Schedule

11/25 09:00 – 10:30 Introduction to Java PathFinder
11/25 10:40 – 12:10 Introduction, part II
12/02 09:00 – 10:30 Socket/network programming
   Stubs
12/02 10:40 – 12:10 I/O caching
12/09 09:00 – 10:30 Centralization
12/09 10:40 – 12:10 Centralization with networking
   Summary, preview of final (large) exercise
Race conditions: A common concurrency problem

◆ February 2004, Japan:

Two clients print their tax sheet at same time, see same tax sheet.

Cyrille Artho, 2013-12-02
Why is concurrency difficult?

1. Send data to server.
2. Read response.

Operations of both clients can occur in any possible order!
Why is concurrency difficult (2)?

Operations of both clients can be interleaved!
Multi-threaded applications

◆ Non-determinism of thread schedule.

◆ Testing misses many possible failures.

◆ Model checking explores all interleavings.
  – Failures due to data races.
  – Finds incorrect synch./shared conditions.
  – Incorrect usage of \texttt{wait/notify\{All\}}.
  – Timing-dependent failures affecting communication and shared data between threads.
Terminology: Context switches in multi-threaded software

- Sometimes more processes + threads ready to run than CPUs available.

- Operation system scheduler manages which threads to run at what time.
  1. Chooses a thread that is ready to run.
  2. Executes that thread for a while.
  3. Switches the context to another process/thread, continues at step 2.

- Thread/process context is saved and restored automatically.
  - Threads are not aware of context switch (*preemptive multi-tasking*).
  - Context switch may occur immediately if a thread is blocked (waiting for lock, file, etc.), or later when its time slice is used up.
  - Switches are benign for a correct program, but may allow other threads to cause race conditions in a faulty program.

- Java PathFinder shows context (thread) switches in error trace indirectly: Thread ID changes between transitions.
Networked programs

◆ Multiple separate processes.

◆ Most SW model checkers can only handle a **single** process!
Network communication

Model checker repeats I/O operations during state space exploration!

- Output of local application is sent several times.
- Input is expected several times.
- Communication with external applications cannot be model checked directly...
Goals of the next four lectures

1. Distributed applications.
   - Multiple interacting processes.

2. How to abstract system libraries:
   Two general problems:
   - Asynchronous communication.
   - Inter-process communication.

3. Network communication.
Basic network model

Client

accepts communication (blocking)

connects (blocking)

Server

established connection

bi-directional communication possible

◆ Model can be applied to several tiers.
Socket-based networking

Client

```
socket = new ServerSocket(port);
conn = socket.accept();
```

*(blocking)*

```
conn = new Socket("example.com", 8080);
conn.connect();
```

*(blocking)*

---

Server

```
BufferedReader in = new BufferedReader(
    new InputStreamReader(conn.getInputStream()));
PrintStream out = new PrintStream(
    new BufferedOutputStream(conn.getOutputStream()));
```

---

◆ Client and server typically read and write from streams from within a loop:

*InputStream.read()*, *OutputStream.write()*,

*PrintStream.println()*,

*BufferedReader.readLine()*,

etc.
### Key classes/methods for Socket-based program in Java

<table>
<thead>
<tr>
<th>Class</th>
<th>Usage/methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.net.ServerSocket</td>
<td>Represents (open) port.</td>
</tr>
<tr>
<td></td>
<td>◆ Constructor or <code>bind()</code> opens port.</td>
</tr>
<tr>
<td></td>
<td>◆ <code>accept()</code> accepts <em>one</em> client connection (typically used inside a loop).</td>
</tr>
<tr>
<td>java.net.Socket</td>
<td>Represents connection handle.</td>
</tr>
<tr>
<td></td>
<td>◆ <code>connect()</code> connects to a server.</td>
</tr>
<tr>
<td></td>
<td>◆ <code>getInputStream()</code> / <code>getOutputStream()</code> obtain channel to read from and write to.</td>
</tr>
</tbody>
</table>

**Counterpart in C library:** (`#include <sys/socket.h>`)  

**Java**  
ServerSocket constructor or `bind()`  
ServerSocket.accept()  
Client: Socket constructor or `bind()`  
operations on input/output streams  
only in C: low-level operations  

**C**  
socket() + bind() + listen()  
accept()  
socket() + bind()  
read(), write()  
recvfrom(), sendto()
Client-server pair

Client

Server

```
socket = new ServerSocket(port);
conn = socket.accept();
```

(blocking)

```
conn = new Socket("example.com", 8080);
conn.connect();
```

(blocking)

established connection

◆ One side cannot run (meaningfully) without the other.

