Verifying Networked Programs Using a Model Checker Extension

Watcharin Leungwattanakit
University of Tokyo
Tokyo, Japan
watcharin@is.s.u-tokyo.ac.jp

Cyrille Artho
RCIS/AIST
Tokyo, Japan
c.artho@aist.go.jp

Masami Hagiya
University of Tokyo
Tokyo, Japan
hagiya@is.s.u-tokyo.ac.jp

Yoshinori Tanabe
University of Tokyo
Tokyo, Japan
y-tanabe@ci.i.u-tokyo.ac.jp

Mitsuharu Yamamoto
Chiba University
Chiba, Japan
mituharu@math.s.chiba-u.ac.jp

Abstract

Model checking finds failures in software by exploring every possible execution schedule. Until recently it has been mainly applied to stand-alone applications. This paper presents the I/O-cache, an extension for a Java model checker to support networked programs. It contains a cache module, which captures data streams between a target process and its peer processes. This demonstration also shows how we found a defect in a WebDAV client with a model checker and our extension.

1 Introduction

Software testing executes an application through only one thread schedule for each run, but the threads in the application may be scheduled by the operating system in a non-deterministic way. As a result, testing would miss some failures. Model checking solves this problem by exploring every possible schedule.

Networked applications are very complex, since they exchange data and interoperate with other entities. Several techniques have been established to verify such applications with a model checker [2]. The aim of our research is to offer another technique together with a tool, which relaxes restrictions of previous approaches. This paper proposes a *Java PathFinder (JPF)* [4] extension using a cache module as an interface to other processes.

A single-process model checker cannot be applied to networked applications, because some processes are not running inside the model checker, and consequently those processes are not backtracked together with the target process. Previous work has shown that a cache can be used to act as a proxy to external processes [1]. Our *I/O-cache* makes use of requests and responses in the past and sends already known responses back to the target application instead of dispatching requests to the peer. As a result, peers do not become aware of the model checker. If a request is not cached, the I/O-cache will physically send the request to the peer, wait for a response, and remember it.

The cache stores information it observed in a data structure that contains communication traces of each channel. When the target program sends a message to a peer, the message is stored in the cache. The I/O-cache then polls the peer process for a response. If there is a response, it will be matched with the last request. Some kinds of programs such as web servers serving dynamic content produce messages differently each time. To handle such programs, the cache module creates a new instance of the peer process to handle a new set of messages.

2 Tool Overview and Demonstration

The I/O-cache is an extension to JPF (see Figure 1). The internal state of JPF changes continually during program verification. Tool developers can write listeners and subscribe them to certain internal events. When a state transition takes place, the cache saves and restores communication data such as stream pointers and the number of active connections.

The Java library class `Socket` is modified to return special `InputStream` and `OutputStream` instances. The customized output stream redirects data sent by a target program to the cache, whereas the customized input stream reads data from the cache rather than the real peer process. `Socket` and `ServerSocket` have *native peer* classes, which run on the standard Java VM. These classes are responsible for the establishment of physical connections to peer processes. The cache is initialized when either class

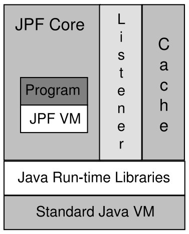

Figure 1. I/O-cache system architecture.

```
transition #3 thread: 0
ThreadChoiceFromSet {>main,SocketTimeout}
  StreamDemultiplexor : timer = t.setTimeout();
transition #153 thread: 1
ThreadChoiceFromSet {main,>SocketTimeout}
  StreamDemultiplexor.java:435 : timer = null;
  StreamDemultiplexor.java:438 : demuxList.remove(this);
transition #154 thread: 0
ThreadChoiceFromSet {>main,SocketTimeout}
  StreamDemultiplexor.java:138 : timer.hyber();
  HTTPConnection.java:2268 : requestList.remove(req);
snapshot #1
thread index=1,name=SocketTimeout,status=RUNNING
  call stack:
 at StreamDemultiplexor.close(StreamDemultiplexor)
 at StreamDemultiplexor.markForClose(StreamDemultiplexor)
 at SocketTimeout.run(StreamDemultiplexor)

NullPointerException: calling 'hyber()V' on null object
 at StreamDemultiplexor.init(StreamDemultiplexor.java)

elapsed time:       0:00:09
states:             new=3400, visited=2833, BT=6073
search:             maxDepth=280, constraints=0
choice generators:  thread=3400, data=0
heap:               gc=6060, new=7122, free=629
instructions:       306142
max memory:         25MB
```

Figure 2. The error trace of the WebDAV client (some transitions are omitted).

```
class StreamDemultiplexor    class SocketTimeout
static {                       extends Thread
  t = new SocketTimeout(10); SocketTimeout(sec) {
  t.start();                   list = new Entry[sec];
}                              for each entry in list
init() {                         entry = new Entry;
  timer = t.setTimeout();    }
  timer.hyber();             run() {
}                              while(alive) {
close() {                        sleep(sec);
  stream.close();                c = (c + 1)%sec;
  socket.close();                if (list[c] is marked
  if (timer != null)               and is not suspended)
    timer = null;          (**)    list[c].demux.close();
}                            }}
                             setTimeout() {
                               list[c].mark();
                           (*) return list[c];
                             }
```

Figure 3. Pseudocode of the WebDAV client.

Socket or ServerSocket is referenced for the first time by the program being verified. After that, it is ready to manage connection requests and data exchange between both sides of communication.

The time at which peers are launched depends on their roles. If a peer is a server, it must be completely initialized before JPF and the target process start. If the peer is a client, it is executed after the target process has called method accept. Execution order is specified in start-up scripts to make sure that each process runs at the right time.

A WebDAV test client [5], which is used for testing WebDAV-supported HTTP servers, was one of the applications we analyzed. Our extended JPF finds a defect consisting of a NullPointerException, as shown in the error trace in Figure 2. The code involving the defect is shown in Figure 3. A countdown thread is created in the static initializer of class StreamDemultiplexor. Any stream that is inactive for 10 seconds will be automatically closed by this thread, which is started immediately after creation. In transition #3, the main thread executes method init and starts counting time by method setTimeout, which also returns an object handler timer at line (*). In the scenario causing a failure, before method hyber is called, thread SocketTimeout gets its turn and continues running until time runs out. The countdown thread closes the corresponding stream and socket at line (**), setting variable timer to null (transition #153). This causes the method call to timer.hyber to fail at transition #154.

The failure is caused by improper synchronization in access to timer objects. It allows another thread to break in on the execution of method init. Such a fault might occur in programs with a high degree of thread-level parallelism. The Java language specification does not guarantee fairness of the thread scheduler [3]. A thread not to run for a certain time, which may cause defects that are very hard to find using traditional testing. This problem is addressed by model checking (and our tool extension).

References

[1] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe. Efficient model checking of networked applications. In *Proc. TOOLS EUROPE 2008*, volume 19 of *LNBIP*, pages 22–40, Zurich, Switzerland, 2008. Springer.

[2] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe. Tools and techniques for model checking networked programs. In *Proc. SNPD 2008*, Phuket, Thailand, 2008. IEEE.

[3] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Third edition, 2005.

[4] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[5] B. Holmes. A Simple PROPFIND/PROPPATCH Client, 2000.