Accurate Centralization for Applying Model Checking on Networked Applications

Cyrille Artho
National Institute of Informatics
Tokyo, Japan

Pierre-Loïc Garoche
Institut de Recherche en Informatique de Toulouse
France*

Abstract

Software model checkers can be applied directly to single-process programs, which typically are multi-threaded. Multi-process applications cannot be model checked directly. While multiple processes can be merged manually into a single one, this process is very labor-intensive and a major obstacle towards model checking of client-server applications.

Previous work has automated the merging of multiple applications but mostly omitted network communication. Remote procedure calls were simply inlined, creating similar results for simple cases while removing much of the inherent complexities involved. Our goal is a fully transparent replacement of network communication. Other language features were also modeled more precisely than in previous work, resulting in a program that is much closer to the original. This makes our approach suitable for testing, debugging, and software model checking. Due to the increased faithfulness of our approach, we can treat a much larger range of applications than before.

1. Introduction

Java [11, 17] is a popular object-oriented, multi-threaded programming language. Verification of Java programs has become increasingly important. However, most non-trivial programs use network communication. Testing distributed systems is extremely difficult. The non-determinism introduced by asynchronous communication and thread scheduling makes testing unreliable, because failures may only occur under particular schedules. *Model checking* finds such failures reliably while still being completely automated [6]. Several model checkers for Java-based programs have been created [2, 8, 23]. Unfortunately, these model checkers can only explore a single process and are not applicable to networked applications, where several processes interact.

Process centralization allows model checking distributed applications: Processes are converted into threads and merged into a single application [21]. Networked applications can then run as one multi-threaded application. This approach is applicable if all programs to be merged are available in the same format and inter-program communication can be modeled accurately. The latter has not been addressed satisfactorily yet. Previous work inlined parts of one program in another one, modeling certain patterns of interaction using Remote Method Invocation (RMI) under Java [21]. Unfortunately, this implementation is very specific to certain RMI-based programs and cannot cover more general, TCP/IP-based communication. Our approach can fully replace the widely used socket mechanism. It is therefore much more generic. Furthermore, we treat certain language features with a higher precision and explain in more detail possible solutions for mapping processes to threads. This increased precision allows us to use our tool on any application where an automated approach is feasible, and correctly centralizes applications that were not treated accurately in previous work. While our goal is to model check the resulting centralized application, the technique may also be used to simplify debugging of multi-process systems, allowing them to run on a single-process debugger. We do not pursue this idea further in this paper.

This paper is organized as follows: Section 2 describes how problems arising with the conversion of processes to threads are solved. Treatment of network communication is described in Section 3. An overview of our tool, and results gained from experiments are given in Section 4. Section 5 described related work, and Section 6 concludes.

2. Process centralization

Centralization of Java program involves four issues [21]:

1. Wrapping applications (processes) as threads, and starting them as such.

*Thanks go to the NII international internship program for its financial support.

2. Keeping the address space of each process separate. This is not a problem for normally created instances, as they are only reachable from the process where they have been created.[1] However, it is in issue when considering static fields. Each static field is unique and globally accessible by its class name. This uniqueness applies per class and thus per Virtual Machine (VM): In the original system, each process has its own copy of each class used, and each static field contained therein. In the centralized version, field accesses to static fields must be disambiguated between processes.

3. Static synchronized methods. Method-level synchronization is performed implicitly by the VM whenever a method is synchronized. This synchronization accesses a class descriptor that should again be unique for each centralized process. Resolving this also entails transforming the `synchronized` flag in the method in question to block-level synchronization.

4. Shutdown semantics. The Java VM closes open sockets and files before exiting. Furthermore, background (daemon) threads are killed. These actions only occur if the entire process terminates, and therefore have to be implemented by the centralization tool.

2.1. Process wrapping and startup

The first task is to convert a process to a thread. In addition to the conversion, a unique process ID has to be maintained for each process. Both problems can be solved quite elegantly, as implemented by the tool described by Stoller [21]. Figure 1 shows the implementation. A simple wrapper class `CentralizedProcess` maintains the process ID, and another class (called `ProcessStartup` in this example) initializes and runs the processes as threads. Classes `Proc1` and `Proc2`, which are not shown, contain the `main` method of each application.

