasses. The key method of `ClassLoader` is `loadClass`. This method takes the name of a class as argument, loads the class, and returns the corresponding `Class` object. The method `loadClass` implements the lookup policies which by default follow the *delegation* pattern (explained in the following section). To access binary data in class files, `loadClass` invokes the method `defineClass` on the class loader object. The method `defineClass` parses the class file and transforms its data into a `Class` object. This method is declared as *final*, that is, it cannot be overridden by subclasses of `ClassLoader`. Moreover, every `ClassLoader` instance is associated with a *search path* which specifies the locations at which the class loader searches for class files.

Next, we discuss some important aspects of the Java class-loading model which

are also crucial to our model.

Delegation Pattern

The standard implementation of Java follows a delegation pattern in which every class loader (except the bootstrap one) has a *parent*. To load a class, first the class delegates the load to its parent, and only defines the class itself if it has no parent, or the parent fails to return a `Class` object. The delegation pattern is implemented by the `loadClass` method of `ClassLoader`. However, it is not enforced, and one can replace it by overriding `loadClass`. The following code, that presents part of the method `ClassLoader.loadClass`, shows how the delegation model is implemented. The field `parent` represents the parent class loader, `c` is a variable of type `Class`, and `findClass` locates the class file and defines the type by invoking `defineClass`.

```
public Class<?> loadClass(String name) {
  ...
  if (parent != null) {
    c = parent.loadClass(name);
  }
  ...
  if (c == null) {
    c = findClass(name);
  }
  ...
  return c;
}
```

The delegation pattern implies that the class loader that initiates the load of the class is not necessarily the defining class loader. Consider the following definitions taken from [86].

Definition 4.1.1. *Let `C` be the return value of the method `L.defineClass`. Then `L` is the defining class loader of `C`.*

Definition 4.1.2. *Let `C` be the return value of the method `L.loadClass`. Then `L` is an initiating class loader of `C`.*

For example, consider a class loader hierarchy that follows the delegation pattern and includes the class loader `B` as the parent of `A`. The application attempts to load the class `C` by invoking `loadClass` on `A`. First, `A` delegates the load to `B` which successfully defines the type and returns it as the result of `defineClass` to `A`. The type defined by `B` is also going to be the return value of the call `A.loadClass(C)`. However, `B` is the defining class loader, whereas `A` and `B` are both initiating class loaders of `C`.

To run a Java application, the JVM uses a standard hierarchy including three system-defined class loaders: the bootstrap class loader, the extension class loader, and the application class loader. This hierarchy follows the delegation pattern, and it exists during the entire execution of the Java application. The bootstrap class loader is the root of the hierarchy, and has the extension class loader as its child

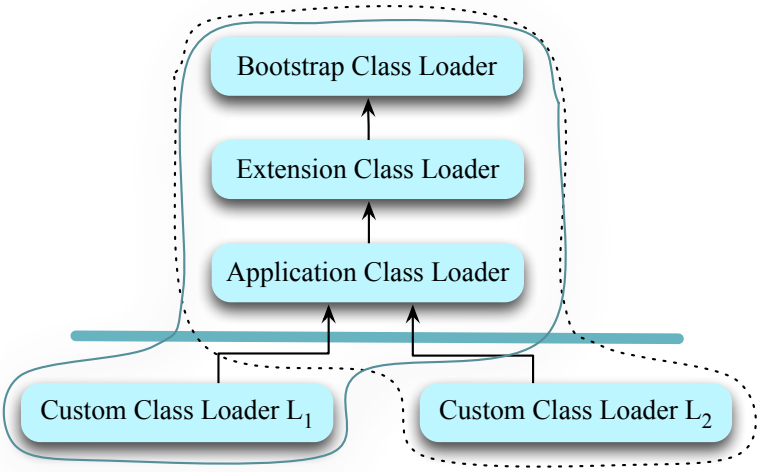

Figure 4.1: The delegation pattern of the Java class loading model. The solid and dashed contours represent the scope accessed by the classes defined by the class loader L_1 and L_2, respectively

which is used to locate and define platform specific frameworks (the locations of which are specified by the `java.ext.dirs` system property). The application class loader is responsible for loading classes declared by the Java application, and it has the extension class loader as its parent. By default, the application class loader is used as the parent of custom class loaders. Figure 4.1 shows the standard hierarchy of Java. The delegation relationships between class loaders are represented by arrows.

Resolution

The Java class-loading model dynamically extends an application at runtime. It loads classes on-demand. The compiler does not determine all the classes that will eventually be part of the application. The JVM starts the execution of a Java application by loading the initial class using the application class loader. From there, it loads classes only when they are referenced by the application. This process of dynamically extending the application is called *dynamic linking*, and it is provided through *resolution*.

Every Java class file has dependencies to entities referenced within its implementation, including its superclass, implemented interfaces, and any other types used in the body of the class. The class keeps symbolic references to the runtime constant pool[13] for these entities.

Consider the following example.

```
class A extends B {
  C m() {
    D d = new D();
  }
}
```

The class file representing `A` includes symbolic references to `B`, `C`, and `D`. As the

[13]Every Java class has a runtime constant pool which is an internal table that contains different kinds of constants, ranging from numeric literals known at compile time to references resolved at runtime.

JVM executes the application, it *resolves* these dependencies dynamically by replacing the symbolic references by concrete references. This process is performed recursively. Resolution of superclasses, implemented interfaces and types used to declare attributes in the class occurs while the class is being defined by the method `defineClass`. Besides, invocation of certain methods in the Java reflection API and certain bytecode instructions (such as `getfield`, `getstatic`, and `instanceof`) contain symbolic references. When executing any of those, the JVM resolves the corresponding symbolic reference on-the-fly.

Resolving symbolic references within a class is initiated by the class loader that defines the class. Consider the example explained above. Let `L` be the defining class loader of `A`. To resolve `B`, `C`, and `D`, the JVM invokes the method `loadClass` on `L`.

Namespace

The namespace of a class loader can be defined as a finite map associating names of classes loaded by the class loader to their corresponding `Class` objects. The JVM enforces class loaders to consistently return the same `Class` object for every type in their namespaces. This policy not only avoids type incompatibilities at runtime, but also prohibits the injection of redefined classes from different locations, which could compromise security.

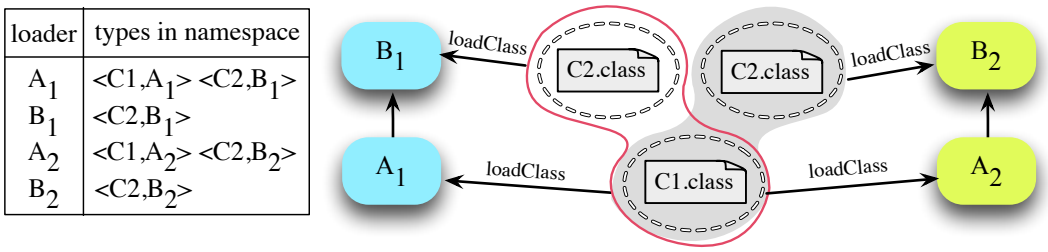

Figure 4.2: The namespaces determine the class view. The solid contour and the gray area determine the view of `C1` when loaded by A_1 and A_2, respectively

Namespaces determine the view of the classes, that is, each class loaded by the class loader L can only reference types within L's namespace. As a consequence, depending on which class loader defines a type associated with a class file, the corresponding `Class` object can have different views. That can be seen from the example shown in Figure 4.2 which includes two distinct class loader hierarchies that follow the delegation pattern. B_1 and B_2 are the parents of the class loaders A_1 and A_2, respectively. A_1 and A_2 have access to the class file `C1.class` in their search paths. B_1 and B_2 have access to different class files with different implementations, but share the same name, `C2.class`. Let the class `C1` extend `C2`. When `C1` is loaded by A_1, the type $\langle C2, B_1 \rangle$ becomes its superclass, whereas, when loaded by A_2, $\langle C2, B_2 \rangle$ becomes its superclass. The solid contour and the gray area in Figure 4.2 determine the view of `C1` when loaded by A_1 and A_2, respectively. Moreover, the table in this figure includes the types that belong to

each class loader's namespace. It shows that classes loaded by class loaders from distinct hierarchies are invisible to each other.

As can also be seen from the example in Figure 4.2, the delegation relationship between class loaders plays a central role in specifying the scope of namespaces and, consequently, the visibility of classes. In our approach, to provide processes with distinct namespaces, we associate each process with class loaders that do not have any delegation relationship with class loaders belonging to other processes. The next section explains our class loading model.

4.1.2 The JPF Class-Loading Model

Our model ensures that processes do not share any types, which is essential for supporting distributed applications [91]. Moreover, this model addresses some other limitations of JPF. Prior to this work, JPF included a basic class loading mechanism which did not follow the delegation pattern of a standard JVM. This model was embedded in the JPF infrastructure without identifying entities that represent class loaders. Therefore, it was not able to support custom class loaders. Moreover, the older version of JPF used a basic type system which identified classes only by their names. As a consequence, it only included a single namespace containing every class in the SUT.

Our model is based on the class-loading model of Java. We divide the develop-

ment of our model into two steps. In the first step, we implement a model in JPF which behaves similar to the Java class-loading model. Our model defines entities that represent class loaders within JPF and links them to the SUT class loaders. It supports custom class loaders, follows the Java delegation pattern, provides per-class loader namespaces, and follows a dynamic linking mechanism similar to that of Java. Finally, it improves the JPF type system by also accounting for class loaders to identify types.

The second step towards implementing our class-loading model includes a modification to allow for partitioning types between processes. In particular, our goal is to map processes to sets of namespaces that do not overlap. To accomplish that, our model, contrary to a standard JVM, includes multiple hierarchies of system-defined class loaders, where each hierarchy maps to a process. These hierarchies are distinct, that is, there is no delegation relationship between hierarchies associated with different processes. Moreover, our implementation ensures that each process has exclusive access to its own hierarchy. That guarantees type separation in the presence of custom class loaders as well, since they cannot access standard hierarchies associated with other processes. Finally, having distinct hierarchies, dynamic linking also maintains separation of types, and resolved types that belong to different processes cannot interfere with one another.

The example in Figure 4.3 shows how our model partitions types for a dis-

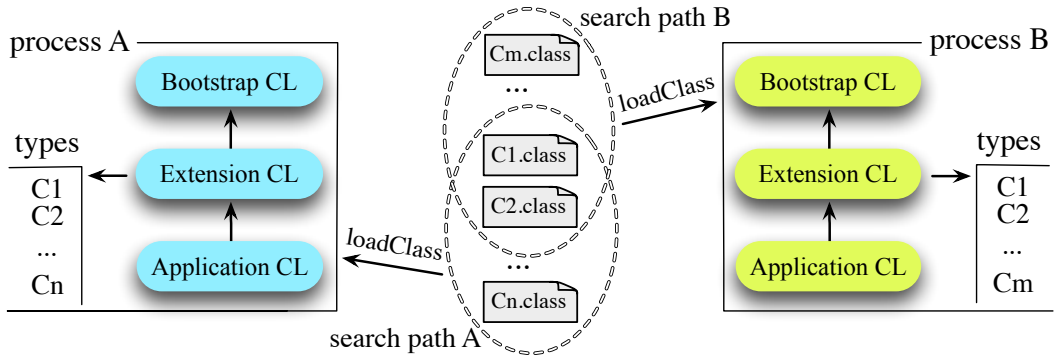

Figure 4.3: The JPF class-loading model separates type spaces between processes by allowing for multiple hierarchies of system-defined class loaders

tributed SUT composed of two processes. Each process has an exclusive hierarchy of system-defined class loaders which is used to define only those classes that belong to that process. Processes can even access the same class files to define types, for example `C1.class`, but the model ensures that different `Class` objects (associated with the same class file) are used in different processes.

Class Loaders Implementation

Our model encapsulates the three system-defined class loaders in a single instance of the class `SystemClassLoaderInfo` which is in the package `gov.nasa.jpf.vm`. Creating `SystemClassLoaderInfo` instances is part of the initialization of JPF's JVM which also ensures that each instance of `SystemClassLoaderInfo` loads only classes that belong to its corresponding process. The bootstrap class loader can-

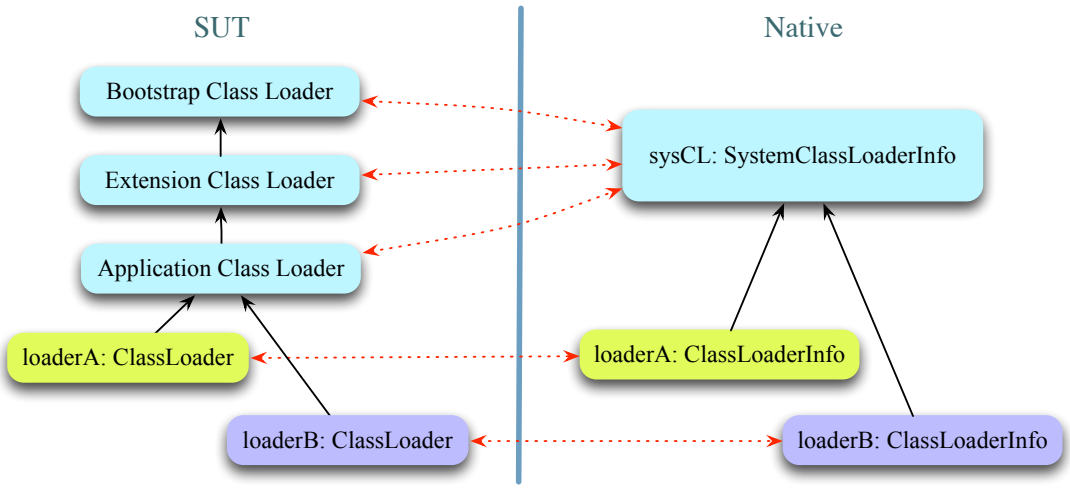

Figure 4.4: For every class loader created in the SUT, there exists a native representation within JPF

not be accessed by the Java application, and since the field of `ClassLoader` that represents the parent class loader is declared as `final`, the parents of extension and application class loaders cannot be changed. However, by combining these class loaders, we cannot support custom class loaders that have the extension class loader as their parents. But we still chose to combine them to considerably reduce the complexity of their implementation and simplify the configuration of their search paths by the user.

Our work models the class `ClassLoader` by including its model class and the corresponding native peer. We also provide support for some of the subclasses of `ClassLoader` in the standard Java library, such as `java.net.URLClassLoader` and

java.security.SecureClassLoader, within JPF. The core functionality of these classes is hardcoded into the internal structure of JPF to optimize the class loading process.

As JPF executes the SUT, for every ClassLoader instance created in the SUT, the model creates an instance of gov.nasa.jpf.vm.ClassLoaderInfo in JPF which is a superclass of SystemClassLoaderInfo. ClassLoaderInfo instances maintain the same hierarchy as their corresponding ClassLoader objects. This can be seen in Figure 4.4. Basically, ClassLoaderInfo instances are native representations of Java class loaders. Invocation of almost every method of ClassLoader within the SUT leads to an invocation of a corresponding method in ClassLoaderInfo. That is accomplished through the MJI and our native peers.

Each ClassLoaderInfo instance keeps its loaded classes in an array of type gov.nasa.jpf.vm.ClassInfo. This array represents the class loader's namespace. To account for class loaders when identifying types, each ClassInfo instance stores a reference to its defining ClassLoaderInfo instance. Every ClassLoaderInfo object has an instance of gov.nasa.jpf.vm.Statics which is used to store the values of the static attributes of classes defined by the class loader. Moreover, each ClassLoaderInfo object has an instance of the class gov.nasa.jpf.vm.ClassPath which encapsulates its search path. Since our implementation provides each SystemClassLoaderInfo with a separate search path, unlike previous work on cen-

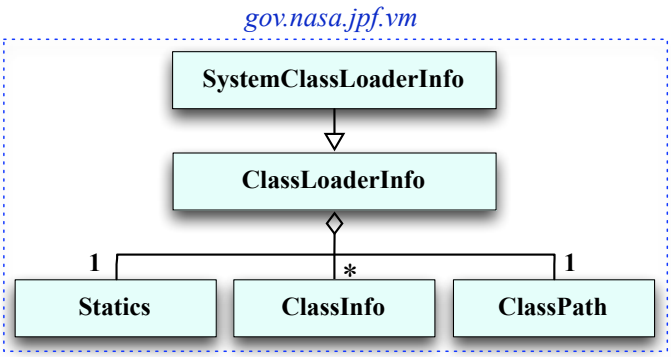

Figure 4.5: Class diagram of JPF class loading model

tralization at the SUT level, it supports distributed SUTs composed of processes written in different versions of Java. The relationships between the classes mentioned above can be seen in Figure 4.5.

Resolution Implementation

As mentioned earlier, our model follows a dynamic linking mechanism similar to that of Java, and resolves classes at runtime accordingly. Resolution takes place in the method `defineClass` which defines a class by accessing its class file. In our model, similar to a standard JVM, defining a class is performed by the final method `defineClass`. This method is declared as native in the `ClassLoader` model class, and it is implemented by a native peer method. Therefore, it executes on the host JVM.

Consider the following example. Let `L` be the defining class loader of `A`.

```
class A extends B {
  ...
}
```

To define `A`, the call `L.defineClass(A)` executes. Since `defineClass` is native, the execution of this call is transferred to the host JVM level. Defining the class `A` requires resolving its superclass `B`. Therefore, before `defineClass` returns, it has to execute the call `L.loadClass(B)`. But at this point the execution is at the host JVM level, whereas `loadClass` has to execute at the JPF level. The `loadClass` method can be overridden in custom class loaders and can contain arbitrary code which may, for example, result in creating `ChoiceGenerators`. Therefore, to capture the SUT behavior, it is essential to execute `loadClass` at the JPF level.

This implies that we need to pause the execution of `L.defineClass(A)` at the JVM level, and invoke the call `L.loadClass(B)` at the JPF level. Once `L.loadClass(B)` returns, the execution needs to transfer back to the JVM level to complete the execution of `L.defineClass(A)`. In other words, to resolve a type, the execution of `defineClass` requires to take the roundtrip $hostJVM \rightarrow JPF \rightarrow hostJVM$. Before describing how we handle such a roundtrip, we explain how a method invocation is handled in JPF. In JPF, similar to a standard JVM, each thread has a stack. When a thread invokes a method, a new frame is pushed onto the stack, and when the method returns, the frame is popped from the stack.

Figure 4.6 shows how the type `B` is resolved by JPF. To resolve a type, the na-

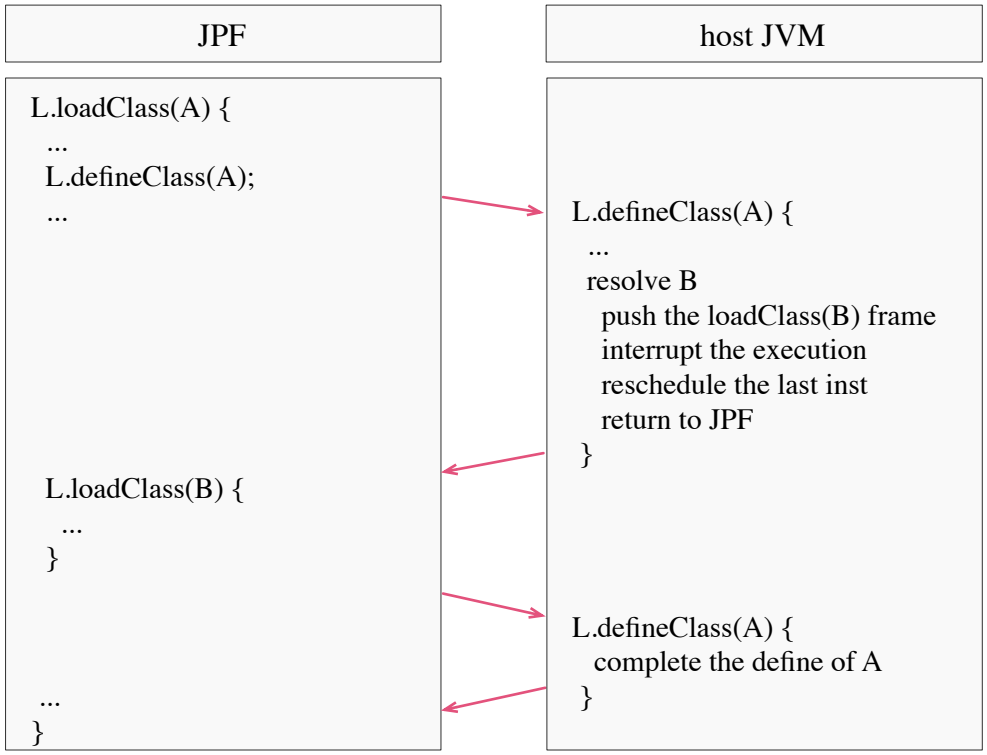

Figure 4.6: The figure shows how the resolution of the superclass `B` of `A` is performed in JPF

tive peer creates a new stack frame corresponding to `L.loadClass(B)` and pushes the frame onto the stack of the current thread running on JPF. Then it interrupts the normal flow of the execution at the host JVM level: this is done by simply throwing an exception of type `gov.nasa.jpf.jvm.ResolveRequired`. This exception is handled within the native peer implementation of `defineClass`. To handle the exception, the bytecode instruction that has invoked the current `defineClass` method is flagged to be re-executed (which is accomplished by invoking the method `MJIEnv.repeatInvocation()`). Note that at this point the class `A` has not been

defined yet, and thus re-execution of `L.defineClass(A)` is essential to complete its definition.

After handling the exception, the native peer method returns, and the execution transfers back to the JPF level. At this point, since `L.loadClass(B)` is at top of the current thread stack, it executes next at the JPF level, and once its execution is completed, the bytecode that triggered the call `L.defineClass(A)` is re-executed by JPF. This procedure repeats for any other types in `A` that need to be resolved.

To avoid the overhead from switching execution between JPF and the host JVM, the roundtrip is only applied if there is a customized `loadClass` implementation involved. Otherwise, the class is resolved internally within the host JVM. In particular, the roundtrip is not required for our `SystemClassLoaderInfo` which is an essential runtime optimization.

4.2 Representing Processes

Next we consider how to represent multiple processes in a Java model checker with a JVM running as a single process. In our approach, processes are modeled as groups of threads with separate sets of namespaces which are provided through our class-loading model, and a specialized JVM (explained in Section 4.3) is implemented to handle groups of threads associated with different processes.

The previous implementation of JPF, similar to a standard JVM, has exactly

one *main thread*, which initiates the run of every Java application. Our approach allows for multiple main threads within JPF. Each main thread maps to a process and initiates the execution on the process from its corresponding main method.

One natural way to map a process to a group of threads is using the `java.lang.ThreadGroup` instances, which is a common way to represent a set of threads in Java. However, this approach requires keeping `ThreadGroup` objects at the SUT level, which may disrupt the expected functionality of these objects. Our aim is to hide the thread-to-process association from the SUT. To accomplish that, we develop the class `gov.nasa.jpf.vm.ApplicationContext`. This class encapsulates per-process information such as initial class, command line arguments, a `SystemClassLoaderInfo` instance, and classpath.

For every process, there exists exactly one instance of `ApplicationContext` which exists at the JPF level and cannot be accessed by the SUT. In the JPF infrastructure, threads are represented by instances of `gov.nasa.jpf.vm.ThreadInfo` which declare an instance field of type `ApplicationContext`. To provides the thread-to-process association, every `ThreadInfo` object stores the reference to the `ApplicationContext` object of its corresponding process.

Upon system initialization, JPF creates a new `ApplicationContext` instance for each process and stores it within the `ThreadInfo` object that represents the main thread of the process. Each new thread that is created by the SUT automat-

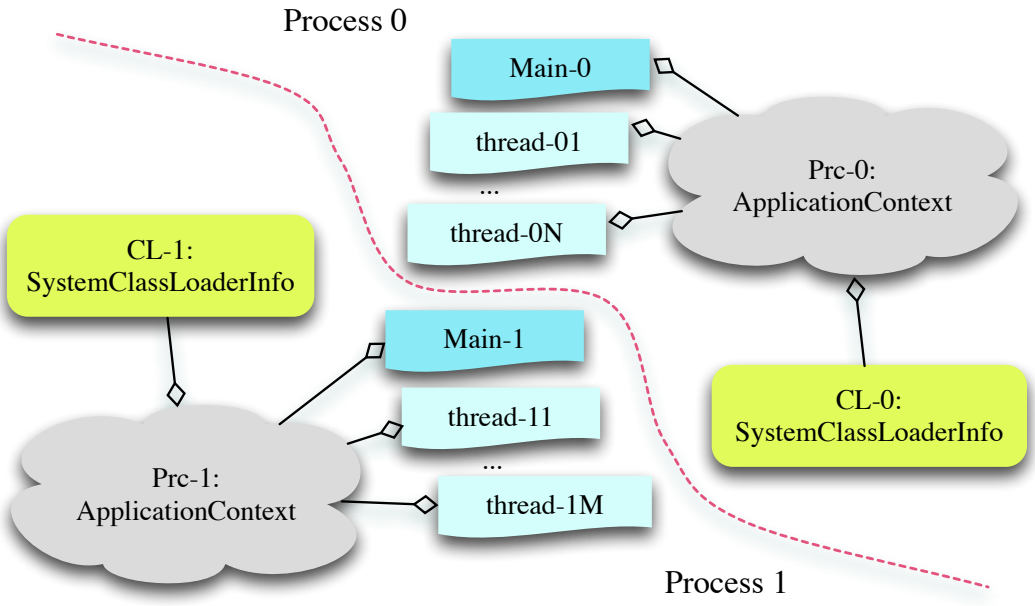

Figure 4.7: The distributed SUT composed of two processes is converted to two groups of threads

ically inherits the `ApplicationContext` of the currently executing thread. This mechanism ensures the thread-to-process association during the entire execution.

To summarize, our approach relies on three key elements: a `SystemClassLoaderInfo` object, a main thread, and an `ApplicationContext` object. That can be seen in Figure 4.7, which also shows how a distributed SUT is captured as groups of threads within JPF. Creating and initializing the elements used to encapsulate processes, and maintaining their states as JPF executes the SUT are performed by our multiprocess JVM, as explained next.

4.3 Implementing Multiprocess VM

The previous version of JPF has a JVM which can only execute single process applications and is not able to handle distributed SUTs. As part of this work, we develop a new JVM within JPF which is multiprocess aware. This JVM identifies parts of the SUT associated with each process and maintains their states as it executes the distributed system. The way that the multiprocess JVM interprets states is also different from the way adopted by the single process JVM. One example is how JPF recognizes that a state is deadlocked. For distributed SUTs, the JVM identifies two different deadlock scenarios, local deadlocks which prevent a single process from progressing and global deadlocks which prevent the whole distributed system from progressing.

The multiprocess VM can also compute non-deterministic choices to capture different interleavings of concurrently running processes. In particular, it uses a new scheduler factory to encapsulate choices. Moreover, the previous version of JPF does not support finalizers which are required to address shutdown semantics of Java processes. This work provides such support in both the single process and multiprocess JVM. The next two sections explain the new scheduler factory of the multiprocess VM and the new model for supporting finalizers.

Since the multiprocess JVM and the single process JVM have some similar

behavior, we use inheritance to capture their similarities. The UML diagram in Figure 4.8 shows the design adopted by our approach. The single process and multiprocess JVMs are encapsulated by instances of `gov.nasa.jpf.vm.SingleProcessVM` and `gov.nasa.jpf.vm.MultiProcessVM`, respectively. Both classes extend the abstract class `VM` and override those methods that behave differently. One of those methods is `initialize`, which is used to set up the initial execution environment. As mentioned earlier, as part of its initialization, the multiprocess JVM associates the process to an instance of `SystemClassLoaderInfo`, a main thread, and an `ApplicationContext` object. Using these objects, at any state, the multiprocess JVM can determine the process associated with each entity in the system.

4.3.1 Distributed Scheduler Factory

As mentioned earlier, to capture scheduling choices in JPF, the type `ThreadChoiceGenerator` is used. The JVM delegates the creation of all `ThreadChoiceGenerator` objects to the scheduler factory. To handle a distributed SUT, JPF requires new scheduling policies. We have developed a new scheduler factory that implements the scheduling policies for distributed SUTs. Our scheduler factory is used by the multiprocess JVM and is encapsulated by the class `gov.nasa.jpf.vm.DistributedSchedulerFactory`, as a subclass of `SchedulerFactory`.

In general, for distributed systems, we can identify two different types of schedul-

ing points: process *local choices* and system *global choices*. Process local choices capture interleavings among intra-process threads, whereas, system global choices capture interleavings among processes. Both local and global scheduling points are captured by instances of `ThreadChoiceGenerator`.

The distributed scheduler factory assumes that there is no communication between processes. Later in Chapter 5 we show how the scheduler is extended to account for communication between processes. There are exactly two places at which the distributed scheduler factory uses global choices to capture thread non-determinism. One place is the initial state from which the execution of the distributed SUT starts. The initial state encapsulates a scheduling choice where there is exactly one transition associated with the main thread of each process. Execution of each transition starts with the first instruction of the process main thread which is unique. Another place that is subject to global choices is processes' termination points. Upon process termination, a global scheduling point is created that includes a choice associated with each remaining process in the SUT.

Any other place that is subject to thread non-determinism is treated as a local choice. Since processes have exclusive access to runtime resources, they cannot interfere with one another during the execution, unless a mechanism is used for them to communicate.

