JPF-Core-X:
Tool Requirements (TR)

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 Requirements" (TR) for JPF-Core-X, and is written according to the guidelines in DO-330 Section 10.2.1.

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 TR will not be relevant. This document is thus a Framework for an actual TR, 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 TR; 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 h) of Section 10.2.1 in DO-330. In each section, we attempt to provide representative content of what should appear in an actual TR 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. Tool Functions and Technical Features

DO-330

"A description of the tool functions and technical features, including modes of operation." [DO330-10.2.1-a]

Java PathFinder (JPF) is a model-checking tool designed to enable customized verification of Java bytecode programs. JPF is designed as a plug-in architecture, composed of the required JPF-Core module, plus several optional add-on modules that each provide specific functions and features. The specific instance of JPF to be qualified for certification is JPF-Core-X, which consists

only of the JPF-Core module. [1]

JPF-Core-X can perform explicit state model checking to check for errors over all possible state values in all possible paths of the program. As such, JPF-Core can be used to verify the requirements of a system when those requirements are expressed in Java source code.

The application of JPF-Core 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's native functionality, without the use of any of the add-on modules available from the JPF maintainers or the addition of any end user-extensions, such as custom `Listener` classes.

a.1. Modes of Operation

JPF-Core-X is itself a virtual machine that executes Java source code. The set of source code supplied to JPF-Core is referred to as the System Under Test (SUT). Each time JPF-Core-X is run, it searches over all possible paths of execution for the SUT, and listens for pre-specified violations (deadlocks, race conditions, and application-specific assertions). This is the only mode of operation.

Discussion

Some of the aspects of how JPF-Core-X executes may be customized, such as the search mechanism and listener properties, for example (see Section c.). In addition, the tool may be executed in two different ways: either from the command line or via an IDE. Regardless of the configuration settings and the manner in which it is started, though, JPF-Core-X has a single mode of operation when it runs.

[1] We obtained JPF-Core-X from the JPF mercurial repository maintained by NASA on March 1, 2016.

b. User Instructions

DO-330

"User instructions, installation instructions, list of error messages, and constraints. This is often packaged as a user manual. The user manual may be part of the Tool Requirements or it may be packaged in one or more documents." [DO330-10.2.1-b]

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.

c. Customization Requirements

DO-330

"Requirements to describe the users ability to customize the tool." [DO330-10.2.1-c]

Customization of JPF-Core-X is managed through the *.jpf application properties file. An overview of the application properties that may be customized for JPF-Core, in the context of this qualification exercise, is provided below.

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 are included.

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

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

`search.properties` This tells JPF-Core-X which types of faults to look for.

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

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

`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.

`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 addeded.

`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.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. of the TOR for more discussion of property violations.

`listener` This defines the set of listeners that JPF-Core-X will use in its analysis. By default, JPF-Core automatically loads the NonNull and Const listeners by listing them in the `listener.autoload` key. In addition, the following listeners are added to the `listeners` key for JPF-Core-X:

d. Functional Requirements

d.1. Introduction

Discussion

DO-330 5.2.1.2.k states "The Tool Requirements should be defined to a level of detail appropriate to ensure proper implementation and to assess correctness of the tool (for example, defining underlying models or mathematical theories)." Accordingly, we include a short recapitulation of the technical background of JPF (similar to that found in PSAC Section c.2) and define some low-level requirements for the implementation of the JPF model checking algorithms.

JPF-Core-X belongs to the class of model checkers that are sometimes called an "execution-based model checker". In this type of model checker, the "model" is represented by an executable description, such as a programming language. The model checker simulates the execution by stepping through the execution of the code. At each choice point in program execution, *e.g.*, a scheduling decision or an input-dependent branching decision, the tool endeavors to enumerate every execution history forward from the decision point. In this way, the checker proceeds through a search of possible model execution.

JPF-Core-X uses a custom Java Virtual Machine (JVM) to perform explicit state model checking of models (such as low-level requirements) represented in the Java programming language. Properties such as high-level requirements are expressed as Java classes which extend the Listener-Adapter interface and check system state as execution progresses. As the surrogate virtual machine enumerates the possible executions of the Java representation, it repeatedly invokes the property checking classes which can inspect arbitrary aspects of the enhanced virtual machine to check for violations of the property.

d.2. Execution Process

The process that JPF-Core-X performs in the course of executing model-checking on any SUT are summarized in the steps below:

1. Initialization.
   (a) Load the JPF Properties file.
   (b) Initialize all listener and reporter objects.
   (c) Initialize the run-time object.
   (d) Initialize the virtual machine object.
2. Search.
   (a) Initialize the search method.

