JPF-Core-X:
Tool Operational Requirements (TOR)

Contents

Introduction

This document is one of a several exemplar documents prepared as part of a research Case Study whose objective is to simulate a Formal Methods Tool qualification exercise under DO-330. The specific tool considered in this case study is the core module of Java PathFinder (JPF-Core). As with any tool qualification exercise, the qualification is done with respect to a specific version of the tool. Therefore, throughout this document, we refer to our version of JPF-Core as "JPF-Core-X", as described in Section a of the Tool Qualification Plan. This particular document provides a representative example of the "Tool Operational Requirements" (TOR) for JPF-Core-X, and is written according to the guidelines in DO-330 Section 10.3.1.

Because this is a research study, in which there is no actual qualifying organization and accompanying context, and because the tool under consideration is a research implementation without specific versions, release control, user documentation, or standard configurations, some of the sections of an actual TOR will not be relevant. This document is thus a Framework for an actual TOR, organized according to the DO-330 enumeration of required contents. Accordingly, some sections will represent content that is concrete enough to be part of an actual TOR; some will discuss how a more concrete implementation of this tool might fulfill the required contents; and some will not be applicable for the purposes of this research simulation.

The sections that follow are organized according to sub-parts a) through i) of Section 10.3.1 in DO-330. In each section, we attempt to provide representative content of what should appear in an actual TOR for a real qualification exercise. In addition, we offer supplemental *meta-level* comments throughout the document.

Discussion

This is how a meta-level comment appears in the text. These comments are meant to provide insight into our process of writing the document, and to suggest interesting or important topics that relate to the qualification of formal methods tools.

a. Description of context

DO-330

"Description of the context of the tool use, including interfaces with other tools and integration of the tool output files into the resultant software." [DO330-10.3.1-a]

We will apply JPF-Core-X to a subset of the Low Level Requirements (LLRs) to verify that they comply with certain High Level Requirements (HLRs) which JPF-Core-X is particularly well-suited to test. These LLRs will express system requirements such as function sequencing and concurrency as relatively short programs in the Java language. By model-checking these LLRs, JPF-Core-X will verify the absence of faults corresponding to these HLRs. (If faults are discovered by the tool at

this stage during the development process, the faulting execution trace discovered by JPF-Core-X can be used to correct the implicated LLRs.) This approach to early validation of the LLRs has been shown in academic research to reveal errors before source code generation [1].

As shown in Figure 1, JPF-Core-X will support four of the Software Verification Processes required by the DO-178C certification regime.

- JPF-Core-X will verify the *compliance* of the LLRs with certain HLRs representing properties of liveness (*i.e.*, freedom from deadlock) and correct sequencing.
- JPF-Core-X will verify the *accuracy and consistency* of the LLRs by enforcing their expression in a formal language (Java) that can be checked for internal consistency (through compilation).
- JPF-Core-X will demonstrate the *verifiability* of the LLRs by enforcing their representation in a formal notation.
- JPF-Core-X will help verify the *algorithm accuracy* of the LLRs by checking application-specific assertions.

Figure 2 shows JPF-Core-X's inputs and outputs in the context of its proposed use in certification activities. As shown, JPF-Core-X consumes files containing properties to be verified represented in its own input language and one or more files containing Java code representing system LLRs. It's output is presented to the user in the form of a report file.

The application of JPF-Core-X will be limited to the high-level requirements corresponding to:

- *Deadlocks* occurring when two or more threads get stuck in a state where each is waiting on the other(s) before continuing execution;
- *Race Conditions* in multi-threaded programs when either a) two or more threads can access the same shared data and try to change it at the same time, or b) shared data is changed by threads in a particular (undesired) order, resulting in a fault;
- *Application-Specific Assertions* - specific conditions that we (as developers) say must be true in order to for us to claim that the application is functioning properly;

These classes of high-level requirements can be verified by JPF-Core-X's native functionality, without the use of any of the "modules" available from the JPF-Core-X maintainers or the addition of any end user extensions, such as custom `Listener` classes. Accordingly, we seek to qualify only the core JPF-Core-X tool. We will not use or qualify any modules or custom extensions.

