Software Model Checking for Distributed Systems with Selector-Based, Non-blocking Communication

Cyrille Artho*, Masami Hagiya†, Richard Potter†, Yoshinori Tanabe‡, Franz Weitl§, and Mitsuharu Yamamoto§
*AIST/RISEC, Amagasaki, Japan
c.artho@aist.go.jp
†The University of Tokyo, Tokyo, Japan
hagiya@is.s.u-tokyo.ac.jp, potter.richard@gmail.com
‡National Institute of Informatics, Tokyo, Japan
y-tanabe@nii.ac.jp
§Chiba University, Chiba, Japan
franz@chiba-u.jp,mituharu@math.s.chiba-u.ac.jp

***Abstract*—Many modern software systems are implemented as client/server architectures, where a server handles multiple clients concurrently. Testing does not cover the outcomes of all possible thread and communication schedules reliably. Software model checking, on the other hand, covers all possible outcomes but is often limited to subsets of commonly used protocols and libraries.**

Earlier work in cache-based software model checking handles implementations using socket-based TCP/IP networking, with one thread per client connection using blocking input/output. Recently, servers using non-blocking, selector-based input/output have become prevalent. This paper describes our work extending the Java PathFinder extension net-iocache to such software, and the application of our tool to modern server software.

***Index Terms*—software model checking; caching; software verification; distributed systems; non-blocking input/output; selector-based input/output**

I. Introduction

Modern client/server architectures are complex software systems. In addition to the processes involved on the client and server sides, the server is usually written as concurrent software using multiple threads [1] internally. This introduces two dimensions of non-determinism: Both the thread schedule of the software, and the order in which incoming messages arrive, cannot be controlled by the application. Application defects that depend on the timing of events are therefore extremely hard to find and reproduce using traditional testing.

Model checking [2] provides an automated analysis of concurrent systems under test (SUT). Classical model checking operates on an abstract model of software. *Software model checking* implemented by tools such as Java PathFinder apply the same principles to implementations, executing the implementation code at run-time [3]. A software model checker analyzes the state space of an application by backtracking the entire application state (thread stacks and program counters, and the heap) to a previously stored checkpoint. A problem arises if this approach is applied to systems communicating with other components; if external components are not managed by the model checker, their state will be inconsistent with the SUT after backtracking. Previous work resolves this by using a caching data structure that stores all previously recorded network traffic and maps the application state of the SUT to communication states [4] to overcome this problem.

Previous work handles various types of applications that use socket-based, blocking input/output (I/O) operations [5], [6]. Such operations suspend the currently active thread until the information is transmitted over the network. This has the advantage that the server side is relatively intuitive to implement, because one worker thread typically handles one connection (and thus one client). On the other hand, modern selector-based I/O libraries allow a single thread to handle multiple connections at once. In a finely-tuned system, this has shown to yield better performance (due to a lower overhead for thread management) although the implementation architecture is more complex [7], [8].

When model checking a server *algorithm*, the distinction in the implementation library may lie below the level of abstraction used. However, when analyzing the application *code* in a software model checker, the use of such I/O libraries requires special tool support. Initially, this appeared to be just a matter of layering the extra semantics on top of the existing tool. Unfortunately, subtle issues arose from the non-blocking behavior of communication operations. In particular, the network data cache polls the peer processes after data has been sent, to determine the structure of the protocol used [4]. By this approach the cache introduces an intermediate data layer where messages are stored before they are seen by the SUT. This made it difficult to handle end-of-file semantics correctly, and required adaptations of the existing tool.

To our knowledge, no other tool exists that exhaustively analyzes application code using non-blocking, selector-based networking against live peer processes. Our tool supports all important aspects of Java's network library `java.nio` on top of the Java PathFinder tool. We have first analyzed our tool using Modbat [9], a model-based test case generator that is designed to test libraries with non-blocking I/O and exceptions. Then, we have applied our tool to a test server and to rupy, a high-performance light-weight HTTP server [10]. We successfully found a seeded bug in the former and a previously unknown race condition in the latter.

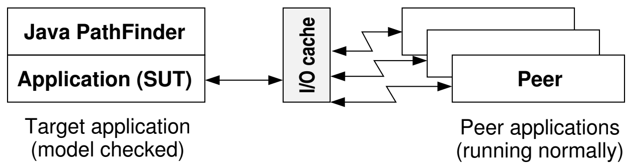
Java PathFinder
Application (SUT)
I/O cache
Peer
Target application (model checked)
Peer applications (running normally)

Figure 1. General architecture of `net-iocache`.

This paper is organized as follows: Section II gives background on our general approach and on blocking and non-blocking I/O. Section III lists related work, and Section IV describes our implementation. Our experiments with various client/server systems are shown in Section V. Section VI concludes and outlines future work.

II. Background

This section briefly introduces our approach to handling client/server systems in a software model checker, and gives the necessary technical background on non-blocking I/O.

A. Cache-based Verification

A software model checker executes a system under test (SUT) until a non-deterministic operation or scheduling decision affects the outcome of the program. The location of such an event is called a *choice point.*[1] At each choice point, the software model checker stores the state of the entire program, adds that state with the current choices to the set of states to be explored, and continues its analysis. Unlike in "traditional" model checking, the structure of the system is not defined a priori as a Kripke structure, but explored dynamically; the structure of program states is likewise complex and dynamic due to dynamic memory allocation in the SUT. Therefore, a software model checker typically executes the SUT at run-time and explores different program states explicitly [3].

In this paper, the term backtracking denotes the restoration of a previous state, even if that state is not a predecessor of the current state. As the execution of the SUT is subject to backtracking, this has implications on networked software. External processes, called *peers,* can usually not be backtracked. If the software model checker backtracks only the SUT but not the peers, then their states (and the state of the communication links) are no longer consistent.

Possible solutions to synchronize the state of the SUT with its peers after backtracking include restarting the peer processes when needed, replaying previously recorded communication to reach the desired peer state again [5], or using a virtualization environment to store snapshots of the peer processes (thus extending the state management capabilities of the model checker to both the SUT and its peers) [6]. In both cases, the use of a cache layer in between is essential for a useful verification performance on realistic systems [5], [6].

Our Java PathFinder extension `net-iocache` supports the analysis of client/server systems on Java PathFinder, by acting as a bridge between Java PathFinder executing the SUT, and peers communicating with the SUT (see Figure 1). This allows the peers to interact with partial executions of the SUT, while `net-iocache` maintains consistency between the different components. An important characteristic of this approach is that it is *process-modular;* it analyzes only one (client or server) process as the SUT in Java PathFinder, while the other components run outside. This results in a much better performance than if all processes were included in the software model checker [4].