   (b) Backtrack step.
   (c) Forward step.
   (d) Check property violations.
   (e) Check search limits.
   (f) Terminate.
3. Reporting.
   (a) Report "Out of Memory" errors.
   (b) Report property violations.
4. Exit.

Requirements are enumerated in this document according to the main steps of the execution process outlined above. The overarching goal of the requirements is to show that JPF-Core-X can discover faults that occur in any part of the SUT under any possible sequence of operations. Consequently, the requirements are written to ensure that JPF-Core-X's coverage is both *correct* and *complete*. First, we describe the underlying algorithms and data structures in more detail to provide sufficient background information, and to lay a foundation for subsequent discussion of specific requirements.

d.3. State Space Model

To accurately simulate every possible execution of the SUT, JPF-Core-X must maintain correct data structures of its search state, the simulated state of the system after each transition, and the state of each branching point (called a "choice" by JPF-Core-X developers).

Discussion

For an actual qualification exercise, a tool provider would be required to provide a more detailed low-level description of their core algorithms and support the description included in the Tool Requirements document with references to published papers or internal technical reports to be submitted as part of the qualification packages. The high-level descriptions provided here are a product of our position outside the tool development organization and limited time and funding.

In the remainder of this subsection, we first describe the data structures used by JPF-Core-X that provide a model of the state space and the search path. Next, we describe JPF-Core-X's implementation of the depth-first search method, which is used to exhaustively explore the state space.

d.3.1. Data Structures

To completely and correctly explore simulated execution paths, JPF-Core-X maintains three critical types of data structures that represent the state of the overall search, the state of each ChoiceGenerator (representing branching points in system execution), and the state of the simulated system before each branching point.

Discussion

For purposes of this exercise, we include only representative, high-level descriptions of JPF's data structures.

The Search data structure includes a pointer to the current state of the simulated JVM and collections of required ChoiceGenerators.

Search
vm: JVM choices: ChoiceGenerators
search () : void

ChoiceGenerator
choiceSet: ChoiceSet
hasMoreChoices() : Boolean advance() : void getNextChoice() : Choice

The JVM data structure maintains system state, including variable values and the current status of all active threads. The JVM also contains a point to a Path object, which contains a linked list of Transitions representing the execution trace to the current state.

JVM
state: SystemState path: Path threads: ThreadInfo
forward () : void backtrack () : void restoreState () : void

d.3.2. Search Method

JPF-Core-X uses the classic depth-first search (DFS) algorithm [2] to explore the space of possible program executions.

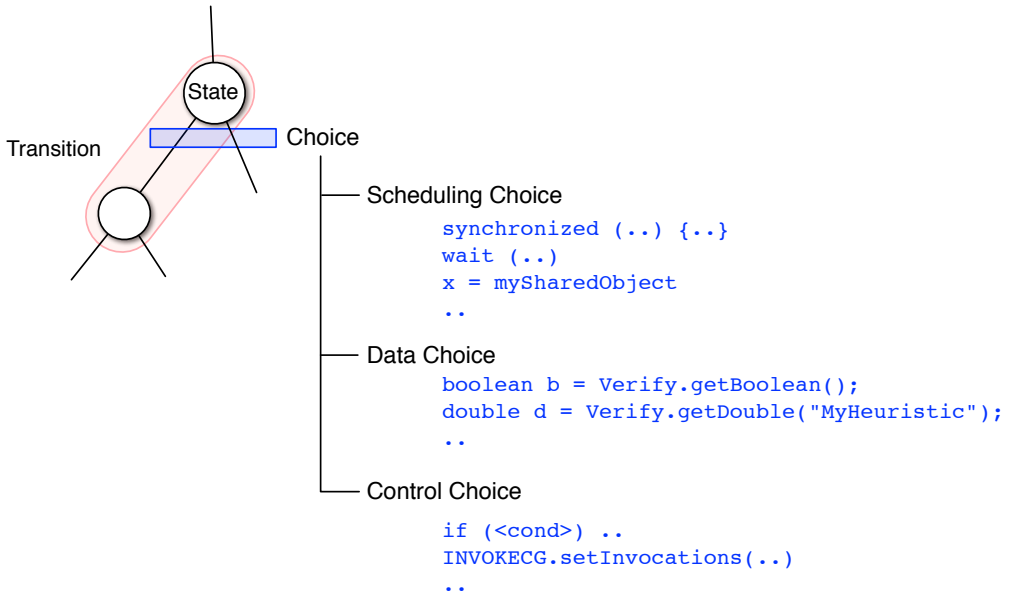

Figure 1: JPF-Core-X searches the possible executions of the SUT by progressing a simulated system until it reaches a "choice." Upon encountering a branch, it creates a "state" data structure to maintain a record of the program state immediately before the branch. Then, JPF-Core-X creates a "choice generator" object to generate each possible outcome of the branch. (This image was copied from Peter Mehlitz's JPF Tutorial slides [1].)