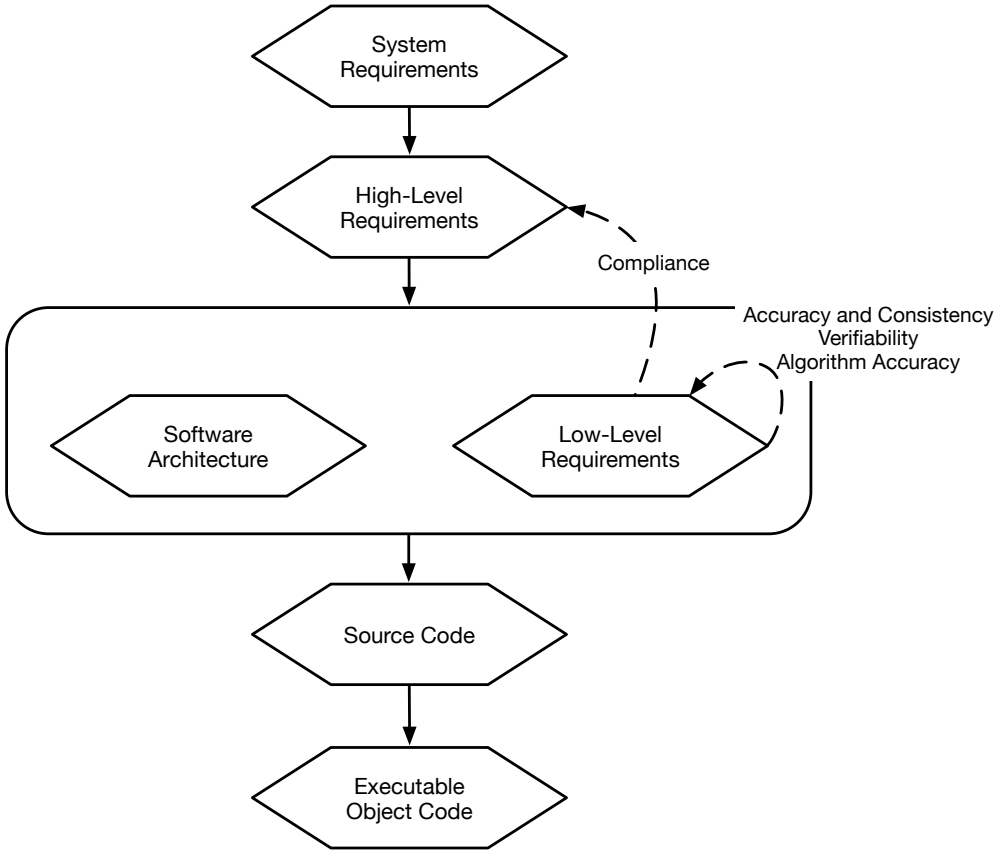

Figure 1: JPF-Core-X will be used to automate the indicated review and analysis DO-178C certification activities, including compliance of LLRs with HLRs, the accuracy and consistency of the LLRs, the verifiability of the LLRs, and algorithm accuracy in the LLRs. This diagram shows how these certification activities supported by JPF-Core-X fit into the overall Level A Software Verification Processes as illustrated in Figure FM.6-1 of DO-333.

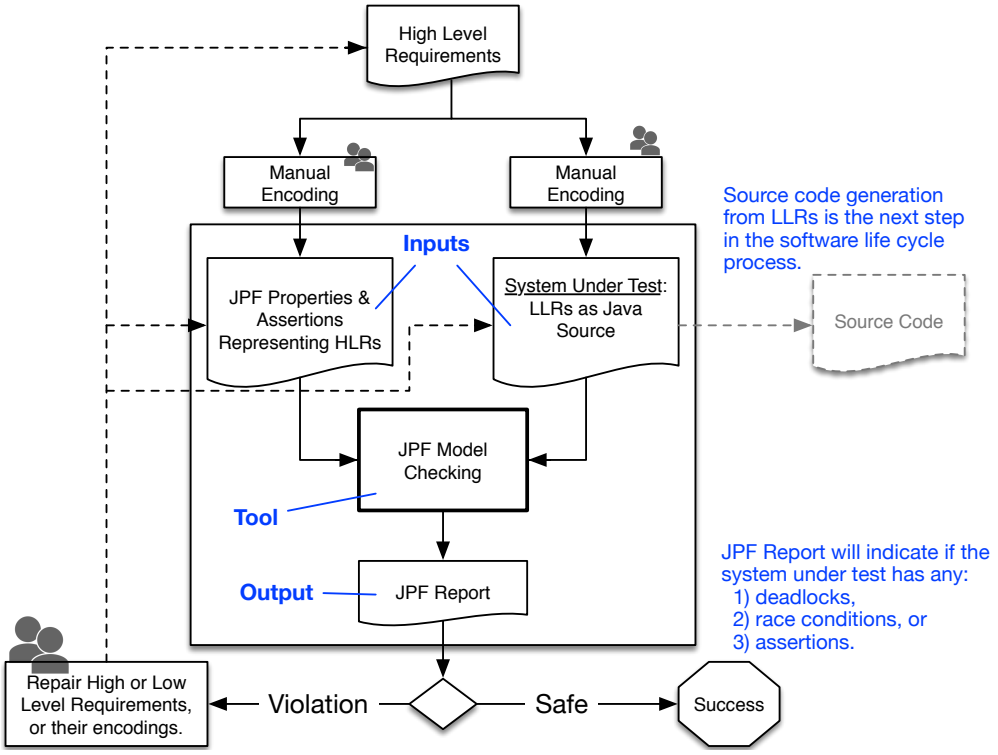

Figure 2: Flow of HLRs and LLRs for JPF-Core-X model checking.

a.1. Functional Overview

Discussion

This section overlaps with Section "c" in the TQP.

JPF-Core-X is a configurable environment with its own virtual machine designed to enable customized verification of Java bytecode programs.

Discussion

The full JPF architecture includes a core virtual machine with a basic set of verification tools (JPF Core), plus several *optional* extension modules that may be added to perform more customized analysis. As discussed above, however, this qualification exercise is for JPF-Core-X only.

A diagram of the architecture is shown in Figure 3. The JPF-Core-X installation resides and runs on top of the native Java installation on the host OS. It is therefore a VM on top of a VM. The JPF-Core-X virtual machine then executes the Java application being tested. This Java source code being checked by JPF-Core-X is also referred to as the System Under Test (SUT).

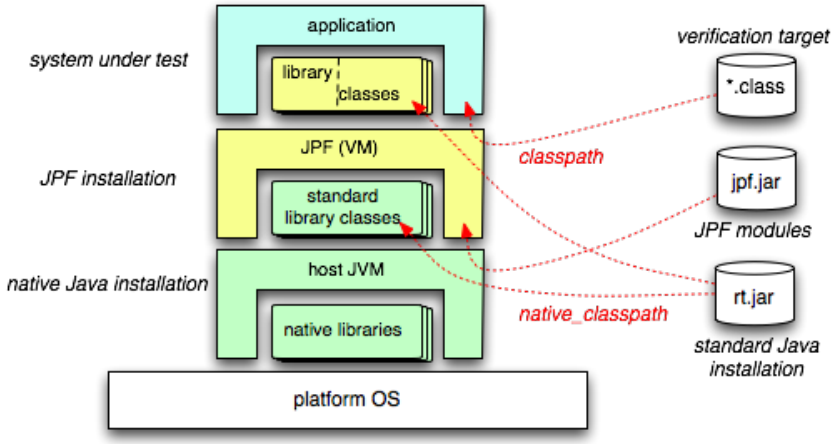

Figure 3: JPF-Core-X Layered Architecture. Image taken from: http://babelfish.arc.nasa.gov/trac/jpf/wiki/user/components

In general, a potential challenge with using JPF-Core-X to verify an application (especially large scale applications) is that JPF-Core-X cannot execute Java libraries that use native code. Doing so would prevent JPF-Core-X from matching and/or backtracking the program states. The workaround is to use *native peers* and/or *model classes*.

Native peers are Java classes that effectively replace native methods. The native peers are executed by the real Java VM (not JPF-Core-X). *Model classes* are simple replacements of standard classes, such as java.lang.Thread. The model classes provide alternatives for native methods which are fully observable and backtrackable.

b. Description of operational environment

DO-330

"Description of the tool operational environment(s) (where the tool will be installed)." [DO330-10.3.1-b]

Discussion

There's nothing interesting with respect to model checkers about this section. However, the fact that JPF is an open-source tool distributed from a source code repository does present possibly interesting issues.