B. Blocking and Non-blocking Input/Output

The conceptually simplest way of communicating with a disk or network is the use of *blocking input/output (I/O).* In a blocking operation, the current thread is suspended until the operation has either completed successfully or failed. This mode is suitable for relatively fast devices or if processing cannot continue until the result of the operation is known.

In modern servers, many clients are served in parallel. Thus, having the server process suspend itself until a message is transmitted would not be useful. Because of this, multiple worker threads [1] are commonly used in a server, such that other workers can continue processing while a given worker thread is suspended. The most common server architecture using blocking I/O therefore consists of one server main thread, which accepts incoming client requests, and one server worker thread per active client connection [8].[2] Excluding possible auxiliary threads (for example, for bookkeeping), this architecture therefore uses a total of $n + 1$ threads to serve n clients. Even with multi-core, hyper-threaded processors being the norm in these days, this architecture has the following shortcomings:

1) The number of processor cores may be lower than the number of active connections.
2) A thread is frequently suspended waiting for I/O.

Both problems require the operating system to switch between threads on a given processor core, introducing overhead. Because of this, *non-blocking I/O* operations are used prevalently in high-performance servers. Unlike its counterpart, a non-blocking operation completes (practically) immediately; however, the result of the operation may be incomplete. In practice, this means that non-blocking `connect` and `accept` operations may fail to obtain a working connection; non-blocking `read` and `write` operations may transmit no data at all, or only incomplete data. Instead of checking multiple communication channels individually, *selectors* can be used to poll multiple channels at once in a single operation, obtaining a set of channels where data is available. The use of non-blocking I/O therefore requires a wider array of operations with the following additional, significant implementation complexity:

1) Application code needs to include extra loops, and buffer management code, to retry transmission if a previous attempt was incomplete.

[1]In Java PathFinder, scheduling choice points are not predefined but detected at run-time based on accesses to shared memory between threads.

[2]The `accept` operation in the main thread blocks, but this does not affect ongoing client operations. Dispatching requests to a worker thread is considered to be sufficiently fast to usually not require more than one thread.

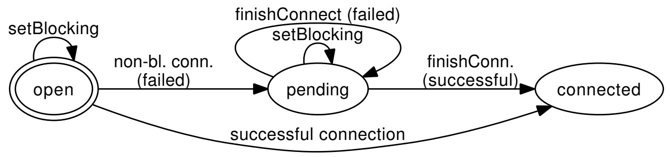

Figure 2. API for the blocking and non-blocking `connect` function.

2) A single worker thread may handle multiple client transactions simultaneously, which makes it harder to separate data between clients and increases the chance of accidental data corruption due to faulty code.

To summarize, non-blocking I/O can utilize processing cores better, but requires careful analysis. This makes it all the more important to bring software model checking to such systems.

C. The `java.nio` Library

Java offers non-blocking I/O as part of the `java.nio` package. For communication over a network, three components of that package are essential [11]:

- *Buffers* are containers for data.
- *Channels* represent connection entities. These include server-side ports that can accept an incoming connection (`ServerSocketChannel`) and connection handles to send and receive data over an active connection (`SocketChannel`).
- *Selectors* can query multiple channels at once on their availability. Channels to be queried are registered using *selection keys.*

Our work models these components (packages `java.nio` and `java.nio.channels`). As `net-iocache` does not model file I/O, and as our work is not concerned with Unicode encoding, packages `java.nio.charset` and `java.nio.file` are elided.

In this library, the semantics of blocking and non-blocking operations are of particular interest. The application programming interface (API) of `java.nio` allows to switch between blocking and non-blocking modes at any time; for example, the blocking mode can be used if the result of an operation is needed to continue. Most operations have the same interface in both modes, with one particular exception: class `java.nio.channels.SocketChannel`, which represents a connection handle, has a special API to connect to a server in non-blocking mode.

In the following discussion of `SocketChannel`, it is assumed that an instance of that class has already been created using `SocketChannel.open`. In that case, the library user may either connect in blocking mode (the default), or change the mode using `setBlocking`. A blocking connection attempt waits until a connection is established; a non-blocking attempt may fail to establish a connection. In that case, the failed connection attempt may not be retried in the same way, but must be finished by invoking `finishConnect` instead [11]. Implicit in this requirement is a state change in the connection object to a *pending* state (see Figure 2). In that pending state, the blocking mode may again be changed. In non-blocking mode, `finishConnect` may again be unsuccessful, and it can be retried until it succeeds; in blocking mode, `finishConnect` completes as expected.

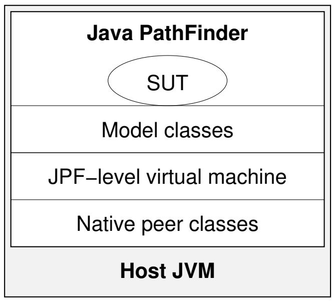

Figure 3. Design of Java PathFinder.

The Java library does not permit the sequence of connection operations to deviate from this prescribed usage; incorrect library calls result in exceptions such as `ConnectionPendingException` [11].

D. Java PathFinder Architecture

Java PathFinder (JPF) implements a Java Virtual Machine (JVM) that is capable of backtracking the program state to an earlier version, to explore multiple outcomes of a non-deterministic decision. JPF is itself written in Java and runs on a standard JVM ("host JVM"). By managing the outcome of each Java bytecode instruction, JPF can execute any Java program that does not include native (machine) code, or any indirect dependency on native code. Unfortunately, most interesting functions such as I/O, depend on native code.

Native code executes outside the JVM, so its side-effects cannot be managed by JPF. Any native code therefore needs to be replaced with a *model class.*[3] The model class implements the same interface as the class to be replaced, but is written entirely in Java. It also executes on the JPF-level VM (see Figure 3). Model classes can be used to represent native code that does not affect the environment.

Non-determinism can be modeled using *choice generators.* A choice generator in JPF creates a set of possible outcomes, each of which is explored as a possible successor state.

Code that interacts with the environment needs to do so by calling the corresponding native function on the host JVM. Such code is implemented as *native peer classes* and runs in a separate namespace. It gets called from the JPF-level VM and then calls the underlying host JVM.

