Software Model Checking for Internet Protocols with Java PathFinder

Jesús Martínez and Cristóbal Jiménez

Departamento de Lenguajes y Ciencias de la Computación
University of Málaga, Spain

Abstract. Java is one of the most popular languages used to build complex and distributed systems. The existence of high-level libraries and middleware makes it now easy to develop applications for enterprise information systems. Unfortunately, implementing distributed software is always an error-prone task. Thus, middleware and application protocols must guarantee different functional and non-functional properties, which has been the field usually covered by tools based on formal methods. However, analyzing software is still a huge challenge for these tools, and only a few can deal with software complexity. One such tool is the Java Pathfinder model checker (JPF). This paper presents a new approach to the verification of Java systems which communicate through Internet Sockets. Our approach assumes that almost all the middleware and network libraries used in Java rely on the protocols available at the TCP/IP transport layer. Therefore, we have extended JPF, now allowing developers to verify not only single multi-threaded programs but also fully distributed Socket-based software.

1 Introduction

Model checking [1, 2] is a mature technique for dealing with complex problems within distributed systems. In contrast to some other existing approaches for verifying communication systems, model checking is an automatic process that normally returns a simple but clear verdict to questions about functional or non-functional properties of the system being analyzed, such as safety, liveness, performance or probability, among others. When a property is not satisfied, the model checker presents a trail of execution steps which lead users directly to the *counterexample*.

Analyzing a concurrent/distributed system with model checking requires the creation of an abstract description (model) of the system with the critical behavior to be analyzed. We also need to specify a set of verification properties using a property-oriented language. The model checking algorithm will produce a reachability graph including all the execution paths for the model in order to check whether these paths satisfy the properties. It is worth noting that the system model must be closed, and the behavior of the environment must be provided in order to fully analyze it. In the context of software applications, this environment consists of the underlying platform on which the software runs, but also includes the networking mechanisms used to communicate distributed entities.

Martínez J. and Jiménez C. (2008).
Software Model Checking for Internet Protocols with Java PathFinder.
In *Proceedings of the 6th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems*, pages 91-100
DOI: 10.5220/0001738600910100

```
import java.io.*;
import java.net.*

public class Client{
  public static void main (String [] args){
    int serverPort = 6666;
    int c;
    try {
      InetAddress ia= InetAddress.getLocalHost();
      Socket sd = new Socket(ia,serverPort);
      InputStreamReader in=  sd.getInputStream();
      while( (c = in.read()) != -1)
      System.out.print((char)c);
      sd.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

```
import java.io.*;
import java.net.*;

public class BasicServer  {
  public static void main (String [] args){
    int port = 6666;
    try {
      ServerSocket ss = new ServerSocket(port);
      while(true) {
        Socket newsd = ss.accept();
        OutputStreamWriter out =
          new OutputStreamWriter (newsd.getOutputStream());
        out.write("Hello World of Sockets!");
        newsd.close();
      }
    }catch( IOException ie ) {
      ie.printStackTrace();
    }
  }
}
```

Fig. 1. Basic examples using Java Sockets: a Client (left) and a Server (right).

Unfortunately, one of the main disadvantages of model checking has been the traditional need for in-depth knowledge of formal methods. Normally, software developers looking for a model checking tool want to consider it a kind of *smart debugger*. However, verifying software directly with model checking (avoiding the creation of the formal model mentioned previously) is a challenging task. In recent years, there has been a considerable effort to make this technique available to developers and designers outside the academic world (p.e. Microsoft' SLAM [3], Berkeley's BLAST [4] or NASA's JPF [5, 6]).

JPF is the first software model checker for Java. Started in 1999 and now released as open source, the tool itself is a Java application which acts as an explicit state model checker for verifying executable Java bytecode programs. JPF explores all potencial execution paths of a Java program in order to find violations of properties such as deadlocks or unhandled exceptions. It includes support for part of the Java API (1.6 and prior versions) [7], making it possible to analyze code without further instrumentation. Moreover, JPF is easy to configure and extend, it being an appropriate platform to explore new methods to model check distributed systems.