When launching networked programs, there exists an additional problem that was not discovered in previous work: The startup procedure shown in Figure 1 does not control the order in which threads are run. Calling `Thread.start` merely enables a thread to run but does not necessarily start it immediately. Therefore, there exist possible thread schedules where Thread t_2 is run prior to the thread which was created before, t_1. Such thread schedules may lead to executions where a client thread attempts to connect to a server thread before the server has finished initialization. This will lead to failures in the client that are generally not interesting for model checking. For example, assume a client starts

[1]The original application cannot share references between processes, as Java does not allow to do so directly. Therefore, instances created within a particular process are always only reachable by threads belonging to that process.

```
public class CentralizedProcess
  extends Thread {
  public int pid;

  public CentralizedProcess
    (int procId) {
    super();
    pid = procId;
  }
}

public class ProcessStartup {
  Thread t1 =
    new CentralizedProcess(0) {
    public void run() {
      Proc1.main(null);
    }
  };
  t1.start();

  Thread t2 =
    new CentralizedProcess(1) {
    public void run() {
      Proc2.main(null);
    }
  };
  t2.start();
}
```

Figure 1. Wrapping a process as a thread.

before the server and cannot connect to the desired port. Some clients may retry a number of times, but there still exist schedules where the server cannot complete initialization on time, and the client will ultimately fail. The server will then wait indefinitely for a client to connect, while the client has already aborted.

Such failures should be avoided in the centralized version. If the case where no server is present should be tested, this can be done by running the client by itself, without a server. Therefore, when model checking the client together with a server, this scenario creates undesired, spurious failures. These behaviors do not only increase the search space, but may also mask more interesting failures.

When simply testing an application, an imprecise solution for organizing the startup sequence can be used. Simply waiting for a short time before starting the client is sufficient. This will give the server enough time to initialize. Thread schedules in which server initialization cannot complete in time are unlikely, and can be dealt with by repeating the test in the rare event of a failure. However, when using model checking, all thread schedules are explored, including the unlikely ones leading to a slow server startup. Hence, a mere heuristic reducing the likelihood of schedule problems cannot be applied to model checking.

Instead, one has to ensure that server initialization has completed before the client starts. In a distributed application, it may not be possible to check whether the server is ready. Fortunately, if the program is centralized, completed server initialization can be communicated to the client by using shared data.[2] Still, such startup synchronization remains difficult. Commonly, the last step required before a server can accept client requests is the system call `ServerSocket.accept`. This is a blocking system call, and it is not easily possible to check whether the socket is actually ready for receiving requests, even when using another server thread to do this. While there exist mechanisms how this could be achieved, these approaches are complex and not fully portable. In principle, it could be implemented by using method `ServerSocket.acceptImpl` to wrap `accept` and then polling file descriptors. Other inter-process communication mechanisms can be polled in different ways. For instance, most operating systems have tools listing open file descriptors. Open sockets can be detected by port scans. In any case, knowing whether initialization of inter-process communication has completed usually requires knowledge of its implementation, not just its API. Different inter-process communication (IPC) means have different ways of being polled or monitored. However, in the centralized version, it is actually desirable to *replace* a given IPC mechanism entirely by a simpler intra-process mechanism, which allows for model checking. This is described in Section 3 for TCP/IP networking.

2.2. Static fields

As described at the beginning of this section, it is not possible to re-use static fields without changes. The fact that different VMs load each class individually implies that they have their own copy of static fields [11, 17]. This must be preserved for the centralized version. This creates a challenge because even in the centralized version, a class can only be loaded and initialized once. The problem becomes exceptionally difficult for static initializers relying on certain data or having side-effects; only some of these initializers may be centralized automatically. This limitation is acceptable because model checking usually entails a certain degree of program abstraction. The focus of this part therefore lies on static fields and their initialization.

The best solution to problems arising with global, static fields lies in replacing each static field with an array of static fields [21].[3] For each static field, an array containing the value of that static field for each process is maintained. Arrays are transformed into arrays of arrays, with an extra dimension for distinguishing the array content between processes. This approach assumes that the number of processes is known at compile-time. This is a very reasonable assumption as model checkers can only handle a small number of threads. Thus the number of processes has to be bounded for scalability reasons. At run-time, the process ID of the current process can be obtained as follows:

```
PID = ((CentralizedProcess)
  Thread.currentThread()).pid;
```

A centralized process can in turn create threads. If this occurs in a program, another change is required: Any thread created has to be replaced with an instance of `CentralizedThread`, which is a superclass of `CentralizedProcess`. Any class extending `Thread` must be changed to extend `CentralizedThread` and initialized with the PID of the "parent process".

Furthermore, field initialization has to be implemented for fields whose value is not initialized to zero. In Java, static fields are initialized by the *static initializer,* a method that is called when a class is loaded. The centralized version has to initialize the array of per-process copies of the field first. Then, for initializing the field content, the original static initializer has to be executed once for each process. This initializes all copies of the original static field and also executes other initialization statements once per process. Care has to be taken for classes where such side-effects should not occur several times; such classes have to be excluded from this transformation step. Automation has other limits: For instance, certain class initializers may be dependent on the context of a process. For instance, an initializer may use a default setting given by an environment variable. If static data is initialized equally for all processes, this may negate the fact that different processes may be run under different settings. Our tool currently supports process-specific command line arguments; process-specific environment variables are subject to future work.