◆ We need the counterpart, or a simplified version of it, to execute the system.
External processes as stubs

◆ Stubs replace complex libraries (and for networking, entire peer process).

◆ Write (application-specific) stub for each I/O operation of each application.

◆ Model check each application individually.

Stubs also replace missing library functions in verification tool.
Stubs

◆ Processes interact via system calls.
  – Replace system call with simpler function.
  – Stub should preserve normal operation for given test(s).

◆ New method does not call original library.
  – No interactions between processes.
  – Values are often omitted or simplified.
  – Uncertainty may be introduced (abstraction).

◆ Simulate exceptions.
  – Non-determinism is ideal tool to test exception handler code.
# Examples for stubs

<table>
<thead>
<tr>
<th>Library call</th>
<th>Stub</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date/time</td>
<td>Return fixed value</td>
</tr>
<tr>
<td>Read from network</td>
<td>◆ Success or failure</td>
</tr>
<tr>
<td></td>
<td>◆ Return empty string</td>
</tr>
<tr>
<td></td>
<td>◆ Return hard-coded string</td>
</tr>
<tr>
<td></td>
<td>◆ Return entry from an array of strings</td>
</tr>
<tr>
<td>Write to network</td>
<td>◆ Success or failure</td>
</tr>
<tr>
<td></td>
<td>◆ Ignore operation</td>
</tr>
<tr>
<td></td>
<td>◆ Compare output to previous run</td>
</tr>
<tr>
<td>RMI/DNS name lookup</td>
<td>◆ Success or failure</td>
</tr>
<tr>
<td></td>
<td>◆ On success, ignore name; assume entity found.</td>
</tr>
</tbody>
</table>
Goals

◆ Model necessary behavior for application.

◆ Use non-determinism to simulate failures.

◆ Stubs cannot model complex data well.
  – But ideal to simulate possible failures.
  – Can also be used to simulate shutdown.

◆ Abstrated application + stubs should focus on these goals!
Contradictory goals...

Independence of system call

Simplification and abstraction

Result should be precise:
Try to avoid impossible paths in caller

Conflicting results! Engineering challenge
Contradictory goals (2)

◆ Independence of system call:

  – Original library call cannot be executed by model checker.
  – Reason: difficult/impossible to „undo” action on library/OS level.

◆ Simplification and abstraction:

  – Original library function too expensive to execute.
  – Testing typically does not cover failure case (of network/hardware).

◆ Precise results:

  – Preserve original behavior (do not eliminate potential failures).
  – Avoid false positives (try not to introduce failures).
Why engineering challenge?

◆ Fully automated abstraction too coarse.

◆ Fine-grained model: same complexity as original system call.

◆ Tool-supported abstraction: (SLAM/SDV)
  – Counter-example guided abstraction refinement.
  – Start with fully non-deterministic behavior.
  – Refine behavior for spurious counter-examples.
  – Requires model of the environment.
Counter-example guided abstraction refinement

- Start with coarse abstraction.
- Check result of MC in concrete application.
- Refine abstraction if failure is not real (spurious).

Not the focus of this lecture.
Goal: Verify Windows NT device drivers by model checking.

- System calls approximated by model.
  - Model includes state changes in kernel.

Model used to check dozens of device drivers.

Continuous effort (over 5 man-years for model alone!)

For MC a single application, stubs are more economical.
  - Generic stubs may grow into a „library” over time.
Application-centric approach

◆ Module boundary: `java.io`, `java.net`.

◆ Model outcome of each operation.
  – First simple version just succeeds or fails.
  – Refine semantics until accurate enough.
  – Include original values returned.
  – Track state where necessary.
  – Minimal changes in application desired (removal of code vs. changes in code).
Implementation

Stub for given class `java.net.X` is called `env.java.net.X`.