A model class executes on the JPF-level VM; its state is therefore backtracked together with the SUT. The native peer classes execute on the host JVM and are thus not subject to backtracking. In designing `net-iocache`, data structures that should be "persistent" across backtracking therefore need to be written as a native peer class while data that should be

[3]JPF throws an exception and stops verification when it encounters native code that is not supported by a model class.

synchronized with the SUT has to reside in a model class. Note that it is not necessary to replace all classes within a given library to make the library compatible with JPF; only classes with native code, or dependencies on native code, need replacement.

III. Related Work

In addition to earlier work on `net-iocache` mentioned above, several other approaches exist to verify distributed systems. In the context of software model checking, a global approach analyzes all components inside the model checker. This means that all processes are executed in tandem.

The earliest tool of that type is *Verisoft* [12], which verifies a target implementation consisting of multiple processes. Processes are executed in a debugger to control the SUT and handle the application state space. However, threads inside a process are not supported, and the state of file descriptors and sockets is not maintained. The former limitation is fundamental as the thread schedule of the SUT is not controlled by Verisoft, and Verisoft relies on re-executing a program to a given point to restore a given system state. Therefore, modern applications composed of multiple threads are not directly applicable. Other tools such as *CHESS* [13] or *inspect* [14] support multiple threads and can therefore control and execute the state space of multiple processes systematically. The platform used in our work, *Java PathFinder* [3], has the advantage that it does not only control program states, but also stores and compares them, enabling certain types of optimizations that are not applicable in a stateless search [15].

Other multi-process software model checkers use virtual execution environments [16]. Although a debugger still controls the processes, the use of virtualization technology allows the software model checker to create and restore SUT checkpoints. This makes the analysis of multi-threaded software possible, but the usage of virtualization on the level of the operating system executing the SUT brings a high overhead.

Finally, program transformation can convert multiple processes to threads and create a self-contained, single-process application. This transformation, called *centralization,* allows distributed systems to be analyzed by existing tools that have not been designed for such tasks [17], [18], [19]. Compared to our work, that approach analyzes all components of the system together. That analysis brings with it a large state space but is more comprehensive than our modular approach, which investigates one process at a time.

Another way to analyze distributed systems in a modular way is to replace peer components by *stubs,* small pieces of code that produce the desired outcome for communication with peers under a given test scenario. Tools like *Netstub* exist to facilitate the creation of such stub implementations [20], including even attempts to synthesize a stub model from a given execution. Netstub supports event injection for code using `java.nio`. Our approach distinguishes itself in that *live data,* not code, is used to represent the behavior of peer components. In our work, peer components execute against the SUT, whereas they are replaced by stubs in Netstub.

IV. Design and Implementation

This section describes the design and key aspects of the implementation of `java.nio` in `net-iocache`, including a first attempt that was not sufficient to handle all cases.

A. Layered Approach

The previous version of `net-iocache` implements the necessary classes in `java.net` and `java.io` to handle socket-based network communication with blocking I/O [4], [5], [6]. This implementation polls peer processes proactively for a response each time data has been sent. Using this technique, a data structure reflecting the request/response pairing of messages is created [4]. This approach hides the exact recorded interleaving of incoming messages from the SUT. Instead, the thread schedule generator of JPF generates all possible schedules of operations that consume the data (or send it), thus ensuring full coverage of all possible message interleavings and program behaviors.

Our first idea was to layer non-blocking I/O on top of blocking operations in the Java library. This is possible in principle because even the “classical” blocking library in `java.io` has one non-blocking operation, `available`, which returns a lower bound on the number of bytes that can be read from a data stream without blocking [11]. Based on this observation, it is possible to build a working, although less efficient, version of `java.nio` using only Java code. In JPF, this would require only model classes but no native peers, allowing 100 % code reuse of `net-iocache`. Such a library could even be used as a compatibility toolkit for an older version of the JVM that does not support `java.nio` yet.

The design works as follows: *read* operations first check `available` to read only available data. Data to be *written* over the network is written to a worker queue instead. This ensures immediate completion of the operation; a worker thread would regularly poll that queue and write the data physically over the network. Accepting incoming connections is done in a similar way with an extra thread accepting connections on a (predefined) port, and queuing accepted (ready) connections in a shared data structure. Finally, connections to a server are made using a blocking call; in the given test environment, the test server is assumed to be always available.

Table I shows a summary of such an implementation in Java-like pseudo code. Note that `conn` refers to the underlying socket of the given `SocketChannel` instance; `buf` (`buf.clone`) is (a copy of) the underlying buffer in `ByteBuffer`; `in` is the underlying input stream of socket `conn`; `writer` is the queue of `(Socket, byte[])` entries to be written; and `pendingAcc` is a queue of incoming connections.

The pseudo code shown here covers all the successful cases (complete operations). Partially completed or failed operations can be emulated using choice generators in JPF. Exceptions also need to be thrown if the connection is in an incorrect state. These features will be explained below.

Channel selectors are built by polling connections (using `available`) instead of using a real selector. This provides

Table I
LAYERED IMPLEMENTATION OF `JAVA.NIO`.

Non-bl. operation	Emulation using blocking I/O (`java.net`, `java.io`)

read	`int read(ByteBuffer sink) {`
	`  int n = conn.available();`
	`  return in.read(sink.buf, 0, n); }`

write	`int write(ByteBuffer src) {`
	`  writer.enqueue(conn, src.buf.clone());`
	`  return src.remaining(); }`

accept	`SocketChannel.accept() {`
	`  return pendingAcc.poll(); }`

connect	`boolean connect(SocketChannel dst) {`
	`  conn.connect(dst);`
	`  return true; }`

the same result at a lower performance: A `select` call on n file descriptors is $O(1)$ while the emulation is $O(n)$. Because the overall state space exploration is exponential in complexity, the small overhead of polling a few file descriptors was not expected to be significant.

B. Limitations of Layered Approach

The layered approach is very appealing, because it is elegant, independent from JPF and `net-iocache`, and it decouples non-blocking operations from actual I/O. Early versions of our work could indeed be tested both on the normal JVM and on JPF. Unfortunately, at some point the limitations of this approach became apparent and insurmountable.

First, the code in Table I includes several background threads that poll the physical network resources in use or to be used. Such background threads, when running on JPF as model classes, are subject to JPF's scheduling and model checking. Because JPF executes one thread at a time using an interleaving semantics, a blocking `accept` operation could potentially endlessly stall verification by blocking JPF itself. As JPF executes the SUT synchronously, it would be waiting, possibly indefinitely, for the `accept` operation to complete. Existing work [4] solves this problem by managing incoming connections and peer processes on a native level.