The specific instance of JPF to be qualified for certification is JPF-Core-X. We obtained JPF-Core-X from the JPF mercurial repository maintained by NASA on March 1, 2016. On that date, JPF-Core-X was the `tip` tag on the `default` branch of the repository at `http://babelfish.arc.nasa.gov/hg/jpf/`. The last changeset included in JPF-Core-X is `29:820b89dd6c97` committed on October 16, 2015. From here forward, we will use JPF-Core-X to refer to this specific version of JPF-Core.

JPF-Core-X is a pure Java application. As such, it runs on the Java virtual machine, which itself can be run on Windows, OS X, or Unix operating systems. The minimal required Java version is Java SE 7, which corresponds to JDK 1.7.

The JPF-Core-X distribution is provided with project configurations for both the NetBeans and Eclipse IDEs. This qualification exercise assumes that only the NetBeans IDE is used.

Given that JPF-Core-X is generally a resource hungry application, it is recommended to run with at least 2 GB of RAM. This is just a general guideline. The actual memory usage will depend on the size of the application being tested.

Discussion

This section overlaps with Section "d" in the TQP.

c. Description of input files

DO-330

"Description of input files, including format, language definition, etc." [DO330-10.3.1-c]

As the diagram in Figure 2 shows, the two main inputs provided to JPF-Core-X are: 1) the System Under Test (SUT), which is the Java application to be tested, and 2) the JPF properties.

c.1. Java Application to Test

The Java application to be tested with JPF-Core-X should be provided as Java source code. This is referred to as the System Under Test (SUT). One class in the set of *.java files must contain the main() method.

This Java source code represents the low level requirements of the software to be certified. In addition, a subset of the high level requirements may be embedded in the Java source code in the form of application specific assertions.

c.2. JPF Properties

JPF-Core-X will be used to verify that the SUT meets certain high-level requirements. In particular, it will check the SUT for deadlocks, race conditions, and application-specific assertions. The manner in which JPF-Core-X searches for these faults can be configured through a .jpf properties file.

Defining the overall JPF-Core-X configuration involves specifying properties at three different levels: site, project, and application.

Discussion

The JPF architecture was designed to be both *extensible*, through the inclusion of additional modules, and *configurable*, through the definition of a large set of configuration parameters. The goal of this design, of course, is to grant users as much freedom as possible so they can shape JPF to meet their own needs. Although this degree of flexibility is extremely useful, it also permits the tool to exhibit a wide range of behaviors. Qualification of a tool with such loosely defined behavior is impractical and ill-advised. With the goal of providing a well-defined set of behaviors, while also allowing some configuration, we therefore restrict the scope of the tool to include only JPF-Core, and further restrict which configuration parameters may be changed.

Site Properties The `site.properties` file is created as part of the install process. Unless otherwise specified, it is installed at: `{user.home}/.jpf/site.properties`. The purpose of the

`site.properties` file is to inform JPF-Core-X where to find all of the installed projects and extensions. An example site.properties file for JPF-Core-X is shown below.

Discussion

Note that this example includes two additional extensions beyond JPF-Core-X. This is, in general, how the `site.properties` file would be defined for a particular installation of JPF-Core-X with multiple extensions. However, for our verification exercise, only JPF-Core will be used, so only the first two (non-comment) lines are necessary.

```
# JPF site configuration

# jpf-core must always be defined and included in the extensions
jpf-core = ${user.home}/projects/jpf/jpf-core
extensions=${jpf-core}

# Additional extensions would be listed here. For example:
jpf-numeric = ${user.home}/projects/jpf/jpf-numeric
extensions+=${jpf-numeric}

jpf-aprop = ${user.home}/projects/jpf/jpf-aprop
extensions+=${jpf-approp}

# and so on...
```

Project Properties

Discussion

Each JPF module contains a project properties file in its root directory, named `jpf.properties`. For this verification exercise, we require only one project properties file, which is specifically for JPF-Core.

The System Under Test (SUT) may also have a jpf.properties file. If so, it would simply define the classpath variable so that JPF-Core-X will know where to find all of the SUT classes.

The jpf.properties files are executed in the order in which they are listed in the site.properties file, with one exception. If JPF-Core-X is started from within a directory that contains a jpf.properties file, this file will be loaded last and will therefore overwrite any previously defined settings in the other files.

The site and project properties must be consistent. This means that the module names (e.g. "jpf-core") in site.properties and jpf.properties need to be identical.

This properties file does two things: 1) it defines the JPF-Core-X specific paths that need to be set for the module or system under test to work properly, and 2) it defines numerous default properties that govern JPF-Core-X functionality. This includes properties for the VM, search methods, choice generation, reports, and listeners.

The first entry in every jpf.properties file must define the module name. For JPF-Core-X, it is:

```
jpf-core=${config_path}
```

JPF automatically expands `${config_path}` with the pathname of the directory in which the jpf.properties file resides.

The next entries in jpf.properties set the classpath, sourcepath, and peer packages.

```
 jpf-core.native_classpath=\
  ${jpf-core}/build/jpf.jar;\
  ${jpf-core}/build/jpf-annotations.jar;\
  ${jpf-core}/lib/junit-4.10.jar

jpf-core.classpath=\
  ${jpf-core}/build/jpf-classes.jar;\
  ${jpf-core}/build/examples

jpf-core.sourcepath=\
  ${jpf-core}/src/examples

jpf-core.test_classpath=\
  ${jpf-core}/build/tests

jpf-core.peer_packages = gov.nasa.jpf.jvm,<model>,<default>
```

The remaining parts of jpf.properties sets values for key/value pairs. Both the site.properties and the jpf.properties files can define or override any key/value pairs recognized by JPF-Core-X.

Important: The default key/value pairs that are defined in the jpf.properties file of JPF-Core-X should not be changed. For the purpose of this verification, it is assumed that these default values will be used. Only a subset of the key/value pairs may be changed, and that those changes should be specific in the Application Properties *.jpf file (see below).

Discussion

For a real qualification exercise, the tool should provide some protection to prevent the user from changing default properties that should remain unchanged.