- Change import statements in applications to use stubs.
- Not all Java classes need replacement:
  Exceptions (such as `IOException`) and some other classes merely carry data, but do not perform any I/O operations.
- Sometimes, minor change in method call necessary to disambiguate context.
- Blocking call: break up MC transition (by inserting call to `Thread.yield`!)

Cyrille Artho, 2013-12-02
Implementation: Import

\begin{verbatim}
import java.net.X \rightarrow import env.java.net.X.
\end{verbatim}

◆ Transparent replacement possible using bootclasspath mechanism.

◆ Transparent replacement not recommended for stubs:
  – Adaptation of code (minor changes) often unavoidable.
  – Stub sometimes needs to use original functionality.

◆ Hence, change import manually and write stubs and/or wrappers for standard library.
**Stub implementation**

- Should preserve normal operation.
  - Easy for simple applications with one (or very few) active connections.

- Simulate exceptions:
  - In top-level API methods (e.g. reading a packet or line).
  - Low-level operations (on characters/bytes): too much noise/overhead.

- Data sources/sinks:
  - If no inter-process communication: hard-coded input.
  - Output operations can usually be ignored (empty stub method).
State of stub model/caller context

◆ Few methods just return constants.

◆ Result depends on
  – Caller context (arguments).
  – Callee state (this instance).

◆ Arguments not always fully used (IP address, delay settings, ...).
  – Ignore as much data as possible for simplicity/performance.
Caller context

- Result often statically known for a particular test scenario.

```java
InputStream in = ...; // java.io.InputStream
...
header = in.readLine(); // e.g. HTML header
...
body = in.readLine(); // any text (constant)
```

can be abstracted to

```java
InputStream in = ...;
...
header = in.readLine_header();
...
body = in.readLine_body();
```
Static stub (stateless behavior)

- `readLine_header` returns e.g. "Content-type: text/html".

- `readLine_body` returns e.g. "<html><body>Simple test</body><html>".

- Sufficient in many cases.
  - Part of abstraction effort.
  - MC targets interaction of processes, not algorithms processing data.

- For more complex cases, stateful stub needed.
Dynamic stub (stateful)

◆ Uses and updates instance fields.

◆ Stub class often requires different/additional fields in its implementation.

◆ Common example: `InputStream` returning a hard-coded array of strings.
  – Index to array is incremented with each access.
  – Implements a simple finite-state machine returning different values.
  – Content of array given by inspection or previous (conventional) test run.

◆ Use wherever necessary, but keep stubs as simple as possible!
  – Simplify application if exact result can be ignored.
### Stateless vs. stateful stubs

<table>
<thead>
<tr>
<th>Stateless stub</th>
<th>Stateful stub</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename/replace method.</td>
<td>Replace class/sets of functions.</td>
</tr>
<tr>
<td>Ignores instance data (state).</td>
<td>Uses instance fields (state).</td>
</tr>
<tr>
<td>Heavy abstraction.</td>
<td>Flexible abstraction.</td>
</tr>
</tbody>
</table>

- Combination of approaches possible!

- Goal: Model each stub operation as simply as possible.
  - Specific to application/test case.
  - Efficiency/scalability major problem in MC.
  - Stub usage may need modifications in application (ignore exact result).
Non-determinism

- "Random" decisions taken by environment:
  - Connection successful or failed.
  - Scheduler action.
- Testing: take random path.
- Model checking: try all paths.
Introducing non-determinism: `Verify.getBoolean()`

- Produces a random boolean.
- Normal „random number”: just one result.
- Model checker, using non-determinism: set of results.
- Similarly, `Verify.getInt(int min, int max)` produces the set of all random numbers.
- Use `getBoolean` to generate non-deterministic exceptions:
  - Many I/O functions may throw an exception on I/O failure.
  - These failures typically do not occur during testing.
  - Use fault injection (simulated faults/exceptions) to test behavior!

```java
if (Verify.getBoolean()) {
    throw new IOException("Simulated exception");
}
```
Blocking calls

◆ Original: Current thread is suspended until underlying library call returns.

◆ Stub: No library call.
  – Potential of thread being suspended must remain!
  – „Atomic” block of code must be broken up.

◆ Call to `Thread.yield` breaks up transition in JPF.
  – Resulting operation is non-atomic.
  – Other threads may run during „blocking” call.
  – Original blocking semantics preserved.
How to guarantee that stub is correct?