This paper presents a novel approach for verifying Java software which communicates through Internet Sockets. We found that JPF did not support the Java networking API. Moreover, its typical usage included only one executable program at a time. Therefore, we have extended JPF, now allowing developers to verify not only a single multi-threaded program but also fully distributed Socket-based software following the Client/Server architecture (with at least two executable programs). Our approach has included TCP/IP networking support to the model checker, making it possible in the future to analyze some other middleware which usually rely on these protocols, such as Java Enterprise applications or Web Services.

The paper is organized as follows. Section 2 introduces the different Java APIs which implement the Client/Server model. Section 3 is focused on the extensible JPF model checker for Java and our proposed solution to deal with Client and Server code. Finally, Section 4 presents our conclusions and lines of future work.

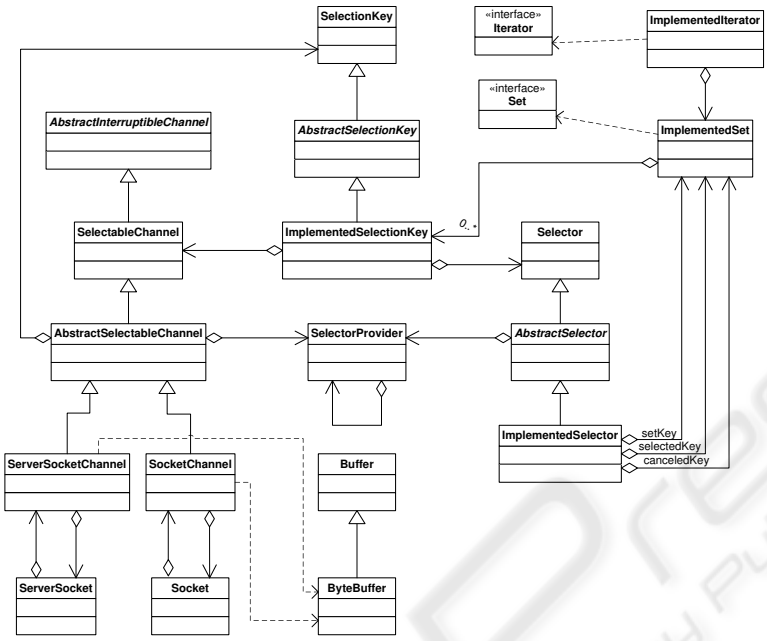

Fig. 2. The Java Socket API after including java.nio.

2 The Java Client/Server Model

From its inception, the Java API has supported Berkeley Sockets. The design of its `java.net` package was focused on avoiding ill usage which was common with the original C library. Thus, object oriented programming provided the kind of type safety not present in the UNIX API. Moreover, Java split the behavior of Sockets for Clients and Servers into two classes: the Socket and ServerSocket. Perhaps one of the main differences between the Java approach and the original C API was the decoupling of the entities responsible for managing connections (the Socket and ServerSocket classes) and for exchanging data over the network (the Input and Output Streams). Fig. 1 shows the basic code for TCP Clients and Servers. The left part shows the Client code, where the Socket constructor includes the classic socket definition and the connection procedures. Data is read from its associated InputStream. The right part in fig. 1 shows an iterative Server, where the ServerSocket constructor defines and binds the socket to a port and then creates a connection backlog (equivalent to the `listen` C primitive). When a new connection is accepted, the server writes some bytes to its associated OutputStream and then returns back to attend a new client. This simple scheme may serve one client at a time although it is also common to spawn a thread in order to service clients in parallel.

The original `java.net` API lacked one of the most used techniques when building short-lived servers: demultiplexing I/O. This technique avoids spawning a new thread for each new service, using instead a single execution thread to serve clients simultaneously. This functionality was added in version 1.4, which included the new `java.nio` API including the Selector framework (implementing the native OS mechanism for demultiplexing I/O). Unfortunately, this framework achieved backward compatibility at the expense of complicating the class structure existing in `java.net`. The new API is partially shown in fig. 2, where some new elements are depicted: Channels (an evolution of Stream objects) and Buffers (representing platform-level buffers which allow zero-copy data operations).