4.3.2 Addressing Shutdown Semantics

When it comes to distributed applications, finalizers and shutdown hooks play an important role, since they are used to clean up communication channels. For example, `java.net.Socket` objects which represent end points of connections are finalizable. Upon the garbage collection of a `Socket` object, its finalizer invokes `Socket.close()` to clean up the corresponding connection. As is explained in the next chapter, closing `Sockets` can affect the behavior of the process at the other end of the connection. Therefore, to capture all possible interactions of communicating processes, it is essential to support finalizers. However, JPF does not provide support for Java finalizers. As part of this work, we have provided such support in both single process and multiprocess VMs of JPF.

To execute finalizers on JPF, we create a special thread, called *finalizer thread*, for each process. This thread is created during the initialization of JPF's JVM and is kept alive during the entire execution of the process. To avoid increasing the state space, as explained later, our approach does not take the finalizer thread into account when computing choices, unless a finalizable object is being garbage collected.

Every finalizer thread is represented by two objects. One thread object exists on the JPF level, and it is of type `gov.nasa.jpf.FinalizerThread` which is a subclass

of `Thread`. This thread is used to execute finalizers, which can include arbitrary code, on the JPF level. The other object exists on the host JVM level, and it is of type `gov.nasa.jpf.vm.FinalizerThreadInfo` which is a subclass of `ThreadInfo`. The `FinalizerThreadInfo` object of each process is kept by its corresponding `ApplicationContext` object. A native peer is implemented as an interface between these two classes that represent finalizer threads.

Our model to support finalizers, shown in Figure 4.9, can be divided into three parts. In the first part, the JPF garbage collection is extended to identify those finalizable objects that are no longer referenced from the SUT and store them in queues associated with finalizer threads. These queues exist on the JPF level, and they are encapsulated by the field `finalizeQueue` declared in the class `Finalizer-Thread`. The second part involves providing the JVM with a functionality to schedule the finalizer threads. Finally, the last part focuses on the way that a finalizer thread processes its `finalizeQueue`. In the remainder of this section, we explain these three parts in more detail.

As explained in Section 2.5.1, garbage collection of JPF is based on the mark and sweep algorithm. The marking phase marks all objects that are being referenced from the SUT, and the sweeping phase removes all objects that have not been marked. We extend the garbage collection mechanism of JPF with an additional phase between mark and sweep. This phase marks any finalizable object that

has not been marked and adds it to the corresponding `finalizeQueue`. Marking finalizable objects prevents them from being removed from the heap in the sweep phase. Moreover, keeping the finalizable objects in a queue, `finalizeQueue`, which exists on the JPF level prevents them from being garbage collected until they are processed and removed from the queue.

Basically, the executions of finalizer threads are closely controlled by JPF's JVM. The JVM allows a finalizer thread to run only if there is a finalizable object to process. Otherwise, the JVM sets the status of the finalizer thread to `WAITING` to prevent it from contributing to the state space. Moreover, there is no point in the SUT that creates and starts the finalizer threads. Creating these threads is part of JPF's JVM initialization. When created, to start each finalizer, a stack frame corresponding to `Thread.run()` is pushed onto the stack of every finalizer thread. To reduce the state space, initially, the status of finalizer threads are set to `WAITING` to prevent them from being included when computing scheduling points. After each garbage collection, the JVM checks if there is any object stored in `finalizeQueue` of the finalizer thread of the process executed last. If so, the JVM schedules this thread to execute next. To accomplish that, the scheduler factory is extended to create a thread choice generator from which only the finalizer thread can proceed. At this point, the status of the finalizer thread is set to `RUNNABLE` to let the JVM execute its `run()` method.

The `FinalizerThread.run()` method iterates over `finalizeQueue` and invokes `finalize()` on every object. Once the entire queue has been processed, the finalizer thread clears the queue, and the scheduler factory creates a global scheduling point. At this point if there is no other thread alive in the process, the finalizer thread is terminated. Otherwise, its status is set to `WAITING` until it is scheduled again by the JVM. Setting the status to `WAITING` avoids the finalizer thread from iterating through the `while` loop (See Figure 4.9).

Note that in the presence of finalizer threads, the end states of processes are interpreted differently. Without finalizer threads, a state is interpreted as an end state, if there are no alive threads in the process. But, in the presence of finalizer threads, a state is considered an end state if there is either no alive thread or the only alive thread is the finalizer thread and it is not runnable.

Since supporting finalizers requires adding a phase to the JPF garbage collection, which is performed very frequently, it can increase the model checking time. Therefore, a property is provided in the JPF configuration file to activate the support for finalizers. By default, finalizers are not processed. To handled them one needs to set `vm.process_finalizers` to `true`.

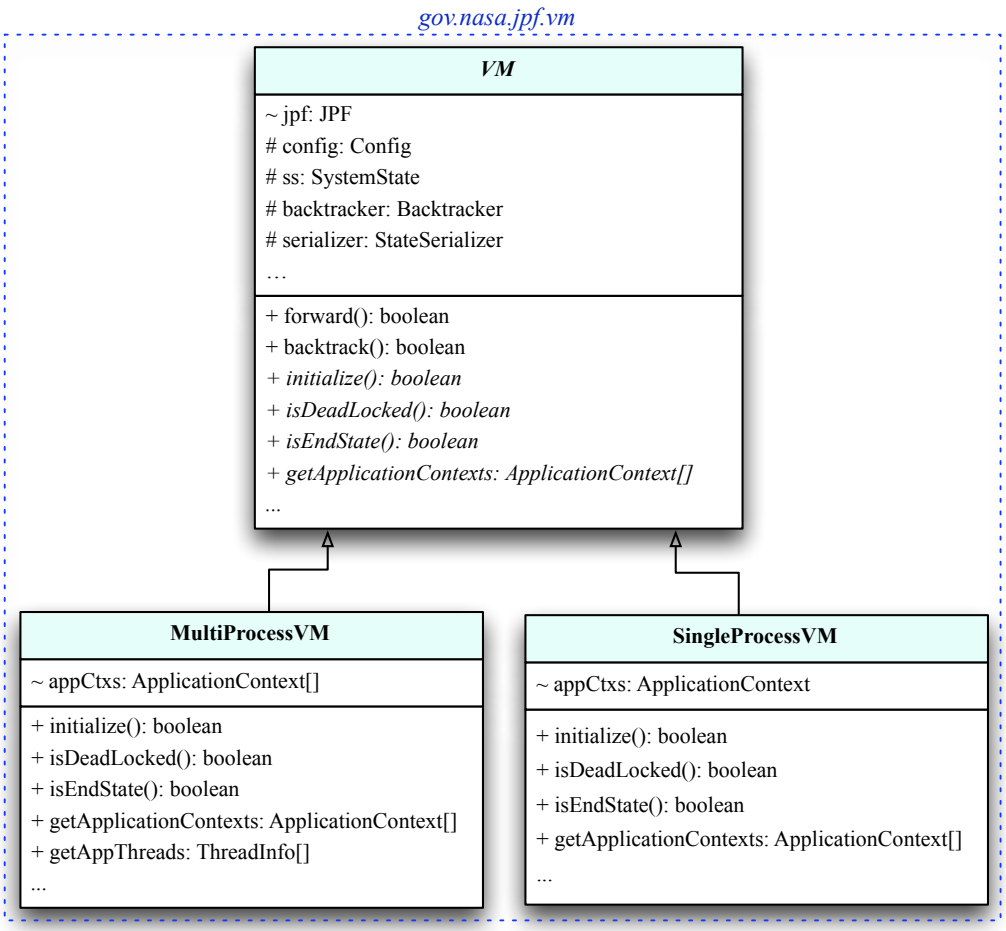

Figure 4.8: The UML diagram including classes that encapsulate the JVMs of JPF

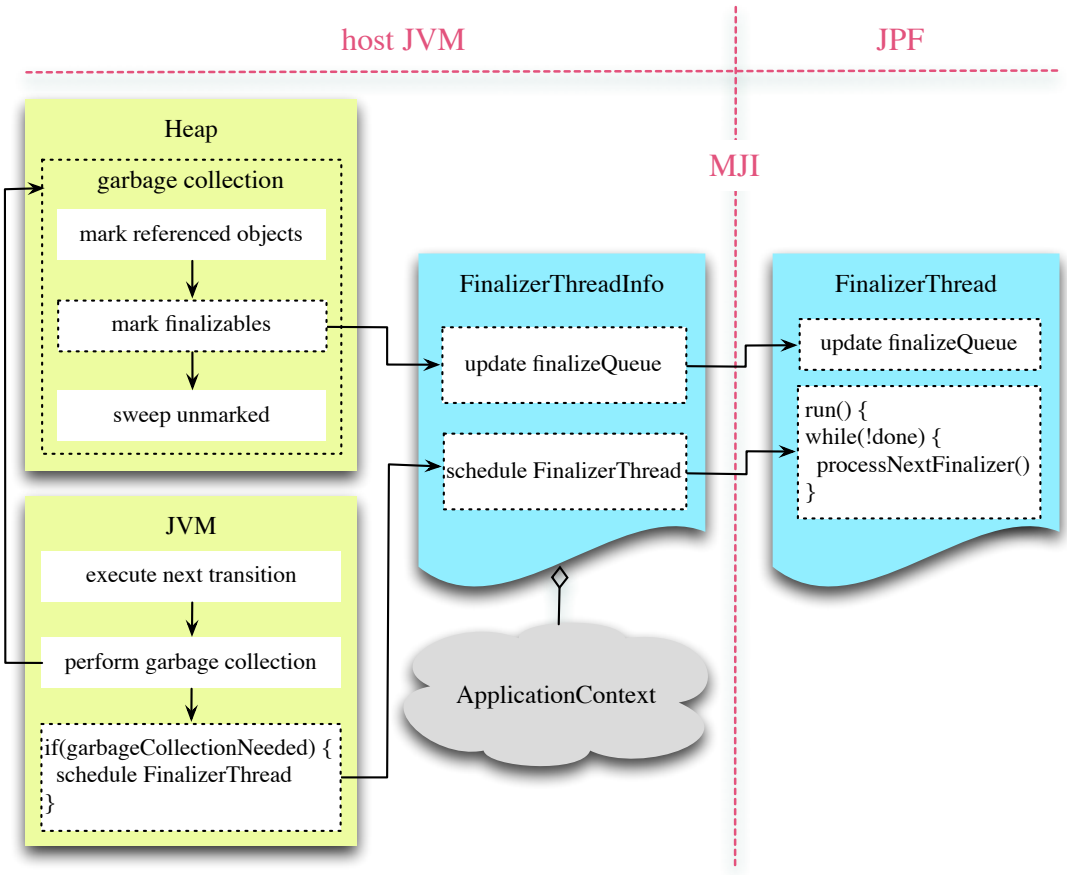

Figure 4.9: Different components of the model that provides support for Java finalizers

5 Modeling Interprocess Communication

In the previous chapter, we explained how our centralization technique provides the basic building blocks in JPF to capture distributed applications. However, to model check such applications, models of communication between processes are required. This chapter explains how communication channels are modeled in our approach. It also presents how we capture different scheduling of threads at process communication points to explore the state space of distributed SUTs. Moreover, it presents our POR technique and its correctness proof with respect to certain properties. First, we explain, using a simple distributed application, how Java processes communicate via TCP sockets. Then, in Section 5.2, we elaborate on the components required to model communication between processes.

5.1 Java Networking

As mentioned earlier, one of the features that makes Java a suitable platform for developing distributed applications is multilevel support for network communica-

tion. The most commonly used networking API for process communication in Java is based on sockets which are also the most primitive networking construct [6]. A socket represents an endpoint of a communication channel between two processes. Every socket is identified by two elements, an *IP address* and a *port number*. An IP address is a unique address to identify a host across a network which is based on the Internet protocol (IP). A port number identifies the communication endpoint within a computer and is usually associated with a specific service or protocol.

One of the main types of sockets in an IP network is based on the transport control protocol (TCP). TCP provides a reliable communication channel that guarantees the successful delivery of data packets in order. Java provides support for TCP sockets through classes in the `java.net` package, such as `java.net.Socket`, `java.net.InetAddress`, and `java.net.ServerSocket`. Data is exchanged through TCP sockets via a pair of input and output streams.

The code in Figure 5.1 demonstrates a simple example in which a server process and a client process communicate using TCP sockets. Both processes are single-threaded. Henceforward, we use $s.i$ and $c.i$ to denote the i^{th} statement of the server code, and the client code, respectively.

A TCP socket in Java can be either passive or active. To establish a connection, the server creates a passive socket (encapsulated by `java.net.ServerSocket`) that is associated with a given port ($s.6$), and then it blocks ($s.7$) until it receives a

```java
import java.io.*;
import java.net.*;
public class Server {
  public static final int PORT = 1024;
  public static void main(String[] args) throws IOException {
    ServerSocket ssocket = new ServerSocket(PORT);
    Socket socket = ssocket.accept();
    InputStream  in = socket.getInputStream();
    byte[] buf = new byte[10];
    in.read(buf);
    socket.close();
    ssocket.close();
  }
}
```

```java
import java.io.*;
import java.net.*;
public class Client {
  public static void main(String[] args) throws IOException {
    final String HOST = "indigo.cse.yorku.ca";
    Socket socket = new Socket(HOST, Server.PORT);
    OutputStream out = socket.getOutputStream();
    String message = "Hello";
    out.write(message.getBytes());
    socket.close();
  }
}
```

Figure 5.1: A simple client and server that communicate through TCP sockets

connection request from a client.

The client creates an active socket (encapsulated by `java.net.Socket`) which sends a connection request to a server that is running on the host `indigo.cse.yorku.ca` and is waiting for connections on the specified port ($c.6$). If there is no server socket associated with the given host and port, Java throws a `java.io.IOException` that has to be handled in the client code. Otherwise, the server accepts the connection request from the client and obtains a new active socket representing the server endpoint of the connection ($s.7$). At this point the connection is established, and the server and the client can start to exchange data.

The server and the client receive and send data through their respective socket streams, which are obtained by calling the `Socket.getInputStream()` and `Socket.OutputStream()` methods ($s.8$ & $c.7$). Next, the server attempts to read a request from the client ($s.10$). If data have not been sent yet, the server input operation blocks until the client writes some data ($c.9$). Once the server is done with the input operation, it terminates by closing its sockets ($s.11$ & $s.12$). Closing the `ServerSocket` object prevents new clients from connecting to this server, and closing the `Socket` object disconnects this server from the client. After the client completes the output operation ($c.9$), it also terminates by closing its endpoint of the connection ($c.10$). Closing the `Socket` object disconnects this client from the sever, and consequently, no more data can be exchanged between the two.

5.2 Model of IPC

At this stage, our model supports communication via TCP sockets [92]. The communication paradigm supported by TCP sockets is based on blocking operations on unbounded buffers. As part of our future work, we intent to support other communication paradigms, such as non-blocking operations on unbounded buffers. Our IPC model consists of two main components: *connection manager* and *scheduler*. The former models the communication channels (see Section 5.2.1) and the latter captures different scheduling of processes at synchronization points at which they communicate (see Section 5.2.2). Our model is implemented as an extension of JPF which is called *jpf-nas* (which stands for networked asynchronous systems).

5.2.1 Connection Manager

Consider the server/client example in Figure 5.1. The two processes use a communication channel to exchange data which is established by two active sockets. The communication channel is implemented natively, i.e., it is kept out of the scope of the JVM runtime environment and is handled by the underlying OS. Therefore, the content of the channel is completely invisible from the Java processes, and they can only access its entry points through Java `Socket` objects to send and receive data.

Modeling a similar mechanism in JPF requires providing shared buffers accessed

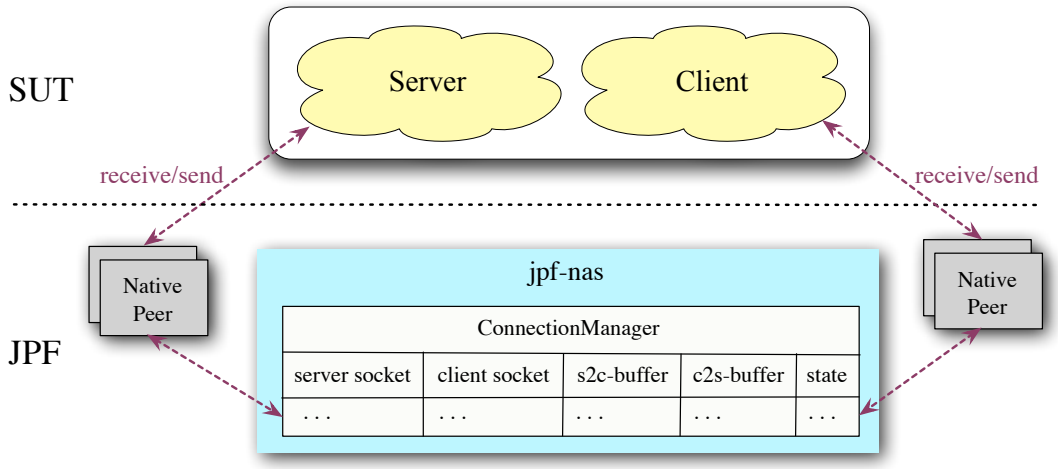

Figure 5.2: The list of connections is kept natively at the same level as JPF

by different processes. Such buffers cannot exist at the SUT level, since the type separation enforced by our centralization technique does not allow different processes of the distributed SUT to share Java types and therefore, cannot share objects. If, in JPF, a process accesses a type defined by a class loader that belongs to some other process, an instance of the exception `java.lang.ClassNotFoundException` is thrown by the process. However, using the MJI feature of JPF (see Section 2.5.5) allows for shared objects at the host JVM level.

To provide the communication channels, the connection manager component of jpf-nas creates and maintains shared buffers including communication data (see Figure 5.2). This component exists at the host JVM level (i.e., it is invisible from the JVM of JPF) and it is accessed by native peers corresponding to `java.net` classes.

The connection manager is encapsulated by the class `nas.java.net.connection.ConnectionManager`. For every communication channel, it includes an instance of the class `nas.java.net.connection.ConnectionManager.Connection`. Henceforth, these instances are referred to as *connections*. Each connection is shared by exactly two processes. The state of a connection is updated by the connection manager each time that a SUT process accesses a socket or its I/O stream. Each connection captures the following information:

- **Endpoints** - each connection is identified by two endpoints, representing a server socket and a client socket. To capture endpoints, each connection stores two integers representing the two corresponding SUT socket objects in JPF and the respective `ApplicationContext` instances in which they were created (to identify processes).

- **State** - we define three possible states for connections, *PENDING* (the connection has not been established yet, and one endpoint is waiting for the other end to connect), *ESTABLISHED* (the connection has been established and is ready for I/O operations), *CLOSED* (at least one socket has been closed, and no more data can be transmitted).

- **Buffers** - to capture the communication data, the connection has two buffers, one buffer is used to store data sent from the client socket to the server socket,

and the other one is used to store data sent from the server socket to the client socket. The buffers store only the data that has been sent by one endpoint, but has not been received by the other end yet. The buffers are instances of `gov.nasa.jpf.util.ArrayByteQueue` which is a cyclic queue to store raw bytes.

As JPF executes the distributed SUT, the connection manager needs to maintain a list of connections that were created along the current execution path. However, since connections exist at the host JVM level, their states are not part of the SUT and they are not handled by JPF. Therefore, JPF does not take into account the state of the communication channels when matching states. Moreover, as JPF backtracks to previously visited states, it does not restore the state of the connections, and their states may not be in sync with the SUT state anymore.

To address the former issue, we implement a hash function within the `Connection` class. This function maps the data that defines the state of a connection to an integer. This integer is stored by an instance field, `hash`, which is declared within the `Socket` model class. In other words, every `Socket` object within the SUT keeps the hash value representing the state of its corresponding communication channel. The connection manager updates the `hash` field upon any changes to the corresponding connection. Using this mechanism, the changes to connections are reflected to the SUT and thus are considered by JPF when matching states. In

the case of a hash collision, different states of a connection object are associated with the same hash value. That can make the model checker treat two different states of the SUT the same, and as a consequence, it can miss certain execution paths. Therefore, hash collisions can affect the completeness of the verification.

To address the latter issue, a mechanism is required to keep the state of the list of connections in synchronization with the state of the SUT. Without such a mechanism, after backtracking or restoring a previously seen state, the list of connections may not reflect the right state. To provide such a mechanism, jpf-nas uses a JPF listener, `gov.nasa.jpf.util.StateExtensionListener`, which captures snapshots of the connection list at every state of the SUT. Basically, this listener keeps a map from state ids to lists of connections. It operates upon three different events, `stateAdvanced` (the SUT execution reaches a new state), `stateBacktracked` (the SUT execution moves back to the previous state) and `stateRestored` (the SUT moves to a previously seen state). Each time JPF has reached a new state it calls the `stateAdvanced` method of the listener, which then creates a deep copy of the current list of connections, stores this snapshot in the map kept by the listener, and associates it to the current state id. This way, a list of connections is associated with every state of the SUT. Upon the events `stateBacktracked` and `stateRestored` at which the SUT moves back to a previously visited state, using the state id, the listener restores the associated connection list from the map to keep this list always

synchronized with the SUT state.

5.2.2 Scheduler

As mentioned in Section 2.5.3, JPF has a scheduler which is used to capture different schedulings of threads within a process. When JPF searches the state space of the SUT, it uses this scheduler to explore different orderings of those concurrent transitions where executing them in different orders may lead to different behaviors of the process. The scope of the JPF scheduler is an individual process, that is, it only deals which those operations which are local to a single process and do not involve interprocess communications. We refer to the JPF scheduler as the *local scheduler* from now on.

A communication between processes requires an access to a communication channel. Different orderings in which processes access communication channels may lead to different behaviors of the distributed system. Therefore, a mechanism is needed to capture different orderings of those concurrent transitions that involve interprocess communications. To address this issue we include a scheduler in our IPC model which is referred to as the *global scheduler*.

Our approach to search the state space of distributed systems involves using both the local scheduler and the global scheduler. Basically, we use the local scheduler to deal with intra-process operations, whereas the global scheduler is used to

deal with interprocess operations. Consider the state s of the distributed system where the transition that takes the system from a previous state to s belongs to a thread in the process p which is referred to as the *current process*. If all those transitions to be executed next by the threads of p from s involve intra-process operations, the local scheduler is used which only interleaves the threads that exist within the process p. Otherwise, the global scheduler is used which interleaves all the threads in the distributed system.

This approach leads to a selective search strategy (see Algorithm 1) which instead of exploring all the enabled transitions at each state, explores a subset of the enabled transitions. In Chapter 6, we provide the precise description of our algorithm to explore the state space of the distributed system. We also prove that our approach is sound with respect to deadlocks.

The jpf-nas scheduler is also used to inject exceptional control flows that correspond to network failures. As mentioned earlier, verifying distributed systems is challenging since failures can happen at so many different levels. Model checking the source code of Java processes does not verify the distributed system against all possible failures, for example, the ones that occur at the OS and hardware levels. Failures occurred at network layers appear into processes as exceptions of type `java.io.IOException`. To account for such network failures in our model, the scheduler component inserts transitions into the state space that represent the

corresponding exceptional control flow. Such functionality is essential to simulate network failures without SUT modifications. This functionality is enabled through the jpf-nas `properties` file, i.e., if the property `scheduler.failure_injection` is set to true, choices for failures are injected. By default, all possible exceptions are injected. However, one can specify the types of exceptions to be injected using the property `scheduler.injected_failures_types`.

Moreover, as part of this work, we provide a visualization of the search graph explored by JPF, as it model checked the distributed SUT. Our visualization tool is implemented in the form of a JPF listener (Section 2.5.4), `gov.nasa.jpf.listener.DistributedSimpleDot`, which extends an existing listener in jpf-core, `gov.nasa.jpf.listener.SimpleDot`. The listener `DistributedSimpleDot` generates a `dot` file that includes the SUT search graph. It distinguishes between local and global scheduling points. It uses circles to show local scheduling points and octagons to show global scheduling points. Figure 5.4 presents a search graph generated by our listener. We elaborate on that in Section 5.3.

5.2.3 Native Peers Implementation

This section focuses on implementation details of jpf-nas and explains how it models classes of the Java networking API. Consider the class `Socket`. To model this class, our extension includes the model class `java.net.Socket`. While running

the SUT, JPF uses this class as an alternative to the actual `Socket` class from the standard Java library (Section 2.5.6). The extension also includes a native peer, `JPF_java_net_Socket`, associated with the model class. That makes JPF establish a correspondence between the methods of the model class and the native peer (Section 2.5.5). For example, when JPF gets to an invocation of the method `Socket.connect` (in our model this method is invoked by the constructors of `Socket` and sends a connection request to a server), the execution transfers to the host JVM level and the method `connect__Ljava_lang_String_2I__V` in the native peer executes.

The flowchart presented in Figure 5.3 shows how the native peer method `connect__Ljava_lang_String_2I__V` is implemented. This example reveals how native peers in jpf-nas interact with the connection manager and the scheduler to model the Java networking API.

First the method checks if the socket to be connected is closed. If so, the method returns by creating a JPF exception object of type `java.net.SocketException` which is thrown by the SUT. Otherwise, it uses the connection manager and goes through the connection objects to check if there exists a server at the given host and port whose corresponding `ServerSocket` object has not been closed yet. If such a server does not exist, a JPF exception object of type `java.io.IOException` is created and thrown by the SUT, and the method returns. Otherwise, a connection

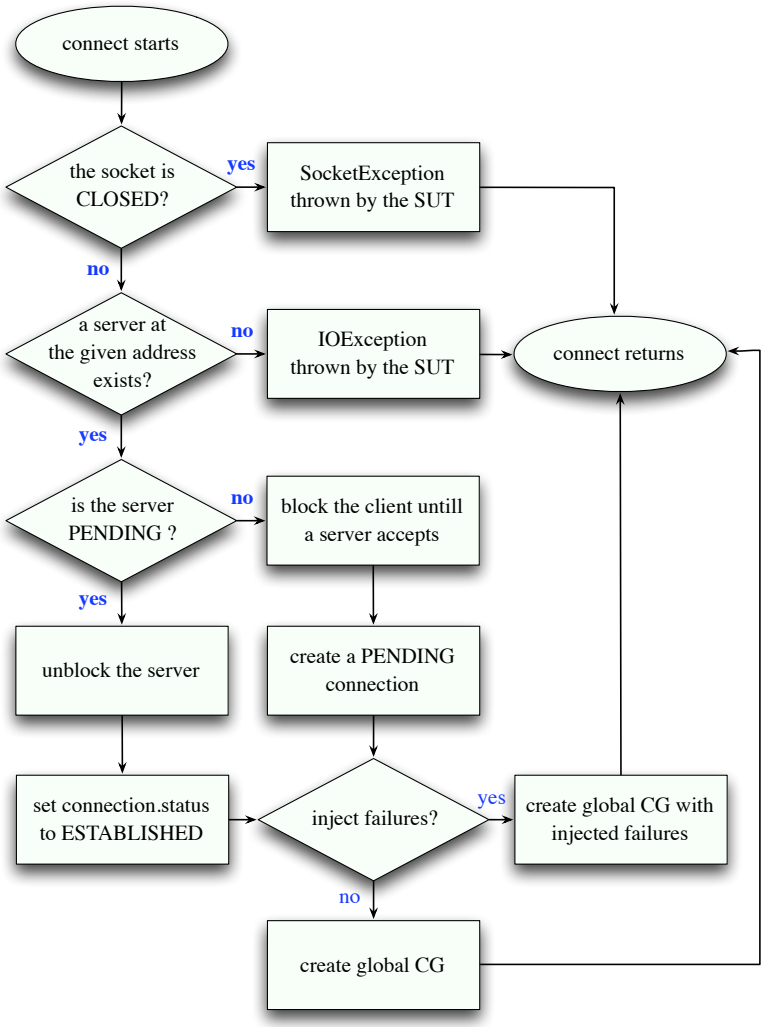

Figure 5.3: The flowchart illustrating the implementation of the method `Socket.connect`

object, c, is retrieved with an endpoint that represents the server.

Depending on the status of c, the method takes different actions. The connection with the PENDING status indicates that the server is blocking on an `accept()` operation to receive a connection request from a client. At this point, the blocked thread at the server gets unblocked. Moreover, the method updates c, i.e., it sets the status to ESTABLISHED and this socket becomes an endpoint of the connection. Next, the method uses the scheduler to create a global scheduling choice. Before creating a new choice generator it checks if `scheduler.failure_injection` is set to true. If so, the scheduler considers choices for possible failures as well. Otherwise, it creates a choice generator that only includes choices for runnable threads in the SUT. Finally, the method returns.

If the status of the retrieved connection object c is not PENDING, the client has to block until the server calls `accept()`. At this point, a new connection object is created and is added to the list maintained by the connection manager. One of the endpoints of the new connection is set to this socket, and its status becomes PENDING. Next, the scheduler is used to create a global scheduling choice. If `scheduler.failure_injection` is set to true, choices are included for possible failures as well. Otherwise, a choice generator is created which only has choices associated with runnable threads in the SUT. Then, the method returns.