Therefore, threads inside `net-iocache` that manage communication with peer processes must execute as native peers. The need to manage connections on the level of native peer classes breaks the total independence from JPF and also requires code to be split into model classes and native peers, losing the simplicity of the approach.

Furthermore, our tests using various client/servers revealed another subtle problem when analyzing applications that depended on the correct simulation of possible end-of-file (EOF) return values. In blocking mode, it is never possible to read zero bytes unless a socket is closed (the end-of-file case).[4] Therefore, previous versions of `net-iocache` returned EOF in case nothing could be read from a socket input stream (i. e, no response to a sent request arrived within a certain time). Unfortunately, for a non-blocking read operation, reading zero bytes is a common occurrence of a partially completed operation. Because `net-iocache` internally polls peer processes and caches their responses, it is not possible to transparently layer a library that needs to check if EOF is reached. Hence, EOF must be detected on the native level and forwarded to the model to be distinct from the "nothing could be read" case.

[4]Instead, the `read` operation blocks until data is available.

C. Adaptation of `net-iocache`

From the limitations above, the adaptations of the preliminary version of our model library follow. We still layer non-blocking I/O on top of blocking I/O. We also still use choice generators at a model class level to decide to what extent a non-blocking operation should be carried out. This guarantees that we observe the full state space of partially completed operations.[5] This choice maps the outcome of complete and partially complete communication to different program states. Failed non-blocking connection attempts are also tracked at the level of model classes (see Figure 3) and tied to a program state. Whether an operation completes or not is therefore not visible at the lower level of native peer classes.

In the adapted implementation, non-blocking I/O is not executed on top of `java.net` and `java.io` alone; instead, we re-use facilities that `net-iocache` provides, in particular its connection management and its ability to cache incoming data from peer processes. Some adaptations in `net-iocache` were also made to cover end-of-file semantics.

Communication buffers were exempt from the redesign; they are only containers for data to be transmitted and therefore can be modeled fully by model classes.[6] We describe the other aspects of the final design below.

1) Connection management: Connections and communication using `java.nio` (including non-blocking I/O) and `java.net` (the traditional library) are managed in a uniform way. A central class `CacheLayer` manages connections, allocating a tree data structure for each communication channel to store (possibly diverging) communication contents that are discovered as the state space is explored [5]. The existing connection management code could be reused.

The data structure storing communication data has been extended to include an end-of-file event that allows `net-iocache` to distinguish between non-blocking read operations returning no data and the actual end of a connection. End-of-file events are converted to return value -1 by the cache layer main class; low-level internal events are thus not visible to model classes.

2) Non-blocking `accept`: The background thread that accepts incoming connections is removed from the model and handled by the cache layer main class, executing as a native peer on the host JVM. This prevents the thread from being included in the state space analysis of JPF. On the functional level, the only difference between non-blocking and blocking

[5]In practice, we restrict the choices to some boundary values for efficiency.
[6]The implementation of buffer classes in the standard Java library includes native code for optimization reasons, which is why an equivalent Java model class was needed.

accept is that non-blocking accept may return null while blocking accept always returns a socket unless an I/O fault occurs. As a result, we can simulate non-blocking accept by introducing a choice point with two branches representing the two possible outcomes: the successful case (a socket channel is returned) and the unsuccessful case (null is returned). On the *successful* branch, a client process is launched, a blocking native-level accept is called, and the obtained connection is returned.

On the native level, accept is blocking. Because JPF simulates different threads with an interleaving semantics, the time spent in the native accept call is not observable by the SUT, as no other SUT threads can execute during that time. This means that successful blocking and non-blocking accept calls can be simulated in the same way on the native level, while the model class implements the different behaviors of the two modes.

The result of the choice made to model a non-blocking call has to be remembered between calls: a connection that is unavailable may become available later, but the opposite is not possible unless the available connection handle has been used (in this case, by calling accept). Indeed, correctly updating the internal state after every operation was one of the biggest challenges of our work, and the source of a subtle defect we found in an earlier version of our model, as described in Section V.

Algorithm 1 shows the implementation of accept.[7] The method call in the first line performs some sanity checks. After that, a call to isReady simulates a possible network delay. If in blocking mode or in the successful branch of non-blocking accept (i. e., isReady returns true), the remote client is launched and a blocking accept on the native level of JPF is executed. Otherwise, null is returned, representing the unsuccessful execution branch of non-blocking accept.

3) Non-blocking connect: Non-blocking connect is implemented according to the state diagram of Figure 2. In non-blocking mode, a call to SocketChannel.connect(dst) returns immediately, but may return false to indicate that the socket channel is in a pending state and and not yet connected. In blocking mode, SocketChannel.connect(dst) always succeeds unless an I/O error occurs.

We use three variables to manage the internal state of a connection: ready (similar as in accept), a state variable connState that reflects the internal state of a connection (see Figure 2), and variable pendingConnAddr to remember the address of the pending connection. The underlying blocking connect call by net-iocache is only made if meant to succeed (see Algorithm 2), similar to accept above.

4) Non-blocking communication: The implementation of read and write operations follows the pattern above in that possibly incomplete outcomes of non-blocking operations are modeled using choice generators. According to the result, the operation is then executed completely by net-iocache. A read of length 0 results in no actual operation; a non-zero read is executed natively in blocking mode.

Algorithm 3 shows the implementation. The first few lines in read perform some sanity checks. After that, a call to isReadable simulates the (end of) a possible network delay. Finally, communication data is read from the cache layer. Note that much of the complexity is hidden in the call to in.read,

[7]For brevity, we elide details such as declaring methods as public, and the full range of exceptions thrown.