Discussion

Off the shelf, JPF allows its users to choose its search strategy. Users can select depth-first search, breadth-first search or another priority-based heuristic to direct the exploration of the possible executions of the SUT. The advantages of a priority-based heuristic search, in general, is that it may be able to guide JPF to a failure state while exploring a smaller number of system states than the depth-first search. However, a priority-based search algorithm requires additional complications in the state recording and the choice generators in the JPF infrastructure. In the case that there are no faults in the SUT, the depth-first variant should be faster, as it incurs less overhead. To simplify the qualification, we would propose to qualify only the depth-first variant, although during the development process, engineers might use the unqualified breadth-first search variant to more quickly discover faults in the early drafts of the LLRs.

One practical challenge for model-checkers is "state explosion", where the full enumeration of possible states to explore grows combinatorially with the number of choice points in the program. One of the practical goals of the tool, therefore, is to prune the search space whenever possible, to avoid searching parts of the space that cannot be reached, or that have already been searched. To help mitigate this issue, JPF-Core-X keeps a record of all states that have been searched. Before proceeding with execution down a new branch, it first compares the next state from the choice generator to the set of previously searched states. If it matches any of these states, then the branch must be a duplicate of one that has already been explored, allowing JPF-Core-X to ignore it and move to the next choice.

In JPF-Core-X, the DFS algorithm is implemented with the `DFSearch` class, which extends the abstract `Search` class. `DFSearch` is called directly from the main JPF-Core-X `run()` method. The UML diagram in Figure 2 illustrates precisely how the `DFSearch()` method is implemented in Java code. The legend to the left of the diagram defines the meaning of the various symbols used. Some further explanation is offered here to ensure clarity. All method calls are represented as rectangles, and are color-coded to indicate which class implements the method. All of the methods called from `DFSearch` belong either to the JVM class (black) or the Search class (green and blue). Blue is used to indicate Search class method of the *notification* type, which are used to notify the registered listener and reporter classes of significant events that occur during the search.

The algorithm consists of six major steps, as shown with the six different colored backgrounds. The steps are outlined below:

0. **Initialize.** Prior to calling `DFSearch()`, the Java virtual machine (JVM) object is initialized from the `run()` method in `JPF.java`. Local variables in `DFSearch()` are then initialized as shown, and all listener and reporter objects are notified that the search has started.

1. **Backtrack Step.** A backtrack is required if no forward steps can or should be taken from the current state. This holds true if any of the following conditions are met: a) a backtrack has been requested following the a property violation or reaching the depth limit, b) the current state is not new (i.e. it is identical to a previously explored state), c) the current state is an

end-state (i.e. if there are no more threads to run), or d) the current state is ignored because the choice generator found zero choices. The backtrack method is implemented in the `JVM` class. If the backtrack attempt was successful, it decrements the depth counter and continues. Otherwise, no further backtracking is possible, at which point the search method exits.

2. **Forward Step.** The `forward()` method is implemented in the `JVM` class, which applies a choice generation technique to identify and select the next state to transition to. If a state transition was made, the depth counter is incremented, otherwise it returns to the backtrack step.

3. **Property Violation Checks.** After the state has moved forward, potential property violations are checked. In JPF-Core-X, four separate listener classes are used to check for property violations: `NotDeadlockedProperty`, `PreciseRaceDetector`, `AssertionProperty`, `NoUncaughtExceptionsProperty`. If no violation is found then the search continues. If instead a property violation is found by any of the listeners, it is reported. The search continues after a property violation *only* if the configuration option to find *all* errors is chosen; otherwise, the search terminates.

4. **Search Limit Checks.** After the state has been checked for property violations, the current depth of the search is checked. If it exceeds a prescribed depth limit, which may be set in the JPF properties file, a backtrack is requested and the method returns to the backtrack step. Otherwise, the state space limit is checked by evaluating the free memory available. If it has fallen below the minimum free memory threshold, which may also be set in the JPF properties file, then the search terminates.