5.3 Tool Demonstration

This section explains how jpf-nas verifies the client/server example in Figure 5.1. Applying this extension requires JPF to use the multiprocess JVM and the distributed scheduler factory (see Section 4.3 and 4.3.1). The default configuration of jpf-nas includes the following setting.

```
vm.scheduler_factory.class =
   gov.nasa.jpf.vm.DistributedSchedulerFactory
vm.class = gov.nasa.jpf.vm.MultiProcessVM
```

To apply jpf-nas on the client/server example, JPF is configured as follows.

```
@using = jpf-nas
target.0 = Server
target.1 = Client
listener+=,gov.nasa.jpf.listener.DistributedSimpleDot
```

The first line makes JPF use the jpf-nas extension. The next two lines specify the initial classes from which the execution of the server and client processes start. Finally, the last line makes JPF use the listener `DistributedSimpleDot` which generates the search graph from running jpf-nas on the example.

Figure 5.4 shows the entire graph. The total number of unique states visited by JPF is 16. Transitions with labels starting with *P0* are taken by the server process main thread and the ones with labels starting with *P1* are taken by the client process main thread. Moreover, the graph shows the transitions' last statement

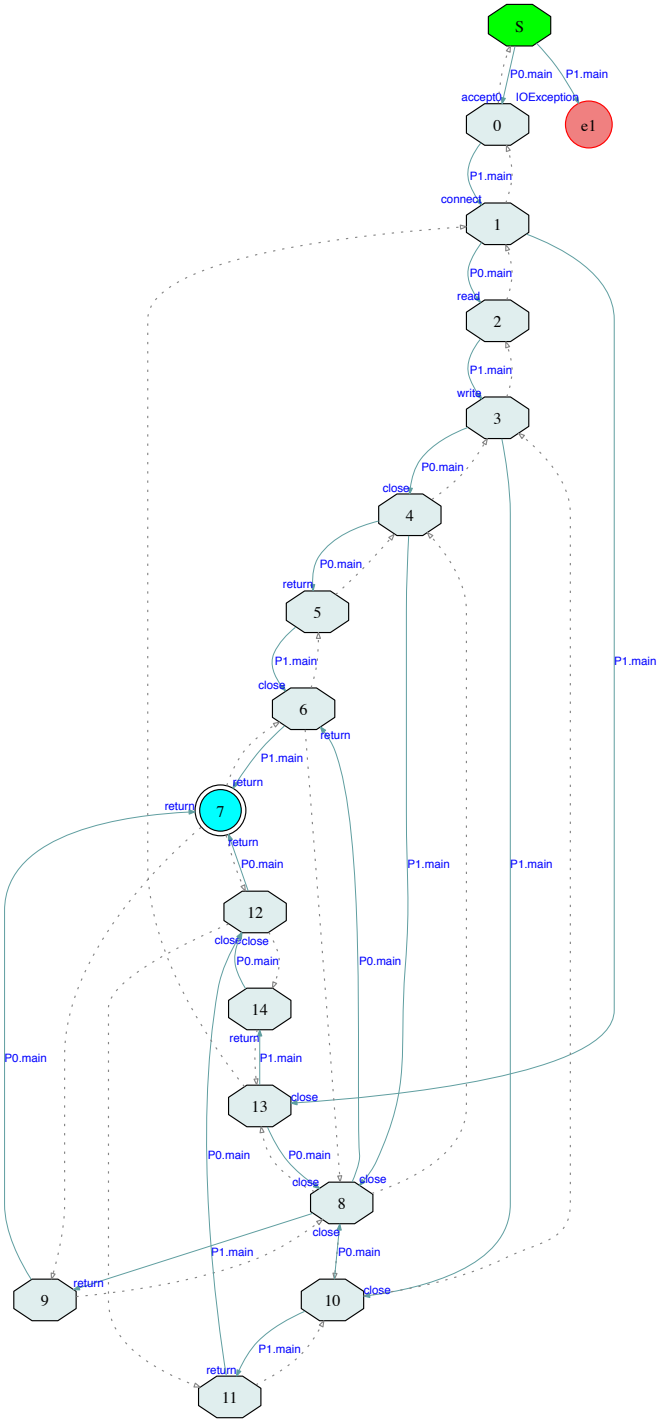

Figure 5.4: Search graph for example in Figure 5.1

that results in generating a new state. The dotted arrows are used when the model checker backtracks to a previous state. The state represented by a double circle is the end state at which all the processes of the SUT terminate. The state *e1* represents an error state at which the normal flow of the SUT's instruction is disrupted and its execution cannot proceed further. The states that capture global scheduling points are represented by octagons. `DistributedSimpleDot` depicts the states capturing local scheduling points by white circles. In this example, since both the server and the client processes are single threaded and do not include any data non-determinism, JPF does not create any local scheduling points. Next, using the search graph, we explain how jpf-nas uses the connection manager and the scheduler to capture the process interactions in JPF. Henceforth, the state with the id i in Figure 5.4 is referred to as S_i.

As was mentioned earlier, different orderings of statements involved in establishing a connection between a server and a client can have different outcomes. Figure 5.4 shows how our extension captures the different orderings of $s.7$ and $c.6$. In the start state, S, both the client and the server main threads are enabled. The execution path that starts with the client execution leads to an error state, e. In this path, since the client sends a connection request without the server waiting, an exception is thrown. The other possible execution starts with the server which blocks until it receives a connection request. To capture this execution, after the

server blocks, the state S_0 is created from which only the client can proceed. After the client unblocks the waiting server by executing $c.6$, jpf-nas creates the state S_1 from which both the client and the server main threads can proceed next. At this state, the connection is established, and a new connection object is added to the connection manager.

Figure 5.4 also presents one of the possible orderings of the I/O operations, $s.10$ and $c.9$, captured by jpf-nas. After the connection is established at S_1, the server proceeds and executes the read operation, $s.10$. Since the client has not written any data yet, the server blocks and a new state, S_2, is created from which only the client is enabled. Next, the client proceeds and unblocks the server by writing a request ($c.9$). At this point, the state S_3 is created from which both processes can continue.

Closing a socket from a process can affect the I/O operations performed by the process at the other end. To capture this effect, jpf-nas creates scheduling choices upon socket close operations. Figure 5.4 shows how different orderings of the write operation at $c.9$ and the socket close operation at $s.11$ are captured. As is shown in the figure, the server continues from state S_3 until it gets to the close operation. Before closing the corresponding connection in the connection manager, jpf-nas breaks the transition and creates state S_4 from which both the client and the server main threads are enabled. The server continues by closing the socket and then,

it terminates at $s.14$. At that point, since a process terminates, the distributed scheduler factory creates state S_5 from which the remaining threads (i.e., only the client main thread) can continue. At state S_5, the client continues and attempts to read from a closed channel. Another possible execution from state S_4 is that the client continues and reads the data before the connection is closed by the server.

To show the effect of the failure injection functionality, we run JPF on the example in Figure 5.1 using the following configuration.

```
@using = jpf-nas
target.0 = Server
target.1 = Client
listener+=,gov.nasa.jpf.listener.DistributedSimpleDot
scheduler.failure_injection = true
```

Setting `scheduler.failure_injection` to true activates the failure injection functionality of jpf-nas which allows for checking the code against possible network failures. Instead of exploring the search graph in Figure 5.4, jpf-nas explores the graph in Figure 5.5. Due to a possible `java.io.IOException` at $s.7$ which is not handled by the code, the choice generator at state S_0 includes a transition which is taken by the server and leads to the error state e_{10}. Similarly, since executing $c.6$ can throw an unhandled `java.io.IOException`, the choice generator at state S_1 includes a transition which is taken by the client and leads to the error state e_9.

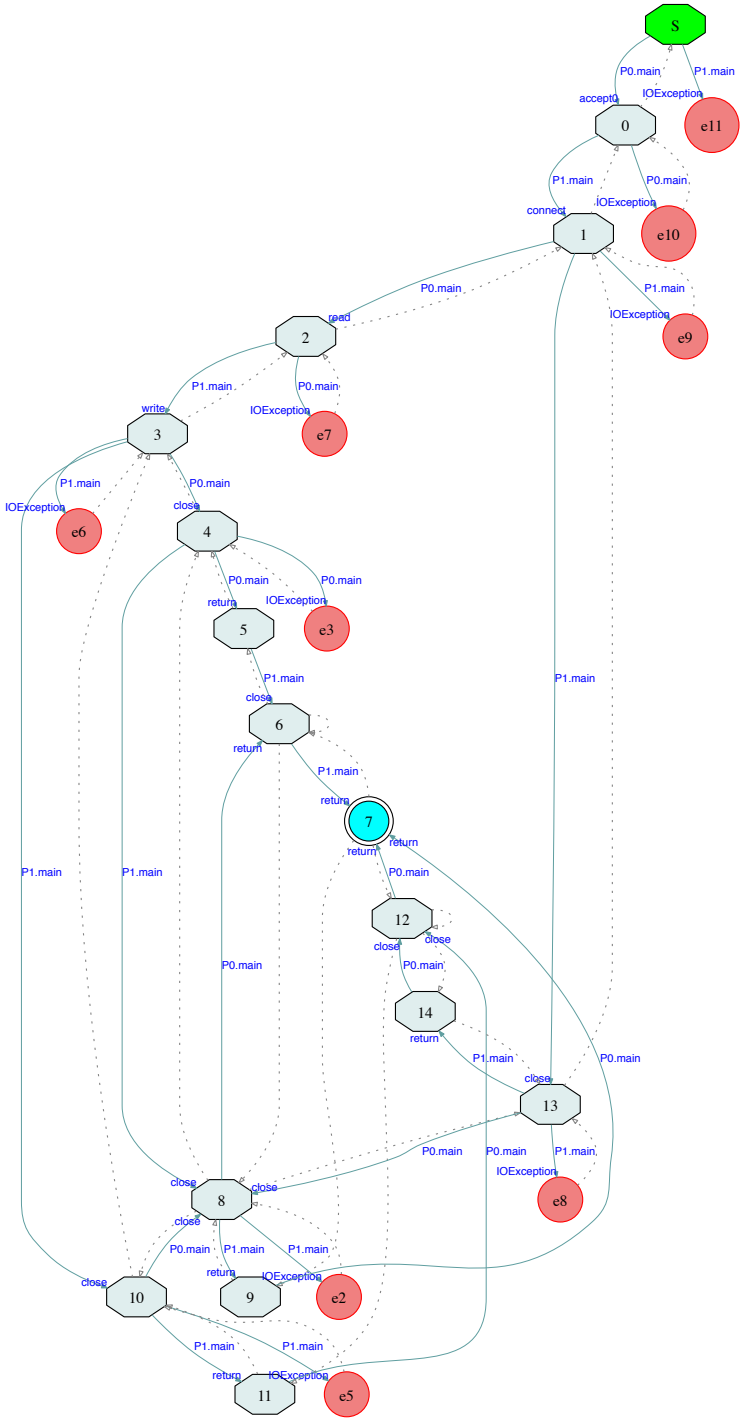

Figure 5.5: Search graph for example in Figure 5.1 when failures are injected

Figure 5.6 contrasts local choices against global choices. The search graph on the left side is obtained from code similar to what is presented in Figure 5.1, except for the sake of simplicity the statements `socket.close()` in both the client and server code are commented out. Since both of the processes in this example are single threaded and there is no data non-determinism in the code, the search graph includes only global choices. To make local choices appear in the search graph, we made the client process multithreaded, i.e., at the beginning of the `main` method of `Client`, a statement is added which creates and starts a new thread. To keep the example simple, the body of the `run` method of the new thread is simply empty.

The search graph on the right hand side of Figure 5.6 is obtained from model checking the distributed application composed of the server process and the multithreaded client. The client threads are interleaved upon the starting of the new thread and the termination of the threads. Since the new thread does not interfere with the client `main` thread at all, there is a one-to-one correspondence between the global choices of the two search graphs.

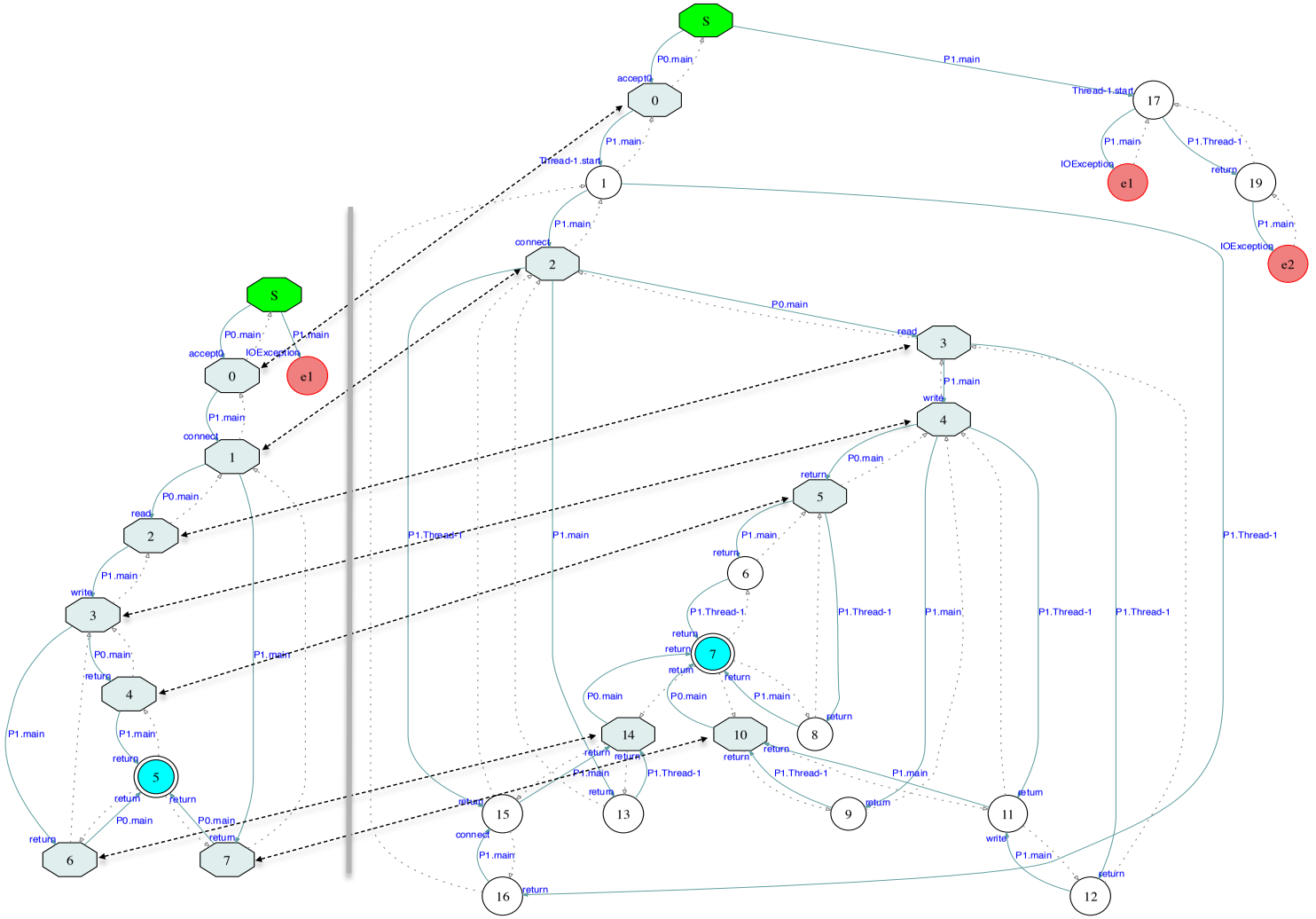

Figure 5.6: Local choices versus global choices

5.4 Correctness of the Model

In this section, we explain two different approaches used to verify that our model behaves consistently with the `java.net` classes. In Chapter 6, we formalize our POR algorithm and show that it preserves deadlocks.

5.4.1 Testing Framework

One approach used to verify the model is testing. JPF includes a unit testing framework. Using this framework, one can implement regression tests in the form of methods, annotated by `org.junit.Test`, which run on top of JPF. The testing infrastructure of JPF has been designed in a way that its test classes adopt the same format as JUnit tests[14]. By running a test method within the JPF testing infrastructure, all possible execution paths of the method are explored by JPF, and for each path it checks if the test passes.

The testing framework of JPF can only model check a single process. As part of this work, we extend the JPF testing framework to model check multiple processes simultaneously. Using our distributed setting, for every annotated method, one can specify an integer representing the number of processes. The code to be executed for each process by JPF is the body of the test method. In order to allow different

[14]Unit Testing Framework for Java: `http://junit.org`

processes to execute different parts of the test method, we include a method in the JPF testing infrastructure which can be used to retrieve the id of the processes. Using ids, one can make a process execute a certain part of the method. As an example, consider the following test method.

```
@Test
public void exampleTest() throws IOException {
  if (mpVerifyNoPropertyViolation(2)) {
    int prcId = getProcessId();
    int PORT = 1024;
    String HOST = "indigo.cse.yorku.ca"
    switch(prcId) {
      case 0:
        ServerSocket serverSocket = new ServerSocket(PORT);
        serverSocket.accept();
      case 1:
        Socket socket = new Socket(HOST, PORT);
        try {
          socket = new Socket(HOST, port);
          assertTrue(socket.isConnected());
        } catch(IOException e) {
        }
    }
  }
}
```

This example shows the format used to develop tests in JPF. By executing `mpVerifyNoPropertyViolation`, JPF is initialized and starts model checking the test method. The integer, 2, sent as argument to this method, represents the number of processes. In this example, JPF model checks a distributed application composed of two processes that execute the code surrounded by the `if` block. As you can see, using the process ids, parts executed by different processes are specified using

a `switch` statement. The process with the id 0 executes the part associated with the `case 0` statement which makes it the server process. The process with the id 1 executes the part associated with `case 1` which makes it the client process.

We use the distributed testing infrastructure explained above to test our model. For each standard class modeled in jpf-nas, we develop a test class, including regression tests that check if the model behaves as expected. We identified the correct behavior of `java.net` classes from the Java specification API. In the majority of the cases, the specifications are not precise enough, and we retrieved the expected behavior by going through the source code of the standard Java classes. For code which is implemented natively, we observed its behavior by running examples of distributed applications.

5.4.2 Runtime Behavior of the `java.net` Model

In order to understand the behavior of the `java.net` classes which are modeled in jpf-nas, we looked into their specifications along with their Java source code. However, due to parts that are implemented natively, some behaviors were not clear, and understanding them required observing the runtime behavior of distributed Java applications.

To verify our understanding of `java.net` classes, we defined invariants, preconditions, and postconditions in form of assertions that capture the behavior we

observed by looking into the actual classes from the Java standard library. To check if these assertions capture the correct behaviors, we included the assertion statements in our own copies of the standard classes. Basically, these classes are identical to the ones within the standard Java libraries, except they include the assertion statements. We execute some distributed Java applications on the host JVM, when forcing the host JVM to use our copies of `java.net` classes, which is accomplished by using the JVM property `-Xbootclasspath`. These applications are presented in Chapter 8. Not getting an assertion violation gave us confidence that our assertions capture the expected behaviors of `java.net` classes.

We used the same set of assertions to check if our model is sound, that is, the behaviors of distributed SUTs captured by the model are consistent with their behaviors captured by the Java libraries. We included the assertions into our model of the `java.net` package. However, that still cannot confirm our model is complete, that is, whether it captures all potential executions of distributed systems. Using this approach, we found an error in our model. The error was related to reading from a socket where the corresponding connection had been closed by the process at the other end. In this case, we missed throwing an exception of type `Socket-Exception`.

5.5 Related Work on Modeling IPC

Existing work that can be compared with our model includes modeling IPC for applications centralized at the SUT level. As mentioned in Section 3.2, these approaches transform the distributed SUT, which includes multiple processes, into a new single process Java application with multiple threads. That also requires providing IPC models that replace the communication between processes with inter-thread communication.

The work by Stoller and Liu [79] models Java RMI. Java RMI allows a process to invoke a method on an object that exists in another process. In RMI applications, one process, called the server, makes objects accessible to other processes, called clients. The server achieves that by entering the references to those objects in a database, called the RMI registry. Then, the clients obtain the references to remote objects in the server and communicate with the server through invoking methods on the remote objects. The arguments to the methods are provided by clients, i.e., they are *serialized* on the client and sent to the server. Serialization is a mechanism that represents a Java object as a sequence of bytes which includes information such as the object type and value. Once the method is invoked, the client blocks until it receives the serialized result from the server. The server unserializes the arguments, executes the method remotely invoked by the client, serializes the result, and passes

it to the calling client. Distributed applications that communicate through Java RMI are also referred to as distributed object applications.

To model Java RMI, the approach by Stoller and Liu simply replaces an RMI call with a ordinary method call and also simulates the RMI registry using a map. Their model considerably simplifies the communication and does not reflect the complexity of bidirectional network communications between processes. The work by Artho and Garoche [80] also provides an IPC model for applications centralized at the SUT level. Their IPC model is similar in flavour to our work, since they also model communication based on TCP sockets. Note that both models encompass a RMI model since Java RMI is a protocol built on top of TCP/IP.

In the IPC model of Artho and Garoche, the communication channels are represented by instances of the `java.io.PipedInputStream` and `java.io.PipedOutputStream` classes and are maintained at the SUT level. However, in our approach they are implemented at the native level and only their entry points are visible to the SUT processes. In a way, our approach is more realistic, since the communication channels used by Java processes are entirely handled by native libraries, not by Java code. Therefore, modeling them at the SUT level may affect the soundness of the approach and may lead to executions which are not consistent with the SUT behaviors. Moreover, the results from comparing our approach with their approach demonstrate that modeling communication channels at the SUT

level intensifies the state space explosion problem (see Section 8).

Moreover, our approach provides a more efficient way to model check distributed SUTs. The reduction technique implemented by the scheduler component leads to smaller state spaces. As explained in Section 5.2.2, we distinguish between two types of scheduling: global and local scheduling points. For operations that do not have any effects outside of the process, the scheduling choices include only those threads that exist in the process. However, in approaches where IPC models are built on top of centralized SUTs, all scheduling points are treated as global.

6 Reduction Techniques for Distributed Applications

In this chapter, we explain two different reduction techniques to reduce the state space of distributed multithreaded applications. These techniques are exploited in our work. One technique (Section 6.1) is a slight modification of a reduction technique presented by Godefroid in [41, Section 2]. This reduction technique by Godefroid reduces the state space of a system consisting of a single process that has multiple threads. These threads are assumed to be deterministic. We apply a similar technique to a system with multiple processes (Section 6.2). We also show that applying such a technique on the state space of a distributed system preserves deadlocks and assertion violations.

Moreover, we propose a partial order reduction technique for distributed systems (Section 6.3). Our technique relies on the fact that each process has its own local data which is not shared with any other processes in the system. We show that applying our POR algorithm on a system, which is constructed by the former

reduction technique, allows for detecting deadlocks. We use the persistent set technique of Godefroid [93] to prove the correctness of our algorithm. Using this technique, we show that in each state, our algorithm explores a sufficient subset of all possible transitions leading out of the state, preserving deadlocks.

The reduction techniques are presented in general settings. In the last section (Section 6.4), we specialize these techniques to JPF. We show that JPF applies the reduction technique of Godefroid [41] for model checking single process Java applications which preserves deadlocks and assertion violations. Moreover, we show that applying our POR algorithm within JPF allows for detecting global deadlocks in distributed Java applications.

6.1 Reduction of Single Process Systems

Consider a system consisting of a single process composed of a finite set Φ of threads. Each thread $T \in \Phi$ executes a sequence of *actions*. The system is modeled by a transition system TS. The transition system is a tuple $(S, Act, \rightarrow, s_0)$ such that

- S is a set of states,
- Act is a set of actions,
- $\rightarrow \subseteq S \times Act \times \Phi \times S$ is the transition relation, and
- $s_0 \in S$ is the initial state.

The set Act of actions is partitioned into the set V of visible actions, and the set I of invisible actions. Instead of $(s, \alpha, T, s') \in \rightarrow$ we write $s \xrightarrow[T]{\alpha} s'$. The action α is said to be *enabled* in the state s, if

$$\exists T \in \Phi : \exists s' \in S : s \xrightarrow[T]{\alpha} s'$$

We use $enabled(s)$ to denote the set of all enabled actions in s, that is,

$$enabled(s) = \{\alpha \in Act \mid \exists T \in \Phi : \exists s' \in S : s \xrightarrow[T]{\alpha} s'\}$$

Each thread is assumed to be deterministic. At any state s, for each thread T, there is at most one action α and state s', where $s \xrightarrow[T]{\alpha} s'$. This is expressed by Assumption 6.1.1.

Assumption 6.1.1. *If* $s \xrightarrow[T]{\alpha_1} s_1$ *and* $s \xrightarrow[T]{\alpha_2} s_2$ *then* $\alpha_1 = \alpha_2$ *and* $s_1 = s_2$.

A state s is called a *global state* if,

$$enabled(s) \subseteq V$$

We denote the set of global states by $g(S)$. We also assume that all actions to be executed from the initial state are visible. That is expressed by Assumption 6.1.2.

Assumption 6.1.2. $enabled(s_0) \subseteq V$

As a consequence, $s_0 \in g(S)$.

We assume that invisible actions of one thread do not have any effect on actions performed by other threads.

Assumption 6.1.3. *If* $s_1 \xrightarrow[T_1]{\alpha_1} s_2 \xrightarrow[T_2]{\alpha_2} s_3$ *where* $T_1 \neq T_2$ *and* $\alpha_1 \in I$ *or* $\alpha_2 \in I$ *then* $s_1 \xrightarrow[T_2]{\alpha_2} s_2' \xrightarrow[T_1]{\alpha_1} s_3$ *for some* $s_2' \in S$.

From the transition system TS, we construct a reduced system, denoted as $r(TS)$. The transition system $r(TS)$ describes the behavior of the system by its set of global states and the transitions (sequence of actions) that take the system from one global state to another. A transition of $r(TS)$ is one visible action followed by a finite maximal sequence of invisible actions performed by a single thread. The transition system $r(TS)$ is the tuple $(g(S), VI^*, \Rightarrow, s_0)$ such that $s \xRightarrow[T]{\alpha_1 \ldots \alpha_n} s'$ if,

- $s = s_1 \xrightarrow[T]{\alpha_1} s_2 \xrightarrow[T]{\alpha_2} \cdots \xrightarrow[T]{\alpha_n} s_n = s'$,
- $\alpha_1 \in V, \alpha_2 \in I, \ldots, \alpha_n \in I$, and
- $s' \in g(S)$.

The example illustrated in Figure 6.1 compares the transition systems TS and $r(TS)$, composed of two threads T_1 and T_2. The code of T_1 includes the sequence of actions $a_1; a_2$, and the code of T_2 includes the sequence of actions $b_1; b_2;$. The state of the system is composed of three variables (v, i, j) where v is shared, i is local to T_1, and j is local to T_2. The initial state of the system is $(-1, -1, -1)$. The actions a_1 and b_1 are visible actions which set v to 0 and 1, respectively. The actions a_2 and b_2 are invisible where a_2 sets i to 1, and b_2 sets j to 1. Figure 6.1(a)

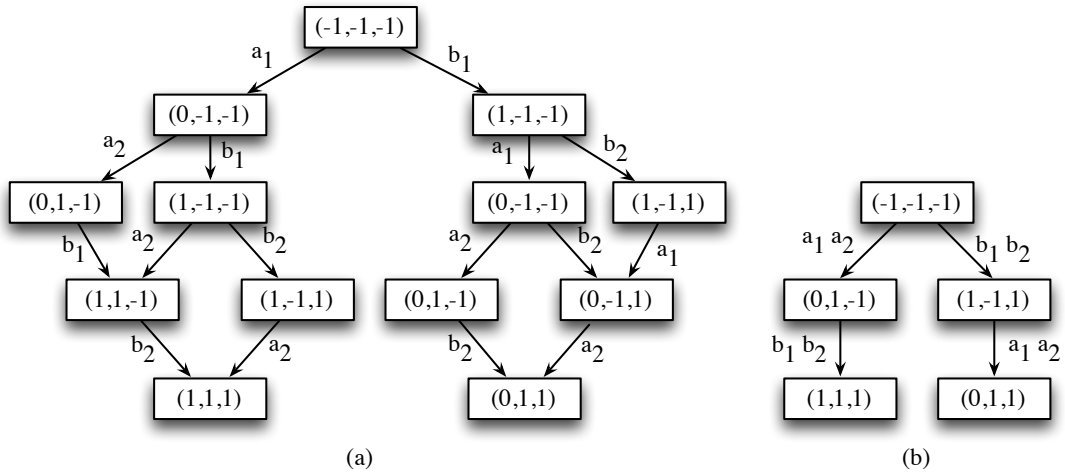

Figure 6.1: Comparing the transition systems (a) TS and (b) $r(TS)$ for a system composed of two threads with actions $a_1; a_2;$ and $b_1; b_2;$, where a_1 and b_1 are visible and the rest of the actions are invisible

illustrates the transition system TS, whereas Figure 6.1(b) illustrates the reduced system $r(TS)$.

The state $s \in S$ is *deadlocked* if,

$$enabled(s) = \emptyset$$

Note that in our model, states at which the system terminates are also identified as deadlock states. Since these final states can easily be identified, we do not distinguish between final and deadlocked states. This simplifies our model and the proofs that follow, without any loss of generality.

It can be shown that $r(TS)$ preserves deadlocks and assertion violations which are reachable from s_0 in TS. This is captured by Theorem 6.1.1 and 6.1.2. First, we present Lemma 6.1.1, which is required to prove these two theorems. This lemma shows that any global state which is reachable from s_0 in TS is also reachable from s_0 in $r(TS)$.