Application Properties When JPF-Core-X is run, it must be provided a *.jpf application properties file. At a minimum, the *.jpf file defines the target, which is the main java class that should be started to begin execution of the application. A list of target_args may be provided if the target class requires input arguments. If necessary, multiple *.jpf files may be created, each defining different values for the target input arguments. An example is shown below to explain the syntax.

```
# Define the target class of the SUT. This class must have a main() method.
```

```
target =  MY_SUT_CLASSNAME
# Target arguments, if necessary.
target_args=1,2,''a'',''b''
```

Discussion

In general, the *.jpf application properties file may also list dependencies on other JPF modules, such as jpf-awt or jpf-shell, for example. However, because our tool qualification includes only JPF-Core, no other modules may be listed.

In addition, the *.jpf file may also define JPF-Core-X properties as key/value pairs. Any JPF-Core-X properties defined in the *.jpf file will overwrite the values set in the jpf.properties file. **Important:** For the purpose of this verification exercise, only the following properties may be changed in the *.jpf file:

`search.class` This defines which method of search JPF-Core-X will use in navigating the state space.

```
# DEFAULT
search.class = gov.nasa.jpf.search.DFSearch
# OTHER OPTIONS
search.class = gov.nasa.jpf.search.BFSHeuristic
search.class = gov.nasa.jpf.search.RandomHeuristic
```

Discussion

Note that, in the Tool Requirements (TR) document, we limit our discussion to functional requirements associated with depth-first search. If other search methods are allowed, the functional requirements in the TR would have to be expanded to include them.

`search.depth_limit` Defines the depth limit for search. The default value is 2147483647.

Discussion

Limiting the search depth may be a practical need for large programs. A greater search depth requires more memory and takes more time for the tool to process. However, this represents one of the fundamental issues with model-checking. If the search is halted at a certain depth, then we have not fully examined the entire search space, leaving open the possibility for faults to occur in the unexamined part. This could lead to a false negative. One mitigating factor for this issue is that, if the search depth is reached, it is indicated in the JPF report. Thus, any interpretation of a negative (no errors found) result would be tempered with the fact that the full search space was not examined. In some cases, it may be possible to make the argument that searching

beyond a given depth is unnecessary, based on how the SUT is implemented. However, this argument would have to be made outside the scope of the tool qualification, and would be specific to the SUT.

`cg.randomize_choices` Defines the randomization policy for choice generation.

```
# choice randomization policy in effect:
#   "NONE" - choice sets are not randomized
#   "FIXED_SEED" - choice sets are randomized using a fixed seed for each JPF
    run (reproducible)
#   "VAR_SEED" - choice sets are randomized using a variable seed for each JPF
     run
cg.randomize_choices = NONE
```

`cg.seed` This defines the standard seed value used for any FIXED_SEED policy.

```
# DEFAULT - the value of 42 below can be changed to any integer.
cg.seed = 42
```

`log.level` This defines how log messages are displayed. A set of preset options are available, each producing log messages of a different format and content.

```
# DEFAULT
log.level=warning
# OTHER OPTIONS: severe, info, fine, finer, finest
```

`report.publisher` This defines what types of reports JPF-Core-X will output during and after the run. The console is defined as the one publisher by default, and this should not be changed. However, additional publishers may be added.

```
# DEFAULT (this should not be changed)
report.publisher=console
# OPTIONAL ADD-ONs:
report.publisher+=xml
report.publisher+=html
```

`report.console.file` This defines a file name for the console output to be redirected to. If added to the *.jpf file, the console output will be saved in the specified file. If the file does not exist, it will be created. If the file does exist, its contents will be replaced with the output from the next run. If this line is not added to the *.jpf file, the console outputs will not be stored in any file.

```
report.console.file=MY_JPF_REPORT
```

`report.console.property_violation` This defines, for the console publisher, which of the property violation topics should be processed and in which order. See Section d.3. for more discussion of property violations.

```
# DEFAULT (this should not be changed)
report.console.property_violation=error,snapshot
# OPTIONAL ADD-ONs:
report.console.property_violation+=trace
report.console.property_violation+=output
report.console.property_violation+=statistics
```

d. Description of output

DO-330

"Description of output files, including format and contents." [DO330-10.3.1-d]

JPF-Core-X can produce three different types of outputs:

Application Output - describes what the application, a.k.a. the System Under Test (SUT), is doing as it runs.

JPF-Core-X Logs - describes what JPF-Core-X is doing as it performs its evaluation of the SUT.

JPF-Core-X Reports - describes the result of the JPF-Core-X run.

The manner in which each of these outputs is provided is controlled by the JPF-Core-X configuration mechanism.

Each of the output types is described in the subsections that follow.

d.1. Application Output

This represents all outputs printed during execution of the application. This output is typically generated from Java source code with a call like: `System.out.println(...)`. Because it is executed by JPF-Core-X as part of the application, the same print statement might be executed several times. Consider the following example:

```
public class MyApplication ..{
    ...
```

```
    boolean cond = Verify.getBoolean();
    System.out.println("and the cond is: " + cond);
    ...
}
```

This will produce the following application output:

```
...
and the cond is: true
...
and the cond is: false
...
```

Following the first print statement, JPF-Core-X performs a backtrack operation. The `Verify.getBoolean()` statement has two possible outcomes, *true* or *false*, which ultimately leads to both results being printed. It is, in general, confusing to view the application output in isolation because the backtracking operations performed by JPF-Core-X are not visible. This makes it difficult (or impossible) to ascertain whether repeated printouts with different values (such as the example above) are due normal to iterations within the application, or the result of JPF-Core-X backtracking.

JPF-Core-X provides two different configuration options to help with this issue:

- `vm.tree_output = {true|false}` – (Default.) Output will be displayed on the console each time a print statement is executed. This corresponds to the above example.
- `vm.path_output = {true|false}` – Output will not be immediately printed to the console. Instead, it will be stored in the path for later processing once JPF-Core-X terminates. This will cause the application output to appear as if it were run on a normal JVM.

d.2. JPF-Core-X Logs

JPF-Core-X logs convey the internal status of JPF-Core-X. This can include high-level notifications, such as errors or assertion violations, as well as low-level information, such as fine-grained details about internal JPF-Core-X operations. The JPF-Core-X logging mechanism extends the standard `java.util.logging` infrastructure. Specialized extensions of the LogHandler and Formatter classes are provided with JPF-Core-X. This enables logging configuration to be performed via JPF-Core-X's own configuration mechanism.