Algorithm 1 Implementation of accept in net-iocache.

```
boolean ready; // remember choice for non-blocking accept

boolean isReady() {
  ready = ready || Verify.getBoolean();
  // use choice generator to simulate (end of) network delay;
  // each I/O attempt may result in a resource becoming available
  return ready;
}

SocketChannel accept() throws IOException {
  checkConnectionState(); // throw exc. if not connected

  if (isBlocking() || isReady()) {
    ready = false;
    return new SocketChannel(nativeAccept());
    // reset ready flag, launch the remote client process,
    // and call native-level blocking accept
  } else return null;
  // simulate possible failure of non-blocking accept
}
```

Algorithm 2 Implementation of connect.

```
boolean ready;// remember choice for non-blocking connect
boolean isReady() { ... } // same as in accept

boolean connect(SocketAddress dst) throws... {
  checkConnectionState(NOT_CONNECTED);
  // throw exception if not initialized or in wrong state
  pendingConnAddr = remote;
  connState = PENDING;
  return tryConnect();
}

boolean finishConnect() throws IOException {
  if (connState == CONNECTED)
    return true;

  checkConnectionState(PENDING);
  return tryConnect();
}

boolean tryConnect() throws IOException {
  if (blocking || isReady()) {
    bl_Connect(pendingConnAddr);
    return true;
  } else return false;
}

void bl_Connect(SocketAddress dst) throws... {
  socket.connect(dst);
  ready = false;
  connState = CONNECTED;
}
```

Algorithm 3 Implementation of `read`.

```
boolean isReadable() {
  readable = readable || Verify.getBoolean();
  // use choice generator to simulate (end of) network delay
}

int read(ByteBuffer buf) throws IOException {
  checkConnectionState(CONNECTED);
  int count = buf.remaining();
  if (count == 0) return 0;  // no space in buffer

  if (!blocking && !isReadable()) return 0;
    // simulate possible delays for non-blocking read

  synchronized (readLock) {
    byte[] bytes = new byte[count];
    count = in.read(bytes);  // fetches data from cache
      // CacheLayer.read may return –1 (EOF)
    if (count > 0) {
      buf.put(bytes, 0, count);
      readable = false;
    }
  }
  return count;
}
```

which takes cached data that has been stored after polling the peer side following a previous write operation [4].

Write operations are handled in the same way as `read` above, except that no end-of-file can occur.

5) Selector-based operations: As mentioned above, a `select` operation is implemented iteratively by checking each channel one by one. Our design uses the current (logical) state of the model class, reflecting the results of any previously made non-blocking calls. This eliminates a need to apply a choice generator in the selector class as well. Queries to `isReady` either return an existing available state or may instead make data available, simulating the possibility of data becoming available prior to the `select` operation. A part of the implementation of select is shown in Algorithm 4. Internal checks that set the key state are all analogous to the code shown in `checkAcceptable`: If a key is set up to query that particular feature, the underlying channel is checked. On a positive result, the key state is updated accordingly.

V. Experiments and Discussion

A. Testing `java.nio` in `net-iocache`

The main challenge in a correct implementation of the `java.nio` API is the fact that most operations can be invoked in blocking and non-blocking mode; the mode can be switched between operations. This creates a very large state space for possible uses of the API. A good test coverage would require many manually written unit tests. Instead, we used only few unit tests to cover some short test sequences involving corner cases. Longer test sequences were generated with Modbat, a model-based test tool [9].

We briefly introduce the key features of Modbat here. Modbat uses an extended finite state machine [21] as its underlying model. State transitions are labeled with *actions* that are defined as program code (functions implemented in Scala [22]). This program code can directly execute the system under test (in our case, parts of the Java API). In addition to that, Modbat also supports exception handling, by allowing a declaration of possible exceptions that may result by (failed) actions. Finally, Modbat supports non-blocking I/O by allowing the specification of *alternative target states* to cover both the successful and the failed (incomplete) outcome of non-blocking I/O.

We have modeled the usage of the key classes `ServerSocketChannel` (see Figure 4) and `SocketChannel` (see Figure 5) with Modbat. Both APIs have in common that a channel object first needs to be created by calling `open`. Our models take the resulting state as the initial state. In the server case, the created object represents the ability to accept incoming connections; the object therefore also needs to be bound to a port and IP

Algorithm 4 Implementation of `select`.

```
int select(long timeout) throws IOException {
  for (SelectionKey key : registeredKeys) {
    key.readyOps(0); // reset the key ready ops
    key = checkAcceptable(key);
    key = checkReadable(key);
    key = checkWritable(key);
    if (key.readyOps() != 0)
      readyKeys.add(key);
  }
  return readyKeys.size();
}

SelectionKey checkAcceptable(SelectionKey k) {
  if ((k.interestOps() &
       SelectionKey.OP_ACCEPT) != 0) {
    if (k.channel().isReady())
      k.setFlag(SelectionKey.OP_ACCEPT);
  }
  return key;
}
```

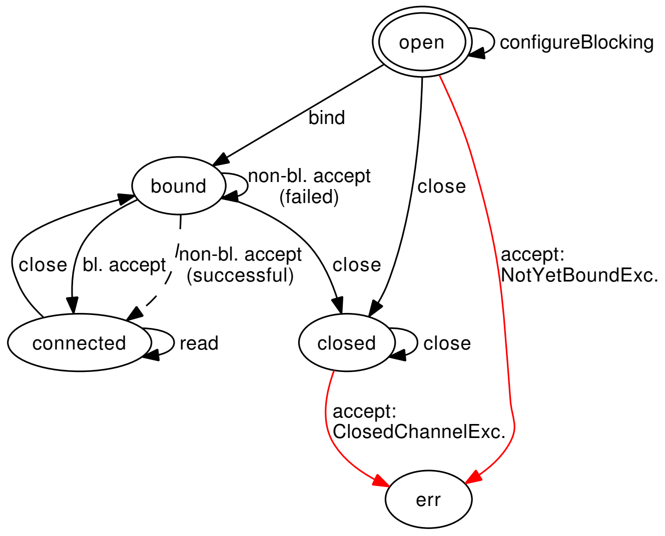

Figure 4. API model for `ServerSocketChannel`.

address before a connection can be accepted. In the client case, the connection can be established directly by supplying the IP address and port of the server as a function argument. However, the client API is slightly more complex in general in the sense that finishing a pending connection attempt requires a different function than the initial attempt (see Section II), and there are more possible exceptions.

In the figures, dashed transitions correspond to the successful (completed) case of a non-blocking operation that would otherwise have to be repeated (non-blocking `accept`) or end up in a *pending* state (non-blocking `connect`). Red, accordingly labeled edges correspond to exceptions resulting from actions that are not allowed in a given state. In these cases, the edge label denotes the exception type, where “Connection” is abbreviated to “Conn”, “Channel” to “Ch”, and “Exception” to “Exc.”. Some nodes have a self-transition that denotes a possible switch from blocking to non-blocking mode using `configureBlocking` (“confBl.”). A self-loop may also denote a retry of a previously failed non-blocking action; in the successful case, the dashed alternative transition is taken to the *connected* state. Finally, there is a self-transition in the *connected* state that reads from the newly connected channel before the connection is closed again.