3 Model Checking Internet Protocols with JPF

Java PathFinder can verify any Java program which does not depend on unsupported native methods. The JPF virtual machine cannot execute platform specific or native code, which imposes a restriction on what standard libraries can be used from within the application under test. For instance, the current version of JPF does not support `java.awt`, `java.net`, `java.nio`, and only has limited support for `java.io` and runtime reflection. The huge state storage requirements for a software system also limit the size of checkable applications. The JPF developers have estimated a maximum of 10.000 lines of code if no application and property specific abstractions are used [7].

Fortunately, JPF was designed as an open tool with extension capabilities. These extensions allow users to adapt the model checking process to their specific applications and properties. The main mechanisms available in JPF for extensions are:

- Listeners. The SearchListener and VMListener interfaces may be implemented to modify the basic behavior of the model checking algorithm. A variant of the Observer design pattern [8] allows listeners to subscribe to different events (bytecode execution, forward or backward steps,...).
- Configurable choice generators. The ChoiceGenerator class is the extension point responsible for implementing non-determinism policies to explore the state space in terms of thread states or data values. After executing a system transition, JPF prepares a ChoiceGenerator object which will be called to select the next transition to run. A ThreadChoiceGenerator will schedule threads to run, whereas other user-defined generators are focused on defining a finite data interval to explore.
- The Model Java Interface (MJI). Even if it is only a Java application (i.e. solely consists of Java classes), JPF can be viewed as a Java Virtual Machine in itself. The consequence is that (*.class) bytecodes are processed in two different ways in a JVM running JPF: i) as ordinary Java classes managed and executed by the host JVM (standard Java library classes, JPF implementation classes) or ii) as *modelled* classes managed and processed (verified) by JPF.

Fig. 3 depicts JPF interfaces and their relationship with the host Java virtual machine. In the bottom layer we find the Java Native Interface (JNI) used by the virtual machine to execute native libraries on top of the platform operating system. The Java layer contains the JPF application that runs on top of the virtual machine and uses the

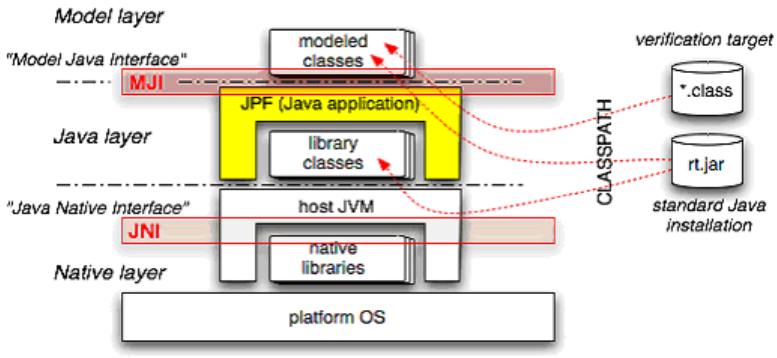

Fig. 3. Layers in JPF and their relationship with the Java virtual machine (from the JPF website [7]).

library classes available in the classpath (p.e. provided by `rt.jar`). The top layer is called the Model layer, and represents classes used by the verification target (the real application being analyzed). The MJI mechanism intercepts class method invocations and decides if they will be replaced instead by modelled classes. Therefore, JPF will execute these classes, which usually behave in a particular way to guide the model checking process. This mechanism is particularly useful to obtain a closed environment or to abstract specific behavior which is not needed for a specific analysis (in order to reduce the so-called state explosion problem).

3.1 Network Emulation in JPF

Following the MJI extension approach, our work has focused on creating appropriate modelled classes to abstract `java.net` and `java.nio` classes. Therefore, Socket and ServerSocket classes will be replaced at execution time by our own versions, which will run inside JPF in atomic mode and will use our abstract version of the network. Our strategy assumes that this *fake* network entity will substitute the real distributed environment for a multithreaded one, where protocol message exchanges will be replaced by thread synchonization mechanisms within the abstract network, allowing JPF to model check the protocols in this new way. Note that original programs must be transformed into threads before this approach can be used (as shown in the following subsection).