5. **Terminate.** The search will terminate if any of the following conditions are met: a) a backtrack was attempted but could not be completed, indicating that all possible states have been searched, b) a property violation was found, and the configuration option to collect *all* errors was *not* selected, or c) the memory available to JPF-Core-X dropped below the minimum free memory threshold.

d.4. Initialization Requirements

Discussion

In a complete TR, the tool requirements that support the "Initialization" step would be provided here. For our sample TR, we instead focus on detailing a subset of the requirements for the "Search" step. These are discussed in the next subsection.

d.5. Search Requirements

To meet the requirements in this section to detect deadlocks, race conditions, and violations of application-specific assertions, JPF-Core-X must correctly simulate all possible paths through the

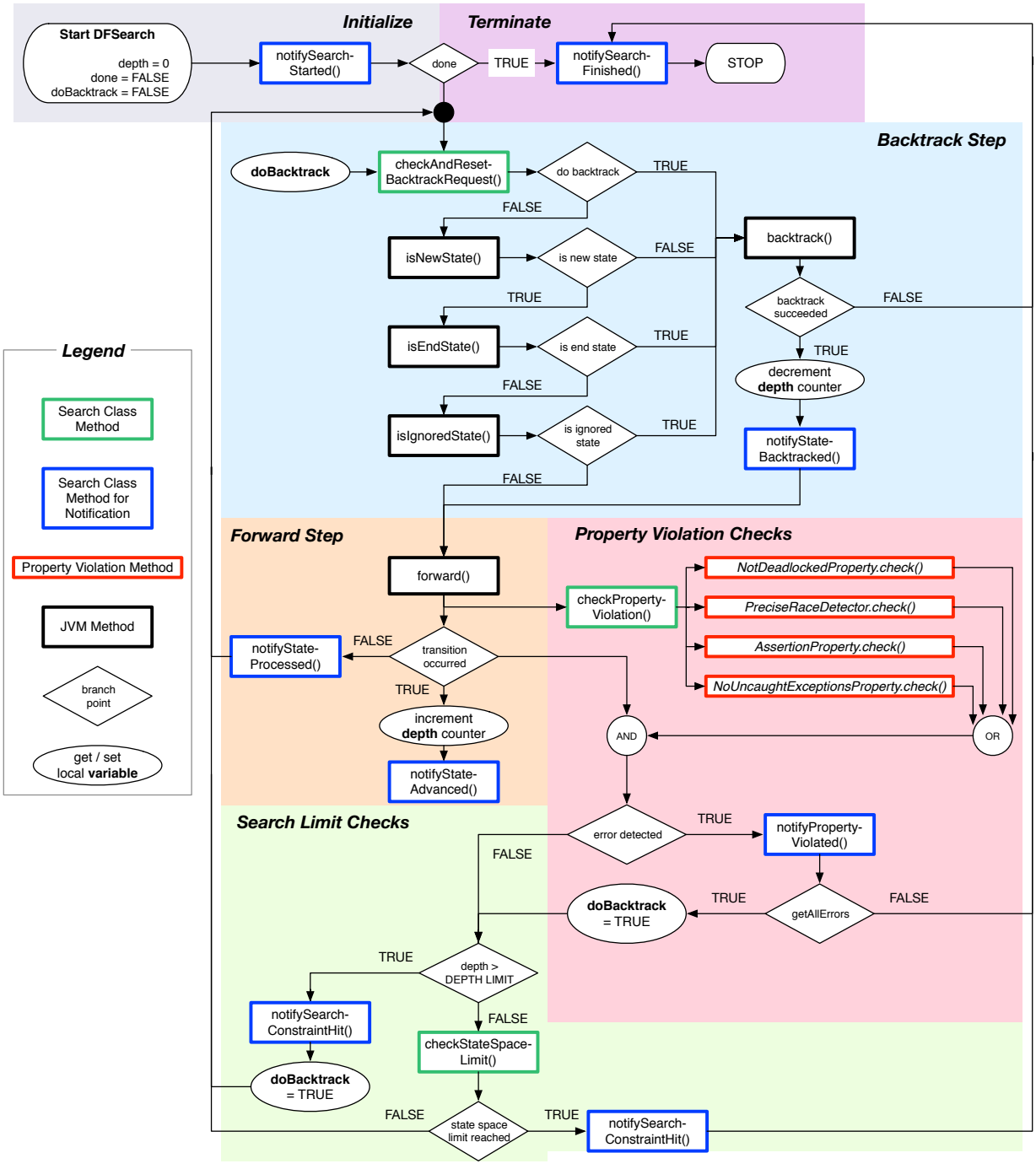

Figure 2: UML Diagram of the Depth-First-Search Implementation in JPF-Core-X

SUT. The following requirements establish the basis for correct and complete path coverage.