When testing the server API, a client that connects to the open port is launched as defined by the test model (to ensure that the test proceeds); when testing the client side, a counterpart server is running in the background.

Modbat uses a random walk through the model, taking one of all available transition with each step. Given the availability of alternative choices and outcomes, this quickly generates thousands of distinct test cases.[8] We first executed the test cases against the standard Java implementation, using it as a reference implementation. This ensures that no false positives are reported by the test model. We then used the given test model in a second test run, against our network model for JPF.

As we used Modbat to reduce the need for manually written unit tests, some of the defects were found against work in progress. However, we also found one interesting defect in our model that would probably have gone unnoticed by manual testing. The necessary test sequence requires switching from non-blocking mode back to blocking mode (see Algorithm 5 for a simplified version). It furthermore requires another operation after the key step (finishing a previously unsuccessful connection attempt) has succeeded. The reason for this is that in the faulty model, the key operation returns the correct result but fails to update its internal state. So only tests involving an extra test operation can find this defect. Such tests tend to be rarely written by human developers [23], which is why existing manually written unit tests have not covered that behavior.

After our implementation passed the test sequences generated by Modbat, we were confident to apply our network model library to real client-server systems.

[8]An exhaustive analysis of such a model is only possible up to certain bounds due to the infinite state space of the model.

Algorithm 5 Test sequence for blocking `finishConnect`.

```
conn = SocketChannel.open();
conn.setBlocking(false);
conn.connect(remoteAddr); // returns false (fails)
conn.setBlocking(true);
conn.finishConnect(); // returns true (succeeds)
conn.finishConnect(); // expect AlreadyConnExc.
```

B. Performance Analysis

Regression testing scripts were created to easily set up automatic tests of new client/server systems. After minimal details of the client and server are given, the tests can be run in various ways. The two major variations are testing the client as the SUT with the server as the peer, and vice versa. In addition, parameters can be passed to the client and server programs for varying runtime behavior such as the number of network connections created. The detailed model checking logs created by JPF are automatically timestamped, saved, and compared with the log file from a previous execution (if available) to check against unexpected changes in the output.

Our test application for regression and performance testing is the alphabet client/server system. In this system, the SUT (the server) waits for a number from a client (peer). After a number n, followed by a newline character, has been received, the nth character of the alphabet is sent back as a response. The server terminates after a preconfigured number of clients has been served, each of which requests a predetermined number of messages. Unlike the old version of the server [4], the new server that is based on `java.nio` is *single-threaded*; its ability to accept and serve multiple connections at the same time lies in the fact that selector-based I/O is used to handle operations on each channel (`accept`, `read`, `write`) without blocking. Instead of blocking when no input is available on one channel, execution immediately proceeds with the next available channel. The complexity of the state space therefore stems from possible combinations of successful and failed I/O attempts, not from thread interleavings.

The experiment was run on an 8-core Mac Pro workstation with 24 GB of physical memory, running Ubuntu 10.10 and JPF 6 (change set `960:465508688048`) running on Java 1.6.0_11. Table II shows the results. The first three columns describe the system and its configuration; for reference, we also ran the same experiment on a `java.net`-based variant of the alphabet server [4] that uses multiple threads instead of selector-based I/O. The central three columns list three key numbers that show the size of the system: Execution time (in minutes and seconds), the number of states, and the number of bytecode instructions (in millions) executed in the entire analysis.[9] Finally, from `net-iocache`, cache hits/misses and the number of connections to peer processes is shown. Analysis

[9]It should be noted that the cost of bytecode instructions is very non-uniform, as an `invoke` instruction may execute native code that does not count towards the instruction tally. Still, the number gives an indication of the state space size.

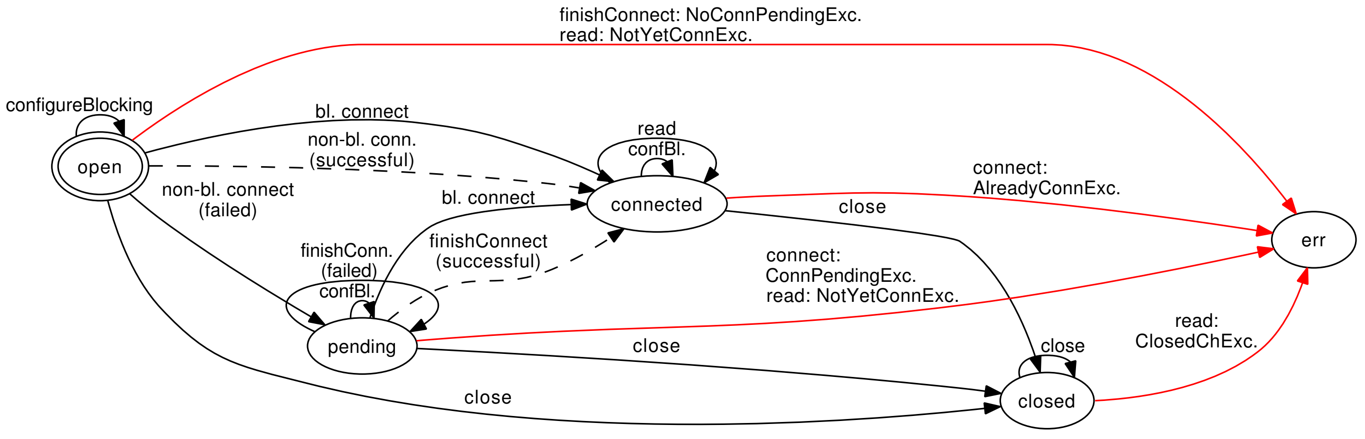

Figure 5. API model for `SocketChannel`.

runs with larger parameter settings than the ones shown did not complete within one hour.

It can be seen from that table that the state space of the variant using `java.nio` is growing much more quickly than for the conventional thread-based implementation. The reason for this is that each non-blocking I/O operation creates a choice point, causing the state space to potentially double. Of course, in practice most operations execute in a loop, so the state after a failed I/O operation often matches with an existing state (as the internal server state does not change and the operation will be retried later). Indeed, as can be seen, a larger number of messages (which affects the number of loop iterations) does not always increase the overall state space, even though the amount of transmitted data is of course affected.