Fig. 4 represents our modelled class hierarchy, where original classes (shaded boxes) have been replaced by their corresponding modelled versions, which also require some additional support. Fig. 5 depicts the complete UML diagram for these supporting classes. The AbstractNetwork class is the mechanism responsible for abstracting the connection procedure along with its mainteinance. It also allows peers to send and receive messages by locating each other in the network through a PortMapper class. This is a data structure that stores so-called Slots, which model different concepts of a Socket connection for each port used. Therefore, Slot is the base class for specialized entities, each of which represents a well-defined role in the communication. There are Slots for abstracting `java.net` Sockets and ServerSockets (derived from SlotNet), `java.nio`

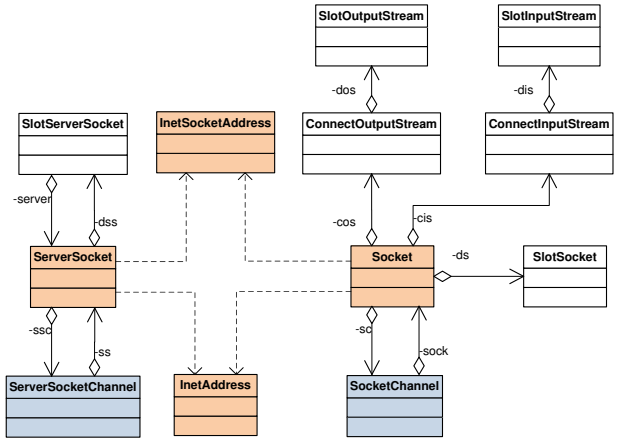

Fig. 4. MJI model for the Socket API.

Channels (derived from SlotChannel) and Streams (derived from SlotStream). The latter encapsulate the buffers used within transport protocols. Thus, Clients and Servers may synchronize their data exchanges through mutexes available in the Slots. This class hierarchy captures the typical behavior of a network which offers a reliable message delivery service, where entities are identified by their addresses and ports. At the moment, we have finished the support for TCP connection-oriented sockets, although UDP and Multicast are now in progress. It is worth noting that our approach provides a flexible way to use model checking for Internet applications, thus opening up the possibility of analyzing more complex Java middleware with JPF in the future. In fact, Java Enterprise applications or Web Services usually rely on the Socket API available, these application being a future target for our analysis.

3.2 The Unify Toolset

The Unify toolset allows verification of Client and Server Java programs with JPF. It is composed of three main components: a transformation tool (UnifyProcess), a GUI called SocketUnify and an executable class analyzed by JPF (UnifyLoader). UnifyProcess is a command-line Java application which accepts at least two executable programs (*.class) along with their original arguments. For each class, UnifyProcess uses the BCEL framework [9] to manipulate it at the bytecode level. The result is a set of ClassWrappers classes encapsulating the original ones and also implementing the Runnable interface. Then, the UnifyLoader application will instantiate every ClassWrapper on it as a thread, it now being possible for JPF to verify programs which were independent before. The BCEL approach used to instrument the original software has demonstrated to be more efficient than other more obvious approaches such as using Java reflection to instantiate original classes within UnifyLoader, which generates more states with JPF.

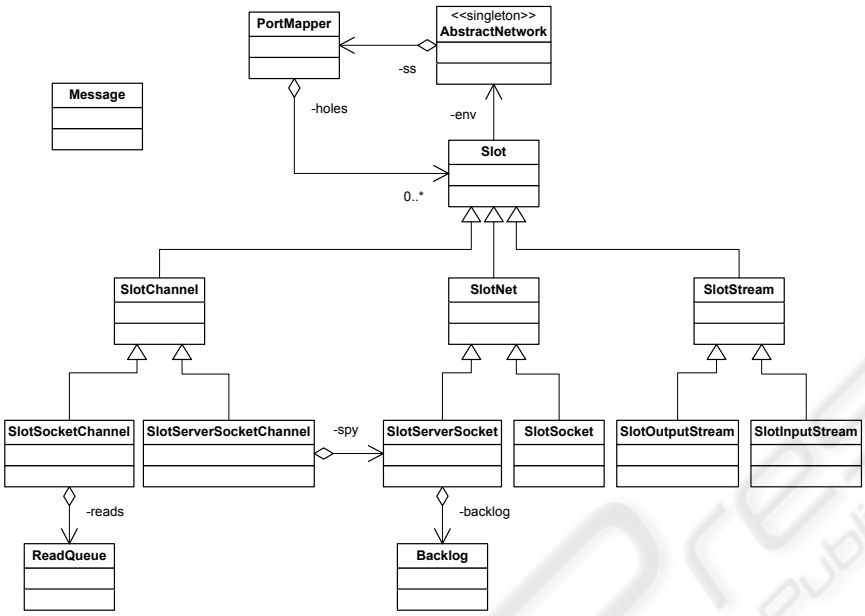

Fig. 5. The Abstract Network class hierarchy.