Lemma 6.1.1. *If* $s_1 \xrightarrow[T_1]{\alpha_1} s_2 \xrightarrow[T_2]{\alpha_2} \cdots \xrightarrow[T_n]{\alpha_n} s_{n+1}$ *and* $s_1,\ s_{n+1} \in g(S)$ *then* $s_1 \Rightarrow^* s_{n+1}$.

Proof. We prove this lemma by induction on the number v of visible actions in α_1, $\ldots$, α_n.

- Base case: $v = 1$. Since $s_1 \in g(S)$, $\mathit{enabled}(s_1) \subseteq V$. As a consequence $\alpha_1 \in V$. Therefore, $\alpha_2, \ldots, \alpha_n \in I$.

  Next, we show that $T_i = T_1$ for all $2 \leq i \leq n$. Towards a contradiction, let j be the smallest index in $[2, n]$ such that $T_j \neq T_1$. Since $\alpha_j \in I$, we can conclude from Assumption 6.1.3 that $s_1 \xrightarrow[T_j]{\alpha_j} s'_2 \xrightarrow[T_1]{\alpha_1} \cdots \xrightarrow[T_1]{\alpha_{j-1}} s_{j+1}$ for some s'_2, $\ldots$, $s'_j \in S$. Hence, $\alpha_j \in \mathit{enabled}(s_1)$. But this contradicts that $s_1 \in g(S)$.

  Combining the above, we get that $s_1 \xrightarrow[T_1]{\alpha_1 \ldots \alpha_n} s_{n+1}$.

- Inductive step: let $v > 1$. As in the base case, $\alpha_1 \in V$. Let α_k be the second visible action in $\alpha_1, \ldots, \alpha_n$. As in the base case, we can show that $T_i = T_1$ for all $2 \leq i < k$. Let ℓ be the smallest index in the interval $[k, n+1]$ such

that $s_\ell \xrightarrow[T_1]{\alpha} s$ for some $\alpha \in V$ and $s \in S$ or $\mathit{enabled}(s_\ell) = \emptyset$. Note that such an ℓ exists, since $s_{n+1} \in g(S)$.

Let i be the number of invisible actions performed by T_1 in $\alpha_{k+1}, \ldots, \alpha_{\ell-1}$. Let $\alpha_{f(1)}, \ldots, \alpha_{f(i)}$ be the subsequence of $\alpha_{k+1}, \ldots, \alpha_{\ell-1}$ of invisible actions performed by T_1. Let $\alpha_{g(1)}, \ldots, \alpha_{g(\ell-k-i-1)}$ be the subsequence of $\alpha_{k+1}, \ldots, \alpha_{\ell-1}$ of the remaining actions. We will prove that

$$s_k \xrightarrow[T_1]{\alpha_{f(1)}} \cdots \xrightarrow[T_1]{\alpha_{f(i)}} s'_{k+i} \xrightarrow[T_{g(1)}]{\alpha_{g(1)}} \cdots \xrightarrow[T_{g(\ell-k-i-1)}]{\alpha_{g(\ell-k-i-1)}} s_\ell \tag{6.1}$$

for some $s'_{k+1}, \ldots, s'_{\ell-1} \in S$ by induction on i.

- The base case, $i = 0$, is trivial.
- Let $i > 0$. Using Assumption 6.1.3, we can conclude

  $$s_k \xrightarrow[T_1]{\alpha_{f(1)}} s'_{k+1} \xrightarrow[T_k]{\alpha_k} \cdots \xrightarrow[T_{f(1)-1}]{\alpha_{f(1)-1}} \cdots \xrightarrow[T_{f(1)+1}]{\alpha_{f(1)+1}} \cdots \xrightarrow[T_{\ell-1}]{\alpha_{\ell-1}} s_\ell$$

  for some $s'_{k+1}, \ldots, s'_{\ell-1} \in S$. By induction,

  $$s'_{k+1} \xrightarrow[T_1]{\alpha_{f(2)}} \cdots \xrightarrow[T_1]{\alpha_{f(i)}} s'_{k+i} \xrightarrow[T_{g(1)}]{\alpha_{g(1)}} \cdots \xrightarrow[T_{g(\ell-k-i-1)}]{\alpha_{g(\ell-k-i-1)}} s_\ell$$

  for some $s'_{k+2}, \ldots, s'_{\ell-1} \in S$. Hence, (6.1) immediately follows.

Next, we will show that $s'_{k+i} \in g(S)$ by showing that $\mathit{enabled}(s'_{k+i}) \subseteq V$. From the choice of ℓ and the construction of the subsequence $\alpha_{f(1)}, \ldots, \alpha_{f(i)}$ we can conclude that $s'_{k+i} \xrightarrow[T_1]{\alpha} s$ implies $\alpha \in V$. Let $T \neq T_1$. Towards a

contradiction, assume that $s'_{k+i} \xrightarrow[T]{\alpha} s$ for some $s \in S$ and $\alpha \in I$. Again using Assumption 6.1.3, we can conclude that $s_1 \xrightarrow[T]{\alpha} s'$ for some $s' \in S$. This contradicts that $s_1 \in g(S)$.

From the above we can conclude that $s_1 \xrightarrow[T_1]{\alpha_1 \ldots \alpha_{k-1} \alpha_{f(1)} \ldots \alpha_{f(i)}} s'_{k+i}$. By induction, $s'_{k+i} \Rightarrow^* s_{n+1}$. Hence, $s_1 \Rightarrow^* s_{n+1}$.

□

Theorem 6.1.1. *All deadlocked states which are reachable from s_0 in TS are also reachable from s_0 in $r(TS)$.*

Proof. Consider that the state s reachable from s_0 in TS is deadlocked. According to the definition of a deadlocked state, we have $enabled(s) = \emptyset$. Therefore, $enabled(s) \subseteq V$ which implies that s is a global state, that is, $s \in g(S)$. According to Lemma 6.1.1, since s is reachable from s_0 in TS, it is also reachable from s_0 in $r(TS)$. □

In our model, an assertion is defined as an action $assert(A)$, where $A \subseteq S$. The assertion A is said to be violated in the state s, if $assert(A) \in enabled(s)$ and $s \notin A$. In our model, it is assumed that the action $assert(A)$ is visible which is expressed by Assumption 6.1.4.

Assumption 6.1.4. $assert(A) \in V$

Moreover, we assume that invisible actions cannot affect assertion results. This is expressed by Assumption 6.1.5

Assumption 6.1.5. *If* $s \xrightarrow[T]{\alpha} s'$ *where* $\alpha \in I$ *and* $assert(A) \in enabled(s)$ *then* $assert(A) \in enabled(s')$ *and* $s \in A$ *iff* $s' \in A$

To prove the following lemma, we need to assume that each thread cannot do a sequence of infinitely many invisible actions.

Assumption 6.1.6. *If* $s_i \xrightarrow[T]{\alpha_i} s_{i+1}$ *for all* $i \in \mathbb{N}$ *then for all* $n \in \mathbb{N}$ *there exists* $m > n$ *such that* $\alpha_m \in V$.

As we will show next, from any state we can reach a global state by doing only invisible actions.

Lemma 6.1.2. *For all* $s_1 \in S$ *there exists* $s_1 \xrightarrow[T_1]{\alpha_1} s_2 \xrightarrow[T_2]{\alpha_2} \cdots \xrightarrow[T_n]{\alpha_n} s_{n+1}$ *where* $\alpha_1, \ldots, \alpha_n \in I$ *and* $s_{n+1} \in g(S)$.

Proof. Consider the following algorithm.

$s \leftarrow s_1$

while $s \notin g(S)$ **do**

pick $\alpha \in I$, $s' \in S$ and $T \in \Phi$ such that $s \xrightarrow[T]{\alpha} s'$

$s \leftarrow s'$

end while

Note that if $s \notin g(S)$, then $enabled(s) \not\subseteq V$. Hence, we can find $\alpha \in I$, $s' \in S$ and $T \in \Phi$ such that $s \xrightarrow[T]{\alpha} s'$. If the algorithm terminates, then the lemma obviously holds.

Towards a contradiction, assume that the algorithm does not terminate. Then for all $i \in \mathbb{N}$, there exist $s_i \in S$, $\alpha_i \in I$ and $T_i \in \Phi$ such that $s_i \xrightarrow[T_i]{\alpha_i} s_{i+1}$. Since the set Φ is finite, one thread, say T, takes infinitely many transitions. Let $(T_{f(i)})_{i\in\mathbb{N}}$ be the subsequence of the transitions taken by thread T. Let $(T_{g_n(i)})_{i\in\mathbb{N}}$ be the subsequence obtained by removing $T_{f(1)}, \ldots, T_{f(n-1)}$ from $(T_i)_{i\in\mathbb{N}}$.

Next, we show that for each $n \in \mathbb{N}$ there exist a sequence $(s_{n,i})_{i\in\mathbb{N}}$ such that

- $s_{n,1} = s_1$,
- for all $1 \leq i < n$, $s_{n,i} \xrightarrow[T_{f(i)}]{\alpha_{f(i)}} s_{n,i+1}$
- for all $i \geq n$, $s_{n,i} \xrightarrow[T_{g_n(i-n+1)}]{\alpha_{g_n(i-n+1)}} s_{n,i+1}$

by induction on n. Note that this contradicts Assumption 6.1.6.

- Base case: $n = 1$. We simply take $s_{1,i} = s_i$ for all $i \in \mathbb{N}$.
- Inductive step: let $n > 1$. We define for all $1 \leq i \leq n-1$, $s_{n,i} = s_{n-1,i}$. Let j be such that $s_{n-1,j} \xrightarrow[T_{f(n-1)}]{\alpha_{f(n-1)}} s_{n-1,j+1}$. Then for all $n-1 \leq i < j$, $s_{n-1,i} \xrightarrow[T_{g_{n-1}(i-n+2)}]{\alpha_{g_{n-1}(i-n+2)}} s_{n-1,i+1}$ and $T_i \neq T$. Hence, according to Assumption 6.1.3, there exist $s_{n,i}$, for $n \leq i \leq j$, such that $s_{n-1,n-1} \xrightarrow[T_{f(n-1)}]{\alpha_{f(n-1)}} s_{n,n}$ and $s_{n,i} \xrightarrow[T_{g_{n-1}(i-n+1)}]{\alpha_{g_{n-1}(i-n+1)}} s_{n,i+1}$ for

all $n \leq i \leq j$ and $s_{n,j} \xrightarrow[T_{g_{n-1}(j-n+1)}]{\alpha_{g_{n-1}(j-n+1)}} s_{n-1,j+1}$. For all $i > j$, define $s_{n,i} = s_{n-1,i}$. The sequence $(s_{n,i})_{i \in \mathbb{N}}$ satisfies the properties by construction.

□

Theorem 6.1.2. *If there is an assertion, A, violated in a state reachable from s_0 in TS, then there is a state reachable from s_0 in $r(TS)$ at which A is violated.*

Proof. Suppose that s is a reachable state from s_0 in TS where an assertion A is violated. Then there is a thread $T \in \Phi$ where $s \xrightarrow[T]{assert(A)} s'$ for $s \notin A$ and some $s' \in S$. From Lemma 6.1.2 we can conclude that

$$s = s_1 \xrightarrow[T_1]{\alpha_1} s_2 \xrightarrow[T_2]{\alpha_2} \cdots \xrightarrow[T_n]{\alpha_n} s_{n+1} \tag{6.2}$$

where $\alpha_1, \ldots, \alpha_n \in I$ and $s_{n+1} \in g(S)$. From Assumption 6.1.5, since $assert(A) \in enabled(s_1)$ then $assert(A) \in enabled(s_{n+1})$ and since $s \notin A$ then $s_{n+1} \notin A$. From Lemma 6.1.1, since $s_{n+1} \in g(S)$, we can conclude that s_{n+1} at which A is violated is reachable from s_0 in $r(TS)$. □

6.2 Reduction of Distributed Systems

Consider a distributed system composed of a set of (multithreaded) processes. Let the set Φ of threads include all threads in the system regardless of which process they belong to. The behavior of such a distributed system can be captured by

a single process system composed of the set Φ of threads which can be modeled by the transition system TS. Therefore, we can use the reduced system $r(TS)$ to model the distributed system which allows for detecting deadlocks and assertion violations. Note that the distributed system is required to satisfy Assumption 6.1.1-6.1.6, otherwise there is no guarantee that deadlocks and assertion violations are preserved by $r(TS)$. In the next section, we apply a POR technique on $r(TS)$ which models the distributed system.

6.3 Partial Order Reduction Algorithm

In this section, we explain our POR technique that reduces the transition system $r(TS)$. Before presenting our technique, we introduce some definitions and notations used throughout the remainder of this section. Consider a distributed system composed of a finite set P of (multithreaded) processes. We use Φ_p to denote the finite set of threads that belong to the process $p \in P$. According to the definition of distributed systems, these sets are disjoint. That is expressed by Assumption 6.3.1.

Assumption 6.3.1. *If* $p_1 \neq p_2$ *then* $\Phi_{p_1} \cap \Phi_{p_2} = \emptyset$

We partition the set V of visible transitions in TS into the set V_l of *locally visible actions*, which involve interactions between threads of a single process, and

the set V_g of *globally visible actions*, which involve interactions between threads that may belong to different processes. We assume that these sets are disjoint. This is expressed by Assumption 6.3.2.

Assumption 6.3.2. $V_l \cap V_g = \emptyset$

We use $next(s, p)$ to denote the set of transitions, in $s \in g(S)$ that belong to the process p, that is,

$$next(s, p) = \{t \in VI^* \mid \exists T \in \Phi_p : s' \in g(S) : s \overset{t}{\underset{T}{\Rightarrow}} s'\}$$

We also use $next(s)$ to denote the set of all transitions in p, regardless of which process they belong to, that is,

$$next(s) = \bigcup_{p \in P} next(s, p)$$

We define the predicate $blocked(s, p)$ for a state s and process p, where $blocked(s, p)$ verifies to true *iff* there exist $t \in G_p$ and

$$s = s_1 \overset{t_1}{\underset{T_1}{\Rightarrow}} s_2 \overset{t_2}{\underset{T_2}{\Rightarrow}} \cdots \overset{t_{n-1}}{\underset{T_{n-1}}{\Rightarrow}} s_n$$

such that,

- $T_1, \ldots, T_{n-1} \notin \Phi_p$,
- $t \notin next(s, p)$, and
- $t \in next(s_n, p)$.

Algorithm 1 Partial Order Reduction Algorithm

1: **Initialize:** $stack \leftarrow \emptyset$; $H \leftarrow \emptyset$; $p \leftarrow nil$;
2: **push**(s_0, p) **onto** $stack$;
3: **while** $stack \neq \emptyset$ **do**
4: **pop**(s, p) from $stack$;
5: **if** $s \notin H$ **then**
6: **enter** s **in** H;
7: $\gamma \leftarrow branch(s, p)$;
8: **for all** $t \in \gamma$ **do**
9: $< s_{succ}, p > \leftarrow < s', p' >$ **where** $s \overset{t}{\underset{T}{\Rightarrow}} s'$ **and** $T \in \Phi_{p'}$;
10: **push**(s_{succ}, p) **onto** $stack$;
11: **end for**
12: **end if**
13: **end while**

In $r(TS)$, we distinguish between different types of transitions. Let $t = \alpha_1 \ldots \alpha_n$ where $s \overset{t}{\underset{T}{\Rightarrow}} s'$ and $T \in \Phi_p$. The transition t is said to be a *local transition* of p if $\alpha_1 \in V_l$. We use L_p to denote the set of all local transitions of p. The transition t is said to be a *global transition* if $\alpha_1 \in V_g$. We use G_p to denote the set of all global transitions of p.

In the remainder of this section, first, we describe our POR algorithm (see Algorithm 1). It adapts the persistent-set selective search of Godefroid presented in Figure 1.4 of [93]. We show that in any state our algorithm explores a persistent set of transitions. Then, using Theorem 4.3 of Godefroid in [93], we conclude that our algorithm can detect deadlocks.

Our POR algorithm starts from the initial state s_0 in $r(TS)$. After exploring each state, it adds it to the set H which used to keep track of visited states.

In any state $s \in g(S)$, reached by $p \in P$ (the process whose thread takes a transition discovering s), the algorithm computes the set $branch(s,p) \subseteq next(s)$, and it continues searching the state space from the transitions in $branch(s,p)$. The following algorithm shows how $branch(s,p)$ is computed for a state $s \in g(S)$.

if $s \neq s_0$ **and** $next(s,p) \neq \emptyset$ **and** $next(s,p) \subseteq L_p$ **and** $\neg blocked(s,p)$ **then**

$branch(s,p) \leftarrow next(s,p)$

else

$branch(s,p) \leftarrow next(s)$

end if

We say that s, where $s \neq s_0$, is a process-local state of p if,

$$next(s,p) \neq \emptyset \wedge next(s,p) \subseteq L_p \wedge \neg blocked(s,p)$$

If the state s is not a process-local state, then s is referred to as a *system-global state*. In process-local states, the algorithm explores the transitions of only the current process. In system-global states, the algorithm explores the transitions of all processes in the system.

Below we define the notation of *independence of transitions*, adapted from Definition 3.1 in [93]. Using this notion, we later show that our algorithm preserves certain properties.

Definition 6.3.1. *Let t_1 and t_2 be two transitions of $r(TS)$. The transitions t_1 and t_2 are said to be independent in $r(TS)$ if for any $s \in g(S)$ the two following properties hold:*

- *if $s \overset{t_1}{\underset{T}{\Rightarrow}} s'$, then $t_2 \in next(s)$ iff $t_2 \in next(s')$, and*
- *if $t_1, t_2 \in next(s)$, then there exists a unique state s'' where $s \overset{t_1}{\underset{T_1}{\Rightarrow}} s'_1 \overset{t_2}{\underset{T_2}{\Rightarrow}} s''$ and $s \overset{t_2}{\underset{T_2}{\Rightarrow}} s'_2 \overset{t_1}{\underset{T_1}{\Rightarrow}} s''$.*

In our model, we assume that local transitions of one process are independent from the transitions taken by any other process in the system. This is expressed by Assumption 6.3.3.

Assumption 6.3.3. *If $t_1 \in L_{p_1}$ and $t_2 \in L_{p_2} \cup G_{p_2}$ where $p_1 \neq p_2$ then t_1 and t_2 are independent.*

Before we discuss the correctness of the algorithm, below, we provide the definition of persistent set taken from Definition 4.1 in [93].

Definition 6.3.2. *A set γ of transitions in a state s is persistent in s iff, for all nonempty sequences of transitions*

$$s = s_1 \overset{t_1}{\underset{T_1}{\Rightarrow}} s_2 \overset{t_2}{\underset{T_2}{\Rightarrow}} \cdots \overset{t_{n-1}}{\underset{T_{n-1}}{\Rightarrow}} s_n \overset{t_n}{\underset{T_n}{\Rightarrow}} s_{n+1}$$

from s in $r(TS)$ and including only transitions $t_i \notin \gamma, 1 \leq i \leq n, t_n$ is independent in s_n with all transitions in γ.

Next, we show that the set of transitions, explored by Algorithm 1 in any state $s \in g(S)$, is a persistent set in s. That is captured by the following theorem.

Theorem 6.3.1. *In any state $s \in g(S)$ and process $p \in P$, the set $branch(s, p)$ of transitions explored by Algorithm 1 is a persistent set in s.*

Proof. For a system-global state s, $branch(s, p)$ is set to $next(s)$. Therefore, there is no nonempty sequence of transitions $s = s_1 \overset{t_1}{\underset{T_1}{\Rightarrow}} s_2 \overset{t_2}{\underset{T_2}{\Rightarrow}} \cdots \overset{t_n}{\underset{T_n}{\Rightarrow}} s_{n+1}$ which includes only $t_i \notin branch(s, p)$ for $1 \leq i \leq n$. Therefore, according to Definition 6.3.2, $branch(s, p)$ is a persistent set in s.

Now we show that, at a process-local state s of the process p, the set $branch(s, p)$ is persistent in s. Towards contradiction, we assume that $branch(s, p)$ is not a persistent set in s. Then, according to Definition 6.3.2, there exists a nonempty sequence of transitions

$$s = s_1 \overset{t_1}{\underset{T_1}{\Rightarrow}} s_2 \overset{t_2}{\underset{T_2}{\Rightarrow}} \cdots \overset{t_{n-1}}{\underset{T_{n_1}}{\Rightarrow}} s_n \overset{t_n}{\underset{T_n}{\Rightarrow}} s_{n+1} \tag{6.3}$$

such that:

(a) $t_1, t_2, ..., t_n \notin branch(s, p)$.

(b) t_n in (6.3) is dependent with some transition $t \in branch(s, p)$.

Without a loss of generality, suppose (6.3) is a shortest such a sequence. Now we show that such a sequence cannot exist. Since s is a process-local state,

$$branch(s, p) = next(s, p) \tag{6.4}$$

First we show that for all $1 \leq i \leq n$,

$$next(s, p) = next(s_i, p) \tag{6.5}$$

by induction on i.

- Base case: $i = 1$, which is trivial.

- Inductive step: let $i > 1$. We assume that (6.5) holds in s_{i-1}, that is,

$$next(s, p) = next(s_{i-1}, p). \tag{6.6}$$

  Then we show that (6.5) holds in s_i, that is, $next(s, p) = next(s_i, p)$.

  According to (a), $t_{i-1} \notin branch(s, p)$, which according to (6.4), implies that $t_{i-1} \notin next(s, p)$. Therefore, according to (6.6), $t_{i-1} \notin next(s_{i-1}, p)$. That implies $t_{i-1} \in L_{p'} \cup G_{p'}$ where $p' \neq p$. Since s is a process-local state, $next(s, p) \subseteq L_p$, and hence, $next(s_{i-1}, p) \subseteq L_p$. Therefore, according to Assumption 6.3.3, t_{i-1} is independent from any transition $t \in next(s_{i-1}, p)$. Therefore, from Definition 6.3.1, we can conclude that for any $t \in next(s_{i-1}, p)$, $t \in next(s_i, p)$. Hence,

$$next(s_{i-1}, p) \subseteq next(s_i, p). \tag{6.7}$$

  Let $t' \in next(s_i, p)$. Towards contradiction, suppose that there exists a transition t', where $t' \in next(s_i, p)$, but $t' \notin next(s_{i-1}, p)$. According to Definition 6.3.1, t' and t_{i-1} are dependent. From Assumption 6.3.3, we can conclude

that $t_{i-1} \in G_{p'}$ and $t' \in G_p$. Since $t' \notin next(s,p)$ (according to (6.5)) and $t' \in next(s_i,p)$, from (a) we can conclude $blocked(s,p) = true$. This is a contradiction, i.e., since s is a process local state, $blocked(s,p) = false$. Hence, such t' does not exist, and

$$next(s_i,p) \subseteq next(s_{i-1},p). \tag{6.8}$$

Therefore, from (6.7) and (6.8), we can conclude that (6.5) holds in the state s_i.

According to (b) t_n is dependent with some $t \in branch(s,p)$ in s_n. According to Assumption 6.3.3, this requires t_n to be a transition of p, that is, $t_n \in L_p \cup G_p$. That implies $t_n \in next(s_n,p)$. From (6.5), this implies $t_n \in next(s,p)$. Hence, from (6.4), $t_n \in branch(s,p)$. This contradicts our initial assumption (a). Therefore, such a sequence of transitions, (6.3), cannot not exist. □

Theorem 6.3.1 shows that our algorithm performs a selective search through $r(TS)$, and in each state s reached by a process p, explores a set $branch(s,p)$ of enabled transitions that is persistent. Moreover, the set $branch(s,p)$ computed by the algorithm becomes empty if and only if $next(s,p) = \emptyset$. The following theorem captures Theorem 4.3 by Godefroid, presented in [93]. According to this theorem our algorithm preserves deadlocks reachable from s_0 in $r(TS)$. Note that the system, on which the algorithm is applied, is required to satisfy Assumption 6.1.1-6.3.3,

otherwise there is no guarantee that deadlocks are detected by Algorithm 1.

Theorem 6.3.2. *Let $s \in g(S)$ be a deadlock state which is reachable from s_0 in $r(TS)$. Then s is reached from s_0 by Algorithm 1.*

6.4 Specialization of the Reduction to JPF

In this section, we describe a specialization of the reduction techniques, presented in this chapter, to JPF. Consider a single process Java application. To model check such a system, JPF applies a reduction technique which is similar to the approach by Godefroid [41]. Basically, JPF combines a sequence of bytecode instructions in a thread, that do not have any effects outside the thread, into a single transition. In a way, given a transition system TS which describes a single process Java application, JPF explores the reduced system $r(TS)$.

Every thread of the Java application represents Java code composed of a sequence of bytecode instructions. To model a Java application using TS, each action is mapped to a bytecode instruction. The set V of visible actions are those bytecode instructions which involve communication between threads. For example, consider `putfield` and `getfield` which access instance fields [94]. If a field being accessed by these bytecode instructions is shared, they are treated as visible actions. Another example of a visible action is a bytecode instruction that invokes the method `java.lang.Object.wait`. This method makes the current thread block

until it gets notified by some other thread. The set of enabled actions in the state s_0 from which the execution of the application starts includes exactly one bytecode instruction executed by the application's main thread. This bytecode instruction is also included in V to satisfy Assumption 6.1.2.

Any other bytecode instruction which is not in V is included in the set I of invisible actions. Side-effects of bytecode instructions mapped to invisible actions are only observable within a single thread. For example, a bytecode instruction that accesses a local variable declared in a method is treated as an invisible action. Intuitively, different orderings of actions that access disjoint variables (that is variables that occupy different memory locations) result in the same state of the system [2]. Variables accessed by invisible actions of a thread in a Java application cannot be accessed by any other thread in the system. Hence, Assumption 6.1.3 is a valid hypothesis.

Assumption 6.1.1 requires threads to be deterministic. For this assumption to hold, the SUTs are restricted to not include any data non-determinism. Data structures that can assume different values represent data non-determinism. For example, the method `java.util.Random.nextBoolean` introduces data non-determinism, since its return value can be `true` or `false`.

Moreover, Assumption 6.1.4 requires an assertion to be a visible action, otherwise it may not be reachable from s_0 in the system $r(TS)$. To satisfy this as-

sumption, we restrict assertions to only involve shared fields since they are accessed through visible operations. That also implies that invisible actions should not access variables used in an assertions. Therefore, Assumption 6.1.5 is also a valid hypothesis. Note that JPF includes the property `vm.por.field_boundaries.break` which is used to specify fields. If a field, f, is specified by this property, then any bytecode instruction that accesses f is treated as a visible action. Therefore, one can use this property to check for assertion violations which also involve non-shared fields.

Furthermore, according to Assumption 6.1.6, a thread cannot execute infinitely many invisible actions. JPF has the property `vm.max_transition_length` which denotes the maximum number of bytecode instructions that form a transition of $r(TS)$. By default, this property is set to 5000. Once JPF reaches this limit while executing a transition, it creates a choice generator which includes a choice associated with every runnable thread in the system. In a way, it creates a global state and it treats the next action of the current thread as a visible action. Hence, Assumption 6.1.6 is satisfied by JPF.

In order to model check a single process Java application, JPF constructs the reduced system $r(TS)$ from the TS that models the application using the above mapping. From Theorem 6.1.1 and 6.1.2, we can draw the following conclusion.

Conclusion 6.4.1. *Consider a single process Java application which does not include any data non-determinism. Given such an application, JPF can detect all*

deadlocks and assertion violations (involving only shared fields) which are reachable from the initial state.

We can also use TS to model a distributed Java application composed of a set P of processes. For a distributed Java application, bytecode instructions, mapped to visible actions, are partitioned into two disjoint sets, V_l and V_g, which satisfies Assumption 6.3.2. The set V_l of locally visible actions includes those bytecode instructions which involve intra-process communication. For example, consider the bytecode instructions `putfield` and `getfield` accessing a shared field and a bytecode instruction that invokes the method `java.lang.Object.wait`. Note that elements of the set V_l for a distributed system can be mapped to visible actions in a single process Java application.

The set V_g of globally visible actions includes those bytecode instructions which involve interprocess communication. For distributed Java applications, in which processes communicate through TCP sockets, V_g includes bytecode instructions that invoke native methods accessing communication buffers. One example is the native method `java.net.SocketInputStream.socketRead0` which reads the content of a communication buffer. The execution of the distributed Java application starts from the initial state s_0. The set of actions, denoted by $enabled(s_0)$ in TS, includes exactly one bytecode instruction associated with the main thread of each process $p \in P$. These bytecode instructions are also in the set V_g, which satisfies

Assumption 6.1.2. Any other bytecode instructions which are not in V_l or V_g are included in the set I of invisible actions.

Since Assumption 6.1.4 requires an assertion to be a visible action, for the case of distributed systems, we limit assertions to only involve shared data including fields shared between threads of a process and communication buffers data shared between all processes. Assumption 6.3.1 holds for any distributed Java application.

Consider the reduced system $r(TS)$ which is constructed from TS that models a distributed Java application. In the system $r(TS)$, Assumption 6.3.3 requires local transitions of one process to be independent of any transitions that belong to any other processes. Our class loading model ensures that each process has exclusive access to its own local data. Our model allows processes to only share communication buffers. Therefore, any variable accessed by a local transition, t, of a process cannot be accessed by a transition, t', that belongs to some other process in the system. Since t and t' access disjoint variables, intuitively, they are independent [2]. Hence, Assumption 6.3.3 is also a valid assumption.