The JPF-Core-X user can use JPF property files (*.jpf) to control two main aspects of the logging: 1) *log output destination* and 2) *log levels*.

Defining the Log Level The default log level is set with the `log.level` property. It may be set to any of the following levels: *severe, warning, info, fine, finer, finest*.

This defines the default manner in which all logs are displayed during JPF-Core-X execution. In addition, users may elect to override the default log level for individual logs in their custom

JPF-Core-X code. To do so, first declare a static logger instance in your custom JPF-Core-X class where the logs will be generated, and then use the desired `log` method:

```
static Logger log = JPF.getLogger(``appropriate ID'');
...
log.severe(``there was an error'');
log.warning(``there was a problem'');
log.info(``there was something of interest'');
```

Defining the Log Output Destination The log outputs may be sent to a different console. From the machine where the log should be displayed, enter the following command in a Unix shell:

```
$ java gov.nasa.jpf.tools.LogConsole <port>
```

Here, `<port>` is the port where JPF-Core-X will direct its output. To specify this port for JPF-Core-X, enter the following on the host machine running JPF-Core-X:

```
$ jpf +log.output=<host>:<port> ... MyTestApp
```

The default host is "localhost" and the default port is 20000.

d.3. JPF-Core-X Reports

JPF-Core-X reports are the primary output of interest. They report any property violations that are found, along with the corresponding traces and statistics of the overall run.

The JPF-Core-X reporting system can be configured to output reports in different formats and to different targets. Users can also control what items of the report are displayed, and in what order.

Reporting is organized according to a predefined set of output phases. In general, each phase may include a configured, ordered list of topics. The distinct set of output phases are:

start – processed when JPF-Core-X starts

transition – processed after each transition

property_violation – processed when JPF-Core-X finds a property violation

finished – processed when JPF-Core-X terminates

If a property violation occurs, different types of information about that violation may be reported. Each JPF-Core-X run can be configured to report specific types of information, or *topics*. The predefined set of topics for the *property_violation* phase is listed below.

error – shows the type and details of the property violation found

trace – shows the program trace leading to this property violation

snapshot – lists each threads status at the time of the violation

output – shows the program output for the trace (see above)

statistics – shows property statistics information

When JPF-Core-X terminates it transitions to the *finished* phase. In this phase, two different topics may be reported:

result – reports if property violations were found, and shows a short list of them

statistics – shows overall statistics information about JPF-Core-X's processing of the current run.

d.4. JPF-Core-X Reporting System

The JPF-Core-X reporting system is comprised of three main components:

1. ***Reporter*** – Performs data collection; manages and notifies *Publisher* objects when a certain output phase is reached.
2. ***Publisher*** objects – Produce output in a specific format.
3. ***PublisherExtension*** objects – Can be registered for specific *Publisher* objects at startup.

Publisher objects are used to produce specific types of output format, such as text or XML. For example, readable text output is displayed to the console by the *ConsolePublisher*. The *PublisherExtension* objects provide a way for users to extend the built-in publishing capabilities of JPF-Core-X. To do so, the user must first implement the required phase- and format-specific methods, then register the extension for a specific Publisher class. An example of a publisher extension provided with JPF-Core-X is the *DeadlockAnalyzer*. This class is a registered listener of *ConsolePublisher* and creates property specific traces (with more information about the deadlock) when a deadlock event is published.

Configuration of the Reporter, Publisher, and PublisherExtension components can be managed by specifying properties in the JPF property file. The listing below provides one example of how the user might specify these properties.

```
report.class=gov.nasa.jpf.report.Reporter
report.publisher=console,xml
report.console.class=gov.nasa.jpf.report.ConsolePublisher
report.xml.class=gov.nasa.jpf.report.XMLPublisher
report.console.property_violation=error,trace,snapshot
report.xml.property_violation=error,trace,snapshot,output,statistics
```

Line 1 specifies the reporter class. The built-in JPF-Core-X reporter can always be used, unless you are implementing your own custom class to perform data collection differently.

Line 2 specifies a list of Publisher objects to use in the reporting. Here we use only two. The values "console" and "xml" are symbolic names for the two types of publication we've chosen. Each of the symbolic names must have a corresponding class name defined for it.

Lines 3-4 specify the class names that correspond to the symbolic publisher names.

Finally, lines 5-6 specify which of the property violation topics should be processed and which order. Here, more information is published to the xml file, and less to the console.

e. Requirements for tool functions

DO-330

"Requirements for all the tool functions and technical features used to satisfy the identified software life cycle process(es)." [DO330-10.3.1-e]

Discussion

The term "identified software life cycle process(es)" here refers to the specific aspects of the software to be certified for which we are seeking certification credit. In other words, this step focuses on identifying all of the requirements for the tool so that it can support the certification credit we are claiming in the TQP.

e.1. JPF-Core-X External Interface Requirements

This section specifies the operational requirements on the external interfaces of JPF-Core-X, with the purpose of enabling the qualification of JPF-Core-X as a verification tool.

e.1.1. Inputs to JPF-Core-X

External inputs to JPF-Core-X include:

- The System Under Test (SUT)
- JPF-Core-X Configuration Options

Requirement TOR-e..1: JPF-Core-X shall run and analyze the SUT, where the SUT is provided as a set of Java classes that compile and run under JRE 1.6 or above.

Requirement TOR-e..2: JPF-Core-X shall execute the SUT by running one of its classes with a main() method.

Discussion

If any third party libraries are involved in the SUT, the user must replace them with abstractions in Java in order for JPF-Core-X to faithfully and completely analyze the SUT.

Requirement TOR-e..3: JPF-Core-X will accept the configuration options that are specified in the application properties .jpf file. See Section c.2. for details.

Requirement TOR-e..4: JPF-Core-X will run the SUT by executing the SUT class that is specified as the "target" in the .jpf file. See Section c.2. for details.

Note that one complete .jpf file must be defined for each distinct use case of the software that is to be verified. See Section c.2. for details.

e.1.2. Outputs from JPF-Core-X

JPF-Core-X's output must reliably inform its consumer whether the SUT has passed its tests (*i.e.*, whether any property violations were found) and whether JPF-Core-X has completed its task without errors. The output consumer may be an operator or an test automation system. JPF-Core-X output must include both human-readable output and process signals to ensure correct interpretation by both engineers and automated testing scripts. Human users and programs should use both the unambiguous log output and the process signals to provide redundant information about JPF-Core-X success and failure.