Incorrect stub code may introduce or hide defects.

1. False (spurious) warning:
   - Usually found when error trace is analyzed.
   - Debugging stub + application takes time.

2. False negative (no defect found):
   - Worse outcome: Problems are missed but we don’t know about it.
   - Possible remedy: Use model-based testing to test stub code.

Model-based testing helps finding defects in model classes; used in:
C. Artho, M. Hagiya, R. Potter, Y. Tanabe, F. Weitl, M. Yamamoto:
Software Model Checking for Distributed Systems with Selector-Based, Non-blocking Communication.
Pros and Cons of stubs

✔ Simple, elegant concept.

✔ Scalable.

✔ Various degrees of precision possible.

✘ Limited automation and precision.

✘ Limited applicability (time-consuming manual abstraction)

✘ High level of abstraction (sometimes too simplistic).
Shutdown behavior

◆ Application uses external resources.
  – Resource must be deallocated on termination.
  – Failure to do so eventually results in problems in peer applications (e.g., connection pool exhausted).

Example: Connections to DB server.

◆ DB connections must be closed in all cases!
  – Whenever exceptions occur.
  – On premature shutdown.

◆ Shutdown hooks allow to achieve this.
Shutdown hooks

◆ Application registers each utilized resource.

◆ Shutdown hook: registered with Java run-time.

◆ Shutdown thread runs after app. termination.
  – Shutdown can occur at any program point.
  – Major challenge (cannot test all outcomes).
  – Ideal target for model checking.
  – No tool support yet (ongoing research).

◆ Implementation by instrumenting application code:
  – Termination of main thread by throwing a new `ThreadDeath` exception.
  – Exceptions are thrown non-deterministically at various locations.
Example/Exercise: Daytime server

- Given program terminates after one successful connection.
- Normal server would run indefinitely.
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Date;

Streams and sockets (and also Date) not supported by JPF!

public class DaytimeServer {
    public static void main(String[] args) {
        try {
            ServerSocket server = new ServerSocket(...);
            Socket connection = null;
            while (true) {
                // main loop: handle requests (next slide)
            }
        } catch (IOException e) {
            // handle failure in ServerSocket init.
        }
    }
}
Daytime server code (2): main loop

```java
// while loop
try {
    connection = server.accept();
    OutputStreamWriter out = new OutputStreamWriter(connection.getOutputStream());
    Date now = new Date();
    out.write(now.toString() + "\n");
    out.flush();
} catch (IOException e) {
    System.err.println(e);
} finally {
    // handle connection problems (next slide)
}
```
Daytime server code (3): finally block

// finally block
try {
    connection.close();
} catch (IOException e) {
    System.err.println(e);
    // add more error handling code if necessary
}
System.out.println("Connection closed.");
Execution of Daytime server in JPF

================================================================================ error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
dev.lang.UnsatisfiedLinkError:
  java.net.PlainSocketImpl.initProto()V (no peer)
at java.net.PlainSocketImpl.<clinit>(PlainSocketImpl.java:84)
at java.net.ServerSocket.setImpl(ServerSocket.java:236)
at java.net.ServerSocket.<init>(ServerSocket.java:178)
at java.net.ServerSocket.<init>(ServerSocket.java:97)
at DaytimeServer.main(DaytimeServer.java:18)
================================================================================ trace #1
================================================================================================================================================================== transition #0 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
  DaytimeServer.java:18: ServerSocket server = new ServerSocket(PORT);
  ...

◆ UnsatisfiedLinkError means that JPF cannot load PlainSocketImpl.

◆ Reason: No JPF-compatible implementation of java.net provided by default!

◆ Solution: Use stubs (env.java.net) instead of java.net packages.
import env.java.io.OutputStreamWriter;
import env.java.net.ServerSocket;
import env.java.net.Socket;
import env.java.util.date;

Streams and sockets (and also Date) not supported by JPF!