To model check a distributed system which can be modeled by TS using the above mapping, JPF's default search strategy explores the reduced system $r(TS)$. This allows JPF to detect deadlocks and assertions involving shared fields and communication buffers data. In order to mitigate the state space explosion problem, our work implements a selective search strategy, presented by Algorithm 1, in JPF.

Note that when computing $branch(s)$ for a state s and process p, JPF evaluates the predicate $blocked(s, p)$ to true only if the next transition of a thread of p is either a blocked or an enabled global transition. JPF uses Algorithm 1 to search the system $r(TS)$ which models the distributed SUT. From Theorem 6.3.2, we can make the following conclusion.

Conclusion 6.4.2. *Consider a distributed Java application which does not include any data non-determinism. Given such application, JPF can detect all deadlocks which are reachable from the initial state.*

In Chapter 8, we apply our approach on four distributed Java applications which do not include any data non-determinism. For these distributed applications, Assumption 6.1.1-6.3.3 are satisfied. Our POR algorithm can be used as a heuristic method to check for properties other than deadlocks or where some assumptions are not satisfied. As a heuristic method, it may not be complete, that is, it may not detect all the errors, however, it is still useful for reducing prohibitively large state spaces of systems which cannot be model checked otherwise. As one of our future plans, we would like to check whether our algorithm can preserve some other properties such as assertion violations and local deadlocks, which prevent only one process from progressing.

7 Communication with External Processes

In Chapter 4, we explained our centralization approach to capture processes of a distributed system within the model checker, and later in Chapter 5, we described how their communication is modeled in JPF. However, in some cases, it might not be feasible to model check the entire distributed SUT. In distributed applications, a component of the system can be a process representing an external resource such as a database or a cloud computing service. It is not possible to verify such a distributed system as a whole if such a process resides on a different machine and its source code is not available, or it is in a format that is not supported by the model checker. Moreover, one might not even be interested in checking the states of the external resources.

For example, consider a Java process communicating with the Google translator which is a cloud computing service. This Java process may only rely on the translation results and not be affected by the internal state of the Google translator. To verify such a system, it suffices to model check the Java process and its commu-

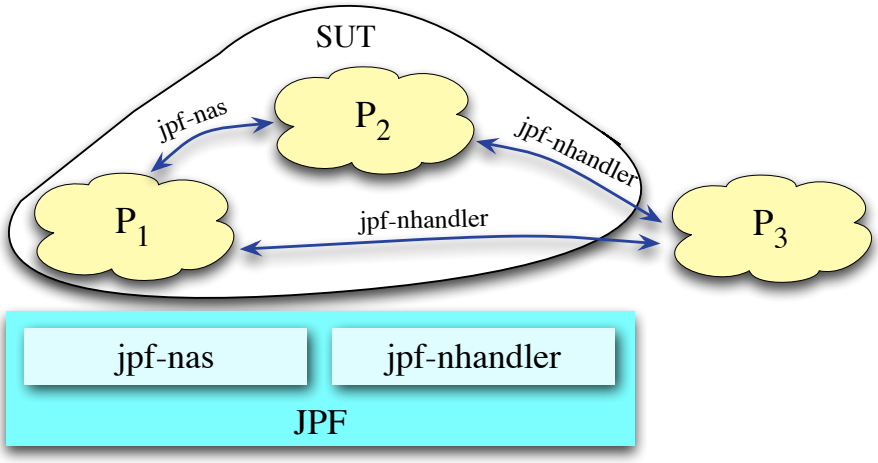

Figure 7.1: Contrasting different ways used by jpf-nas and jpf-nhandler to handle IPC

nication with the translator. Moreover, there is no need to provide backtracking of the external service, since regardless of the state of the translator application, given a translation request, the Java process always receives the same input.

As part of this work, we provide a way in JPF to connect the SUT to external resources. We have implemented this functionality as a JPF extension called *jpf-nhandler*[15]. Figure 7.1 contrasts the different ways adopted by the extensions jpf-nas and jpf-nhandler to capture process communications. In this figure, the communication channel captured by each extension is labeled by the name of the

[15]https://bitbucket.org/nastaran/jpf-nhandler

extension.

The approach implemented in jpf-nhandler is *generic* since it is not specific to certain types of communication and resources. This extension is based on transferring the execution of the SUT from JPF to the host JVM and back. The main idea is to delegate any call, which involves communication with external processes, to the host JVM. The next section elaborates on jpf-nhandler, discusses its design and implementation, and finally, outlines its limitations.

7.1 Delegating Calls in JPF

As explained in Section 2.5.5, the interface MJI of JPF is used to transfer the execution from the JPF level to the host JVM level. That is achieved through native peers, the methods of which are associated with the methods of the SUT classes. Whenever a method associated with a native peer is invoked, the execution is transferred to the host JVM which executes the native peer. The parts of code executed on the host JVM are completely invisible to JPF and are not model checked at all.

The approach taken by jpf-nhandler mainly relies on MJI and native peers. This extension makes the SUT switch between two different modes of execution, one mode is within JPF and the other one is within the host JVM. Whenever JPF encounters certain calls that involve communication with an external process, jpf-

nhandler intercepts the call and uses MJI to delegate it from JPF to the host JVM. To delegate the execution, it creates bytecode for native peer classes on-the-fly, referred to as *OTF peer* classes from now on. Basically, upon the invocation of certain methods, jpf-nhandler creates an OTF peer class (if it does not exist yet) and adds a native peer method, associated with the invoked method, to the OTF peer class (if it does not exist yet).

The design of jpf-nhandler consists of three main components: the forwarder, code generator, and converter. The *forwarder* component identifies and flags the calls to be delegated by jpf-nhandler. The *code generator* creates native peer code on-the-fly upon the invocation of methods flagged by the forwarder. Finally the *converter* component translates classes and objects from their JPF representations to their host JVM representations and back. The converter is used in the body of OTF peer methods, and it is an essential part of this extension, since the way that classes and objects are represented in JPF is different from the way that they are represented in a standard JVM. When the execution switches to a different mode, relevant classes and objects need to be converted from one environment to the other.

The actual communication between the SUT and external processes is performed by the OTF peer methods. Next, we use an example to elaborate on the body of these methods generated by the code generator component.

7.1.1 Example

Consider the following Java application that uses the Google translator web service[16] which translates phrases between natural languages.

```
1  import com.google.api.GoogleAPI;
2  import com.google.api.translate.*;
3  public class GoogleTranslator {
4    public static void main (String[] args) throws Exception {
5      String link = args[0];
6      String key = args[1]
7      GoogleAPI.setHttpReferrer(link);
8      GoogleAPI.setKey(key);
9      String result = Translate.DEFAULT.execute("Hello", Language.
           ENGLISH, Language.FRENCH);
10   }
11 }
```

This example represents a distributed application composed of a Java process and the Google translator. Running the example requires two inputs: a Google translate API key[17] and a link to associate the service with this program. The statement at line 7 requests translation for the word *Hello* from English to French. The execution of this statement leads to the invocation of the method `GoogleAPI.retrieveJSON(URL)`. This method captures the entire communication between the Java process and the Google translator. It commences the communication by creating and sending an HTTP (a protocol to exchange data on the internet) request

[16] code.google.com/p/google-api-translate-java

[17] https://developers.google.com/translate/

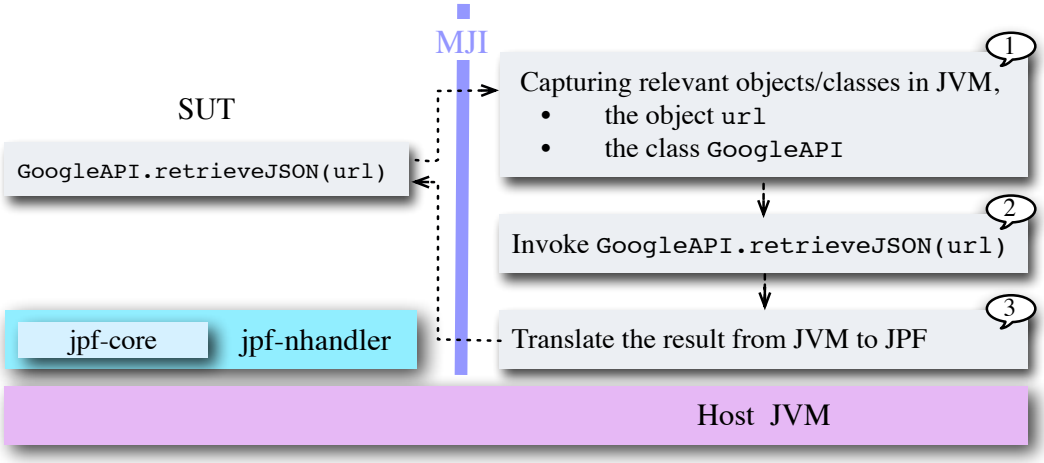

Figure 7.2: jpf-nhandler delegates the execution of `retrieveJSON(URL)` to the JVM level

to the translator, and it returns an object including the result.

Using jpf-nhandler allows for model checking the Java process as the Google translator executes in its normal environment. To model check this example, we configure jpf-nhandler to delegate the method `retrieveJSON`. The delegation of this call is presented in Figure 7.2. The left side executes on JPF and the right side represents code generated on-the-fly and executes on the host JVM. Once JPF gets to the bytecode invoking `retrieveJSON`, jpf-nhandler generates an OTF peer method associated with `retrieveJSON`. This native peer method implements the following three steps:

S1: First, the converter is used to transform the JPF representation of the class `GoogleAPI` and the object `url` to a corresponding object and class in the host

JVM.

S2: Then, using the Java reflection API, the execution is delegated to the host JVM by calling the original method `retrieveJSON` on the JVM representation of the class `GoogleAPI` with the JVM representation of `url` as its argument. By executing the method, the connection is established with the translator and the request is sent to the translator.

S3: Finally, after the method returns, its result, which is an instance of `String`, is transformed from its JVM representation to its JPF representation. Once this step is completed, the OTF peer method terminates and the execution transfers back to JPF.

The jpf-nhandler distribution comes with a `properties` file to configure the tool. To extend the tool's flexibility, we declare a wide variety of properties outlined in Table 7.1. One of the properties, `nhandler.spec.delegate`, is used for specifying the methods to be delegated. To connect the SUT to an external process, one should set this property to all the methods that involve communication with the external process.

7.1.2 Implementation

This section goes through the implementation details of the jpf-nhandler main components.

Forwarder

Using the configuration file, the forwarder identifies the methods to be delegated and makes them trigger jpf-nhandler when invoked on JPF. This component is implemented in the form of a JPF listener. As explained in Section 2.5.4, listeners run at the same level as JPF and are notified upon certain events, as JPF model checks the SUT. The forwarder is encapsulated by the class `gov.nasa.jpf.vm.ExecutionForwarder` and is triggered upon the event `ClassLoaded`. Before going through details, we discuss how JPF implements the association between methods and their corresponding native peers. The respective classes involved in the implementation are shown in the UML diagram in Figure 7.3.

JPF uses instances of `gov.nasa.jpf.vm.ClassInfo` to represent classes. `ClassInfo` includes a map, called `methods`, which keeps track of all the methods declared in the class. The keys of this map are string representations of the method signatures. The values of this map are instances of `gov.nasa.jpf.vm.MethodInfo` which represents methods in JPF. The class `gov.nasa.jpf.vm.NativeMethodInfo`

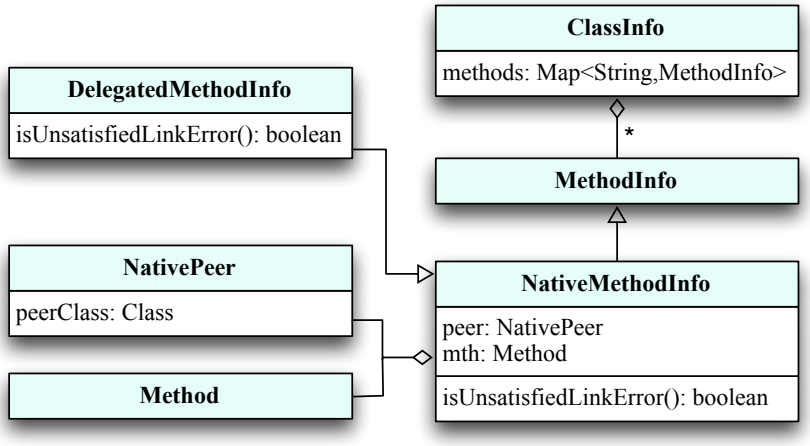

Figure 7.3: A UML diagram of the classes involved in associating methods with native peers

is a subclass of `MethodInfo` and represents those methods associated with native peer methods. To associate the method with its corresponding native peer method, `NativeMethodInfo` declares two fields: `peer` and `mth`. The field `peer` is of type `gov.nasa.jpf.vm.NativePeer` which contains the native peer class, and `mth` is of type `java.lang.reflect.Method` which represents the native peer method. Whenever JPF encounters a call to a method represented by a `NativeMethodInfo` object, instead of model checking it, JPF delegates its execution to the host JVM. JPF uses Java reflection to invoke the method `mth` of the class `PeerClass` (declared as a field in `peer`) on the host JVM. After the method `mth` returns, JPF resumes its model checking effort.

The listener forwarder receives notifications from JPF whenever a class is loaded.

By the time this notification is received, JPF has already created a `ClassInfo` object that represents the loaded class and initialized its `methods`. Once the forwarder is notified, it goes through the collection of `MethodInfo` objects of the loaded class. Using the `properties` file, it checks if the method needs to be delegated. If so, the forwarder replaces its corresponding `MethodInfo` object with a `gov.nasa.jpf.vm.DelegatedMethodInfo` object. The class `DelegatedMethodInfo` is a subclass of `NativeMethodInfo`. This class is part of jpf-nhandler and is used to specify methods to be delegated. Figure 7.3 shows this class in regards to JPF classes.

The class `NativeMethodInfo` contains the method `isUnsatisfiedLinkError`, which checks whether there is a corresponding native peer method (i.e., it checks whether its `mth` field is null). JPF executes the `isUnsatisfiedLinkError` method before it attempts to invoke the native peer method `mth`. To trigger jpf-nhandler on-the-fly, in our `DelegatedMethodInfo` we override the `isUnsatisfiedLinkError` method. Consider, for example, the method `retrieveJSON` from the previous example. The first time the method `isUnsatisfiedLinkError` is invoked on `retrieveJSON`'s `DelegatedMethodInfo` object, jpf-nhandler creates the corresponding OTF peer class and method (if they do not already exist) and initializes the fields `peer` and `mth` of the `DelegatedMethodInfo` object to the OTF peer class and method, respectively. As a consequence, whenever JPF encounters a call to `retrieveJSON`, it delegates the execution of its associated OTF peer method `mth` to the host JVM.

Code Generator

The functionality to generate OTF peer classes is implemented in the package `nhandler.peerGen`. By default, jpf-nhandler generates both bytecode and Java code for the OTF peer classes. To generate bytecode, this component uses the bytecode engineering library (BCEL)[18]. The OTF peer classes and methods follow the same naming pattern as the JPF native peer classes and methods, with the exception that the name of the OTF peer classes is prefixed by `OTF_`. The files containing the bytecode and Java code generated by jpf-nhandler can be found in the `onthefly` directory of jpf-nhandler.

A key class in the package `nhandler.peerGen` is `PeerClassGen`. As mentioned earlier, the method `isUnsatisfiedLinkError` of `DelegatedMethodInfo` creates OTF peer classes. In particular, it uses an instance of `PeerClassGen` for each OTF peer class. The diagram in Figure 7.4 shows how a `PeerClassGen` decides to generate OTF peer classes and methods.

Before generating the OTF peer class, the `PeerClassGen` object checks whether such a class already exists in the `onthefly` directory. If not, it generates the OTF peer class. Otherwise, it loads the existing class, and extends it as it gets to methods, which need to be delegated and whose associated OTF native peer

[18] commons.apache.org/bcel

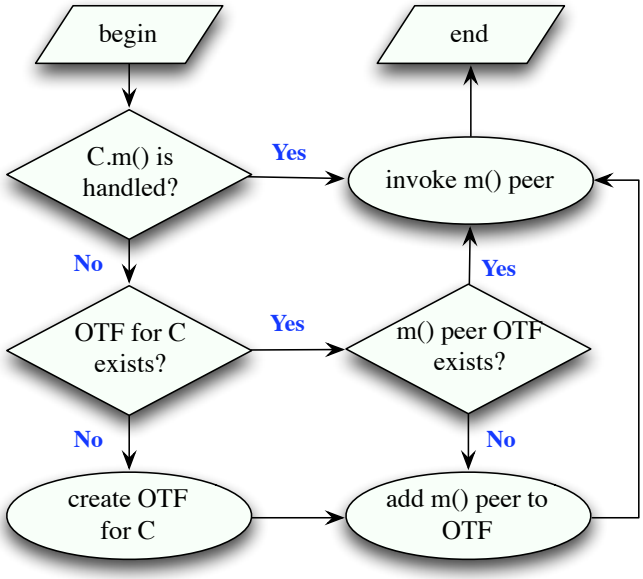

Figure 7.4: A UML diagram depicting how `PeerClassGen` decides to handle a call to the native method `m` of the class `C`

methods are missing. To extend an OTF peer class with an OTF peer method, the method `createMethod(NativeMethodInfo)` is invoked on the `PeerClassGen` object. If such a method does not already exist, it adds it to the OTF peer class. The body of the OTF peer method implements similar steps to the ones explained in Section 7.1 for the call `GoogleAPI.retrieveJSON(url)`.

Note that we can implement similar functionalities in jpf-nhandler without the need to generate the OTF peer classes at all. That can be accomplished by directly implementing the invocation of the native method using Java reflection. However, due to limitations of the tool (discussed in Section 7.1.3), jpf-nhandler may fail

to automatically handle a call. In that case, one can modify the generated Java code to possibly address those limitations, rather than having to start from scratch. Since generating the bytecode and Java code is expensive, we included a property, `nhandler.clean`, to make jpf-nhandler reuse existing OTF peer classes. Later in this chapter we present results indicating how effective it is to reuse existing classes.

Converter

As mentioned earlier, the way that objects and classes are represented in JPF is different from the way they are represented by the host JVM. JPF uses the `ClassInfo` and `ElementInfo`, in the package `gov.nasa.jpf.vm`, to represent classes and objects, respectively. Figure 7.5 contains an example, contrasting the different representations for a simple object. The way that JPF represents primitive types (e.g., `int` and `char`) is similar to the way adopted by the host JVM.

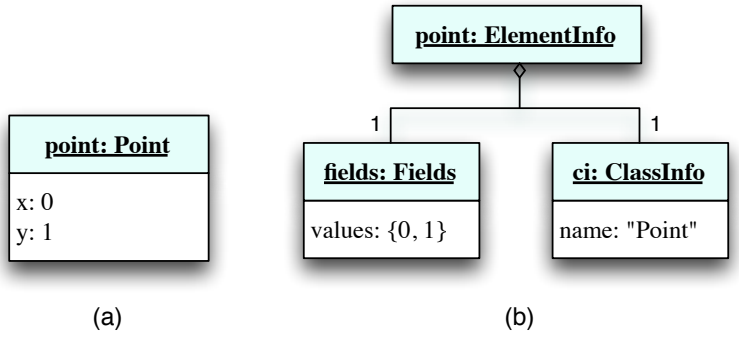

Figure 7.5: UML diagrams of a Point object represented in a JVM (a) and JPF (b)

Since jpf-nhandler interacts with the host JVM (steps 1 and 3 in Figure 7.2), it needs to convert objects and classes from JPF to the host JVM and back. The converter component is implemented in the package `nhandler.conversion`. Given an object or a class in one environment, the converter obtains the entity with the same state in the other environment. The state of a class is identified by the values of all static fields declared in the class, and the state of an object is identified by the values of all non-static fields declared in the object's class. Consider converting a JVM object to a JPF object. Conversion is performed recursively. Using the Java reflection API, the converter goes through the fields of the JVM object and for each non-primitive field it performs another conversion from JVM to JPF.

However, this *generic* converter does not work if there is an inconsistency between a model class and the actual class it models. Since model classes (Section 2.5.6) abstract away details from the original ones, they usually do not declare the same fields as declared in the original classes. However, the converter, explained above, relies on a one-to-one correspondence between the fields of the model class and the actual class. To address this issue, the converter uses the *abstract factory* design to instantiate objects of type `Converter`, the subclasses of which implement type-specific conversions. The factory returns the generic converter if there is no inconsistency, and a hand crafted converter otherwise. This design is illustrated in Figure 7.6. The generic converter is captured by two classes, one converts object

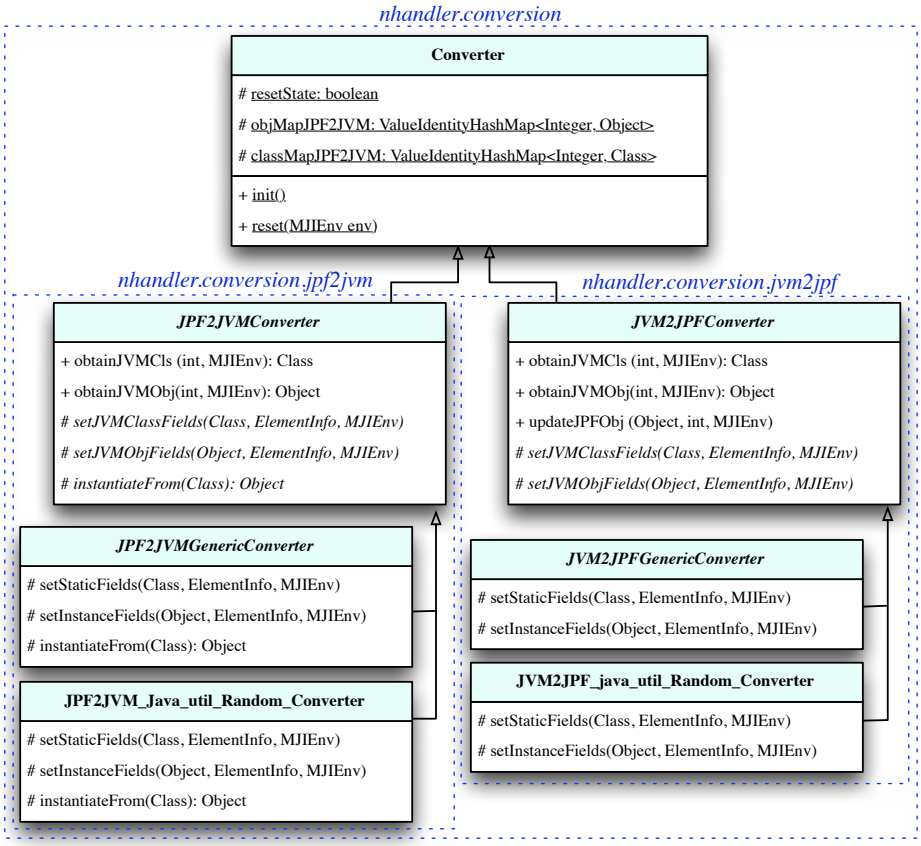

Figure 7.6: UML diagram representing classes that implement the converter component

and classes from JPF to the host JVM, and the other one does the conversion from the host JVM to JPF. Similarly, each hand crafted converter is encapsulated by two types. For example, since the `java.util.Random` model class and the actual class do not declare the same fields, the factory returns a hand crafted converter, implemented by `JPF2JVM_java_util_Random_Converter` and `JVM2JPF_java_util_-`

`Random_Converter`.

7.1.3 Limitations

There are some limitations to jpf-nhandler which are outlined in this section. Native code can modify arbitrary objects and classes through JNI. Currently, we only reflect in JPF the changes made by the native code to some objects and classes. For example, consider the call `GoogleAPI.retrieveJSON(url)`. Only changes made to the object `url`, its declaring class `java.net.URL`, and the class `GoogleAPI` are reflected in JPF. However, if the method were to change any other objects or classes, those changes would not be reflected in their corresponding JPF representations. This can be partly addressed by enhancing the generated code.

Moreover, delegation of a method to the host JVM amounts to the assumption that its execution is atomic. For example, consider an application consisting of two threads that share an integer variable `x` which is initialized to zero. The one thread simply consists of the statement `assert x % 2 == 0`. The other thread calls a method whose body consists of `x++; x++;`. JPF detects an uncaught exception, since there exists an interleaving of the two threads in which the assertion is not true. But, by delegating the method to the host JVM, it is assumed to be atomic, and JPF misses an assertion error. However, we focus on detecting errors, not on proving that code is error free. Using jpf-nhandler we can exercise some possible

executions of code that without jpf-nhandler requires huge modeling effort of the user in order to be checked by JPF.

To explore all potential executions of an application, JPF may backtrack to states that have been already visited. That causes jpf-nhandler to delegate a method multiple times which can lead to undesirable consequences. Consider, for example, an application that adds an entry to a database. To avoid a call from being delegated more than once, we intend in future work, to use a cache to record the effects of a delegated method call. If JPF encounters the same call later, we simply reflect the cached effects in JPF.

7.2 Tool Demonstration

Consider the example presented in Section 7.1.1. As mentioned earlier, to model check this example, we configure jpf-nhandler to delegate the call `GoogleAPI.retrieveJSON(url)`. To use our extension, we run JPF on the following application properties file, with extension `.jpf`, which also includes the tool configuration.

```
1 @using = jpf-nhandler
2 target = GoogleTranslator
3 nhandler.spec.delegate = com.google.api.GoogleAPI.retrieveJSON
4 nhandler.genSource = true
```

The first line makes JPF use jpf-nhandler. The second line specifies the initial class of the target application. The third line tells jpf-nhandler to delegate the

call `retrieveJSON`. Finally, by setting `getSource` to `true`, the source for the OTF peer of `GoogleAPI` is generated which includes the following OTF peer method associated with `retrieveJSON`.

```
1  public static int
      retrieveJSON__Ljava_net_URL_2__Lorg_json_JSONObject_2 (MJIEnv env
      , int rcls, int arg0) throws Exception {
2    ConverterBase.reset(env);
3    Class<?> caller = JPF2JVMConverter.obtainJVMCls(rcls, env);
4    Object argValue[] = new Object[1];
5    argValue[0] = JPF2JVMConverter.obtainJVMObj(arg0, env);
6    Class<?> argType[] = new Class[1];
7    argType[0] = Class.forName("java.net.URL");
8    Method method = caller.getDeclaredMethod("retrieveJSON", argType);
9    method.setAccessible(true);
10   Object returnValue = method.invoke(null, argValue);
11   int JPFObj = JVM2JPFConverter.obtainJPFObj(returnValue, env);
12   JVM2JPFConverter.obtainJPFCls(caller, env);
13   JVM2JPFConverter.updateJPFObj(argValue[0], arg0, env);
14   return JPFObj;
15 }
```

Lines 3 and 5 obtain JVM representations of `GoogleAPI` and `url`, respectively. After the method `retrieveJSON` is retrieved (line 8), at line 10, using Java reflection, it is invoked on the host JVM. Once the method returns, at line 11, its return value is converted to its JPF representation. Next, at lines 12 and 13, it updates the JPF representations of the caller class (`GoogleAPI`) and the input parameter (`url`), respectively. Finally, once the OTF peer method returns the JPF representation of the delegated method return value (line 14), the execution transfers to the JPF environment, and JPF proceeds by executing the next bytecode instruction.

As mentioned earlier, providing the source code of the OTF peers allows for refining the automatically generated code and possibly addressing some limitations. The refined OTF peer can be compiled using the script, `compileOTF`, provided in the `bin` directory of jpf-nhandler.

In this chapter we specifically use jpf-nhandler to capture communications with processes and services running outside of the model checker. However, by appropriately configuring jpf-nhandler, it can be used for different applications which are not the focus of this thesis. One such application, which is the main motivation behind the development of the tool, is to handle native calls. Recently, our tool has been used in verifying a prototypical next-generation air traffic controller system, Autoresolver, at NASA, to handle native calls.

Table 7.1 outlines the properties declared in jpf-nhandler. One can set these properties in the jpf-nhandler `properties` file to configure the tool. If a property is not set, the default value presented in the second column is used instead. Consider the last six properties outlined in the table. If the specification of a method matches the value of more than one of these properties, jpf-nhandler gives priority to the property that appears last in the table.

property	default	description
`nhandler.clean`	`true`	If false, jpf-nhandler reuses the OTF peers created in previous runs. Otherwise these OTF peers are removed
`nhandler.resetVM-State`	`true`	If false, upon each delegation, jpf-nhandler reuses the JVM objects created in previous delegations
`nhandler.genSource`	`false`	If true, the source code for OTF peers is created along with bytecode
`nhandler. updateJPFState`	`true`	If false, jpf-nhandler does not update the JPF objects after the native call returns on the host JVM
`nhandler.delegate-UnhandledNative`	`false`	If true, jpf-nhandler delegates all unhandled native calls
`nhandler.skipNative`	`false`	If true, the OTF peer methods for unhandled native calls become empty methods returning default values
`nhandler.spec. delegate`	`null`	Any method that matches this specification is delegated by jpf-nhandler, e.g., using `java.lang.String.*`, all methods of `String` are delegated
`nhandler.spec. delegateNative`	`null`	Any native method that matches this specification is delegated by jpf-nhandler
`nhandler.spec.skip`	`null`	The OTF peer for any method that matches this specification becomes an empty method that returns a default value
`nhandler.spec. filter`	`null`	Any method that matches this specification is not handled by jpf-nhandler

Table 7.1: The list of properties to configure jpf-nhandler

We were able to apply jpf-nhandler on several examples to successfully model check a Java process connecting to an external service. Here, we outline a few such examples.