Other problems with respect to centralization and class loading may occur if one uses a custom class loader. It then has to be ensured that the class loaded is the same for each centralized process, as it is not possible to have different classes under the same name in a centralized program. Again, this problem should not occur frequently for programs that are going to be model checked.

2.3. Class descriptors and locking on classes

The third issue in centralization is given by synchronized static methods. A static synchronized method is a shortcut for having an entire block of code which is synchronized on the descriptor of the current class (as given by the type in which the current method was declared).[4] Such a descriptor is always a unique instance of type `java.lang.Class`.

[2]This usage of global data only works for inter-thread communication. Thus this change should be done in the wrapper class, as this modification does not work when running the applications individually as processes.

[3]An alternative solution consists of the duplication of each class file, creating one renamed copy per process. This would create a separate "name space" for each process. However, in order to accommodate for callbacks, the entire Java library would have to be duplicated this way. Furthermore, this approach would require type casting for passing object data between processes, e.g. when using RMI.

[4]Sun's official JDK documentation states that a static synchronized method uses the reference returned by `java.lang.Object.getClass` as

Again, different processes have to use different locks. Previous work used proxy locks to replace the class instance for locking in synchronized methods [21]. However, this has the following problems:

- Block-wise synchronization on a given class descriptor using `X.class` or `getClass()` is not covered. Applications using such locking would exhibit data races when centralized.
- If any use of a class descriptor instances is replaced with a proxy object, then this will not only cover locking, but also other uses of class descriptors, e.g. when using reflection. Programs relying on such features will no longer work without proper class descriptors.

Our first approach was therefore an attempt to replace all instances of `java.lang.Class` with an array of class descriptors. The class descriptor of each class can be seen as a static field; the first attempt was therefore to treat them as such, replacing each class descriptor with an array. Unfortunately, class `java.lang.Class` has no public constructor and does not support the `clone` method. Trying to support this option artificially, through code instrumentation, obviously violated internal invariants in Sun's JVM, because running such code resulted in internal errors in the VM.

We therefore reverted to proxy locks. As class descriptors can be used for locking or for other purposes, the final solution works as follows: First, static synchronization is replaced with block-wise synchronization on the current class descriptor.[5] Then, for any block-wise synchronization, a run-time check is added, which tests whether the lock used is of type `java.lang.Class`. If so, a proxy lock is used instead; otherwise, the given lock is used directly. For proxy locks, the invariant that the lock is unique per process is maintained by the transformation, and the program semantics are not changed. The proxy object is used for locking only; other uses of class descriptors are not changed. This allows for arbitrary uses of class descriptors.

2.4. Shutdown semantics

The final problem with centralization concerns application termination. There are two ways to terminate a running Java VM immediately: `System.exit` and `System.halt`. The first one runs any registered *shutdown hooks,* tasks that free resources during shutdown. Such tasks can also be added by the user through `Runtime.addShutdownHook`. The second one terminates the VM abruptly, without freeing any resources [22]. When centralizing an application, these system calls must be replaced, as threads representing other centralized processes must still continue to run. Therefore, any call to `System.exit` and `System.halt` has to be replaced with throwing a new instance of `java.lang.ThreadDeath`. This causes the current thread to terminate and allows other centralized processes to continue [21, 22].

This solution is adequate for single-threaded applications. The complex shutdown semantics of multi-threaded applications have not yet been addressed by previous work. The key problem is that `System.exit` has to shut down other threads belonging to the same process. Similarly, daemon threads have to be terminated when all normal threads have died. In the centralized program, this mechanism is not automatically triggered by the VM. Hence, daemon threads belonging to the "dead" process will not terminate as usual. If daemon threads are not shut down properly in the centralized version, spurious failures may occur because the daemon thread may still attempt to use shared resources which are no longer available.

Unfortunately, it is difficult to shut down other threads in Java. Methods that allow to shut down other threads have been deprecated because they can lead to deadlocks. Therefore, daemon threads cannot be shut down directly. Fortunately, a closer look at the situation reveals a solution: A daemon thread of a dead process performs either invisible operations in memory, or externally visible operations such as I/O. Operations in memory affect only the state of the dead process; therefore, their results, including possible assertion violations or exceptions, can just be ignored. Externally visible operations are prevented by automatically closing all I/O resources. Any attempt to still use these resources will result in a failure. Therefore, the solution consists of ensuring that any possible failures in a dead process are recognized as spurious failures and ignored. This can be achieved by implementing a custom search class in JPF, which overrides the normal search and failure report functions for these special cases.