<table>
<thead>
<tr>
<th>Class</th>
<th>Stub</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputStreamWriter</td>
<td>does not write anything</td>
</tr>
<tr>
<td>ServerSocket</td>
<td>does nothing but may throw exceptions</td>
</tr>
<tr>
<td>Socket</td>
<td>returns hard-coded OutputStream instance</td>
</tr>
<tr>
<td>Date</td>
<td>returns constant date/string</td>
</tr>
</tbody>
</table>
Stub example: `java.net.ServerSocket`

```java
package env.java.net;
import gov.nasa.jpf.jvm.Verify;
import java.io.IOException;

public class ServerSocket {
    public ServerSocket(int p) throws IOException {
        if (Verify.getBoolean()) {
            throw new IOException("Simulated exception");
        }
    }

    public Socket accept() throws IOException {
        if (Verify.getBoolean()) {
            throw new IOException("Simulated exception");
        }
        return new Socket(); // returns socket stub
    }
}
```
Exercise (DaytimeServer)

◆ Example uses non-privileged port 1024 instead of 13 for easier testing.

1. **Enable stub usage.**
   - Stubs are already provided in env/java, but not yet used.
   - Change all relevant import statements from java.X to env.java.X.

2. **Analyze exception handling code in server.**
   - Possible network failures (to simulate):
     - Creation of server socket.
     - Accepting an incoming connection.
   - Add non-deterministic exceptions.

3. **Study error trace, fix exception handling code.**
   - Connection should still be closed at the end, when necessary.
   - No large code changes required for fixing problem in initial version.
   - **Deliverable:** Code difference/patch (see „How to submit“).
Exercise: How to build/run

◆ Extract: `tar -xzf daytime.tar.gz`
  `cd daytime/src` to edit the files for the exercise.

◆ Build: `./build.sh` or
  `javac -cp ~/jpf/jpf-core/build/jpf.jar:. *.java env/java/*/.*.java`
  Requires correct location of JPF.

◆ Run the server with JPF: `./run.sh` or `jpf DaytimeServer.jpf`. 
Exercise: How to submit (by 2013-12-09 09:00 AM JST)

Several small changes needed in `DaytimeServer.java` (stubs, bug fix) and some file(s) in `env/java` (fault injection).
- Yet over 90% of the code will be unchanged.
- Submission of entire archive not practical (at least for large programs).

Solution: Submit only difference between versions.

1. If you have not worked on a copy of the sources, copy your solution:
   ```bash
   cp -r daytime daytime-solution
   # if edits were made before this step, unpack original archive again
   ```

2. Create a “recursive diff” using the `diff` command:
   ```bash
   diff -urw daytime daytime-solution > solution.diff
   # -u = unified diff
   # -r = recursive
   # -w = ignore whitespace (indentation etc.)
   # > redirects output to a file
   ```

3. E-mail solution to `c.artho@aist.go.jp` or print it out (and add your name).
Summary

◆ Most model checkers (including JPF) handle only one process.

◆ Use stubs to replace environment:
  – Replace original library call with abstract method.
  – Retain enough functionality for meaningful test execution (in application).
  – Simulate network failures by injecting exceptions (non-deterministically).
Resources

- Lecture material for this lecture block:
  http://staff.aist.go.jp/c.artho/teaching/smc/

- Exercise (daytime server):
  http://staff.aist.go.jp/c.artho/teaching/smc/daytime.tar.gz

- How to unpack archives:
  `tar -xvzf archive.tar.gz`

- JPF home page:  http://babelfish.arc.nasa.gov/trac/jpf/
  Documentation, other tutorials, latest version.

- Tutorial paper from RV 2010 conference:
  C. Artho. Run-time Verification of Networked Software. RV 2010.
  http://staff.aist.go.jp/c.artho/papers/rv-2010-tutorial.pdf
  15-page summary of lectures from December.
Change of JPF behavior with net-iocache installed

This only applies after the installation of net-iocache.

Original installation (without I/O cache) will yield

```
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.UnsatisfiedLinkError:
    java.net.PlainSocketImpl.initProto()V (no peer)
```

◆ Meaning: Implementation of `java.net.ServerSocket` not available.

With net-iocache, message changes to:

```
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.reflect.InvocationTargetException:
    in java.net.ServerSocket.native_accept :
        java.lang.NullPointerException
```

◆ Meaning: I/O cache network model class is loaded (but not parameterized).
◆ We don’t want to use the cache for this exercise.
◆ Solve exercise by using stubs: `env.java.net`, `env.java.io`.