Querying a Database - jpf-nhandler applied on a Java application, that connects to an Apache Derby database[19], creates a new table, inserts two records into the table, and finally closes the connection to the database. By delegating calls that involve querying the database, jpf-nhandler model checked the Java process as it is connected to a database outside of the model checker.

Scraping the Web - We have developed a web scraper which simply reads the HTML of a web page. By configuring jpf-nhandler appropriately, we successfully model checked the Java process connecting to the web.

Communicating using Sockets - Using jpf-nhandler, we were able to model check a Java application composed of a server and a client that communicate through sockets. We ran the server on one machine and model checked the client on another machine.

Communicating using JGroups - Red Hat's JGroups[20] provides a framework for reliable multicast communication. In our example, two applications communicate by using an `org.jgroups.JChannel`. One application sends a message which is received by the other one. Using jpf-nhandler, we ran the receiver on one machine while model checking the sender on another machine.

[19] db.apache.org/derby

[20] www.jgroups.org

7.3 Correctness and Experimental Results

To verify jpf-nhandler, we selected four Java types which are not subject to the limitations outlined in Section 7.1.3. These types are outlined in the first column of Table 7.2. For each type we used a class including a test suite that checks the correctness of methods declared by the type. These test suites are based on tests developed for verifying standard Java libraries and they all pass when running on a standard JVM. The tests used in our experiments are released either as part of the Java specification requests (JSRs) or under the Apache software foundation (ASF) license.

To test our tool, we ran JPF on each test suite using jpf-nhandler. For each type, jpf-nhandler was configured to delegate all the methods declared by the type. We applied jpf-nhandler with two different settings: In setting A, it generated OTF peers, and in setting B, it reused the OTF peers created in previous runs. All the tests ran successfully.

We also ran jpf-core (without using jpf-nhandler) on the same test suites and all tests passed. Using the results from this setting, we could see how expensive delegation of calls in jpf-nhandler is compared to executing them normally in the JPF environment. Table 7.2 presents our results which are the average of ten runs. The time measurements are in milliseconds. The standard deviation is small, on

average 14 ms. The second column includes times from running JPF without using our extension. The third and fourth columns present times from running JPF with jpf-nhandler, using settings A and B, respectively. The overhead from the settings A and B are shown in the columns *overheadA* and *overheadB*, respectively. Our results show, on average, when reusing the OTF peers, the overhead is almost two times less than the overhead when jpf-nhandler generates the peers.

type	core(ms)	nhandler(ms)	reuse peer(ms)	overheadA	overheadB
`lang.String`	3767	4582	4038	21%	7%
`lang.Math`	6169	6974	6574	13%	6%
`lang.reflect.Array`	44406	5007	4802	13%	8%
`util.concurrent-.atomic.AtomicLong`	4250	4719	4485	11%	5%

Table 7.2: Results representing the overhead of jpf-nhandler

7.4 Related Work

The work most closely related to ours is that of d'Amorim et al. [95]. Although their extension of JPF also model checks some parts of the code and executes the other parts of the code, their objective is reducing the execution time of JPF. Although they have a different objective, their approach shares several ingredients with ours.

First of all, they also translate JPF objects to JVM objects and back. However, their translations have several limitations from which ours does not suffer. For example, their translation from JPF objects to JVM objects handles neither arrays nor instances of classes without a default constructor. In our case studies, we have to handle both. Secondly, they also use reflection to invoke methods on the host JVM. Whereas they only handle methods, we also deal with several other elements of Java such as constructors and static initializers.

Moreover, the approaches that use caching to model check distributed applications [76, 77, 78] are similar in flavour to jpf-nhandler. They also model check the SUT as it communicates with external processes. Unlike cache-based techniques, our approach does not provide backtracking for external processes. As mentioned earlier (see Section 3.1), the cache-based techniques introduce a cache layer that keeps track of communications between the SUT and the external processes. Using the cache, they keep the SUT in synchronization with the external processes. The existing cache-based techniques can only support socket-based communication over a network, whereas, the automatic technique adopted by jpf-nhandler is generic and it is not limited to network communications, e.g., we have applied jpf-nhandler to capture communication with external databases, web pages, and cloud computing services (see Section 7.2). Note that one can extend cache-based approaches to account for more types of communications. However, that requires manually

modeling their respective classes which is time consuming and error prone.

Gligoric and Majumdar [96] have developed DPF, an extension of JPF to model check database applications. They consider both in-memory databases and on-disk H2[21] databases. In their approach they reimplemented methods accessing the database. In Section 7.2, we already mentioned that jpf-nhandler can deal with a simple database application without having to reimplement any Java class. However, unlike their approach, jpf-nhandler is not able to keep the SUT in synchronization with the changes to the database when backtracking.

[21] h2database.com

8 Results

In this chapter, we apply our approach on several Java applications. We compare our approach with the most recent work based on centralization at the SUT level [80, 84]. This work is most similar to ours since it merges processes of a distributed SUT into one and supports socket-based interprocess communication. Moreover, this work uses JPF to model check four distributed Java applications. In our experiments, we use the same Java applications as SUTs , the source code of which is provided by Artho et al. Table 8.1 outlines these applications. The second column presents the size of each application in terms of lines of code. All the applications follow the server/client architecture. The simplest application is Echo in which both the server process and the client process are single threaded, whereas the most complex application is Alphabet in which both the server process and the client process are multithreaded. The last column of the table presents the maximum number of threads used for each application in our experiments.

We model check these applications using JPF in two different settings. One

application	size (loc)	server characteristics	client characteristics	max #threads
Echo	96	single-threaded	single-threaded	2
Daytime	72	single-threaded	single-threaded	6
Chat	134	multithreaded	single-threaded	5
Alphabet	162	multithreaded	multithreaded	9

Table 8.1: Java Applications used in are experiments

setting is referred to as *centralized-jpf* which applies our approach. In this setting, JPF is configured to run in the multiprocess mode, i.e., it uses the multiprocess VM and the distributed scheduler factory. It is also configured to use our jpf-nas extension. In the centralized-jpf setting, the original Java code of the distributed Java applications are fed to the model checker as SUTs. The distributed applications used in our experiments require that the server process always initiates the executions, otherwise, an exception of type `IOException` is thrown. To allow for only those executions which start with a particular process, the property `vm.nas.initiating_target` was introduced in jpf-nas. In centralized-jpf, this property is set to the server process to let only the server main thread proceed from the initial state. Moreover, the property `vm.process_finalizers` is set to true to enable finalizers which are essential to capture certain executions of the distributed applications.

The other setting is referred to as *centralized-sut*. To model check the distributed

applications in this setting, first, using centralization at the SUT level [80, 84], these applications are transformed into single process, multithreaded, applications. Then, the transformed applications are fed to JPF as SUTs. In this setting, JPF is configured to run in the single process mode, i.e., it uses the single process VM and the default scheduler factory. To handle the communication between processes, the centralization at the SUT approach models some classes from the `java.io` and `java.net` packages. This approach requires JPF to use these classes as alternatives to the existing JPF model classes and also alternatives to classes from the Java standard library. That is achieved by setting the JPF property `vm.boot_classpath` to the path of these classes.

All the experiments were performed on a Mac OS X machine with a 2.8 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 memory. For these experiments, we used the Java version 1.7.0_51 of Java and the Java HotSpot(TM) 64-Bit Server VM.

Previous studies reveal that there are various factors that impact the overall performance of Java applications [97, 98]. One factor is Just-In-Time (JIT) compilation [99] performed by the JVM. JIT compilation identifies those parts of bytecode which are frequently executed, usually referred to as *hotspots*. To improve the performance, JIT translates those parts into optimized native code during the execution. Applying JIT compilation can also affect the time intervals at which

the garbage collection occurs. The effect of the garbage collection on the memory system performance is another factor that affects the overall performance. When performing the experiments, we also observed impacts on the model checking time.

To run our experiments, we developed a driver that repeats each experiment 13 times. After each time JPF returns, the garbage collection of Java is invoked (`System.gc()`) to clean up the memory before the next run. Observing the data shows that the performance improves after the first few iterations of the loop. In all the experiments, the first run is slower which is mainly due to static initializations. In the majority of the cases, there is still a noticeable time difference between the next couple of runs and the rest of the runs which is explained by the effect of JIT compilation. In our analysis, we always disregard the first 3 runs since according to our observation, in most cases, the execution times become more consistent after the second or third run. The time provided for each experiment is the average of the last 10 runs.

Next, we present, and discuss the results obtained from applying JPF in the two settings. For each Java application, we provide the execution time in milliseconds, which represents average of ten runs, along with a standard deviation. We also provide the total number of states explored by the model checker which represents the size of the state spaces. Moreover, for each experiment, we present the total number of bytecode instructions executed by the JVM of JPF during the entire

model checking process.

Furthermore, the depth of the JPF search tree, in terms of number transitions, explored by the model checker is presented for each experiment. In our experiments, we use the default search algorithm of JPF to explore the state space which is depth-first search (DFS). If a bug is detected, the approach with the shallower search graph provides shorter execution paths leading to the bug. This makes understanding the bug easier.

We also use a listener, called `gov.nasa.jpf.listener.StateSpaceAnalyzer`, to collect information about the choices explored during the model checking process. Using this information, we identify the main factors that contribute to the size of the state space in both approaches. We also compare the scalability of our approach versus the other approach in terms of different variables such as number of processes and number of messages exchanged between processes. Finally, we present the memory consumption of each approach by providing the maximum amount of heap memory, in megabytes, occupied during the entire model checking process.

8.1 Echo Example

In our first experiment, we use an application called `Echo`. It consists of one server process and one client process. These processes are single-threaded and use TCP sockets to communicate. Once a connection between them is established, the client

sends some messages to the server and the server sends the same messages back to the client in the same order.

Applying our approach on the `Echo` application reported a deadlock which was not detected when JPF was applied in the centralized-sut setting. Consider the following two code snippets which present part of the server and the client code, respectively.

```
1  public class Server {
2    ...
3    while (!done) {
4      try {
5        socket = server.accept();
6        ...
7        while ((msg = in.readLine()) != null) {
8          out.write(msg + "\n");
9          ...
10       }
11     ...
12     done = true;
13   } catch(IOException e) {
14     System.err.println(e);
15   } finally {
16     ...
17     if (socket != null) {
18       socket.close();
19     }
20   }
21 }
```

```
1  public class Client {
2    ...
3    socket.connect(address);
4    ...
5    out.write (msg);
```

```
6   while ((n++ < num_of_msgs) && (line = in.readLine()) != null) {
7     ...
8   }
9   out.close()
10 }
```

The error trace produced by the model checker is illustrated in Figure 8.1, which is an execution path leading to a deadlock. This execution belongs to a run in which the client sends one message to the server. We use $s.i$ and $c.i$ to denote the i^{th} statement of the server and the client code snippets presented above. In this execution, after the connection is established ($c.3$), the client sends a message to the server ($c.5$), and blocks by attempting to read ($c.6$) until the server writes something. Then, the server proceeds by reading the message sent earlier from the client ($s.7$) and writes the same message back to the client ($s.8$). The maximum number of passes through the `while` loop in the client code is the same as the number of messages sent to the server, which is one in this example. Therefore, the client does not go through the loop again, and it terminates after closing the socket ($c.9$).

Upon termination, the finalizer thread invokes the `finalize()` method on the socket which closes the socket if it has not been closed yet. Next, since the server attempts to read from a broken connection ($s.7$), an exception of type `SocketException` is thrown. Since `SocketException` is a subclass of `IOException`, the exception is caught at $s.13$. Then, the server closes the socket, and since the

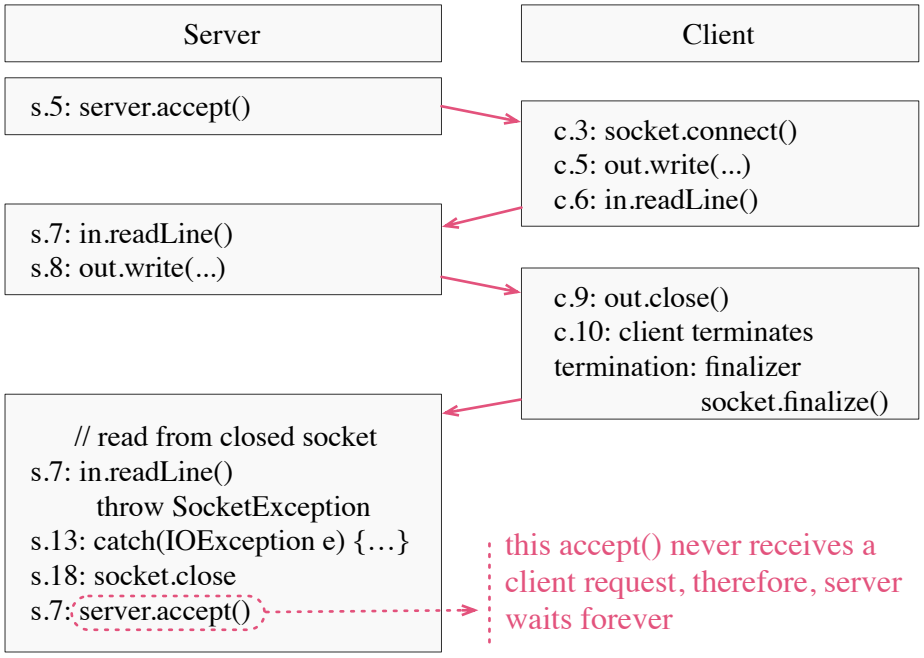

Figure 8.1: An execution of the `Echo` application leading to a deadlock

`while` loop condition evaluates to true, it goes through the loop again and executes `server.accept()` ($s.5$). This blocks the server until a connection request is received from a client. However, there is no client, and thus, the server blocks forever.

Looking into the code used by the centralization approach at the SUT level reveals that missed paths are caused by lack of precision in their model classes. Reading from a broken connection leads to an exception of type `SocketException`. This scenario is not captured by their model classes. To fix the bug, we included a `catch` block for the type `SocketException` as follows.

```
1 public class Server {
```

```
2     ...
3     while (!done) {
4       try {
5       socket = server.accept();
6       ...
7       while ((msg = in.readLine()) != null) {
8         ...
9       }
10      ...
11    } catch(SocketException e) {
12      done = true;
13    } catch(IOException e) {
14      ...
15    } finally {
16      ...
17    }
18  }
```

The only way the server can get to the `SocketException` catch block is reading from a broken connection. This implies that the connection has been established with the client at some point in the past, and therefore further execution of `server.accept()` leads to deadlock. By setting `done` to `true` in the `SocketException` catch block, the server never gets to `server.accept()` again, and thus the deadlock is avoided.

In this experiment, we model checked `Echo` in two different settings: centralized-jpf and centralized-sut. In each setting, the experiment was performed with different numbers of messages, ranging from one to ten, sent from the client to the server. The experiment for each message was repeated ten times. The size of the SUT code for centralized-jpf and centralized-sut is 77 and 74 classes, respectively. These

numbers are the total number of classes loaded by the VM of JPF through dynamic linking applied by the class-loading model. The centralization approach at the SUT level simplifies modeling of the `java.net` package by merging the implementation of a hierarchy of classes into one class, and therefore ends up using fewer classes.

Table 8.2 presents the execution times in milliseconds along with the standard deviations obtained from applying both approaches. The last column of the table presents the ratio of the execution time in the centralized-sut setting to the time obtained when applying centralized-jpf. The ratios show that, in every case, our approach has better performance. Moreover, the nonlinear increase in ratios shows that as the number of interactions between processes increases the overhead of their approach versus our approach becomes more significant.

The data presented in Table 8.3 is also obtained from model checking `Echo`. The first two sections of the table include the size of the state space, the number of bytecode instructions executed by the JVM of JPF, and the depth of the JPF search tree, respectively. Later in this chapter, we demonstrate this effect by seeding the SUTs with bugs (Section 8.5). The values outlined in the last column of each section of the table represent ratios of the centralized-sut values to the centralized-jpf values.

The results show that our approach is more efficient. In all cases, our approach leads to the smaller states spaces, fewer bytecode instructions, and shallower search

#echos	centralized-jpf		centralized-sut		time ratio
	time (ms)	st.dev	time (ms)	st.dev	
1	121	22	204	24	1.7
2	131	27	489	8	3.8
3	144	26	1046	11	7.3
4	152	22	1848	7	12.2
5	165	23	2869	13	17.4
6	180	25	4032	20	22.4
7	197	22	5317	22	27
8	221	23	7107	36	32.2
9	239	23	9228	36	38.6
10	264	25	11053	27	41.9

Table 8.2: Execution times obtained from model checking `Echo`

trees. It can be seen that the difference in the size of the state space becomes pronounced as the number of network interactions increases. One of the main factors that amounts to this difference is the way that communication channels are modeled. In our approach, the communication buffers exist on the host JVM level and are hidden from the SUT. In the centralization at the SUT level, the objects that capture communication buffers exist at the SUT level. They are modeled as shared objects between processes.

Note that the set of model classes used for this experiment in the centralized-sut setting are specific to this particular Java application. The mechanism used by these model classes to interleave threads upon network interactions is through

#echos	#states			#bytecode			max depth		
	centralized-jpf	centralized-sut	ratio	centralized-jpf	centralized-sut	ratio	centralized-jpf	centralized-sut	ratio
1	21	460	22	11237	230677	20.5	12	29	2.4
2	39	1519	39	16356	772744	47.2	14	54	3.9
3	60	3532	59	25098	1812712	72.2	16	83	5.2
4	85	6148	72	37464	3164913	84.5	18	110	6.1
5	114	9484	83	54434	4891326	89.9	20	137	6.9
6	147	13199	90	75866	6810049	89.8	22	162	7.4
7	184	18316	100	105256	9467057	89.9	24	191	8
8	225	24269	108	141680	12560364	88.7	26	220	8.5
9	270	30541	113	184371	15813404	85.8	28	247	8.8
10	319	37168	117	234254	19260655	82.2	30	273	9.1

Table 8.3: Data obtained from model checking the `Echo` application

accessing shared attributes. Communication buffers are encapsulated by a static array, which is declared in the `java.io.InputStream` model class and shared between the processes. Every time an array element is accessed (for example for some internal check) a new choice generator is created from which all the threads in the system can proceed.

To verify the justification above, we configured JPF to use the listener `gov.nasa.jpf.listener.StateSpaceAnalyzer`. This listener collects information about the choice generators created during the model checking process. Table 8.4 shows the total number of choices explored by JPF and the percentages of choices created due to socket accesses. It can be seen that, in the centralization at the SUT level, reading from and writing to sockets contribute a large portion of the state

space.

setting	1 echo			3 echos		
	#total	#buffer	%buffer	#total	#buffer	%buffer
centralized-jpf	24	9	%38	72	35	%49
centralized-sut	1083	974	%90	8641	8236	%95
setting	6 echoes			10 echoes		
	#total	#buffer	%buffer	#total	#buffer	%buffer
centralized-jpf	180	104	%58	394	252	%64
centralized-sut	32630	31499	%97	92302	89600	%97

Table 8.4: Information on choices generated due to accesses of communication buffers for `Echo`

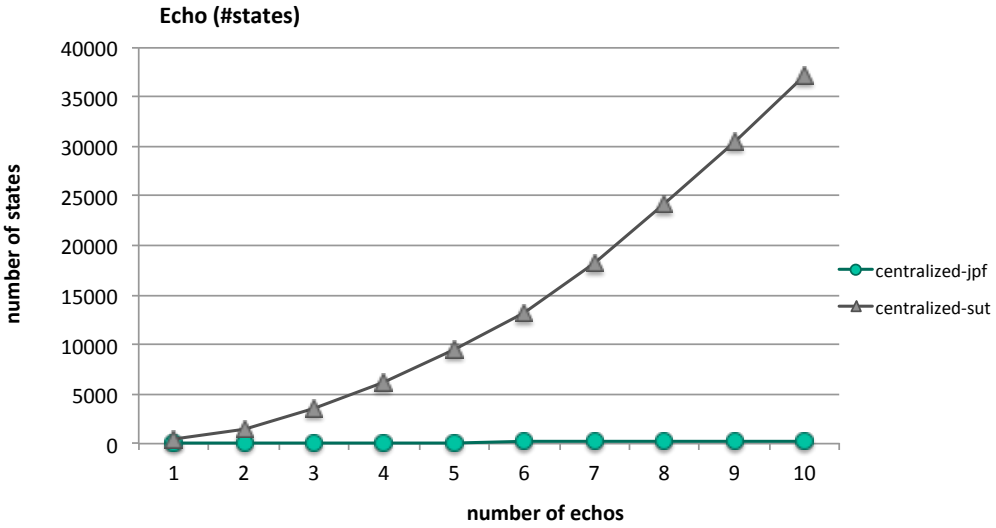

Figure 8.2: Number of states explored when model checking `Echo` in the centralized-jpf and centralized-sut settings

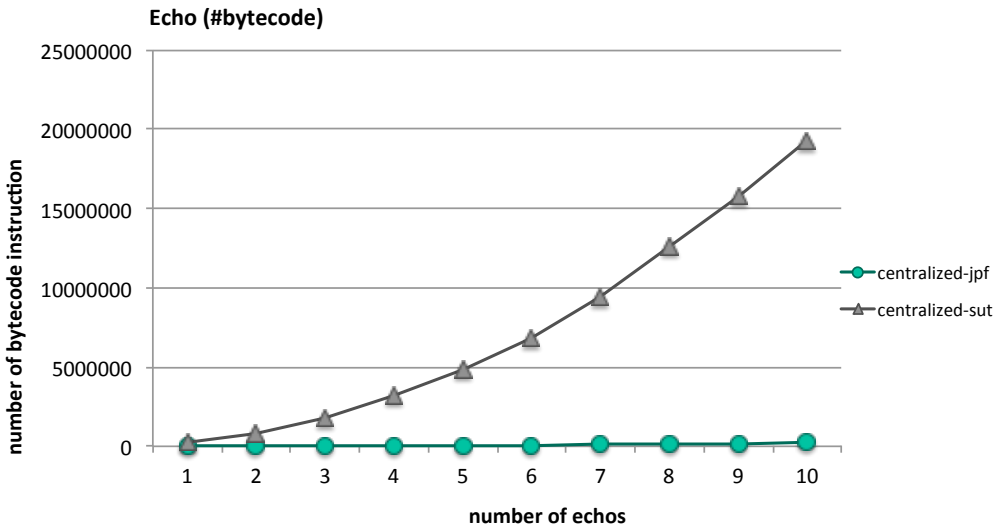

Figure 8.3: Number of bytecode instructions executed when model checking `Echo` in the centralized-jpf and centralized-sut settings

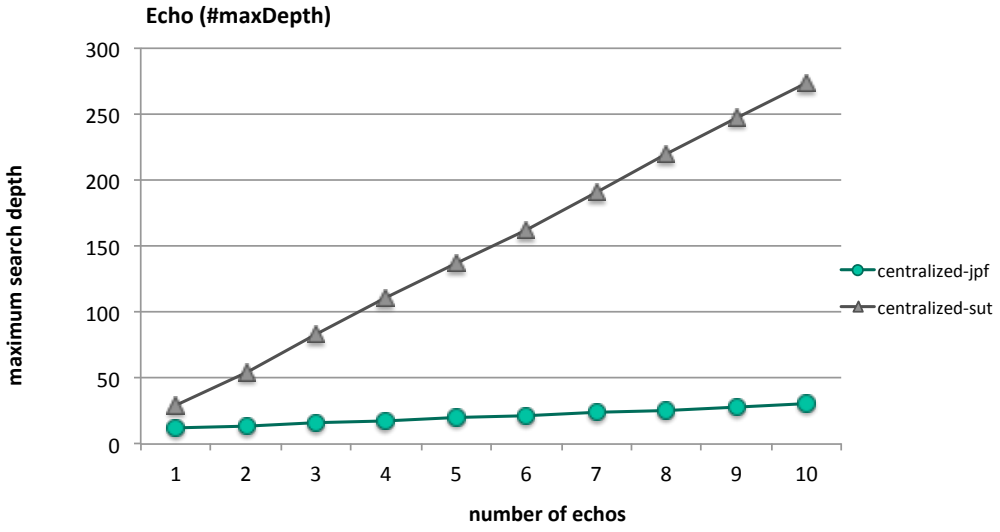

Figure 8.4: The depth of the search tree explored when model checking `Echo` in the centralized-jpf and centralized-sut settings

The graphs presented in Figure 8.2, 8.3, and 8.4 show how the approaches scale, in terms of the size of the state space, the number of bytecode instructions, and the depth of the search tree, as the number of communications between processes increases. It can be seen that our approach scales significantly better than the other approach.

8.2 Daytime Example

For our second experiment, we used the `Daytime` application. It includes a server which communicates with one or more clients. The server and the client processes are single-threaded and use TCP sockets to communicate. Once a connection between the server and a client is established, the server creates a `java.util.Date` object, capturing the current time, and sends its `String` representation to the client. The server repeats the same procedure for every client and can only serve one client at a time. When serving a client, the remaining clients block until a connection with the server is established.

We applied JPF on this application using centralized-jpf and centralized-sut settings. The size of the SUT code for centralized-jpf and centralized-sut is 83 and 89 classes, respectively. The reason for having more classes in centralized-sut is that it declares some helper classes which are used by the `java.net` model. The experiment is performed with different numbers of clients ranging from one to five.

Table 8.5 presents the execution times in milliseconds along with the standard deviations obtained from applying both approaches.

The ratios, presented in the last column of the table, show that overall, the performance of our approach is better. Even though the experiment performed with one client is faster when using the other approach, we can still conclude that overall, our approach exhibits a better performance, since the standard deviation in this experiment is high compared to the rest of the experiments. For instance, in the centralized-jpf setting, for the experiment with one client, the standard deviation is about 19% of the average execution time, whereas in the experiments performed with three or more clients, the standard deviation is less than 1% of the average execution times.

#clients	centralized-jpf		centralized-sut		time ratio
	time (ms)	st.dev	time (ms)	st.dev	
1	126	24.4	121	10	0.96
2	369	46	1396	16	3.8
3	5750	38	31141	83	5.4
4	142880	379	632300	1742	4.4
5	2848372	8689	10648446	39529	3.7

Table 8.5: Execution times obtained from model checking `Daytime`

The data presented in Table 8.6 is also obtained from model checking `Daytime`. In every case, centralization at the model checker level results in a smaller state

space, fewer bytecode instructions, and a shallower search graph.

#clients	#states			#bytecode			max depth		
	centralized-jpf	centralized-sut	ratio	centralized-jpf	centralized-sut	ratio	centralized-jpf	centralized-sut	ratio
1	20	144	7	18035	85643	5	9	19	2.1
2	367	2848	8	303011	2507243	8	17	37	2.2
3	7648	47452	6	6535605	57611690	9	25	54	2.2
4	310789	703912	5	147481441	1096959361	7	33	71	2.2
5	2413212	9700480	4	2764883787	18492964176	7	41	88	2.1

Table 8.6: Data obtained from model checking the `Daytime` application

The set of model classes used for this example in the centralized-sut setting uses `Object.wait()` and `Object.notify()` operations to interleave processes upon network interactions. These model classes are developed to model the package `java.net`. The communication buffers captured by these model classes exist at the SUT level. They are encapsulated by attributes of types `java.io.PipedInputStream` and `java.io.PipedOutputStream` declared in the `Socket` model class, i.e. socket writes to its `PipedOutputStream` object and reads from its `PipedInputStream` object which are read and written, respectively, through the socket at the other end of the connection. The same set of model classes is used for the remaining experiments (on the applications `Chat` and `Alphabet` in Section 8.3 and 8.4) performed in the centralized-sut setting.

Using the listener `StateSpaceAnalyzer`, we obtained information about choices created to model check `Daytime` in both settings. Table 8.7 compares the choices

created due to accessing communication buffers. The results show that accessing communication buffers kept at the SUT level amounts to large state spaces in the centralized-sut setting.

setting	2 clients				3 clients			
	#total	ratio	#buffer	ratio	#total	ratio	#buffer	ratio
centralized-jpf	579	14.8	91	42.7	14503	12.2	21111	44.3
centralized-sut	8569		3884		176789		93509	
setting	4 clients				5 clients			
	#total	ratio	#buffer	ratio	#total	ratio	#buffer	ratio
centralized-jpf	312663	10	45811	33.2	5605777	9	813355	27.7
centralized-sut	3135127		1520259		50436635		22554169	

Table 8.7: The percentages of choices generated due to access communication buffers for `Daytime`

The graphs presented in Figure 8.5, 8.6, and 8.7 show how the approaches scale, in terms of the size of the state space, the number of bytecode instructions, and the depth of the search tree, as the number of clients increases. It can be seen that also in this experiment, our approach scales better than the other approach.