Beyond shutting down other threads belonging to the same application, there is the problem of shutting down the VM itself and executing shutdown hooks. Such shutdown hooks close open files or network connections. In order to model this correctly, each newly created such resource has to be registered in class `CentralizedProcess` and closed manually if it is still open. This is very important because some of these resources, such as an open server socket on a given port, can only be allocated once. If a centralized application terminates, such resources have to be freed automatically by the run-time library of the centralization tool.

its lock [22]. This is slightly imprecise: First, `getClass` is not available in a static context. Second, the lock used is actually determined by the compile-time type of a method, not by its run-time type. Assume a static synchronized method is inherited by a subclass. Said method, even when called via its subclass, will still lock on the class descriptor of the super class, which is not what a call to `getClass` in a non-static context would return.

[5]This entails installing an exception handler that releases the given lock should an exception occur inside the given static method.

$$\begin{tikzcd}
\textbf{Client} & \textbf{Server} \\
 & \texttt{accept}\text{s communication (b)} \\
\texttt{connect}\text{s (b)} \arrow[r] \arrow[d] & \text{established connection} \arrow[dl] \\
\text{bi-directional communication possible} &
\end{tikzcd}$$

Figure 2. Client-server communication.

In order to be able to close allocated resources at process shutdown time, each allocation has to be registered at run-time. I/O resources can only be allocated by a handful of classes. Creation of such instances can thus be tracked automatically through code instrumentation in a few key methods. When a centralized process terminates, any of its allocated resources that have not yet been closed are closed by our shutdown routine.

Our current implementation of the shutdown semantics covers `System.exit`, `System.halt`, and normal process termination, keeping track of open sockets and files. It is still work in progress. We have not yet implemented shutting down external processes that were created by `java.lang.Runtime.exec`. Experiments on Sun's JVM on Linux have shown that the VM sends an `INT` (interrupt) signal to such processes, and detaches them from the dying VM process. Nonetheless, as a centralized application that starts external processes cannot be model checked directly, we did not investigate this feature further.

3. Network communication, RMI

Network communication can be modeled as an interaction of two peers, a *client* and a *server.* The server allows for incoming communication by accepting connections to a certain port. The client can subsequently connect to that port. After a connection is established, a bidirectional communication channel exists between the client and the server. Communication then occurs asynchronously. Underlying transport mechanisms (commonly TCP/IP) ensure that sent messages arrive eventually (if a connection is available), perhaps with some delay. This applies to messages in both directions. A connection can be closed by the client or the server, terminating communication.

Figure 2 illustrates this: The client has to wait until the server is available, and retry a connection attempt if necessary. When the server accepts a new connection, its execution blocks until a client has connected. A blocked state is shown as (b) in the figure. As soon as a client connects, the server is unblocked, and a connection is established by underlying system libraries. The corresponding `connect` call is in turn blocking for the client, unblocking as soon as the response from the server is received. In Java, TCP/IP

$$\begin{tikzcd}
\textbf{Client} & \textbf{Server} \\
\text{Client looks up stub} \arrow[d] & \\
\text{Stub serializes arguments} \arrow[r] & \text{Remote stub called} \arrow[d] \\
\text{Client stub receives result} \arrow[d] & \text{Result sent by stub} \arrow[l] \\
\text{Client receives result} &
\end{tikzcd}$$

Figure 3. Remote Method Invocation.

$$\begin{tikzcd}
\textbf{Client} & \\
\text{Client looks up stub} \arrow[d] & \\
\text{Stub serializes arguments} \arrow[d] & \textbf{Local computation} \\
\text{Stub returns result} \arrow[d] & \\
\text{Client receives result} &
\end{tikzcd}$$

Figure 4. Inlined RMI call.

communication mainly involves two classes: `Socket` and `ServerSocket`. Instances of `Socket` are handles to a connection between a client and a server. A `ServerSocket`, on the other hand, allows a server to accept incoming connections. This operation instantiates a `Socket` object as a result. On the client side, internal `Socket` data will be initialized when its `connect` call returns.

Remote Method Invocation (RMI) in Java builds in sockets and behaves in a similar way, as shown by Figure 3. RMI offers a naming mechanism which allows clients to find a server. After the lookup is performed, the client communicates with the server via *stubs,* which hide network communications and make (blocking) RMI calls appear to be local. Arguments for calls have to be serialized (converted into raw data) before transmission. The client blocks until it receives the response from the server. The stub receives the serialized result, unblocks execution of the client, and relays the result to the calling client.

Previous work [21] has inlined the remote computation of the server, replacing a blocking remote call with a local call, as shown by Figure 4. This idea "cannibalizes" on the blocked client thread for simulating remote computation. In doing so, it greatly simplifies the original program, which is desirable for model checking. However, it also removes a lot of the structural complexity, which may be harmful because it may mask problems. The key problem is that this idea cannot be extended to generic TCP/IP networking.