High quality, easily understood output regarding the cause of a SUT failure is a secondary consideration for qualification purposes, but an important aspect for reduction of development costs. JPF-Core-X should provide clear failure messages to facilitate the debugging of tool operation, LLRs, and safety properties, so that engineers can quickly determine whether identified failures are actual problems with the LLRs or improper use of the tool.

There are four important outcome categories that must be readily apparent to the user:

- *SUT failure*, when JPF-Core-X discovers one or more property violations in the LLRs;
- *SUT passes*, when JPF-Core-X completely explores the system state space without finding any property violations;
- *JPF-Core-X error*, when either inputs or internal fault cause JPF-Core-X to fail without completing its exploration of the state space.
- *JPF-Core-X interruption*, when an external source (such as the operating system, a user, or a test automation system) terminates JPF-Core-X before it has completed its test.

Correct indication of the last two conditions must be signaled clearly to the user, because a partial run of JPF-Core-X does not provide any information about the absence of property violations and must not be treated as such.

Requirements related to SUT failure The following set of requirements specify the correct behavior of JPF-Core-X when a SUT failure is detected.

Requirement TOR-e..5: JPF-Core-X shall report every property violation that it detects.

Requirement TOR-e..6: JPF-Core-X shall return an exit status of one to the calling process when it completes a run in which it finds one or more property violations.

Requirement TOR-e..7: JPF-Core-X shall direct reports of detected faults to the console.

Reporting detected faults to the console is part of JPF-Core-X's default behavior. The following lines must be present in the jpf.properties file in the JPF-Core-X project directory, and they must not be overwritten in the application properties .jpf file.

```
report.publisher=console
report.console.class=gov.nasa.jpf.report.ConsolePublisher
```

Requirement TOR-e..8: JPF-Core-X shall print the full error to the console for every property violation, if the user requests it.

To request this behavior, add the following line to the .jpf file:

```
report.console.property_violation=error
```

Requirement TOR-e..9: JPF-Core-X shall report other topics (in addition to "error") for each property violation, if the user requests it.

To report all topics, the *.jpf file should include the following:

```
report.console.property_violation=error,trace,snapshot,output,statistics
```

The order in which each topic is listed is the order in which it will be published in the report.

Requirement TOR-e..10: JPF-Core-X shall direct reports to an xml file if the user requests it.

Reports will be directed to an xml file if the following lines are added to the *.jpf file:

```
report.publisher+=xml
report.xml.class=gov.nasa.jpf.report.XMLPublisher
```

Requirement TOR-e..11: JPF-Core-X shall report all topics of property violations in the xml file if the user requests it.

To request that all topics of property violations be reported in the xml file, the *.jpf file should include the following:

```
report.xml.property_violation=error,trace,snapshot,output,statistics
```

Requirements specifying behavior when SUT passes The following set of requirements specify the correct behavior of JPF-Core-X when it completes a run without finding any property violations in the SUT.

Requirement TOR-e..12: JPF-Core-X shall output, to the console and to a log file, the string "JPF-Core-X TEST COMPLETE. RESULT: SUCCESS." followed, on the same line, by a unique test identifier whenever it completes a test without detecting any property violations in the SUT.

Discussion

JPF-Core-X must be written to ensure that the only circumstance in which it returns a zero exit status is when it completes a full exploration of the system without discovering any property violations. For example, invocations of JPF-Core-X to display usage information or any functionality that only simulates test operations should return an exit status of one to prevent the misinterpretation of JPF-Core-X output as a successful test.

Requirement TOR-e..13: JPF-Core-X shall return an exit status of zero to the calling process when it completes a run without finding any property violations.

Requirements related to JPF-Core-X error output The following set of requirements specify the correct behavior of JPF-Core-X when JPF-Core-X encounters an error condition in its own operation.

Requirement TOR-e..14: JPF-Core-X shall output, to the console and to a log file, the string "JPF-Core-X ERROR. RESULT: ERROR." whenever it encounters an internal error condition caused by abnormal inputs or incorrect internal operation.

JPF-Core-X can be configured to provide additional error output when desired to assist in the debugging of possible errors in input files and internal function.

Requirement TOR-e..15: JPF-Core-X shall return an exit status of one to the calling process when it encounters an internal error condition caused by abnormal inputs or incorrect internal operation.

Requirements related to JPF-Core-X interruption The following set of requirements specify the correct behavior of JPF-Core-X when JPF-Core-X is interrupted by an external signal or event.

Requirement TOR-e..16: JPF-Core-X shall output, to the console and to a log file, the string "JPF-Core-X INTERRUPTED. RESULT: INTERRUPT." whenever it terminates operation in response to an external signal or event.

Requirement TOR-e..17: JPF-Core-X shall return an exit status of one to the calling process when it terminates operation in response to an external signal or event.

e.2. JPF-Core-X Verification Operational Requirements

This section specifies the operational requirements on the verification methods implemented by JPF-Core-X, with the purpose of enabling the qualification of JPF-Core-X as a verification tool.

e.2.1. Deadlocks

Requirement TOR-e..18: JPF-Core-X shall verify whether or not the SUT satisfies the "Not-Deadlocked" property.

Discussion

Req. TOR-e..18 is expanded in the Tool Requirements document to list several supporting functional requirements.

Requirement TOR-e..19: When JPF-Core-X detects a deadlock, it shall provide a printout in its report that includes the terms "NotDeadlockedProperty" and "deadlock encountered".

An example of a deadlock in a console report is shown below: [1]

```
1 ========================================================== error #1
2 gov.nasa.jpf.jvm.NotDeadlockedProperty
3 deadlock encountered:
4   thread FirstTask:{id:1,name:Thread-1,status:WAITING,priority:5,lockCount:1,
      suspendCount:0}
5   thread SecondTask:{id:2,name:Thread-2,status:WAITING,priority:5,lockCount:1,
      suspendCount:0}
```

Requirement TOR-e..20: When JPF-Core-X detects a deadlock, it shall provide a program trace that includes the complete execution history leading to the deadlock.