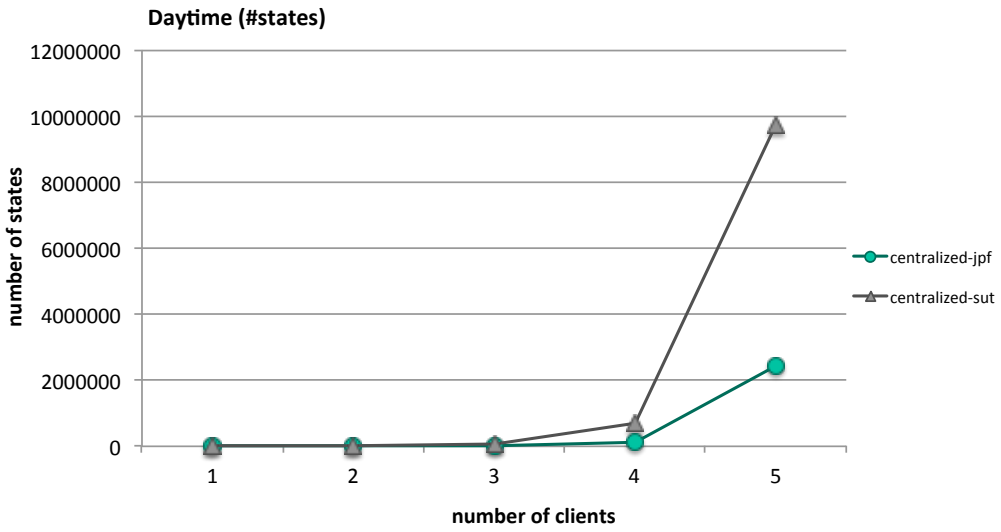

Figure 8.5: Number of states explored when model checking `Daytime` in the centralized-jpf and centralized-sut settings

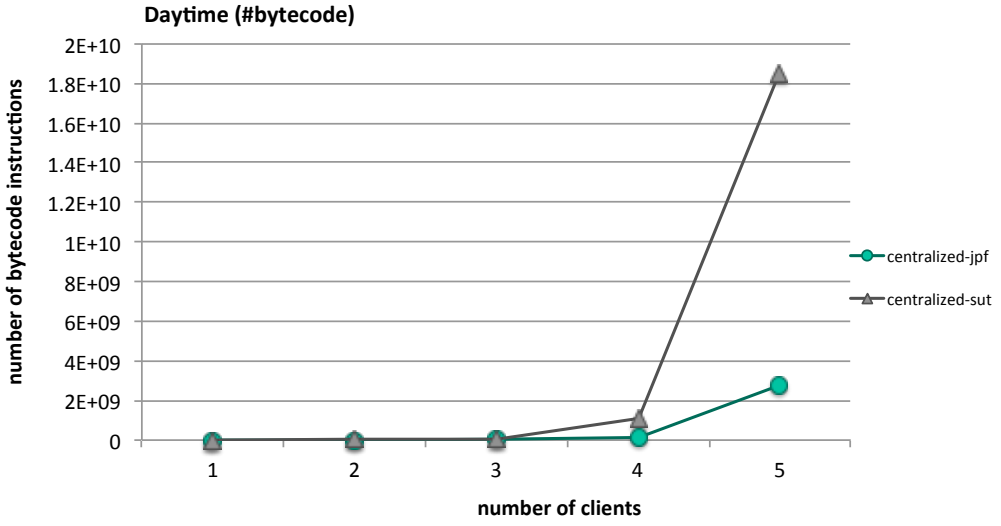

Figure 8.6: Number of bytecode instructions executed when model checking `Daytime` in the centralized-jpf and centralized-sut settings

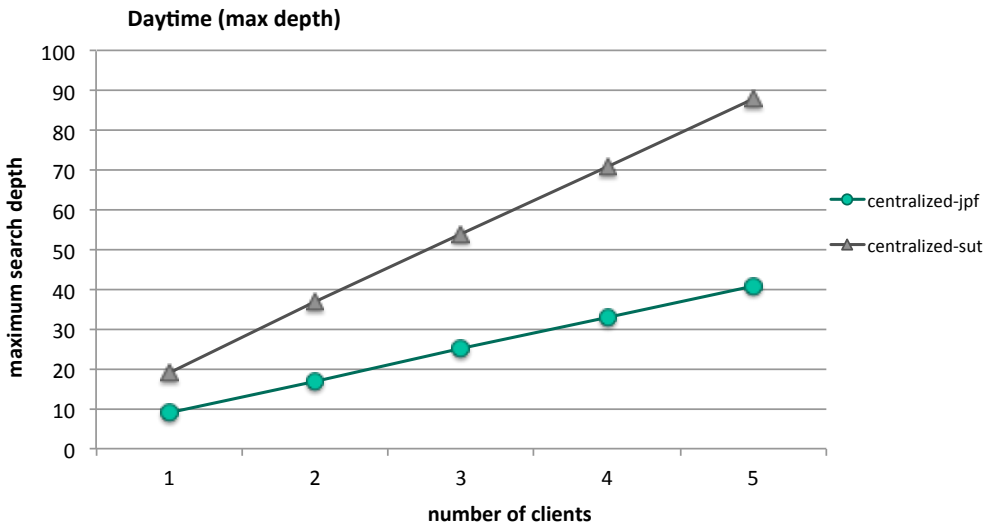

Figure 8.7: The depth of the search tree explored when model checking `Daytime` in the centralized-jpf and centralized-sut settings

8.3 Chat Example

Our next experiment is performed on the `Chat` application. It contains one server that communicates with one or more clients. The server is multithreaded, whereas the clients are single threaded. They use TCP sockets to communicate. In this application, the clients interact with each other through the server, and each interaction between them is handled by a particular thread, referred to as a *worker*. The server main thread listens to incoming connection requests from clients. Once a connection is established with a client, it creates a worker thread to handle interactions with that particular client. This way, more than one client can be served at

the same time. When the worker thread receives a message from the client, it sends it to every other client in the system. The worker threads use a shared array to send a message to the other clients. The array stores references to all the workers. Using these references, the worker obtains the sockets connecting the chat server to each client.

One of the input parameters of `Chat` is the size of the array that keeps references to the worker threads. The maximum number of workers that can simultaneously serve clients is the same as the size of the array. We perform the experiments on `Chat` using different numbers of clients and different sizes for the workers array. Table 8.8 presents the execution times in milliseconds along with the standard deviation obtained from applying both approaches. Overall, our approach exhibits better performance.

In the first experiment presented in Table 8.8, the execution time of the other approach is smaller. However, the standard deviation in this experiment is high compared to the rest of the experiments. For instance, in the centralized-jpf setting, for the first experiment, the standard deviation is about 17% of the average execution time, whereas in the last two experiments performed successfully, the standard deviation is less than 1% of the average execution times. Therefore, it can be concluded that in general, our approach performs better.

The data presented in Table 8.9 is also obtained from model checking `Chat` in the

#clients/#workers	centralized-jpf		centralized-sut		time ratio
	time (ms)	st.dev	time (ms)	st.dev	
1c/1w	160	28	150	10	0.9
2c/1w	1865	49	13952	39	7.5
3c/1w	71146	160	1279186	4038	18
2c/2w	78956	208	722864	2087	9.2
3c/3w	out of memory		out of memory		-

Table 8.8: Execution times obtained from model checking `Chat`

centralized-jpf and centralized-sut settings. The first three experiments presented in the table are performed with one, two and three clients, respectively. In all three of them, a worker array of size one is used. This means that only one client can be served at a time. If there are more clients that need to be served, they wait until the last worker terminates, and a space in the array becomes available for a new worker. The last experiment presented in the table is performed with two clients and a worker array of size two. In this experiment, the maximum number of clients to be served simultaneously is two. When verifying the `Chat` example with three clients and a worker array of size three, the host JVM runs out of memory, due to the infamous state space explosion problem.

It can be seen from Table 8.9 that our approach leads to smaller state spaces and shallower search graphs, and executes less bytecode instructions. In this application, there are two factors that contribute to this difference. As mentioned earlier, one

factor is that, in our approach, the communication buffers are hidden from the model checker, whereas in centralized-sut they exist at the SUT level and therefore, contribute more to the size of the state space.

#clients/#workers	#states			#bytecode			max depth		
	centralized-jpf	centralized-sut	ratio	centralized-jpf	centralized-sut	ratio	centralized-jpf	centralized-sut	ratio
1c/1w	73	192	3	26506	55774	2	14	24	1.7
1c/2w	3148	22841	7	981094	8022851	8	46	77	1.7
1c/3w	93016	1691302	18	28202807	573027106	20	77	123	1.6
2c/2w	139056	1271696	9	30087951	467362721	16	73	101	1.4

Table 8.9: Data obtained from model checking the `Chat` application

Another factor that improves the results obtained from our approach is the POR technique (see Section 5.2.2). This factor did not play a role in the previous examples since they only include single-threaded processes. For operations that do not have any effect outside of the process, for example accessing a shared field, our approach interleaves only the threads that belong to the process by creating a local choice generator. Therefore, an access to the shared array that stores workers in the centralized-sut setting leads to more choices compared to the centralized-jpf setting. In the case of centralized-sut, a global choice generator is created from which all threads in the system, including the clients' main threads, can proceed. In the case of centralized-jpf, a local choice generator is generated from which only the server's main thread and the worker threads can proceed.

Table 8.10 presents information on the choices explored by JPF which is ob-

tained by applying the listener `StateSpaceAnalyzer`. This table includes a section for each experiment. Similar to Table 8.7, the columns *#total* and *#buffer* represent the total number of explored choices and the number of choices created by accessing communication channels. The column *#server* represents the total number of choices created within the server code. This number does not include those choices created to capture network communications. The next column is the ratio of the *#server* value obtained in the centralized-sut setting to the *#server* value obtained in centralized-jpf.

As can be seen from the table, in the majority of the cases, the ratio for *#buffer* is higher than the ratio for *#total*. This implies that the model of communication at the SUT level is leading to larger state spaces. Moreover, in most of the cases, the ratio for *#server* is higher than the ratio for *#total*. This also shows the effectiveness of the POR technique.

setting	1c/1w						2c/1w					
	#total	ratio	#buffer	ratio	#server	ratio	#total	ratio	#buffer	ratio	#server	ratio
centralized-jpf	100	5	21	4.2	24	8.5	5571	14.6	937	24.3	1561	20
centralized-sut	498		89		204		81142		22813		31245	
setting	3c/1w						2c/2w					
	#total	ratio	#buffer	ratio	#server	ratio	#total	ratio	#buffer	ratio	#server	ratio
centralized-jpf	194191	36.1	42950	53.2	62666	41.9	258666	14.7	85664	19.7	98440	12.7
centralized-sut	7004299		2287002		2623778		3807778		1688283		1252509	

Table 8.10: Information on choices generated due to access communication buffers for `Chat`

The graphs presented in Figure 8.8, 8.9, and 8.10 show the scalability of the approaches in terms of the size of the state space, the number of bytecode instructions, and the depth of the search tree for the worker array of size one as the number of clients increases. It can be seen from the graphs that also in this experiment, our approach scales better.

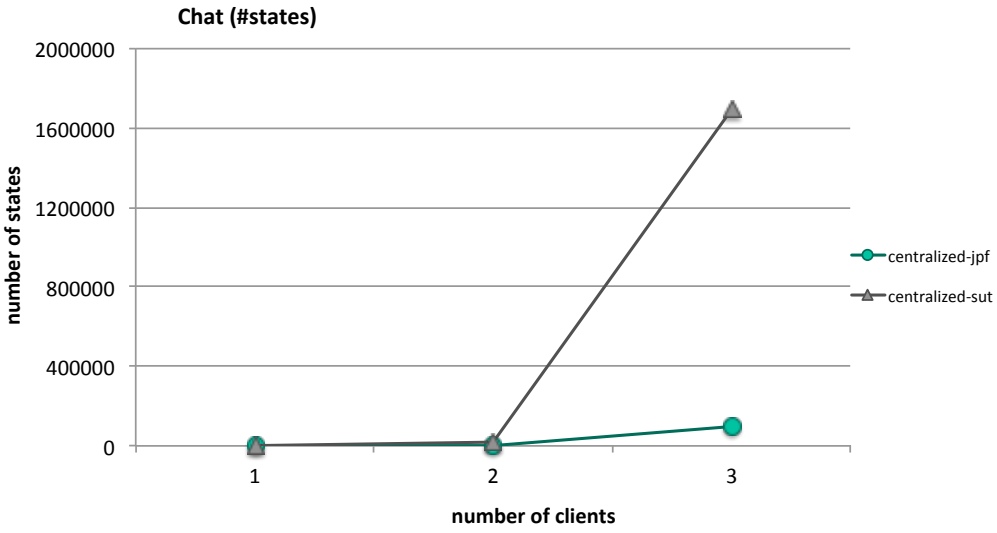

Figure 8.8: Number of states explored when model checking `Chat` in the centralized-jpf and centralized-sut settings

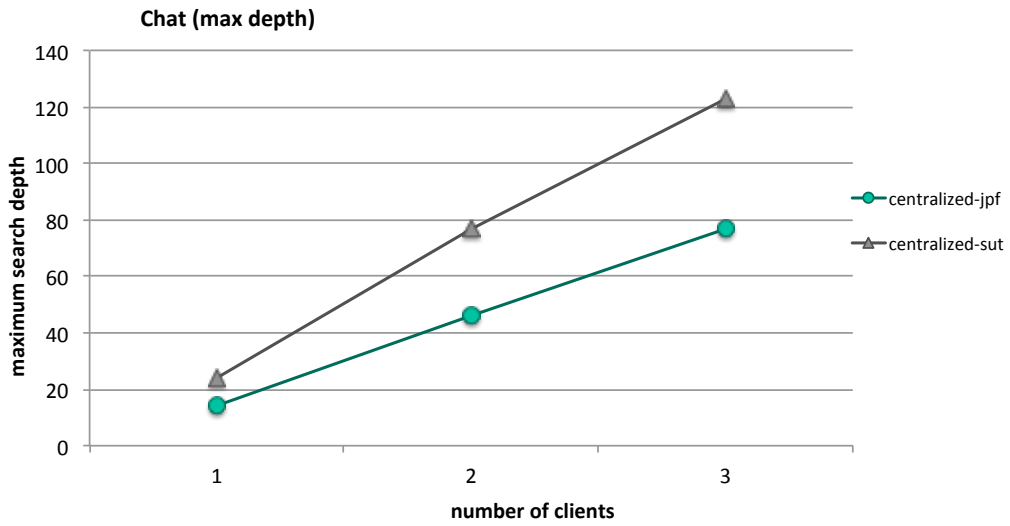

Figure 8.9: Number of bytecode instructions executed when model checking `Chat` in the centralized-jpf and centralized-sut settings

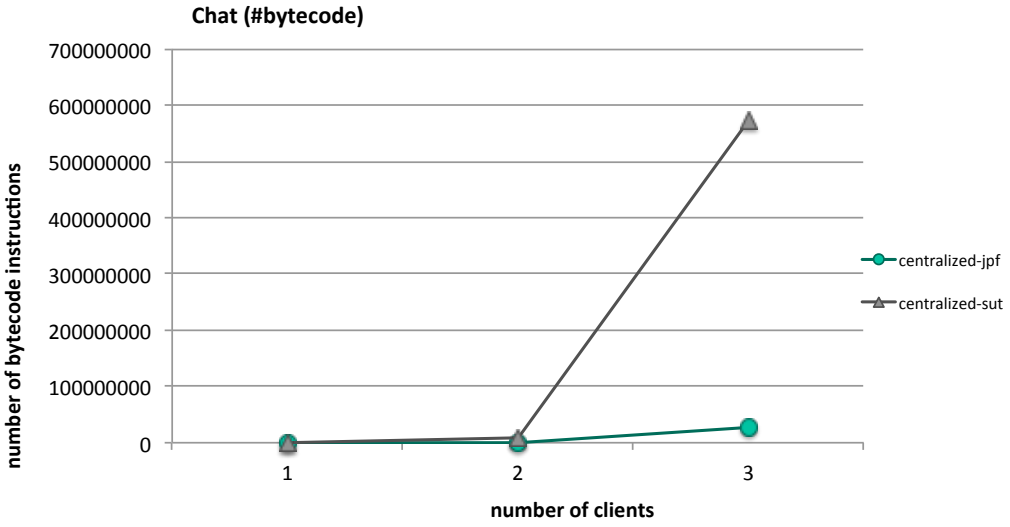

Figure 8.10: The depth of the search tree explored when model checking `Chat` in the centralized-jpf and centralized-sut settings

To precisely observe the effect of our POR technique, we model check `Chat` in the centralized-jpf setting with the POR disabled and compare it with the case in which POR was in effect. With POR disabled, JPF is simply configured to use the default scheduler factory which only allows for global choices. Table 8.11 presents the results from this experiment. It compares the data obtained from running JPF with POR enabled and disabled. In each section, the column *por-on* and *por-off* represent values obtained when POR is enabled and disabled, respectively, and the last column is the ratio of the *por-off* value to the *por-on* value. In all sections, the ratio for the experiment presented in the last row of Table 8.11 is the highest, which implies that experiment is impacted the most by disabling POR. This is what we also expected as this is the only case in which worker threads serve clients simultaneously. Therefore, there are more accesses to the shared array, and consequently, POR plays a more effective role.

#clients/#workers	#states			#bytecode			#total choices			#server choices		
	por-on	por-off	ratio	por-on	por-off	ratio	por-on	por-off	ratio	por-on	por-off	ratio
1c/1w	73	77	1.05	26506	30116	1.14	100	113	1.1	24	32	1.3
1c/2w	3148	3245	1.03	981094	1394858	1.42	5571	6443	1.2	1561	2166	1.4
1c/3w	93016	100270	1.08	28202807	46909151	1.66	194191	239202	1.2	62666	93342	1.5
2c/2w	139056	237985	1.71	30087951	63512124	2.11	258666	500009	1.9	98440	175169	1.8

Table 8.11: Impact of the POR technique applied by the centralization at the model checker level

8.4 Alphabet Example

Our last experiment is performed on the `Alphabet` application. It contains one server which is connected to one or more clients. Both the server and clients are multithreaded. The server and the clients processes use TCP sockets to communicate. A string of digits is associated with each client. The client's goal is to have the server transform a string of digits to a string of letters, for example, the string "123" becomes "ABC". To accomplish that, the client creates pairs of producer/consumer threads. Each pair is responsible for transforming a substring of the client string. The number of the pairs and the size of the substrings are input parameters to the client processes. Each producer/consumer pair is independently connected to the server. The server process creates a worker thread to handle each connection. For every digit in the substring, the producer thread sends a byte representing the digit to the server. The server converts the digit to a letter and sends the result to the consumer thread. Basically, there can exist more than one communication channel between each client and the server, and both the server and the client can handle multiple connections simultaneously.

We applied JPF on the `Alphabet` application in the centralized-jpf and centralized-sut settings. We performed the experiments on `Alphabet` using different sizes of strings transformed by the server, different numbers of producer/consumer

#clients/#pairs/#bytes	centralized-jpf		centralized-sut		time ratio
	time (ms)	st.dev	time (ms)	st.dev	
1c/1p/1b	297	44	5688	19	5.3
1c/1p/5b	3809	33	631938	696	146.6
1c/2p/2b	77995	63	1835485	4757	28.6
2c/1p/1b	320899	894	22438331	67866	163.7

Table 8.12: Execution times obtained from model checking `Alphabet`

pairs per client, and finally different numbers of clients. Table 8.12 presents the execution times in milliseconds along with the standard deviations obtained from applying both approaches. Table 8.13 also presents the results obtained from our experiment. The results obtained for this example also indicate that our approach is superior.

#clients/#pairs/#bytes	#states			#bytecode			max depth		
	centralized-jpf	centralized-sut	ratio	centralized-jpf	centralized-sut	ratio	centralized-jpf	centralized-sut	ratio
1c/1p/1b	597	5688	10	30989	501782	16	22	38	1.7
1c/1p/5b	3809	631938	166	141359	63584724	450	39	74	1.9
1c/2p/2b	77995	1835485	24	3824342	261890278	68	44	74	1.7
2c/1p/1b	320899	41961084	131	15233025	6441690358	423	41	79	1.9

Table 8.13: Data obtained from model checking the `Alphabet` application

In the first experiment presented in both tables, the SUT includes one client with a producer/consumer pair which has the server transform a string of size one. The second experiment in both tables is similar to the first one except the client uses a string of size five. The increase in the ratios shows that increasing the number of

read and write operations on sockets leads to significant overhead in centralization at the SUT level. That shows the impact of keeping the communication buffer at the SUT level.

In the third experiment presented in the tables, the SUT includes one client with two producer/consumer pairs. The clients have the server transform a string of size two. Each producer/consumer pair handles a substring of size one. Finally, in the last experiment, the SUT includes two clients. Each client has a producer/consumer pair which has the server transform a string of size one.

Using the listener `StateSpaceAnalyzer`, we obtained information about choices created to model check `Alphabet` in both settings. The results are presented in Table 8.14. The column *#total* is the total number of choices explored by JPF and the next column is the ratio of total choices explored in centralized-sut to the choices explored in centralized-jpf. The column *#buffer* is the total number of choices created by classes that encapsulate and access communication buffers, and the next column is the ratio of the *#buffer* value obtained in the centralized-sut setting to the *#buffer* value obtained in centralized-jpf. Finally, the column *#prc-internal* represents the total number of choices created within either the server code or the client code. This number does not include those choices created to capture network communications. The next column is the ratio of the *#prc-internal* value obtained in the centralized-sut setting to the *#prc-internal* value obtained in

centralized-jpf.

Similar to what we observed in the previous experiments, the ratios for *#buffer* show that one of the factors that makes our approach more efficient is keeping the communication buffers on the host JVM level. Moreover, comparing the *#prc-internal* choices shows that our POR technique is also another factor that makes our approach superior.

setting	1c/1p/1b						1c/1p/5b					
	#total	ratio	#buffer	ratio	#prc-internal	ratio	#total	ratio	#buffer	ratio	#prc-internal	ratio
centralized-jpf	1173	19	306	7	611	21	8473	289	2826	200	4867	314
centralized-sut	22810		2161		12673		2447731		564057		1528131	
setting	1c/2p/2b						2c/1p/1b					
	#total	ratio	#buffer	ratio	#prc-internal	ratio	#total	ratio	#buffer	ratio	#prc-internal	ratio
centralized-jpf	239224	47	64605	20	142847	46	778147	387	246005	121	413772	423
centralized-sut	11351631		1263643		6502023		301045170		29720826		175099081	

Table 8.14: Information on choices explored by JPF when model checking `Alphabet`

To precisely observe the effect of our POR technique, similar to the previous experiment, we model check `Alphabet` in the centralized-jpf setting with the POR disabled and compare it with the case in which POR was in effect. Table 8.15 presents the results from this experiment. Similar to Table 8.11, the last column of this table outlines the ratio of the *por-off* value to the *por-on* value. In all cases, enabling POR improves the performance considerably. Comparing the ratios in this table with Table 8.11 shows that overall, POR plays a more effective role in the

`Alphabet` experiment than `Chat`. One reason is that in `Alphabet` all processes are multithreaded, whereas in `Chat` only the server process is multithreaded. Therefore, in the `Alphabet` experiment, POR directly impacts all processes by replacing the global choices (created to handle the process internal operations) with local choices.

#clients/#pairs/#bytes	#states			#bytecode			#total choices			#prc-internal choices		
	por-on	por-off	ratio	por-on	por-off	ratio	por-on	por-off	ratio	por-on	por-off	ratio
1c/1p/1b	597	1421	2.4	30989	91132	2.9	1173	3507	3	611	1879	3.1
1c/1p/5b	3809	6665	1.7	141359	399194	2.8	8473	17127	2	4867	9437	1.9
1c/2p/2b	77995	159940	2.1	3824342	12008747	3.1	239224	630202	2.6	142847	389727	2.7
2c/1p/1b	320899	2571817	8	15233025	218932643	14.4	778147	11182101	14.4	413772	7129720	17.2

Table 8.15: Impact of the POR technique applied by the centralization at the model checker level

8.5 Seeding Bugs

This section presents our experiment which includes seeding some of the applications, presented earlier in this section, with errors. We used the same error scenarios used by Artho et al. [80]. In this experiment, the erroneous version of the applications is model checked using both our approach and the other approach. In this experiment, to obtain the shortest trace leading to the error, we configure JPF to use the breadth-first search (BFS). The traces are measured by the number transitions.

In the erroneous version of `Daytime`, the server gets into an infinite loop where at each iteration, it attempts to connect to a client. However, the number of

clients is limited, and thus at some point, the server `accept()` operation leads to a deadlock. Both approaches are able to detect the error when model checking up to four clients. When applying our approach on `Daytime` using five client processes, JPF runs out of memory after exploring 277,428 states. The other approach also runs out memory, with five client processes, after exploring 2,061,398 states. The other approach could examine a lot more states before running out of memory. The reason is that in our approach, a lot of memory is occupied by connections that exist on the host JVM level (see Section 8.6).

Note that in general, exploiting the BFS algorithm requires more memory than DFS, i.e., as presented in Section 8.2, JPF could handle `Daytime` with five clients when using DFS. The depth at which the error was found in each run is presented in Table 8.16. The last column outlines the ratio of the *centralized-sut* value to the *centralized-jpf* value. The ratios show that the length of the error traces obtained from our approach are around half that of the traces obtained from the other approach. In this example, a deadlock occurs after all clients have been connected to the server. Therefore, by increasing the number of clients the size of the error trace increases.

#clients	centralized-jpf	centralized-sut	ratio
1	7	14	2
2	14	29	2.07
3	21	43	2.05
4	28	57	2.04

Table 8.16: Length of error traces obtained from model checking `Daytime`

The traces provided by our approach are shorter and easier to analyze. That can be seen from comparing Figure 8.11 and 8.12 which illustrate the error traces, *trace*1 and *trace*2, produced by JPF in the settings centralized-jpf and centralized-sut respectively. The traces are from the `Daytime` experiment performed with one client. To make understanding the traces easier, we added descriptions on the right side of each figure. In *trace*1, it takes the first two transitions until the connection between the processes is established, whereas in *trace*2, after the first eight transitions, the connection is established. A few of these transitions in *trace*2 are taken by a driver thread, which is required in the centralized-sut setting to create and start the processes of the distributed system. However, the driver thread is not part of the SUT code in its original form, and it is added to the code after it is centralized at the SUT level. Moreover, the communication buffers are shared objects. Since in the centralized-sut these buffers exist at the SUT level, any access to them requires holding a lock which also leads to additional scheduling

Figure 8.11: The error trace produced by JPF when verifying `Daytime` in the centralized-jpf setting

points and thus additional transitions in the trace.

The `Chat` application is also seeded with two different errors, which only appear if there are more than one worker serving clients simultaneously. The `Chat` application used in this experiment includes two clients and the workers array of size two. Both approaches detect the errors. The first error causes the application to throw an instance of `NullPointerException`. In the first erroneous version of `Chat`, creating the output stream of workers is moved out of the `Worker` constructor. Therefore, before one worker creates its output stream, the other worker may

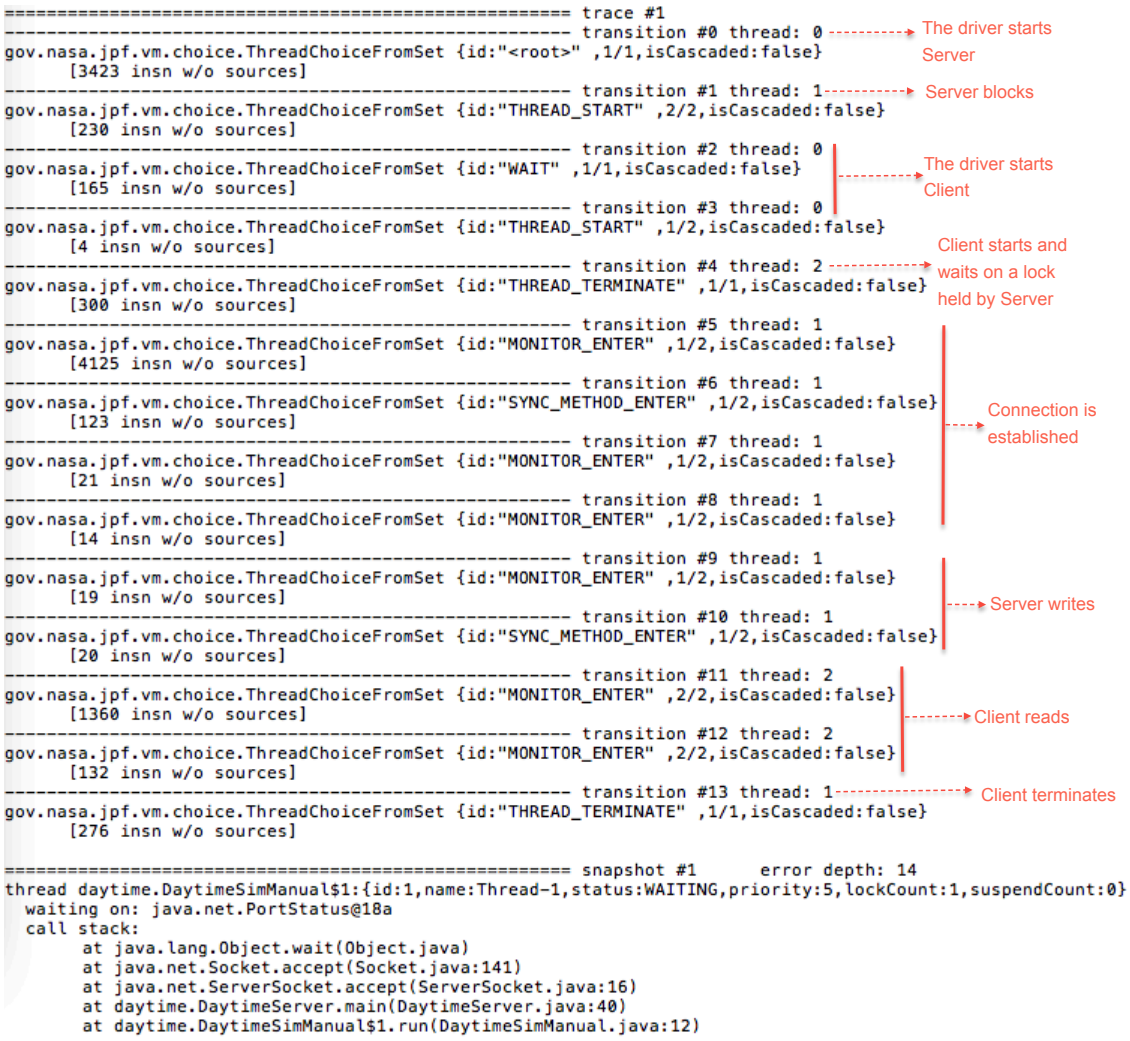

Figure 8.12: The error trace produced by JPF when verifying `Daytime` in the centralized-sut setting

attempt to access the reference to the stream which is null at this point. Using our approach, the path to the error is of depth 10, and using the other approach, the path to the error is of depth 18.