We keep the idea of replacing multiple processes with multiple threads in a single process. The client and server

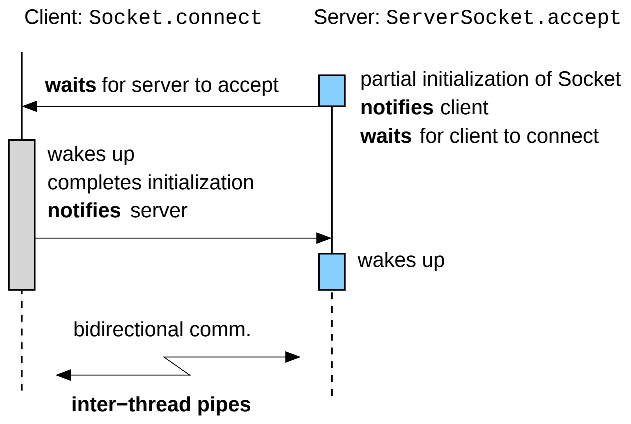

Figure 5. Implementation of Socket.connect and ServerSocket.accept.

applications are still merged into a single application as done before [21], but communication is treated in a totally different way. Due to the complex nature of true bidirectional communication in client-server programs, a simple inlining mechanism is no longer applicable. RMI-based applications are subsumed by our approach since RMI is just a protocol on top of TCP/IP.

General network communication was broken down into two steps: Connection establishment, and bidirectional communication. We used a two-step barrier [16] to model the blocking connection mechanism shown in Figure 2. In a first step, the server blocks during the `accept` call. When the client calls `connect`, the server is unblocked while the client blocks and waits for completion of the connection. This ensures that the sequence of each original application passing through blocking library calls is preserved in the centralized version. Upon connection, two unidirectional inter-thread pipes are set up, as available through `java.io.PipedInputStream` and `java.io.PipedOutputStream`. They model the underlying network communication normally provided by system libraries, replacing inter-process communication by inter-thread communication.

Figure 5 illustrates this idea. If a client connects to an open port, it will be suspended until the server thread has had an opportunity to complete its part of the `Socket` initialization. The server thread will initialize its side of the `Socket` and then suspend itself after having notified the client about completion of its part. The client then wakes up and finishes initialization. After that, the `connect` method of the client returns. Its data streams are already fully initialized at that point, ready to be used by the client. Data will not actually be received or sent by the server until it wakes up, having been notified by the client. After that, bidirectional communication can take place.

Input		**Output**
	Centralizer	
Application bytecode	→	Instrumented bytecode
Startup script	→	Startup wrapper program

Figure 6. Overview of our centralizer tool.

The fact that blocking calls between different processes are modeled using `wait` and `notify` also allowed us to solve the problem of scheduling the start of centralized processes. The point where the server has completed its part of socket initialization and suspends itself is where the first client process can be started. Therefore, the same signal used in `notify` is used by the process wrapper described in Figure 1, which waits for the same notification.

For fine-tuning the implementation, the possibility of a refused connection (when a socket is closed), management of open connections, and other details made the code more complex than indicated in Figure 5. Due to these complex inter-thread interactions, development of a correct socket replacement was very difficult. We have used the Java PathFinder model checker [23] to find failures and later on verify correctness of our implementation.[6] We have also successfully used this socket replacement for running and model checking centralized example applications, such as a daytime server, an echo server, and a chat server.

4. Experiments

Based on the transformation described above, we implemented a centralization tool that automatically transforms all class files of a given application. Figure 6 describes the general way this tool operates: As input, it takes the bytecode (class files) of the application, and a startup script that specifies the processes to be launched. The startup script also includes dependencies, such as whether the client should not be started before the server is ready to accept connections on a given port. As output, the centralizer produces instrumented class files and a wrapper program that is used to launch the centralized processes. The instrumented bytecode also uses a run-time library which implements process and thread wrappers, and proxy locks.

Unlike the tool described in previous work [21], which operates on source code, our tool directly works on bytecode. This improves its performance, as source code does not have to be parsed, and allows transformation when source code is not possible. Another advantage is the fact that Java bytecode has not changed significantly in the last few years. This is in contrast to changes in the Java programming language, which unfortunately had the consequence that the tool developed by Stoller et al. [21] no

[6]Properties verified included built-in properties, such as deadlock freedom and no uncaught exceptions, and user-defined assertions.

Application	Description
Daytime server	Returns the current time (RFC 867).
Echo server	Returns all input received (RFC 862).
Chat server	Sends messages of one client to all clients.

Table 1. Example applications used.