[1] This example is taken from the "oldclassic" example that is included with the JPF-Core distribution.

e.2.2. Race Conditions

Requirement TOR-e..21: JPF-Core-X shall detect and report the presence of a race condition in the SUT, for applications with up to 8 threads.

Discussion

Tool requirements that support Req. e..21 should be developed in a manner similar to that shown for Deadlocks in Section e.2.1..

e.2.3. Application Specific Assertions

Requirement TOR-e..22: JPF-Core-X shall detect the presence of an Application Specific Assertion in the SUT.

JPF-Core-X's `AssertionProperty` class is a property listener that will intercept assertion errors before they are caught, and generate property violations for them. These property violations are automatically reported in JPF-Core-X's output.

To detect application specific assertions, the user must first add the `AssertionProperty` class to the list of listeners in the JPF properties file.

```
listener+=gov.nasa.jpf.listener.AssertionProperty
```

Then, individual assertions must be added to the Java source code that implements the LLRs. In general, the assertions are triggered when specific logical expressions evaluate to false. For example, the following will throw an assertion if variable "x" is greater than a prescribed threshold.

```
assert ( x <= THRESHOLD ) : "Variable x exceeded the prescribed threshold."
```

Discussion

Tool requirements that support Req. e..22 should be developed in a manner similar to that shown for Deadlocks in Section e.2.1..

f. Requirements to address abnormal activation modes

DO-330

"Requirements to address the abnormal activation modes or inconsistency inputs that should be detected by the tool. These requirements should consider the impact of those modes on the functionality and outputs of the tool. (This item is not applicable to TQL-5.)" [DO330-10.3.1-f]

Discussion

One priority: The tool should indicate input errors in a fashion that cannot be inadvertently ignored by users or scripts/tools that might be used to automate its application.
This section has some overlap with section b of the Tool Requirements Document.

Requirement TOR-f..1: The tool shall exit with an error and clearly indicate to the user the type of the error when it encounters a syntactically incorrect low-level requirement.

Requirement TOR-f..2: The tool shall exit with an error and clearly indicate to the user the source of error when it completes the execution of a test, but does not invoke one of the Listener objects associated with a property representing a high-level requirement.

Requirement TOR-f..3: The tool shall exit with an error and clearly indicate to the user the source of error when it is invoked in an environment which causes it to use a standard Java class library in place of a JPF-enabled analog class supporting simulated execution or when a method is called by the application that is not supported by the JPF-enabled analog class.

Discussion

As described earlier in Section a.1., JPF uses a combination of "model classes" and "native peer classes" to emulate the behavior of parts of system libraries and third-party libraries which cannot be emulated directly by JPF. For example, methods which call operating system library functions cannot be emulated by JPF's instruction by instruction execution of Java bytecode and may have consequential side effects on the application's behavior. When JPF encounters these methods, it will use an appropriate model class, many of which have been implemented by the JPF team to accurately model libraries with the fidelity necessary to support model checking. The model class may, in turn, invoke a method in a native peer class to support its emulation. In the event that an application invokes a method that is not supported by the suite of model classes, JPF must signal an error because it is impossible for JPF to emulate and verify the program effects of such functions without a model class implementation.

Requirement TOR-f..4: The tool shall exit with an error and clearly indicate to the user the

source of error when it is invoked by JPF-Core-X test file that specifies a nonstandard Listener library (*i.e.*, one that has not been distributed with the qualified tool distribution).

Discussion

The requirement below addresses the maximum allowable number of threads. In actuality, JPF-Core does not impose a limit on the number of threads. However, for a real tool qualification, it is important to consider the size and complexity of the software that the tool is expected to verify. Limiting the qualification scope of JPF-Core-X to a maximum number of threads is one way to define a meaningful bound. The advantage of imposing such a limit, in general, is that it provides some concrete bounds on the test cases and procedures to be used to qualify the tool. A disadvantage, of course, is that it also limits the applicability of the tool to programs that adhere to this thread limit.

For this exercise, we chose the maximum number of threads to be 8. This selection was based on our own observations of how long it took JPF-Core to complete analysis of simple models with varying numbers of threads. For a real qualification effort, the decision should be based on the maximum number of threads in programs that the tool is expected to analyze / verify.

Requirement TOR-f..5: The tool shall exit with an error and clearly indicate to the user the source of error when it determines that the number of threads in the SUT exceeds the maximum allowable number of threads. The maximum allowable number of threads for this verification is 8.

g. User information

DO-330

"The applicable user information, such as a user manual and installation guide or a reference to it, if not provided as part of the Tool Requirements data." [DO330-10.3.1-g]

Discussion

An online wiki for JPF is maintained by the developers at NASA Ames. It provides an introduction to JPF, a description of the installation process and separate guides for both users and developers. The online wiki may be found here:
http://babelfish.arc.nasa.gov/trac/jpf/wiki/WikiStart.
It is important to point out that the wiki describes the full JPF tool, which includes JPF-Core plus an optional set of add-on modules. The only required module is JPF-Core itself, which is described here: http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-core. Because the scope of this tool qualification is limited to JPF-Core, any discussion in the user's guide

related to other, optional modules is not relevant.

h. Description of operational use

DO-330

"Description of the operational use of the tool (including selected options, parameters values, command line, etc.)." [DO330-10.3.1-h]

JPF-Core-X may be run from the command line of a terminal window, or via the NetBeans IDE. Both methods are described in the subsections below.

h.1. Execution from the Command Line

Open a terminal window and navigate to the root directory of the SUT. A JPF-Core-X properties file with extension .jpf, specific to this SUT, should reside in this root directory. At the prompt, execute the following:

```
$ PATH-TO-JPF-CORE/jpf SUT-PROPERTIES.jpf
```

where "SUT-PROPERTIES.jpf" is the JPF-Core-X properties file for the SUT.

h.2. Execution from the IDE

Open the SUT project in the NetBeans IDE, and in the "Projects" explorer tab, navigate to the desired JPF-Core-X properties file. Control-click on the filename to bring up a context menu, and select verify. A sample view taken from the JPF-Core set of examples is shown in Figure h.2..

i. Performance requirements

DO-330

"Performance requirements specifying the behavior of the tool output." [DO330-10.3.1-i]

Discussion

The purpose of this section is to define requirements for how the tool should present information to the user under all of the possible modes of operation. We discuss some representative requirements for JPF-Core-X, distinguishing between *normal* and *abnormal* operation modes.