The Unify toolset also includes a graphical front-end shown in fig. 6. This GUI simplifies the selection of Client and Server classes, and helps the user to define options to guide the creation of ClassWrappers and the execution of UnifyLoader with JPF. The figure shows in the foreground a dialog box where users may select the JPF and Unify binary folders, along with a set of useful parameters such as the kind of analysis (quick or normal), the abstract network configuration, memory usage or different ways to show results, among others. Regarding verification modes in Unify, a quick verification mode assumes that a server is always available for connections in the port which clients will use, a situation which avoids the exploration of some execution paths and saves time and resources in the verification. However, this mode may result in false positives (OK verdicts) even when a server is not bound to the same port where clients are trying to connect. The normal verification mode takes care of this situation in JPF and notifies the corresponding error.

3.3 Details and Results

Our work has modelled twenty-two classes from `java.net` (covering 15% of the package), `java.io` (13%), `java.nio` (20%), `java.nio.channel` (39%) and `java.nio.channel.spi` (100%). All methods from these classes were modified, except the ones in ByteBuffer related with conversions to other formats (they were not a priority in this work). We have tested our JPF extension with different Java source code for Clients and Servers, where the limitations encountered were usually more

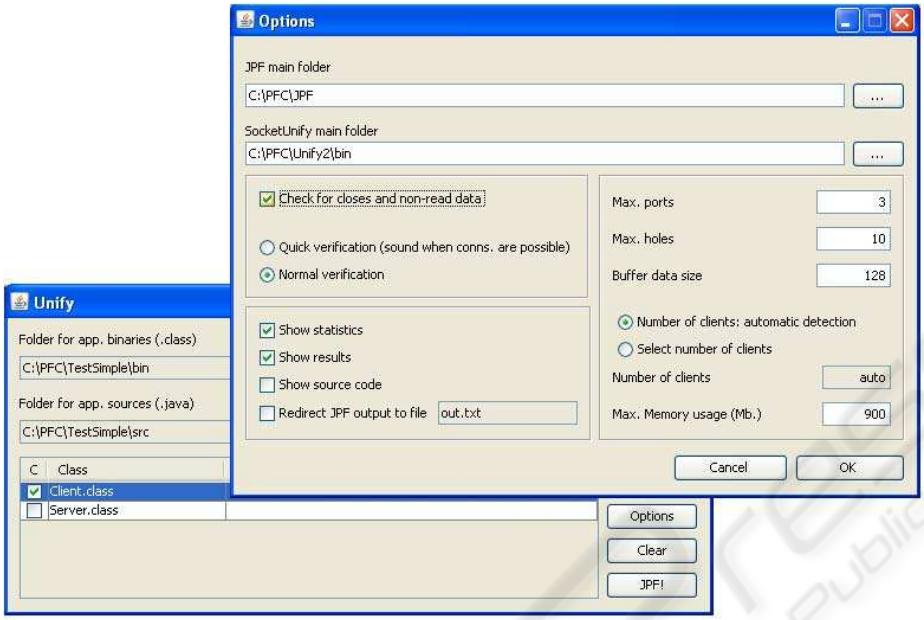

Fig. 6. The Unify GUI front-end.

Table 1. Experiment results using the JPF extension for basic Internet applications.

Experiment	Visited States	Max. Depth	Instructions	Time	Memory
Echo_C_S	726	25	393515	0:00:01	8MB
Echo_C_S_Selector	1153	30	870930	0:00:02	9MB
KnockKnock-Protocol	4220	203	6917403	0:00:08	23MB

JPF-related issues (e.g. some Java classes did not have their corresponding MJI version in the model checker). JPF verifies Socket code smoothly in terms of time and resources (states). For instance, table 1 shows some results obtained after verifying three Client/Server systems implementing variations of a request/response protocol (basically as shown in fig. 1): Echo_C_S, Echo_C_S_Selector and KnockKnock-Protocol. The first one uses an iterative server, whereas the second one uses demultiplexed I/O with a Selector, which is internally more complex and also more expensive to analyze. The latter is the example protocol implemented in the Java Tutorial [10]. All these experiments have been carried out using a PC Intel Core 2 Duo with 2,4 GHz and 2GB of memory.