Cyrille Artho, 2013-12-02
Schedule

11/25  09:00 – 10:30  Introduction to Java PathFinder
11/25  10:40 – 12:10  Introduction, part II
12/02  09:00 – 10:30  Socket/network programming
     Stubs
12/02  10:40 – 12:10  I/O caching
12/09  09:00 – 10:30  Centralization
12/09  10:40 – 12:10  Centralization with networking
     Summary, preview of final (large) exercise
Most SW model checkers can only handle a single process!
External processes as stubs

- Stubs replace complex libraries (and for networking, entire peer process).
- Write (application-specific) stub for each I/O operation of each application.
- Model check each application individually.

![Diagram showing client and server applications with model checkers and I/O stubs]

Client application (model checked)  Server application (model checked)

(Conventional) test combining client and server
NetStub [ASE2007]

- Framework for verification of distributed Java applications.
- Replaces Java standard libraries.
- Event generator simulates network events.
- Publication at ASE 2007: Barlas, Bultan. *NetStub: A Framework for Verification of Distributed Java Applications*
Event generator communicates with target process.

User manually writes event generator.

Work in progress: Recorder that replays previously captured events.
I/O caching approach

- Model check one process.
- Allow for inter-process communication.
- Problem: Network communication/environment not controlled by JPF!
- Goal: Run SUT against real peer processes (without using stubs).
Model checker repeats I/O operations during state space exploration!

- Output of local application is sent several times.
- Input is expected several times.
- Communication with external applications won’t quite work this way...
Solution: Cache layer between model checker and peer

- Each I/O operation is captured by (model-checking aware) cache layer.
- Duplicate send operations: „ignored“ (not relayed to peer).
- Duplicate read operations: previous input from peer is replayed.
- Uses **live data** of **real applications** instead of stub!
I/O cache approach

New state: Store I/O data streams „globally”.
- Cache I/O data and message sizes.
- Map program state to stream position.

Backtracking: Restore I/O state „locally”.
- Reset data streams back to old state.
- „Rewind” seek positions in files/streams.

Continued exploration: Replay previous I/O.
- Input: replayed locally from cache.
- Output: compared to cached output.
  - If output inconsistent, report failure.
Matching requests with responses

Alphabet server with multi-threaded client:

- Client sends a number, followed by a newline character.
- Server responds with the corresponding letter in the alphabet.
- Problems with sequence of requests and responses in cache:
  - Where does a complete request end? → poll server response
  - How far does a response go? → keep track of response size
Example

State space exploration

Comm. trace

Response limit

- Keeps track of prev. data
- Maps request to response pos.
- Arrows = current pos.

0 → 0
Example – Client sends „1”, server replies with „A”

State space exploration

Comm. trace

Response limit

- Keeps track of prev. data
- Maps request to response pos.
- Arrows = current pos.

Client reader thread not yet scheduled, so response is not processed.

Reflected by communication position in trace.
Example – Client sends „2”, server replies with „B”

State space exploration

Comm. trace

Response limit

- Keeps track of prev. data
- Maps request to response pos.
- Arrows = current pos.

Client reader thread still not scheduled.
Example – Model checker backtracks, client reads „A‟

State space exploration

Comm. trace

Response limit

Keep track of prev. data

Maps request to response pos.

Trace request position backtracked.

Trace response position advanced.

Limit is persistent.
Problem with linear cache: non-deterministic responses

Web server with page hit counter

✧ Value of returned counter depends on order of GET requests.

✧ Blue thread first (left): 1; blue thread second (right): 2.
Cached data allows for divergence in communication.

Peer process is restarted to obtain new communication trace.

Low overhead of new data structure/restarts.

Model checking a server

**Client #1**
- Spawn client.
  - Spawn client upon **accept**.
  - Cache comm. data from/to client.

**Backtracking:** **Restore** I/O pos.
- Client not referenced anymore.
- Use cached data.

**Client #2**
- **Diverging data:** **New client.**
  - **Replay** cached data, up to mismatch.
  - Create new branch in cache.
Model checking a client: caching communication

Server runs in background.

**New state:** Store I/O data.
- Cache data, response pointers.
- Map program state to stream pos.

**Backtracking:** Restore I/O pos.
- Reset data streams to old state.
- No communication with server.
- After BT and exploring state 4: I/O data node ref’d by states 3,4.
Model checking a client: using cached data

Matching data: Read from cache.
- Input: use cache.
- No communication with server.