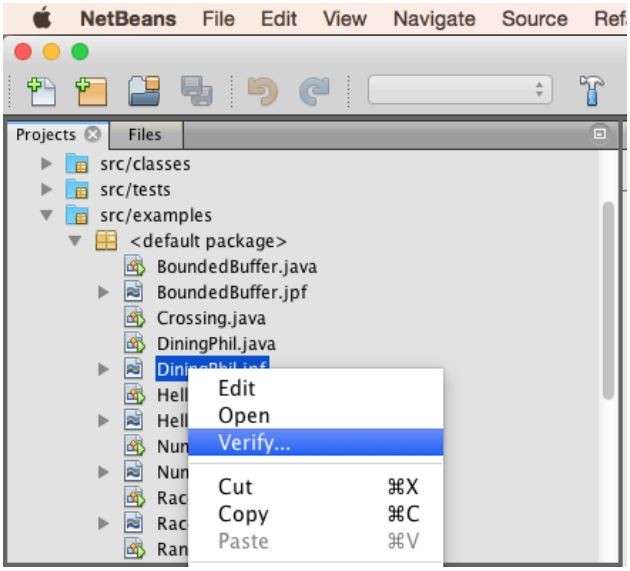

Figure 4: Control-Click on a JPF-Core-X properties file from the NetBeans IDE to run JPF-Core-X on your SUT.

Here, we list additional performance requirements associated with the output of JPF-Core-X that have not already been addressed. We consider two general modes: *normal operation*, when the tool is free from errors and able to function as intended, and *abnormal operation*, when some type internal error or environmental issue prevents the tool from functioning properly.

i.1. Normal Operation

Several operational requirements are already listed for normal operation of JPF-Core-X in Section e.1.2.. Those focus on reporting requirements that govern the output of the tool (either to the console or to files) when a property violation is detected and when the tool completes its analysis.

During normal operation, JPF-Core-X should also provide status information to the user to indicate that it is functioning normally, and to provide some information regarding the overall progress of the search.

Discussion

The requirements listed below actually go beyond the built-in features of JPF Core. Some limited status information is provided in the IDE console (log messages resulting from property violations), but this does not serve as a useful status window. Given that typical searches with JPF and other model checkers on complex programs could take several hours or even days, it is important to provide useful status information to the user so that they can 1) be certain it is still functioning properly and 2) maintain an awareness of how far it has

progressed. The requirements outlined below aim to meet these objectives.

Requirement TOR-i..1: JPF-Core-X shall display a "search status window" to contain a text-based summary of the current search.

Requirement TOR-i..2: JPF-Core-X shall display in the search status window: the name of the SUT and the name (and path location) of the JPF configuration file being used.

Requirement TOR-i..3: JPF-Core-X shall display the number of threads in the SUT in the search status window.

Requirement TOR-i..4: JPF-Core-X shall display one of the following messages at the top of the search status window during normal operation: 1) "Currently Searching" when the search method is actively running, 2) "Search Complete - Full State Space Searched" if the search method exits after a complete search of the state space, 3) "Search Finished - Partial State Space Searched" if the search method exits after an incomplete search of the state space.

Requirement TOR-i..5: JPF-Core-X shall display and update the time elapsed in the search status window when the search method is actively running. It shall continue to display the last updated time elapsed after the search method exits.

Requirement TOR-i..6: JPF-Core-X shall display and update the number of states searched in the search status window when the search method is actively running. It shall continue to display the last updated number after the search method exits.

Discussion

Displaying a progress bar or a percent completion would be desirable, but this would of course require knowledge of the total number of states to search. This knowledge could be obtained only if JPF-Core-X (or another tool) already performed a complete search of the state space for the *same* SUT and counted the number of different states. If the tool is run manually on a given SUT, then a separate manual step of importing that recorded total state number would be required, which leaves room for human error. If, however, the tool is run repeatedly on the same SUT (but perhaps with different configuration options), and if one of the runs leads to a complete search, then the total number of states could be recorded and used to display a progress bar or percentage in all subsequent runs.

Requirement TOR-i..7: JPF-Core-X shall display "Depth Limit Reached (1 time)" in the search status window the first time that the depth limit is reached.

Requirement TOR-i..8: JPF-Core-X shall display "Depth Limit Reached (X times)" in the search status window each time the depth limit is reached after the first occurrence, where X is incremented by 1 each time the depth limit is reached.

Requirement TOR-i..9: JPF-Core-X shall display "Memory Limit Reached" in the search status window when the memory limit is reached.

i.2. Abnormal Operation

Discussion

Requirements for abnormal operation are aimed at the goal of JPF-Core-X providing graceful degradation. Degrading gracefully essentially means that the tool must provide useful and accurate information to the user even when it encounters errors that prevent it from continuing to function. The most important aspect of these requirements is to ensure that a false negative does not occur. In other words, JPF-Core-X should never give the false impression that the SUT was found to be safe when an error prevented the tool from completing its verification.

Requirements e..14 through e..17 describe the required outputs for JPF-Core-X when it encounters different types of errors. An additional requirement to help ensure graceful degradation is for the search status window to indicate clearly that an error has occurred.

Requirement TOR-i..10: JPF-Core-X shall display "ERROR - JPF-Core-X internal error. Search incomplete." at the top of the search status window whenever JPF-Core-X encounters an error.

Note that the message above will replace the message displayed under normal operating conditions, as described in Req. i..4.

Detailed information describing the nature of the error will be provided in the report, as outlined in the requirements from Section e.1.2..

Another possible error condition would arise if there is insufficient disk space to write the output report files. If this were to occur, then the user should be alerted to this fact and the information contained in the report should be made visible to the user before JPF-Core-X exits.

Requirement TOR-i..11: JPF-Core-X shall display "ERROR - Unable to write report output files to disk." if the operating system does not allow JPF-Core-X to write any of the report output files.

Requirement TOR-i..12: JPF-Core-X shall make all of the reports visible within a console window if the operating system does not allow JPF-Core-X to write any of the report output files.

References

[1] S. P. Miller, A. C. Tribble, M. W. Whalen, and M. P. E. Heimdahl, "Proving the shalls: Early validation of requirements through formal methods," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 4, pp. 303–319, 2006. doi:http://dx.doi.org/10.1007/s10009-004-0173-6.