To allow an analysis of larger systems, we also included variants of our model where certain non-blocking calls were modified such that they would always succeed. This essentially limits the simulation of network delays to operations other than the one indicated.

Finally, we also tested the capability of our model to find defects. We seeded a defect involving an incorrect use of selection keys in the alphabet server. We also added a check against a zero-byte read to discover the outcome of our mutation. Without the prior select-based check on data availability, the non-blocking read may return zero data. JPF always finds the defect in one second, regardless of the number of clients (shown as "*" in Table II). The statistics suggest that even in large settings, the fault is always detected after the third client connects.[10] As connections are independent of one another, the fault also manifests itself with only one or two clients or one message.

In the thread-based version, the size of the state space is mostly influenced by different thread interleavings. As the operation of one thread does not affect the global application state of the alphabet server, the state space grows more slowly than for the selector-based version. This means that despite their efficiency in normal execution [7], [8], common selector-based server implementations have a large inherent complexity that makes them more challenging to verify automatically in a software model checker.

[10]We verified this also for 99 clients.

C. Rupy HTTP Server

To test our tool against a real program, we chose the rupy HTTP server, version 0.4.4, "probably the smallest Java nio HTTP application server in the world" [10]. Despite its small size (4,500 lines of code), it implements many features; its non-blocking asynchronous design makes it ideal for high-concurrency applications.

The architecture of rupy involves three types of threads: a selector thread, which polls the open port and all current connections; a number of worker/event handler threads, which process HTTP requests; and a heart-beat thread, which controls termination. Because the high number of active threads combined with selector usage create a state space that is too large for JPF to handle, we remove the code in the heart-beat thread and add a hard-coded exit condition instead, which makes rupy terminate after a fixed number of connections has been served. We also stub out code related to date and date formatting functions, which are currently not supported by JPF.

With this setup, we are able to analyze the complete state space of rupy for one accepted connection (see Table III). While the analysis for two accepted connections did not terminate within one hour, a look at the log files revealed that many null pointer exceptions were caught and logged in the event framework. The intention of that code was to catch exceptions related to I/O, not to internal defects in the code. We therefore added an assertion to check against such null pointer exceptions. With that assertion, the case with one connection passed with the state space being increased by a mere two states. With two active connections, though, the defect is found almost immediately.

We identified a data race on the current worker thread event as the root cause of the null pointer exception. We therefore added `synchronized` statements wherever a shared object `Event` was used, to protect accesses with a lock. With the modified (fixed) version, the state space is actually smaller

Table II
RESULTS ON VARIOUS CONFIGURATIONS OF THE ALPHABET CLIENT/SERVER SYSTEM.

SUT	Param.		JPF performance			Cache statistics		
	#conn	#msg	time	#states	#M ins	#hit	#miss	#conn
Alphabet server using `java.nio`	3	2	0:20	21,167	58	6,144	6	3
		3	0:20	21,132	59	9,174	9	3
		4	0:21	21,085	59	12,248	12	3
		5	0:20	19,319	55	14,025	15	3
	4	2	14:32	1,024,777	2,689	253,690	8	4
		3	14:39	1,027,160	2,719	382,641	12	4
		4	15:01	1,030,336	2,752	512,256	16	4
		5	12:54	878,633	2,378	553,835	20	4
Alphabet server using `java.nio`; `accept` always succeeds	3	2	0:05	3,515	10	714	6	3
		3	0:05	3,515	10	1,071	9	3
		4	0:05	3,515	10	1,428	12	3
		5	0:05	3,523	10	1,785	15	3
	4	2	1:17	87,650	223	15,692	8	4
		3	1:17	87,062	224	23,607	12	4
		4	1:16	86,146	224	31,448	16	4
		5	1:18	85,447	224	39,625	20	4
	5	2	45:48	3,234,209	8,034	503,554	10	5
		3	45:35	3,180,206	7,972	760,629	15	5
		4	46:01	3,230,047	8,156	1,015,688	20	5
		5	46:54	3,225,251	8,223	1,254,225	25	5
Alphabet server using `java.nio`; `read` always succeeds	3	2	0:04	2,308	7	444	6	3
		3	0:04	2,300	7	666	9	3
		4	0:04	2,259	7	888	12	3
		5	0:04	2,249	7	1,115	15	3
	4	2	0:35	40,466	107	6,688	8	4
		3	0:36	40,374	108	10,125	12	4
		4	0:36	41,980	113	13,344	16	4
		5	0:36	41,233	112	16,355	20	4
	5	2	10:36	825,675	2,038	108,590	10	5
		3	10:50	824,890	2,061	164,463	15	5
		4	11:40	890,592	2,256	233,988	20	5
		5	11:32	870,881	2,228	284,035	25	5
Alphabet server using `java.nio`; `write` always succeeds	3	2	0:06	3,630	13	2,172	6	3
		3	0:06	3,765	13	3,426	9	3
		4	0:06	4,564	14	4,564	12	3
		5	0:06	3,533	13	5,350	15	3
	4	2	1:18	75,002	262	41,396	8	4
		3	1:23	78,371	274	65,733	12	4
		4	1:22	77,822	275	86,144	16	4
		5	1:12	67,664	241	95,980	20	4
	5	2	33:33	1,983,290	6,886	986,068	10	5
		3	35:23	2,080,656	7,246	1,555,407	15	5
		4	35:38	2,073,928	7,280	2,071,668	20	5
		5	29:34	1,707,459	6,018	2,155,315	25	5
Alphabet server using `java.net`	3	2	0:03	4,113	1	2,736	6	3
		3	0:04	6,433	1	5,009	9	3
		4	0:06	9,391	2	8,282	12	3
		5	0:07	13,059	3	12,393	15	3
	4	2	0:34	62,515	14	53,205	8	4
		3	0:55	107,849	25	109,501	12	4
		4	1:27	171,367	40	193,993	16	4
		5	2:07	257,083	62	313,911	20	4
	5	2	6:23	612,340	175	670,041	10	5
		3	12:29	1,190,399	347	1,555,940	15	5
		4	21:28	2,100,138	622	3,064,573	20	5
		5	35:38	3,461,419	1,038	5,453,376	25	5
Faulty alphabet server using `nio`	*	2	0:01	9	0	0	6	3
		3	0:01	9	0	0	9	3
		4	0:01	9	0	0	12	3
		5	0:01	9	0	0	15	3

Table III
RESULTS ON VARIOUS CONFIGURATIONS OF THE RUPY WEB SERVER.