Regarding the kind of properties available in our extension, we may check at the moment:

- Deadlocks in the system (e.g. two peers waiting on a blocking read operation).
- Bad uses of the Socket API (usually throwing a Java Exception).
- Socket closures with data still pending in the input buffer.

It is worth noting that this extension is fully compatible with existing verification methods and other optimizations in JPF, such as partial order reduction, abstract matching, race detectors or observers, among others.

4 Conclusions and Future Work

There are many model checking tools available to perform different analyses for distributed and concurrent systems. Unfortunately, users normally have to deal with the representation of their systems in the input language of the tool or tools selected, which is a typical source of new errors (or inconsistencies). In contrast, software model checkers constitute a smart solution for developers demanding an easy-to-use automatic verification tool. However, complex or specific domain problems may need additional adjustment and changes in the tools, following the objective of keeping these issues hidden from the user.

This paper has introduced a set of JPF extensions to deal with a domain-specific problem: the verification of Internet Protocols. The extensions include a way to analyze Clients and Servers provided directly in bytecode form, where no further instrumentation is required by users. The Java Model Interface has provided us with all the features necessary to create an API modelling a network abstraction. Our class hierarchy is able to verify processes which communicate through TCP, but it is scalable enough to deal with other transport protocols. Thus, it is now being improved for non-connection-oriented ones such as UDP and Multicast Sockets.

It is worth noting that our proposal is the first step towards the verification of more complex distributed middleware. Our main objective is the analysis of programs which rely heavily on middleware or high-level libraries such as J2EE or Web Services, where the basic communication tasks are hidden although they still belong to the basic Client/Server model.

Our future work is focused on introducing more flexible ways to define and embed properties for Internet software in JPF, along with techniques to optimize the verification process in terms of resources and time consumption. We also plan to combine our proposal with emerging methodologies such as Model Driven Engineering (MDE) [11], as shown in [12], where MDE and UML concepts were applied in order to simplify the design of network services for the Internet by proposing a UML2 Communication Profile. This would make it more straightforward to use the automatic code generation facilities available in modern modelling tools in order to obtain Socket-based Java source code to be analyzed with JPF.

References

1. Clarke, E., Emerson, E.A., Sistla, A.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. on Programming Languages and Systems **8** (1986) 244–263
2. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (2000)
3. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: Proceedings of CAV01. Volume 2102 of Lecture Notes in Computer Science. (2001)

4. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with BLAST. In: SPIN Workshop on Model Checking Software. Volume 2648 of Lecture Notes in Computer Science. (2003) 235–239
5. Havelund, K., Pressburger, T.: Model Checking Java Programs using Java Path Finder. In: Software Tools for Technology Transfer. Volume 2. (2000) 366–381
6. Havelund, K., Visser, W.: Program model checking as a new trend. In: Software Tools for Technology Transfer (STTT). Volume 4. (2002) 8–20
7. NASA: The Java PathFinder open source project. Available at http://javapathfinder.sourceforge.net/ (2008)
8. Gamma, E., Helm, H., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley Pub Co. (1995)
9. Apache Software Foundation: The Bytecode Engineering Library. Available at http://jakarta.apache.org/bcel/ (2008)
10. Sun Microsystems: The Java Tutorial: all about Sockets. Available at http://java.sun.com/docs/books/tutorial/networking/sockets/index.html (2008)
11. Kent, S.: Model Driven Engineering. In: Proceedings of IFM 2002. LNCS 2335, Springer-Verlag (2002) 286–298
12. Martínez, J., Merino, P., Salmeron, A.: Applying MDE Methodologies to Design Communication Protocols for Distributed Systems. In: First International Conference on Complex, Intelligent and Software Intensive Systems, IEEE Computer Society (2007) 185–190