Diverging data: New connection.
- Reconnect to server.
- Replay cached data, up to mismatch.
- Create new branch in cache.

- Peer process (server) is polled proactively.

- Response (not shown) would be added after nodes 4, 6.
Handling of peer connections

- Server is model checked.
- Server serves a given number of clients.
- Client can be restarted from same init. state.
- Each client connection carries different data.
- **Spawn** new client proc. for new response.
- Client is model checked.
- Client issues requests and terminates.
- Server runs indefinitely.
- Connection to same server: same data.
- **Reconnect** to server for new response.
Implementation architecture

- JPF implements JPF-level VM with backtracking on top of "host JVM".
- No execution of native code possible from model classes in JPF-level VM.
- Stubs (last week) impl. as model classes, cannot interact with environment.
- **Native peer classes** can access native code from underlying JVM.
- Native peers are not managed or backtracked by JPF.

Native peers are usually needed when writing a JPF extension; usage is documented on JPF wiki but outside the scope of this lecture.
## Experiments with alphabet client/server

<table>
<thead>
<tr>
<th># conn./ threads</th>
<th># messages per thread</th>
<th>Linear cache</th>
<th>Branching cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Client</td>
<td>Server</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0:06</td>
<td>0:03</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0:40</td>
<td>0:04</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2:39</td>
<td>0:06</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2:16</td>
<td>0:10</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>35:22</td>
<td>0:21</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>&gt; 1 h</td>
<td>0:45</td>
</tr>
</tbody>
</table>

- Implemented on top of **Java PathFinder** model checker (from NASA).
- I/O caching can check client and server separately.
- Product of state spaces reduced to sum of state spaces.
- Full table with other benchmark cases in ASE 2009 paper.
I/O cache approach: Summary

◆ **Store** state of I/O data streams „globally”.
  
  – Global view applies to OS and external processes.
  – Keep local view of I/O state for model checker.

◆ Backtracking: **restore** I/O state „locally”.
  
  – „Rewind” seek positions in files/streams.
  – Reset local data streams back to old state.

◆ Continued exploration: **replay** previously seen „global” I/O actions.
  
  – Previously received input: replayed locally.
  – Output: compared to previous output.
  * Inconsistency: restart peer, create new cache branch.
I/O cache approach – details

◆ Input or output that is not cached is relayed to “real” system.
  – Cache is inactive until model checker stores program state.
  – Repeated operations use cached data.

◆ Peer has to be deterministic, but SUT may be non-deterministic.
  – Cannot check any type of application.
  – Applicability subject to further research.

◆ Open/close operations also virtualized.
  – Connections not physically opened/closed.
  – Optional re-use of connection data from same server (for web service).

Maps program states to communication states.

◆ Equivalent program states ↔ equivalent comm. data.
I/O cache: pros and cons

✔ Transparent to model checker.

✔ Scalable (only one process in model checker).

✘ Handles many, but not all types of applications.

✘ Work in progress, not all possible network APIs supported.

◆ Alternative for random-access I/O: array.
  – All information in memory (and thus model checker).
  – No interaction with outside world.
  – Array is “seeded” with hard-coded contents.
  – Current state of `RandomAccessFile` implementation in JPF.
Obtaining the I/O-cache module for JPF

◆ JPF home page: http://babelfish.arc.nasa.gov/trac/jpf/

◆ Extension for I/O caching is listed under „Projects”, called net-iocache.
  – Wiki (latest version moved to bitbucket):
    https://bitbucket.org/cyrille.artho/net-iocache
  – Old web page (on babelfish) hosts version for JPF 6.
  – Source repository is set up for JPF 7 by default (v6 still available).
Installing net-iocache

◆ Source code must be checked out and compiled at correct location.

◆ After that, configuration is automatic.

◆ Assumption: jpf-core is at ~/jpf/jpf-core.

◆ jpf-net-iocache must go to ~/jpf/net-iocache:

```bash
cd ~/jpf
hg clone https://bitbucket.org/cyrille.artho/net-iocache
cd net-iocache
hg update jpf6 # changes to branch for JPF 6
ant
```
Running net-iocache

◆ Exercise configured for JPF 6; .jpf configuration file sets all necessary options:

~/jpf/jpf-core/bin/jpf AlphabetServer.jpf

◆ jpf-net-iocache loads java.net model classes through its own options, and activates its listener to enable caching.