SUT	Par.	JPF performance			Cache statistics		
	#conn	time	#states	#M ins	#hit	#miss	#conn
Rupy (defect)	1	5:22 (no defect found)	223,129	890	0	74	1
	2	0:03 (defect found)	1,303	65	0	148	2
Rupy (fixed)	1	2:34 (no defect found)	108,209	427	0	74	1
	2	timeout					

because fewer interleavings between threads are possible. In the modified version, no more null pointer exceptions are found by JFP within five hours. The developers of rupy confirmed the bug and fixed it in a newer version.[11]

VI. CONCLUSIONS AND FUTURE WORK

Servers are often implemented using non-blocking, selector-based input/output operations for efficiency. The verification of these servers is important, yet there is little tool support to analyze the outcome of non-blocking input/output operations exhaustively. We present an approach that layers non-blocking operations on top of `net-iocache`, which implements blocking networking for the Java PathFinder model checker.

Taking advantage of the interleaving execution semantics of Java PathFinder, we model each type of operation (`connect`/`accept`, `read`/`write`, and selector usage) by using a choice generator to determine the outcome of a (possibly incomplete) non-blocking operation before executing the result on `net-iocache`. As a result, we can handle client/server implementations that use the `java.nio` library, which allows us to verify complex asynchronous server applications. Experiments show that non-blocking input/output operations create a large, complex state space for the application to be analyzed. We believe this also has an implication on the mental burden of the developer, who has to write code that manages partial operations in many possible states. This fact makes automated verification tools all the more important. Our tool successfully found a data race in an existing HTTP server, which shows its practical usefulness.

Future work includes optimization of the state space (through limitations on the number of simulated failed communication attempts) and the implementation of a wider range of protocols and case studies. In particular, we are looking for protocols like FTP, where the information sent in one channel affects the input of another channel.

ACKNOWLEDGEMENTS

Thanks go to David Amar for this preliminary work on the implementation; the project had to be suspended after the 2011 Tohoku Earthquake. Thanks also go to Shouichi Kamiya for the continuation of that work, to Watcharin Leungwattanakit for his help in maintaining `net-iocache`, and to Eric Platon for his helpful feedback. This work was supported by *kaken-hi* grants 23240003 and 23300004.

[11] https://code.google.com/p/rupy/issues/detail?id=22

References

[1] A. Tanenbaum, *Modern operating systems*. Prentice-Hall, 1992.

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

[3] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering Journal*, vol. 10, no. 2, pp. 203–232, 2003.

[4] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe, "Efficient model checking of networked applications," in *Proc. 46th Int. Conf. on Objects, Models, Components, Patterns (TOOLS EUROPE 2008)*, vol. 19 of *LNBIP*, (Zurich, Switzerland), pp. 22–40, Springer, 2008.

[5] C. Artho, W. Leungwattanakit, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Cache-based model checking of networked applications: From linear to branching time," in *Proc. 24th Int. Conf. on Automated Software Engineering (ASE 2009)*, (Washington, DC, USA), pp. 447–458, IEEE Computer Society, 2009.

[6] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Model checking distributed systems by combining caching and process checkpointing," in *Proc. 26th Int. Conf. on Automated Software Engineering (ASE 2011)*, (Lawrence, USA), pp. 103–112, 2011.

[7] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 173, 2008.

[8] D. Kegel, "The C10K problem," 2013. http://www.kegel.com/c10k.html.

[9] C. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, and M. Yamamoto, "Modbat: A model-based API tester for event-driven systems," in *Proc. 9th Haifa Verification Conference (HVC 2013)*, LNCS, (Haifa, Israel), Springer, 2013. To be published.

[10] M. Larue, E. Martino, M. Funk, A. Chen, A. Lee, C. Lung, D. Hoyt, and H. Baghdasaryan, "rupy — A tiny Java nio HTTP application server," 2013. https://code.google.com/p/rupy/.

[11] Oracle, "Overview (Java platform SE 7)," 2013. http://docs.oracle.com/javase/7/docs/api/overview-summary.html.

[12] P. Godefroid, "Software model checking: The VeriSoft approach," *Form. Methods Syst. Des.*, vol. 26, no. 2, pp. 77–101, 2005.

[13] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing Heisenbugs in concurrent programs," in *Proc. 8th USENIX conference on Operating systems design and implementation (OSDI 2008)*, (Berkeley, CA, USA), pp. 267–280, USENIX Association, 2008.

[14] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan, "Dynamic model checking with property driven pruning to detect race conditions," in *Proc. ATVA 2008*, vol. 5311 of *LNCS*, pp. 126–140, Springer, 2008.

[15] C. Artho, W. Leungwattanakit, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Model checking of concurrent algorithms: From Java to C," in *Proc. Conf. on Distributed and Parallel Embedded Systems (DIPES 2010)*, vol. 329 of *IFIP AICT*, (Brisbane, Australia), pp. 90–101, Springer, 2010.

[16] Y. Nakagawa, R. Potter, M. Yamamoto, M. Hagiya, and K. Kato, "Model checking of multi-process applications using SBUML and GDB," in *Workshop on Dependable Software: Tools and Methods*, (Yokohama, Japan), pp. 215–220, 2005.

[17] S. D. Stoller and Y. A. Liu, "Transformations for model checking distributed Java programs," in *Proc. 8th Int. SPIN Workshop (SPIN 2001)*, (NY, USA), pp. 192–199, Springer-Verlag New York, Inc., 2001.

[18] C. Artho and P. Garoche, "Accurate centralization for applying model checking on networked applications," in *Proc. 21st Int. Conf. on Automated Software Engineering (ASE 2006)*, (Tokyo, Japan), pp. 177–188, 2006.

[19] L. Ma, C. Artho, and H. Sato, "Analyzing distributed Java applications by automatic centralization," in *Proc. 2nd IEEE Workshop on Tools in Process*, (Kyoto, Japan), IEEE, 2013.

[20] E. D. Barlas and T. Bultan, "NetStub: A framework for verification of distributed Java applications," in *Proc. 22nd Int. Conf. on Automated Software Engineering (ASE 2007)*, (Georgia, USA), pp. 24–33, 2007.

[21] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, USA: Morgan Kaufmann Publishers, Inc., 2006.

[22] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-step Guide*. USA: Artima Inc., 2nd ed., 2010.

[23] R. Ramler, D. Winkler, and M. Schmidt, "Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code?," in *Proc. 38th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA 2012)*, pp. 286–293, 2012.