Requirement TR-d..1: Path Coverage JPF-Core-X shall observe all possible program states by executing all possible paths through the SUT.

This requirement amounts to searching the full state space of the SUT. For this qualification, we consider only the depth-first search method, as described in Section d.3.2..

The following subrequirements establish that the search advances correctly through transitions (instructions), that it identifies branches in the search (choicepoints), and that it correctly maintains sufficient execution state to identify identical system states.

Requirement TR-d..1.1: JVM Initialization JPF-Core-X shall properly initialize the JVM object.

Initialization of the JVM involves the following main steps:

1. Load class structures (e.g. fields, interfaces)
2. Create a thread to generate the stack
3. Create class objects on the thread
4. Push the `clinit` (class initialization) methods onto the stack

Discussion

The full initialization of the Java virtual machine involves the steps outlined above. In a complete TR document, each of these would spawn a set of specific requirements to ensure the steps are implemented correctly. For this sample TR, however, we omit the requirements associated with JVM initialization and instead focus on those associated with the depth-first search.

Requirement TR-d..1.2: Search Object Initialization JPF-Core-X shall properly initialize the `DFSearch` object. The vast majority of the initial program state is contained in the JVM object, which is the subject of the previous requirement. This requirement focuses only on the proper initialization of the local variables for the `DFSearch` object. As Figure 2 shows, this involves setting the integer `depth` to 0, and the booleans `done` and `doBacktrack` each to FALSE.

Requirement TR-d..1.3: Backtrack JPF-Core-X shall correctly implement the **backtrack** step within the depth-first search method.

Requirement TR-d..1.3.1: Backtrack Request JPF-Core-X shall check whether a backtrack has been requested.

The requesting of a backtrack is performed by setting the `doBacktrack` variable to TRUE. In general, this may be achieved from outside the `DFSearch` class by calling its `requestBacktrack()`

method. In JPF-Core-X, however, the only times that a backtrack is requested is from within the `DFSearch` class, either following the detection of a property violation, or after the depth limit has been reached.

Requirement TR-d..1.3.2: isNewState Check JPF-Core-X shall determine whether the current state is new, or if it is identical to a previously explored state.

This state-matching functionality is implemented in the `isNewState()` method of the JVM class.

Discussion

In a complete TR, a separate set of requirements would be provided to describe the required functionality of the state matching method.

Requirement TR-d..1.3.3: isEndState Check JPF-Core-X shall determine whether the current state is an end-state.

A state is considered an "end-state" if there are no more threads to be run. This evaluation requires that the JVM maintain sufficient knowledge of all of the threads in the SUT: whether each is a daemon, whether it has been terminated, and whether it is runnable.

Discussion

In a complete TR, a separate set of requirements would be provided to describe the proper implementation of the end-state evaluation.

Requirement TR-d..1.3.4: isIgnoredState Check JPF-Core-X shall determine whether the current state is to be ignored.

When using DFS as the search method for JPF-Core-X, the current state is ignored *only* if the choice generator finds that there are no choices for forward transition. Requirements associated with the choice generation method are discussed later.

Requirement TR-d..1.3.5: Backtrack JPF-Core-X shall properly execute the backtrack step if the any of the conditions for executing the backtrack are satisfied.

The `backtrack()` method for depth-first search is implemented with the DefaultBacktracker class. The explored paths for kernel states and system states are stored as immutable lists. The backtrack method is a straightforward process. When it is called, the state that caused the backtrack is still on the stack, and so it is removed first. The backtrack process therefore involves popping two states and restoring the second one.

Requirement TR-d..1.3.6: Backtrack Success JPF-Core-X shall indicate whether the backtrack step was successful or not, and notify all listener objects when a backtrack occurs.

Backtrack steps will continue as long as there are no unexplored forward transition choices. If the search algorithm has returned to the initial state, a backtrack step cannot be executed from

this state.

Requirement TR-d..1.3.7: Terminate on Backtrack Failure JPF-Core-X shall terminate if a backtrack step is attempted, but cannot be executed.

If no search constraints have been hit, then a termination of JPF-Core-X under this condition means that the full state space of the SUT has been explored. Termination here without having identified any search constraints or property violations would indicate that the SUT satisfies the high level requirements.

Requirement TR-d..1.4: Forward JPF-Core-X shall correctly implement the **forward** method within the depth-first search method.