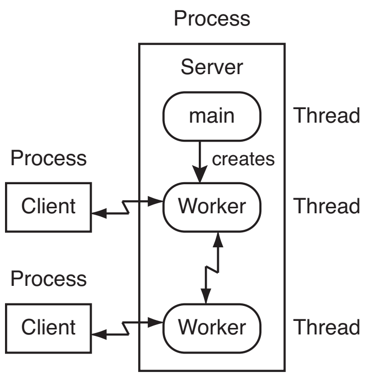

Figure 7. Chat server architecture.

longer functions in current environments. Therefore we could not perform a cross-comparison for centralization of difficult language features and had to rely on tool documentation when describing the extensions in our approach. For model checking, we used the Java PathFinder (JPF) model checker [23]. Prior to our work, JPF could not support multiple processes or any inter-process communication. With our library, JPF (or any other Java model checker) can now handle such programs.

Table 1 gives an overview of the examples used as benchmarks. The daytime server sends a fixed string back to the client; the echo server returns the input received.[7] The chat server sends the input of one client back to all clients, including the one that sent the input. All applications were supplied with two test clients. The daytime client reads a single line, while the echo client sends three lines and expects the same number of lines back. The chat client sends and reads a single line. While the daytime and echo server were quite simple applications, the architecture of the chat server is fairly complex, involving a main server thread to handle connections and one worker thread per connection. This architecture is comparable to more complex applications such as modern web servers [9]. While the first two applications mainly served as test beds for our tool, the chat server gives a good evaluation of the performance of model checking a centralized application and the capabilities of this approach for complex frameworks.

In the chat server, the main server thread listens to incoming connections and creates a worker thread for each connection that is handled. These worker threads use shared data structures to send a message to all other clients. This results in complex possible interactions between several applications (Figure 7). The main thread inside each client process is the only thread and therefore not shown separately. Because the server contains several threads, the main thread is listed as such. The main thread in the server creates worker threads to handle a connection whenever a client connects. Each message from a client is handled by the corresponding worker thread on the server side. This worker thread then sends that message to each other client by accessing the shared array that contains references to all the other workers. Through this array, the sockets connecting the chat server to each client can be retrieved. Therefore any interaction between clients always occurs via the server application, where it is handled by a particular worker thread.

Experiments were executed on a dual-processor 2.7 GHz PowerPC G5 with 8 GB of RAM and 512 KB of L2 cache per CPU, running Mac OS 10.4.5. After the chat server application was centralized, model checking found two faults that were not detected by testing. The first one was a missing `synchronized` statement for a method. It may have been detected by other means such as the Eraser run-time verification algorithm [19]. Model checking the application containing this defect only took a couple of seconds, though, and delivered a detailed counter-example trace, which contains more data than just the field access in question. Nonetheless, we will focus on the second fault found, which is much more interesting and could not have been detected by run-time monitoring.

The parts of the code which are relevant for the observed failure are shown in Figure 8. The failure scenario involves two (or more) clients connecting to the chat server at the same time. The main thread of the server keeps accepting incoming connections on the open socket. As long as the number of available worker threads is not exceeded, the server handles incoming connections by instantiating a worker object for each connection. Each worker instance contains the data which is relevant for itself (a reference to the socket and its ID number). It also implements interface `Runnable`, allowing for creation of new threads. This can be seen in the lower part of Figure 8, where method `run` implements the code that handles messages sent by a given client.

The failure scenario looks as follows: A client connects to the server. The connection is handled inside the `synchronized` block in method `ChatServer.serve`, which creates a new instance of `Worker`. After worker thread w_1 is initialized, the new instance is added to the array of worker threads, and `start` is called. After this, the `for` loop inside the `synchronized` block of `ChatServer.serve` completes. Worker thread w_1 is now ready to run, but at this moment, the scheduler does not

[7]For the purpose of model checking, the "date" used was hard-coded in the replacement class for `java.util.Date`.

```
public class ChatServer {
  public serve(int maxServ) {
    try { while (maxServ-- != 0) {
        Socket sock = servsock.accept();
        synchronized(this) {
          for (int i=0; i < workers.length; i++) {
            if (workers[i] == null) {
              workers[i] = new Worker(sock, i);
              new Thread(workers[i]).start();
              break;
    } } } } } catch(IOException ioe) { /* ... */ }
  }
  static synchronized void sendAll(String s) {
    for (int i = 0; i < workers.length; i++) {
      if (workers[i] != null) workers[i].send(s);
} } }

class Worker implements Runnable {
  Socket sock; int n;
  PrintWriter out = null; BufferedReader in = null;
  public Worker(Socket s, int id) { sock=s; n=id; }
  public void run() {
    try {
      out=new PrintWriter(sock.getOutputStream());
      in=new BufferedReader(sock.getInputStream());
      while ((s = in.readLine()) != null)
        ChatServer.sendAll(s);
    } catch(IOException ioe) { /* ... */ } } }
```

Figure 8. Excerpt of the chat server source.

yet execute any of its code. Instead, another client connects to the server. This results in initialization of another worker thread, w_2, which is also ready to run at this point. Now the scheduler decides to execute w_1, which receives input from its client. This input is then sent to all active connections. Method `ChatServer.sendAll` checks whether each array entry is `null`, and then sends the message to each registered worker via `Worker.send`. However, `PrintWriter out` of w_2 is not initialized yet, which results in a `NullPointerException`. Initialization of `out` would have been completed in the first statement of `Worker.run`, which did not execute yet. So the error consists of the fact that w_2 is not yet fully initialized but already registered in the array of worker instance. This error is quite complex and does not correspond to any easily classifiable data races or atomicity problems which find similar errors [1]. The error trace shown by JPF involves over 100 transitions. Fixing the error required moving the initialization of field `out` from method `run` into the constructor. This allows initialization to complete before the worker thread is registered globally.