The second error seeded to `Chat` is a race condition which allows both worker threads to access the workers array at the same time. Once a worker is done serving a client, it removes itself from the array. Consider a scenario in which one worker, w_1, checks the array to see if the other worker, w_2, exists. The array element containing w_2 is not null at this point, and w_1 is going to obtain the w_2 socket to send a message to the w_2 client. But before w_1 attempts to do so, w_2 finishes serving its client and removes itself from the array. Then, the w_1 attempt to access the w_2 socket leads to throwing an instance of `NullPointerException`. Our approach detects the error at depth 24, whereas the other approach detects the error at depth 34. Therefore, in the case of both errors seeded to `Chat`, our approach makes analyzing the error trace easier for the user.

The erroneous version of the `Alphabet` application includes an execution that leads to an assertion violation error. Both approaches are able to detect the bug. Each approach is applied using the same configurations used in our `Alphabet` experiment (see the first column of Table 8.12). Using our approach, the path to the error is of depth 5 in all configurations. Using the other approach, the path to the error is of depth 9 in all configurations. This example also shows that our approach

makes it easier for the user to analyze the error scenarios.

8.6 Memory Analysis

To compare the memory consumption of the approaches, we use a tool called VisualVM [22]. VisualVM is a Java profiler that can be used to monitor memory and CPU usage of the application. We used VisualVM to monitor JPF when using our approach and the other centralization approach to model check the distributed applications presented in this chapter. Table 8.17 presents the maximum amount of heap memory, in megabytes, occupied by JPF during the entire model checking process. The last column outlines the ratio of the centralized-sut value to the centralized-jpf value.

application	parameters	centralized-jpf	centralized-sut	ratio
`Echo`	10 echos	63	102	1.6
`Daytime`	4c	547	225	0.4
`Chat`	2c1w	299	1185	4
`Chat`	2c2w	2309	3278	1.4
`Alphabet`	1c2p2b	216	241	1.1

Table 8.17: Maximum amount of memory, in megabytes, occupied by JPF during the entire model checking process

The results show that, in all cases except `Daytime`, our approach does better

[22] http://visualvm.java.net

than the other approach. We took several snapshots of the Java heap during model checking `Daytime` using our approach. The snapshots show that `StateExtensionListener` contributes a lot to the memory, i.e., on average about 80% of the memory is occupied by the `StateExtensionListener` object. As explained in Section 5.2.1, this object is used by the connection manager to keep the state of connections in synchronization with the state of the SUT as JPF backtracks. Each time JPF has reached a new state, a deep copy of the current list of connections is added to the `StateExtensionListener` object. To improve our approach, we are considering using persistent data structures [100] to store the states of connections.

The graphs presented in Figure 8.13 and 8.14 show heap memory consumption over time when model checking `Daytime` using our approach and the other approach, respectively. These graphs are automatically generated by VisualVM. Note that these graphs use different scales, and they are included to show the pattern of memory usage. The similar graphs are presented in Figure 8.15 and 8.16 for `Chat` when running with two clients and the workers array of size two.

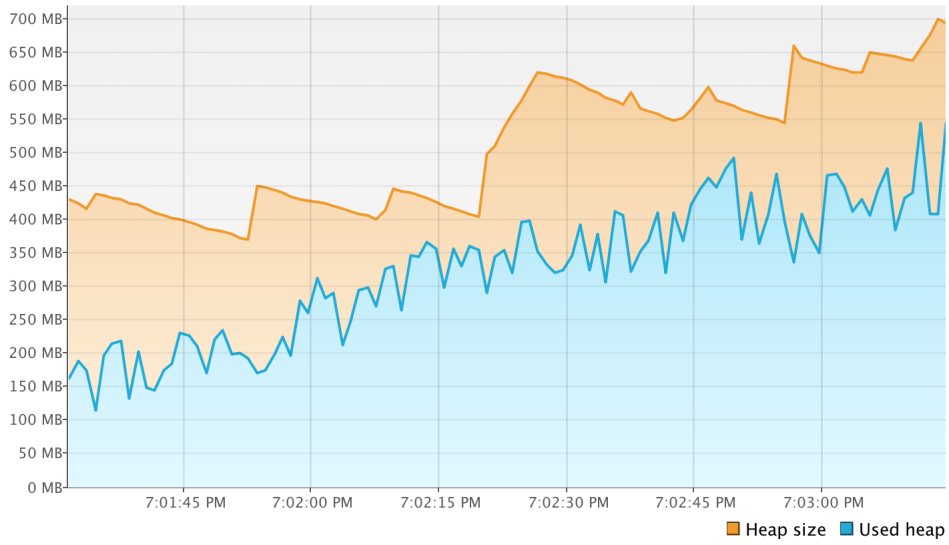

Figure 8.13: Heap memory occupied by JPF when model checking `Daytime` using our approach

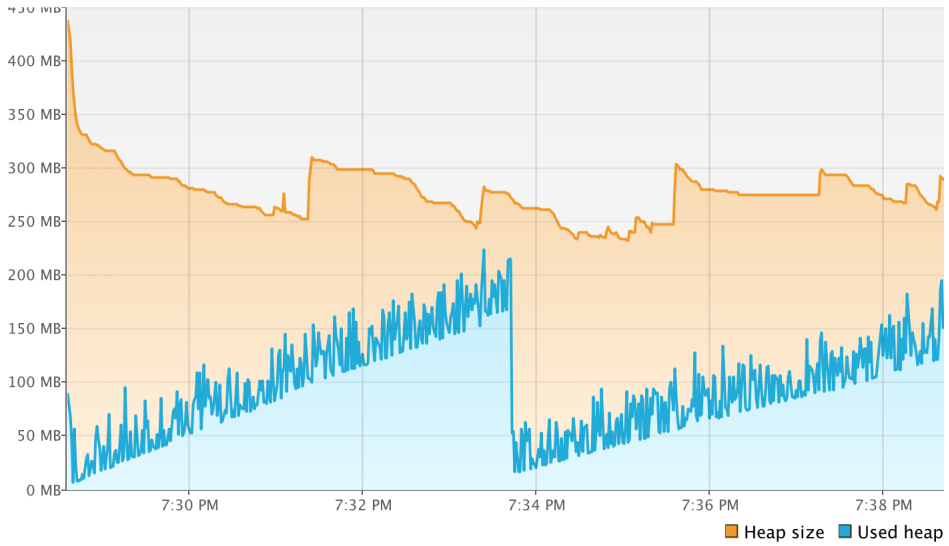

Figure 8.14: Heap memory occupied by JPF when model checking `Daytime` in centralized-sut setting

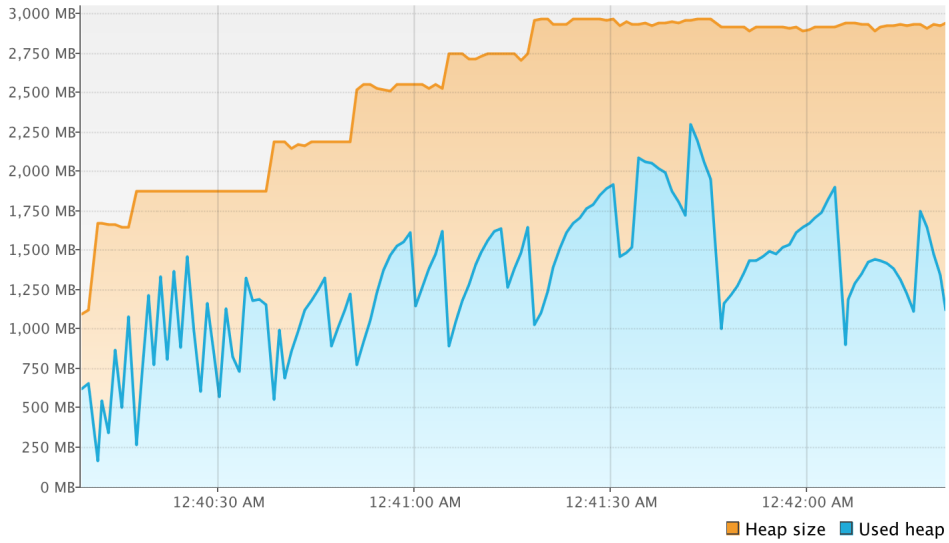

Figure 8.15: Heap memory occupied by JPF when model checking `Chat` using our approach

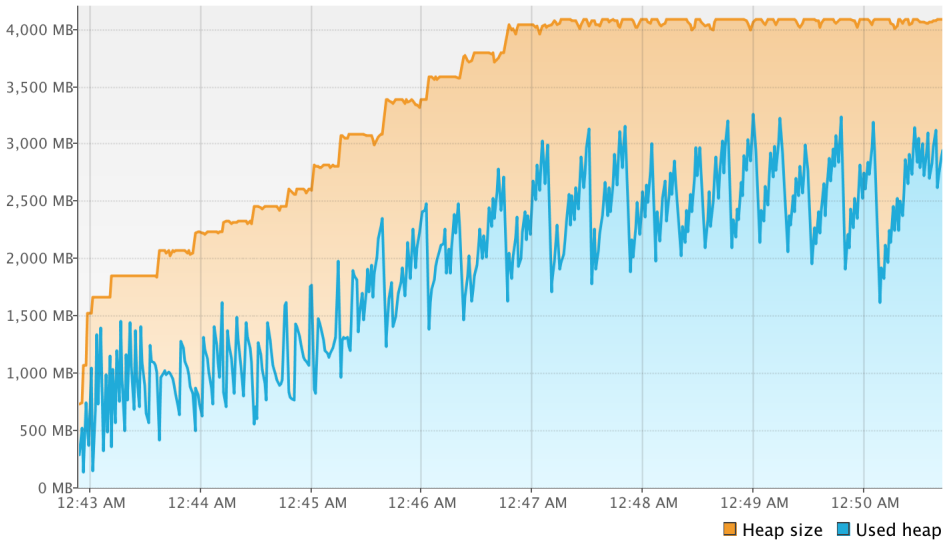

Figure 8.16: Heap memory occupied by JPF when model checking `Chat` in centralized-sut setting

9 Conclusion

In this dissertation, we presented our work on verifying distributed multithreaded applications. We specifically focused on the model checking technique. We extended an existing Java model checker, JPF, which can verify single process applications, to verify distributed applications composed of multiple multithreaded processes.

The first step of our work was expanding the infrastructure of the model checker to accept multiple processes. To achieve that, we used an instance of the centralization technique, applied at the model checker level, in which processes of a distributed application are mapped to communicating groups of threads within the model checker. Processes within a distributed application do not share memory. The main challenge of applying centralization is to prevent different processes from sharing the same data. We addressed this by proposing a novel technique: introducing a new class-loading model within the model checker.

Processes in distributed applications have distinct execution environments. To provide that, our centralization technique mapped the threads within a process to

an object within JPF that uniquely identifies the process. By representing each process using this unique object, along with a main thread and a distinct hierarchy of class loaders, our approach provided a distinct execution environment for each process. Moreover, as part of our centralization technique, we implemented a new JVM within JPF that is able to execute distributed applications. This multiprocess JVM uses new thread scheduling policies which are specific to distributed applications. Finally, we proposed a model for finalizers which has been added to the multiprocess JVM (as well as the single process JVM) of JPF.

Our centralization approach is superior to existing centralization approaches. Some of the main advantages over other centralization techniques, which are applied at the SUT level, are keeping types within standard Java libraries separated for different processes and handling SUTs which use Java's reflection API. Unlike the existing centralization technique at the OS level, our approach does not take the state of the OS into account, which reduces the size of the state space.

Centralization provides the basic building blocks for JPF to capture distributed applications, but it does not capture interactions between processes. To address this, we implemented a model of communication between processes as an extension of JPF, named jpf-nas. At this stage, jpf-nas supports Java processes that communicate via TCP sockets, and models the package `java.net`. It consists of two main components: a connection manager and a scheduler. The former maintains

communication channels along the current execution path, and the latter captures scheduling points upon network interactions. Since the communication channels exist at the host JVM level, JPF does not take their state into account when backtracking and matching states. The former issue was addressed by using a listener maintaining a map from states to lists of connections. The latter issue was addressed by including fields that store hash values of connections at the SUT level. Our model of communication within jpf-nas relies on the finalizer model to address the shutdown semantics of processes by cleaning up corresponding communication channels.

We proposed a POR technique which has been implemented in jpf-nas. The technique distinguishes between two types of choice generators: local and global. Local choice generators are used to interleave threads within one process, whereas global choice generators are used to interleave processes within the distributed SUT. Our POR approach only allows for creating a global choice generator if the operation can have an effect outside of the process. The results showed that POR can significantly impact the performance, e.g. in one case, by disabling POR, the state space became eight times larger (Table 8.15). One of the main challenges of developing distributed systems is dealing with errors occurring at levels, such as the hardware level, which are hidden from the code. To verify the SUT against such errors, we added a mechanism which injects such failures to the SUT by including

choices associated with interrupting the execution flow.

We also provided a way to capture interactions of processes centralized within JPF with external resources such as cloud computing services. This is useful if one is not interested in checking the states of the external resources. We have implemented this functionality as a JPF extension called jpf-nhandler which was originally developed to handle native calls. It is based on delegating calls from the JPF level to the host JVM level. The approach is generic, as it is not tied to certain communication means.

The results, presented in Chapter 8, confirmed that our work provides a more accurate IPC model to capture interactions between processes, for example, it detected a bug which was not detected using the recent work on centralization at the SUT level [84]. Moreover, the results showed that, overall, our implementation choices along with our POR technique led to significantly better performance and scalability.

10 Future Work

One of our future plans is to study whether the POR approach (presented in Chapter 6) preserves additional properties such as assertion violations. We are also planning to improve the POR algorithm to eliminate more redundancies from the state space of distributed SUTs. As mentioned in Section 6.3, in global states at which processes communicate, our POR algorithm explores the set of all enabled transitions. We are planning to use the independence relation between global transitions performed on disjoint communication objects to explore a subset of transitions enabled from the global state instead of all enabled transitions.

In addition, our future work includes modeling the new I/O (NIO) [6] introduced into JDK 1.4. NIO provides scalable I/O by supporting non-blocking I/O operations and *multiplexing*. Multiplexing allows one thread to manage multiple communication channels simultaneously using an event notification mechanism. Most of the existing Java APIs, supporting distributed computing, are based on NIO. One such application, to which we are planning to apply our approach, is

Zookeeper[23]. Zookeeper is a widely used open-source server which is written in Java and provides reliable distributed coordination. We are also planning to apply the approach on JGroups[24] which also relies on NIO. JGroups is a reliable multicast system which is written in Java.

Another future plan is to extend the scope of our approach to a different type of VM, Dalvik, which executes Android applications [101]. To achieve this, we are going to use Pathdroid, which is a model checker for binary Android applications. It shares the same engine as JPF, but it implements its own instruction factory. Pathdroid uses a class loading model similar to JPF's, but at the low level it deals with different types of binary files, which are of type `dex`. Our goal is to be able to verify distributed systems composed of communicating Java and Android processes. Android provides its own version of an IPC protocol [101]. To capture communication between Android processes, we are going to extend jpf-nas with the Android IPC model.

Finally, we are planning to look into the PhoneSat project[25] which is an ongoing project at NASA, started in 2009. This project includes building nanosatellites which are controlled by smartphones and launched into a low orbit around Earth. We are particularly interested in the coordination problem of multiple nanosatel-

[23] http://zookeeper.apache.org

[24] http://www.jgroups.org

[25] http://www.phonesat.org

lites. PhoneSat is written in Java, but relies on native libraries. We believe jpf-nhandler can be very useful in verifying this system since one of the primary challenges is handling native calls.

Bibliography

[1] P. Liggesmeyer, M. Rothfelder, M. Rettelbach, and T. Ackermann. Qualitätssicherung Software-basierter technischer Systeme. *Informatik Spektrum*, 21(5):249–258, October 1998.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[3] Kunle Olukotun and Lance Hammond. The Future of Microprocessors. *Queue*, 3(7):26–29, September 2005.

[4] Valentino Lee, Heather Schneider, and Robbie Schell. *Mobile Applications: Architecture, Design, and Development*. Prentice Hall, 2004.

[5] Jim Farley. *Java Distributed Computing*. O'Reilly, 1998.

[6] Esmond Pitt. *Fundamental Networking in Java*. Springer, 2010.

[7] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology, http://www.nist.gov/itl/cloud/upload/cloud-def-v15.pdf, 2009.

[8] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, March 1953.

[9] Glenford J. Myers. *The Art of Software Testing*. Wiley, second edition, 2004.

[10] Edsger W. Dijkstra. Notes on structured programming. Technical report, Technological University Eindhoven, Department of Mathematics, The Netherlands, 1970.

[11] Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 169–186. The MIT Press, 2000.

[12] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[13] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504, Copenhagen, Denmark, July 2002. Springer.

[14] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.

[15] Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime Assurance Based On Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 279–287, Las Vegas, NV, USA, June/July 1999. CSREA Press.

[16] Klaus Havelund and Grigore Rosu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 200–217. Elsevier, October 2001.

[17] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Formal Methods in System Design*, 24(2):129–155, March 2004.

[18] Patrick Cousot. Abstract Interpretation. *ACM Computing Surveys*, 28(2):324–328, March 1996.

[19] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[20] Francesco Logozzo. Cibai: an abstract interpretation-based static analyzer for modular analysis and verification of Java classes. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 283–298, Nice, France, 2007. Springer.

[21] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.

[22] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 219–232, Vancouver, British Columbia, Canada, June 2000. ACM.

[23] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 56–69, Tampa Bay, FL, USA, October 2001. ACM.

[24] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, Seattle, Washington, USA, November 2002. ACM.

[25] Kenneth L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, 1993.

[26] Joseph Y. Halpern and Moshe Y. Vardi. Model Checking vs. Theorem Proving: A Manifesto. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 325–334, Cambridge, MA, USA, April 1991. Morgan Kaufmann.

[27] Derek Bruening and John Chapin. Systematic testing of multithreaded programs. Technical report, Massachusetts Institute of Technology, Cambridge, MA, 2000.

[28] Madan Musuvathi and Shaz Qadeer. CHESS: Systematic Stress Testing of Concurrent Software. In *Proceedings of the 16th International Symposium on Logic-based program synthesis and transformation*, volume 4407 of *Lecture Notes in Computer Science*, pages 15–16, Venice, Italy, 2006. Springer.

[29] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, New York, NY, USA, January 2002. ACM.

[30] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: a model checker for concurrent software. In *Proceedings of the 16th International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 484–487, Boston, MA, USA, July 2004. Springer.

[31] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[32] Shoham Ben-David, Cindy Eisner, Daniel Geist, and Yaron Wolfsthal. Model Checking at IBM. *Formal Methods in System Design*, 22(2):101–108, August 2003.

[33] Carl-Johan H. Seger, Robert B. Jones, John W. O'Leary, Thomas F. Melham, Mark Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, 2005.

[34] Klaus Havelund, Michael R. Lowry, and John Penix. Formal Analysis of a Space-Craft Controller Using SPIN. *IEEE Transactions of Software Engineering*, 27(8):749–765, August 2001.

[35] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[36] H. R. Andersen. *An introduction to binary decision diagrams*. Lecture Notes for 49285 Advanced Algorithms E97. Department of Information Technology, Technical University of Denmark, October 1997.

[37] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207, Amsterdam, The Netherlands, March 1999. Springer.

[38] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[39] Cindy Eisner and Doron Peled. Comparing symbolic and explicit model checking of a software system. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, Lecture Notes in Computer Science, pages 230–239, London, UK, 2002. Springer.

[40] David Y. W. Park, Ulrich Stern, Jens U. Skakkebæk, and David L. Dill. Java Model Checking. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 253–256, Grenoble, France, September 2000. IEEE.

[41] Patrice Godefroid. Model checking for programming languages using Verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, Paris, France, January 1997. ACM.

[42] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: a pragmatic approach to model checking real code. *SIGOPS Operating Systems Review*, 36(SI):75–88, December 2002.

[43] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. ACM.

[44] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[45] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 342–354, Albuquerque, New Mexico, USA, 1992.

[46] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–535, May 1995.

[47] Rajeev Alur and Thomas A. Henzinger. Local liveness for compositional modeling of fair reactive systems. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 166–179, Liege, Belgium, July 1995.

[48] Dimitra Giannakopoulou, Corina S. Pasareanu, and Jamieson M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *Proceeding of 26th International Conference on Software Engineering*, pages 211–220, Edinburgh, United Kingdom, May 2004. IEEE Computer Society.

[49] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the 3rd International Workshop on Computer Aided Verification*, pages 332–342, Aalborg, Denmark, July 1991.

[50] Gerard J. Holzmann, Patrice Godefroid, and Didier Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proceedings of the IFIP TC6/WG6.1 12th International Symposium on Protocol Specification, Testing and Verification*, pages 349–363, Lake Buena Vista, Florida, USA, June 1992.

[51] Patrice Godefroid and Pierre Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1994.

[52] Robert P. Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. Static partial order reduction. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 345–357, Lisbon, Portugal, March/April 1998.

[53] Cormac Flanagan and Patrice Godefroid. Proceedings of the 32nd acm sigplan-sigact symposium on dynamic partial-order reduction for model checking software. In *Symposium on Principles of Programming Languages*, pages 110–121, Long Beach, California, USA, January 2005.

[54] Robert Netzer and Barton Miller. What are race conditions?: some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[55] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[56] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: Efficiently computing the happens-before relation using locksets. In *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 193–208, Seattle, WA, USA, August 2006. Springer.

[57] Jeremy Manson, William Pugh, and Sarita Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–391, Long Beach, CA, USA, January 2005.

[58] Edith Schonberg. On-the-fly detection of access anomalies. *ACM SIGPLAN Notices*, 39(4):313–327, April 2004.

[59] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on*

Operating Systems Principles, pages 237–252, Bolton Landing, NY, USA, October 2002. ACM.

[60] Cormac Flanagan and Stephen Freund. Type inference against races. In *Proceedings of the 11th International Symposium on Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 116–132, Verona, Italy, August 2004. Springer.

[61] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, Ottawa, Canada, June 2006. ACM.

[62] Polyvios Pratikakis, Jeffrey Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–331, Ottawa, Canada, June 2006. ACM.

[63] Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 1–13, Washington, DC, USA, June 2004. ACM.

[64] Klaus Havelund. Java PathFinder, A Translator from Java to Promela. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, page 152. Springer, 1999.

[65] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A Deadlock Detection Tool for Concurrent Java Programs. *Software Practice and Experience*, 29(7):577–603, June 1999.

[66] Gerard Holzmann. *The SPIN model checker: primer and reference manual*. Addison-Wesley Professional, 2004.

[67] Claudio Demartini, Radu Iosif, and Riccardo Sisto. dSPIN: A Dynamic Extension of SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 261–276. Springer, 1999.

[68] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In

Stephen A. MacKay and J. Howard Johnson, editors, *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, page 13, Mississauga, ON, Canada, November 1999. IBM.

[69] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 267–276, Helsinki, Finland, September 2003. ACM.

[70] Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 179–180, Antwerp, Belgium, 2010. ACM.

[71] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. JST: an automatic test generation tool for industrial Java applications with strings. In *Proceedings of the 35th International Conference on Software Engineering*, pages 992–1001, San Francisco, CA, USA, May 2013. IEEE.

[72] Radu Iosif and Riccardo Sisto. Using garbage collection in model checking. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 20–33, London, UK, 2000. Springer.

[73] Sergey Kulikov, Nastaran Shafiei, Franck van Breugel, and Willem Visser. Detecting Data Races with Java PathFinder. Available at: http://www.cse.yorku.ca/~franck/research/drafts/race.pdf, 2010.

[74] Kristal Pollack. Extending IMP to support threads with race detection by model checking. Available at: http://classes.soe.ucsc.edu/cmps203/Winter13/sample-kristal-final.pdf, 2004.

[75] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley, Boston, MA, USA, first edition, 1999.

[76] Cyrille Artho, Watcharin Leungwattanakit, Masami Hagiya, and Yoshinori Tanabe. Efficient model checking of networked applications. In *Proceedings of the 46th International Conference on Objects, Components, Models and Patterns*, volume 11 of *Lecture Notes in Business Information Processing*, pages 22–40, Zurich, Switzerland, June/July 2008. Springer.

[77] Cyrille Artho, Watcharin Leungwattanakit, Masami Hagiya, Yoshinori Tanabe, and Mitsuharu Yamamoto. Cache-based model checking of networked

applications: From linear to branching time. In *Proceedings of the 27th International Conference on Automated Software Engineering*, pages 447–458, Auckland, New Zealand, November 2009. IEEE.

[78] Watcharin Leungwattanakit, Cyrille Artho, Masami Hagiya, Yoshinori Tanabe, and Mitsuharu Yamamoto. Model checking distributed systems by combining caching and process checkpointing. In *Proceedings of the 26th IEEE/ACM International Conference on International Conference on Automated Software Engineering*, pages 103–112, Lawrence, KS, USA, November 2011. IEEE.

[79] Scott D. Stoller and Yanhong A. Liu. Transformations for model checking distributed Java programs. In *Proceedings of the 8th International SPIN Workshop on Model checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 192–199, Toronto, Canada, May 2001. Springer.

[80] Cyrille Artho and Pierre-Loic Garoche. Accurate centralization for applying model checking on networked applications. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 177–188, Tokyo, Japan, September 2006. IEEE.

[81] Elliot Barlas and Tevfik Bultan. Netstub: a framework for verification of distributed Java applications. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 24–33, Atlanta, Georgia, USA, November 2007. ACM.

[82] Yoshihito Nakagawa, Richard Potter, Mitsuharu Yamamoto, Masami Hagiya, and Kazuhiko Kato. Model checking of multi-process applications using SBUML and GDB. In *Proceedings of Workshop on Dependable Software: Tools and Methods*, pages 215–220, Yokohama, Japan, 2005.

[83] Richard Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, 2002.

[84] Lei Ma, Cyrille Artho, and Hiroyuki Sato. Analyzing distributed Java applications by automatic centralization. In *Proceedings of the 37th Annual International Computer Software and Applications Conference*, pages 691–696, Kyoto, Japan, July 2013. IEEE.

[85] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proceedings of the 23th International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium*, pages 1–12, Rome, Italy, May 2009. IEEE.

[86] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36–44, Vancouver, Canada, October 1998. ACM Press.

[87] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Addison-Wesley, 2005.

[88] T. Jensen, D. Le Métayer, and T. Thorn. Security and dynamic class loading in Java: A formalisation. In *In Proceedings of the 1998 International Conference on International Conference on Computer Languages*, pages 4–15, Chicago, IL, USA, May 1998. IEEE.

[89] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. In *In Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 325–336, Minneapolis, MN, USA, October 2000. ACM.

[90] Akihiko Tozawa and Masami Hagiya. Formalization and Analysis of Class Loading in Java. *Higher-Order and Symbolic Computation*, 15(1):7–55, 2002.

[91] Nastaran Shafiei and Peter C. Mehlitz. Modeling class loaders in Java PathFinder version 7. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, November 2012.

[92] Nastaran Shafiei and Peter C. Mehlitz. Extending JPF to verify distributed systems. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–5, January 2014.

[93] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[94] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Java SE 7 edition, 2013.

[95] Marcelo d'Amorim, Ahmed Sobeih, and Darko Marinov. Optimized execution of deterministic blocks in Java PathFinder. In Zhiming Liu and Jifeng He, editors, *Proceedings of the 8th International Conference on Formal Engineering Methods*, volume 4260 of *Lecture Notes in Computer Science*, pages 549–567, Macao, China, November 2006. Springer-Verlag.

[96] Milos Gligoric and Rupak Majumdar. Model checking database applications. In Nir Piterman and Scott A. Smolka, editors, *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 549–564, Rome, Italy, March 2013. Springer-Verlag.

[97] Lieven Eeckhout, Andy Georges, and Koenraad De Bosschere. How java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 169–186, Anaheim, California, USA, October 2003. ACM.

[98] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 57–76, Montreal, Quebec, Canada, October 2007. ACM.

[99] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[100] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Proceedings of the 18th ACM Symposium on Theory of Computing*, pages 109–121, Berkeley, California, USA, May 1986. ACM.

[101] Marko Gargenta. *Learning Android*. O'Reilly, 2011.