Discussion

The forward method involves multiple steps with calls to several different methods. This is also where choice generation is predominantly used. In a complete TR, another set of pseudocode and a UML diagram similar to Figure 2 would be appropriate to illustrate the actions of the forward method. For this sample document, however, we instead limit our discussion to a textual description of the method below.

The forward method performs the following sequence of actions:

1. Initialize the next transition. Uses the method: `initializeNextTransition()`. This checks if the next proposed transition would have a valid choice before actually performing the transition to that state. If this step returns FALSE, the forward method immediately returns FALSE. Otherwise, continue.
2. Push the kernel state. Uses the method: `pushKernelState()` in the DefaultBacktracker class.
3. Cache the current saved path. Uses the `getLast()` member function on the local `path` object and save.
4. Execute the next transition. Uses the method: `executeNextTransition()` in the SystemState class.
5. Push the system state. Uses the method: `pushSystemState()` in the DefaultBacktracker class.
6. Update the path. Uses the method `updatePath()` in the JVM class. This stores the current SystemState's trail in the path, after updating it with annotations from the JVM.
7. If the new state is not ignored, perform garbage collection and store the new state ID. Otherwise, continue.
8. Exit with TRUE.

Note that step 1 above applies an assertion that the new choice generator object is not NULL before returning. Otherwise, if the new choice generator object is NULL, JPF-Core-X exits with an error stating "transition without choice generator".

Discussion

In a complete TR, each of the steps outlined for for the forward method would spawn one or more requirements stating that they are implemented properly.

Requirement TR-d..1.4.1: Forward Success If execution of the forward method results in a state transition, then JPF-Core-X shall increment the depth counter and notify listeners and reporters if the state has advanced.

Following the call to the forward method, the depth counter is incremented if the method returns TRUE, indicating a transition to a new state has occurred.

Requirement TR-d..1.4.2: Forward Failure If execution of the forward method does *not* result in a state transition, then JPF-Core-X shall notify listeners and reporters if that the state has been processed.

We next consider the requirements associated with detecting property violations.

Requirement TR-d..1.5: Property Check JPF-Core-X shall correctly implement the step of **property violation checks** within the depth-first search method.

As shown in Figure 2, if a state transition occurs in the forward method, the `checkPropertyViolation()` method of the Search class is called. This method calls the `check()` method of all property listener objects that have been initialized. In JPF-Core-X, the set of property listeners includes:

- `NotDeadlockedProperty`
- `PreciseRaceDetector`
- `AssertionProperty`
- `NoUncaughtExceptions`

If any property violation is found, it is reported.

Requirement TR-d..1.5.1: Property Check Notification If a property violation is found, JPF-Core-X shall notify all reporter objects of the property violation.

If the configuration option to "get all errors" is selected, then JPF-Core-X will continue to search after a property violation occurs. Otherwise, if this option is not selected, then JPF-Core-X will terminate after the first property violation.

Each of the above property violation checks must be implemented correctly. This leads to the sub-requirements below:

Requirement TR-d..1.5.2: Property Check for Deadlock JPF-Core-X shall correctly implement the method of checking for the "not-deadlocked" property.

Requirements for detecting the presence of a deadlock are described in Section d.5.1..

Requirement TR-d..1.5.3: Property Check for Race Conditions JPF-Core-X shall correctly implement the method that detects race conditions.

The result of a race condition is either an assertion or an uncaught exception, each of which can be detected *without* the PreciseRaceDetector listener. Including this listener will determine whether either of the above errors, if found, are caused by the presence of a race condition.

Requirements for detecting the presence of a deadlock are described in Section d.5.2..

Requirement TR-d..1.5.4: Property Check for Assertions JPF-Core-X shall correctly implement the method of checking for the "assertion" property.

The AssertionProperty class intercepts assert calls that are made from the SUT and labels them as property violations. Requirements for detecting the presence of an assertion are described in Section d.5.3..

Requirement TR-d..1.5.5: Property Check for Uncaught Exceptions JPF-Core-X shall correctly implement the method of checking for the "no-uncaught-exceptions" property.

The NoUncaughtExceptions class uses Java's built-in error handling methods to intercept uncaught exceptions, such as "DIV BY ZERO" for example, and labels them as property violations.

Requirement TR-d..1.6: Search Limit Check JPF-Core-X shall correctly implement the step of **search limit checks** within the depth-first search method.

Two search limits are evaluated: the depth, and the available memory.

Requirement TR-d..1.6.1: Depth Limit Check If the depth counter meets or exceeds the prescribed limit, JPF-Core-X shall notify listener and reporter objects that a search constraint has been hit and proceed to the backtrack step.