A manually and an automatically centralized version were compared to evaluate the overhead our tool introduces. This overhead consists of unneeded centralization of static data. Knowledge about classes used by one process only can avoid centralization of some static fields. This produces a leaner centralized version, because both the class initializer and all field accesses of static fields require less code. It was therefore interesting to see the quantitative effect of automatic centralization of field accesses.

Application	Manual centr.		Tool-based centr.	
	Time	Cex [tr.]	Time	Cex. [tr.]
Daytime server	1:08 m	n/a	1:49 m	n/a
Echo server	2:07 m	n/a	4:22 m	n/a
Chat v. 1	2.8 s	112	3.0 s	142
Chat v. 2	6.3 s	154	6.8 s	186
Chat, 1 cl., final	3:54 m	n/a	7:21 m (est.)	n/a
Chat, 1 cl., opt.	3:41 m	n/a	7:06 m (est.)	n/a
Chat, final	13:12 h	n/a	65 h (est.)	n/a
Chat, opt.	7:43 h	n/a	11:04 h (est.)	n/a

Table 2. Comparison of manual with automated centralization.

Table 2 shows the results of these experiments. For different versions of each application, the time required to model check the centralized version is given. Two faulty versions of the chat server exist. The correct version has been tested with the ability to serve only one chat client, and two clients. These final versions were run using a general-purpose network replacement library ("final"), and with one that was specially optimized for two connections ("opt."). For faulty versions of the chat server, the length of the counterexample (measured in number of transitions) is also given, since it is a good measure of the overhead introduced by centralization of static fields.

Unfortunately, we encountered internal JPF errors for the larger benchmarks for which we could not find a workaround. The model checking terminated prematurely due to this internal error. Based on the size of the log file, we calculated the percentage of the state space explored, and estimated the total time accordingly. The estimate for the optimized version of the chat server is rather precise, because about 75 % of the state space has been explored; only the estimate for the unoptimized "final" version with two clients is less accurate, because only 22 % of the state space had been explored when JPF aborted. For the smaller benchmarks, JPF had problems with initialization of constants; these problems could be fixed by replacing the constant fields with the values they represent.

When comparing the time required for model checking and number of transitions required for the counterexamples, one can see that our centralization tool normally introduces an overhead of about factor two, which usually is acceptable considering that model checking is exponential in the size of the state space. The roughly 30 extra transitions mainly reflect initialization of run-time data structures of helper classes used by the instrumented, centralized code. The overall time required when model checking the complete state space of a system increases by roughly a factor of two. This still leaves room for improvement. We hope that leaner implementations of certain internal data structures will reduce this overhead in the future.

5. Related work

The classical application domain of model checking consists of the verification of algorithms and protocols [13]. More recently, model checking has been applied directly to software, sometimes even on concrete systems. Such model checkers include the Java PathFinder system [12, 23], JNuke [2], and similar systems [4, 8, 10, 15, 20].

Regardless of whether model checkers are applied to models or software, they suffer from the state space explosion problem: The size of the state space is exponential in the size of the system, which includes the number of threads and program points where thread switches can occur. Most systems are therefore too complex for model checking. *Partial-order reduction* eliminates certain independent interleavings and can thus reduce the number of states to be explored [5, 14]. System *abstraction* reduces the state space by merging several concrete states into a single abstract state, thus simplifying behavior. In general, an abstract state allows for a wider behavior than the original set of concrete states, preserving potential failure states. System abstraction in software is crucial to reduce the state space [4, 8]. Results of system-level operations have been successfully modeled this way to detect failures in applications [7] and device drivers [4]. These techniques analyze one process at a time, with a modular environment (that has been generated automatically or written manually). They work well on low-level programs where the actual content of messages is irrelevant; however, complex applications cannot be analyzed automatically in this way.