◆ For server-side verification, command to start client needs to be specified: jpf-net-iocache.boot.peer.command

◆ This script is provided as ./peer.sh for this exercise.
Verifying a server: configuration of the peer

◆ Server is started in JPF, client is managed by \texttt{jpf-net-iocache}.

  – Peer command is specified as option \\
    \texttt{jpf-net-iocache.boot.peer.command}.
  – Automatically starts a peer process when needed \\
    (whenever server is ready to accept an incoming connection).
  – Peer command has to reside inside shell script, \\
    to avoid possible parse problems with longer command strings.

◆ Edit \texttt{peer.sh} to contain the command(s) to start a client process:

```bash
#!/bin/sh
java AlphabetClient 1 2
```
Verifying a client (not part of the exercise)

1. Create a JPF configuration file AlphabetClient.jpf.
   - Configure client to send two requests (matching the number of requests that the server accepts).
   - In this example:
     ```
     target = AlphabetClient
     target_args=2,2
     ```
   - Arg is 2 because only 1 process (containing 2 client connections) is started.

2. Start a server first. Example:
   - `java AlphabetServer 2 &`
   - Ampersand starts process in the background.

3. Start JPF: 
   - `~/jpf/jpf-core/bin/jpf AlphabetClient.jpf`
   - JPF will verify client application, server will terminate afterwards.
   - Process is similar to server verification, but no peer script is needed; instead, the server is started before JPF runs.
Exercise: alphabet server; due 2013-12-09, 09:00 AM JST

◆ Alphabet server returns \( n \)th character of alphabet.
  – New worker thread is started for each connection.

◆ Slightly modified from example on web: includes counter of # of connections.
  – Counter is not synchronized, maybe not accurate.
  – In given version, it may even still be 0 when first worker thread runs.
  – JPF gives error trace of possible assertion violation.

◆ Task: Fix AlphabetServer.java!
  – Run JPF again to ensure no failure occurs.
  – Submit patch (very simple and short solutions are possible).

1. If you have not worked on a copy of the sources, copy your solution:
   ```
   cp -r alphabet alphabet-solution
   # if edits were made before this step, unpack original archive again
   ```
2. Edit the files to create your solution.
3. Create a „recursive diff” using the diff command:
   ```
   diff -urw alphabet alphabet-solution > solution.diff
   ```

◆ Note: You can choose to do either the „Daytime server” or this exercise (or both).

Cyrille Artho, 2013-12-02
Resources

◆ Lecture material for this lecture block:
  http://staff.aist.go.jp/c.artho/teaching/smc/

◆ Exercise (alphabet server):
  http://staff.aist.go.jp/c.artho/teaching/smc/alphabet.tar.gz

◆ How to unpack archives:
  
  `tar -xvzf archive.tar.gz`

◆ JPF home page:  http://babelfish.arc.nasa.gov/trac/jpf/
  Documentation, other tutorials, latest version.

◆ Papers about current version of net-iocache (for those who want to know more details):
  http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=06645368
Utilities other than JPF

◆ Scripts to compile/run JPF are written for **bash** (Bourne Again Shell).

◆ Some scripts will use **grep** and **sort**, from package **coreutils** (**textutils** on older systems).

◆ On Cygwin, these packages may have to be installed separately; on other systems, the default version should work.
Change of JPF behavior with net-iocache installed

Manual usage of JPF (without .jpf file) may cause this issue.

Last week’s exercise: Original installation (without I/O cache) will yield

```
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.UnsatisfiedLinkError:
    java.net.PlainSocketImpl.initProto()V (no peer)
```

◆ Meaning: Implementation of `java.net.ServerSocket` not available.

With `net-iocache`, message changes to:

```
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.reflect.InvocationTargetException:
    in java.net.ServerSocket.native_accept :
        java.lang.NullPointerException
```

◆ Meaning: I/O cache network model class is loaded (but not parameterized).

◆ We don’t want to use the cache for last week’s/next week’s exercises.

◆ Problem is taken care of by script `run.sh`, unless you run program manually.