Discussion

It is important to consider how the tool handles a situation when the depth limit is reached. The above requirement is consistent with the actual implementation of the depth-first search method in JPF-Core. If the depth limit is reached during the search, it notifies the report listeners, backtracks, and continues. This is a useful feature for a development tool because at development time, one is interested in finding (and later fixing) as many errors as possible. Continuing to search, rather than stopping now and later starting over, is an efficient use of time for the developer. It may be argued, though, that a verification tool should immediately exit with an error if the depth limit is reached. Reaching a depth limit necessarily means that parts of the state space will remain unexplored, making it impossible to verify properties within that unexplored space. A verification tool, therefore, may as well terminate with an error once it knows that it cannot complete the search. Of course, if this argument is accepted, it calls into question the value of using a depth-limit in the search

method in the first place.

The most critical aspect of the depth limit is to ensure that it does not lead to a false negative result. A false negative would occur if 1) the depth limit prevents the tool from exploring a state with property violations, and 2) the report indicates that no property violations are found, leading the user to conclude that the SUT was verified to be safe. However, this problem is mitigated by including information in the report about the depth limit being reached. As discussed previously in the search method description, if the depth limit is reached, the first action is to notify the report listeners. Therefore, the report will include this critical information about the depth limit being reached. Going a step further, it is important to make this fact clear and obvious to the user in the report. It must be understood that the search was incomplete, and therefore the SUT could not be verified to be safe, even if no proprety violations were found.

Requirement TR-d..1.6.2: Memory Limit Check If the available memory drops below the prescribed limit, JPF-Core-X shall notify listener and reporter objects that a search constraint has been hit and terminate.

Requirement TR-d..1.7: Terminate JPF-Core-X shall correctly implement the **terminate** step within the depth-first search method.

If any of the termination conditions are met, all listeners and reporters are notified that the search has finished, and the search method exits.

d.5.1. Detection Requirements for Deadlocks

Requirement TR-d..2: Deadlock Check JPF-Core-X shall verify whether or not the SUT satisfies the "Not-Deadlocked" property. (This requirement is identical to Req. TOR-e..18 in the TOR.)

Discussion

The set of requirements below, from TR-d..3 to TR-d..5, represent functional requirements for JPF-Core-X that support Req. TR-d..2.

The following types of functionality are required in order to support Req. TR-d..2.

Requirement TR-d..3: Deadlock Detection JPF-Core-X shall be capable of identifying the presence of a deadlock by examining the current program state.

Requirement TR-d..3.1: Thread Access JPF-Core-X shall have access to all threads spawned by the program.

Requirement TR-d..3.2: Thread State JPF-Core-X shall be capable of identifying the current state of each thread. It will determine whether each thread is: Suspended, Running, Unblocked, Sleeping, or Timed-Out.

Requirement TR-d..3.3: Thread Runnable Based on the current state of each thread, JPF-Core-X will determine whether each thread is "runnable".

Requirement TR-d..3.4: Deadlock Condition Detection If the "runnable" status of at least one thread is true, JPF-Core-X will NOT detect a deadlock. Otherwise, if all threads are not runnable and none of the threads are daemons, then JPF-Core-X will detect a deadlock.

Requirement TR-d..4: Deadlock Output When JPF-Core-X detects a deadlock, it shall provide a printout in its report that includes the terms "NotDeadlockedProperty" and "deadlock encountered". (This requirement is identical to Req. TOR-e..19 in the TOR.)

An example of a deadlock in a console report is shown below: [2]

```
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
6   SecondTask:{id:2,name:Thread-2,status:WAITING,priority:5,lockCount:1,
      suspendCount:0}
```

Requirement TR-d..5: Deadlock Output Trace When JPF-Core-X detects a deadlock, it shall provide a program trace that includes the complete execution history leading to the deadlock. (This requirement is identical to Req. TOR-e..20 in the TOR.)

d.5.2. Detection Requirements for Race Conditions

Requirement TR-d..6: Detect Race Conditions 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. TR-d..6 should be developed in a manner similar to that shown for Deadlocks in Section d.5.1..

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

d.5.3. Detection Requirements for Application Specific Assertions

Requirement TR-d..7: Detect Violations of Application-Specific Assertions JPF-Core-X shall detect the presence of an Application Specific Assertion in the SUT.

Discussion

In order for JPF-Core-X to detect a violation of an application-specific assertion, the conditions for the assertion must be implemented and checked in SUT itself. Once the assertion error has been encountered, JPF-Core-X must then properly handle the violation, by first noticing it and then logging and reporting it. This aspect of handling and reporting the property violation can be verified. However, the initial task of identifying and throwing the assertion error can only be implemented within the SUT, and is therefore outside the scope of the tool qualification.