Most software model checkers are written to analyze a single OS-level process, and cannot handle distributed systems [2, 4, 8, 10, 20, 23]. When using manual program abstraction, I/O operations can be replaced by stubs that mimic each possible operation. However, creation of such stubs is application-specific and can be quite labor-intensive for complex applications. A better solution to this problem is to lift the power of a model checker from process level to OS level. This way, the effect of any I/O operation is still visible inside the model checker. An existing system that can indeed store and restore full OS state is a tool based on user-mode Linux, which uses the GNU debugger to store states and intercept system calls [18]. The effects of system calls are modeled by hand, but several processes can be model checked together without modifying the application code. The difference to our approach is that centralization transforms a multi-process system down to a single-process one, while the other tool expands the scope of model checking to several processes. Our tool is therefore not dependent on a special model checker and can act as a "preprocessor" of a distributed program, allowing it to be model checked with any software model checker supporting Java bytecode.

Centralization was first proposed and implemented by Stoller et. al [21]. This allows several processes to run in the same model checker. However, this does not solve the problem of modeling inter-process communication. Previous work has been restricted to programs using remote method invocation (RMI), where I/O was not modeled but simulated using process inlining. Generic TCP/IP networking, which is the foundation of many other protocols, has not been supported before. Our work therefore expands the applicability of centralization to a much large set of programs. Furthermore, an accurate treatment of Java semantics such as usage of class descriptors and process shutdown behavior ensures correct results when such rarely-used features occur in Java programs.

6. Conclusion and future work

Distributed programs include several processes and typically use network communication. For model checking such programs in a single-process model checker, processes have to be replaced by threads and merged into a centralized application. Replacement of communication mechanisms is also necessary for non-trivial applications. While Remote Method Invocation can be replaced by inlining, this replacement is not very accurate and cannot be extended to arbitrary socket-based communication. Our approach replaces the socket connection mechanism by inter-thread communication. We have successfully model checked our implementation using client-server applications. Our tool also treats certain Java language features with increased accuracy compared to previous work.

Future work includes running the experiments on other Java-based model checkers, and optimizations in the runtime library of our centralizer. In code instrumentation, reference analysis may eliminate the need for instrumenting certain classes. We also plan to complete the implementation of shutdown semantics in our tool, as described in this paper. Finally, there is ongoing work in adding extensions to the network library, which will allow us to check applications using features such as datagrams. We also intend to explore another alternative to centralization, which consists of I/O extensions in the model checker, which transparently relay inter-process communication packets to external processes [3]. This approach model checks only a single process at a time and prevents the process inside the model checker from physically sending the same data several times.

Acknowlegdements Thanks go to Shinichi Nagano for writing the initial version of the chat server application.

References

[1] C. Artho. *Combining Static and Dynamic Analysis to Find Multi-threading Faults Beyond Data Races*. PhD thesis, ETH Zürich, 2005.

[2] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Proc. 16th Intl. Conf. on Computer Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 462–465, Boston, USA, 2004. Springer.

[3] C. Artho, B. Zweimüller, A. Biere, and S. Honiden. Software Model Checking of Distributed Applications with I/O. To be published.

[4] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. 7th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 268–285, Genova, Italy, 2001. Springer.

[5] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

[7] C. Colby, P. Godefroid, and L. Jagadeesan. Automatically closing open reactive programs. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 1998)*, pages 345–357, Montreal, Canada, 1998.

[8] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd Intl. Conf. on Software Engineering (ICSE 2000)*, pages 439–448, Limerick, Ireland, 2000. ACM Press.

[9] The Apache Foundation, 2006. `http://www.apache.org/`.

[10] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL 1997)*, pages 174–186, Paris, France, 1997. ACM Press.

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

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

[13] G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.

[14] G. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. 12th Intl. Conf. on Protocol Specification, Testing, and Verification, INWG/IFIP*, Orlando, USA, 1992.

[15] G. Holzmann and M. Smith. A practical method for verifying event-driven software. In *Proc. 21st Intl. Conf. on Software Engineering (ICSE 1999)*, pages 597–607, Los Angeles, USA, 1999. ACM Press.

[16] D. Lea. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, 1999.

[17] T. Lindholm and A. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[18] Y. Nakagawa, R. Potter, M. Yamamoto, M. Hagiya, and K. Kato. Model checking of multi-process applications using SBUML and GDB. In *Proc. Workshop on Dependable Software: Tools and Methods*, pages 215–220, Yokohama, Japan, 2005.

[19] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[20] S. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In *Proc. 7th Intl. SPIN Workshop (SPIN 2000)*, volume 1885 of *LNCS*, pages 224–244, Stanford, USA, 2000. Springer.

[21] S. Stoller and Y. Liu. Transformations for model checking distributed java programs. In *Proc. 8th Intl. SPIN Workshop (SPIN 2001)*, volume 2057 of *LNCS*, pages 192–199. Springer, 2001.

[22] Sun Microsystems, Santa Clara, USA. *Java 2 Platform Standard Edition (J2SE) 1.5*, 2005. `http://java.sun.com/j2se/1.5.0/`.

[23] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.