JPF-Core-X's `AssertionProperty` class is a property listener that will intercept assertion errors before they are caught by the JVM, 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-Core-X properties file.

```
1 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.

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

Discussion

Tool requirements that support detection of specific assertions, Req. TR-d..7, should be developed in a manner similar to that shown for Deadlocks in Section d.5.1..

d.6. Reporting Requirements

Discussion

In a complete TR, the requirements that support the "Report" step should be provided here.

e. Requirements Imposed by Tool Operational Environment

DO-330

"Specific requirements, if necessary, for compliance with tool operational environment." [DO330-10.2.1-e]

Discussion

There is nothing about this section of the TR of special interest in the case where the tool to be qualified is a formal methods tool.

JPF-Core is an open source research project developed by NASA. It is a pure Java application. As such, it runs on the Java virtual machine, which itself can be run on Windows, OSX, or Unix operating systems. The specific release of JPF-Core considered here (JPF-Core-X) requires Java version Java SE 7, which corresponds to JDK 1.7. If this were a TR for an actual JPF-Core qualification, there would be a specific version of JPF-Core that was being qualified, and thus it would have a specific operational environment. Any requirements specific for JPF-Core's operational environment would be described here.

f. Failure Mode Requirements

DO-330

"The Tool Requirements should be defined such that abnormal behavior is detected and that invalid output is prevented. For example, it is better for the tool to produce no output than to produce wrong results. Therefore, the Tool Requirements should address failure modes and describe responses to failure conditions." [DO330-5.2.1.2-f]

DO-330

"Failure modes and errors: The objective is to ensure that the Tool Requirements define the behavior of the tool in response to error conditions and specific requirements addressing failure modes and incorrect inputs are identified." [DO330-6.1.3.1-d]

DO-330

"Robustness tests should be performed to address all failure modes (for example, abnormal activation modes, inconsistency inputs, etc.) identified in Tool Requirements." [DO330-6.1.4.2-c]

DO-330

"Specific requirements to address the failure modes and response to inconsistent inputs." [DO330-10.2.1-f]

Discussion

Qualified model checking tools will be used as an adjunct to code reviews and requirements inspections to increase the confidence in the these development artifacts and decrease the cost and labor required to detect deadlocks, races, and application specific errors. To the end, they should be employed by trained engineers who understand the limits of the tools and are aware of the effects of possible errors caused by incorrect tool inputs. We have identified four types of errors, their effects, and tool considerations which should mitigate the consequences.

- *False positives*: False positives caused by incorrect inputs such as inaccurate requirements or misconfiguration of the tool. Requirements might be inaccurate so that either error conditions are incorrectly specified or that the simulated execution of the system model enters states that would not be reachable if the requirements were expressed correctly. The first case should be easy to identify by a user who examines the state that causes the violation and compares it to the reported violation. If the state is not in fact a violation of the requirements, the properties expressing the requirement can be corrected. It may be more difficult to detect cases where an input error has caused the system model to enter a state that should not be reachable. In that case, an engineer who understands the requirements and the execution simulation use by the tool must examine the trace provided by the model checker and identify a transition in the simulation that should not be allowed.

  The tool can provide some features that mitigate the cost of these possible failures. First, it should strictly check that inputs adhere to syntax requirements. Further error-checking could also be applied to the inputs to identify expressions which may correlate with errors, such as test expressions that are either always or never true. Second, messages for the user that describe the property violation must be as clear and specific as possible, displaying both the property expression that was violated and the input file and line number from which it came. If possible, the tool should also identify the locations in the input files of the simulated execution (possibly specifying the execution point when the error occurred for several threads). Third, state descriptions must be complete and easy to understand. Finally, the output of execution traces must be easily understood. In particular, users must be able to follow the particular interleaving of threads that led to the reported failure condition.

- *False negatives:* False negatives can be caused by incorrect inputs that incorrectly specify test properties or the system model. These errors may have serious consequences for the correctness and safety of the overall program. The apparently successful com-

pletion of the model checking tool may lead to a decrease in the effort expended to identify requirements violations by other means such as code inspection. As a result, an error, which would have been identified by a correct use of the model checking tool, could persist in the product until it is identified in testing, causing an expensive process of debugging and re-engineering a piece of software that was believed to be complete.

Errors of this type may be caused by input errors that cause the tool to fail to explore possible execution paths of the system model or errors that misrepresent the system requirements as safety properties to be tested. In the former case, the tool will fail to reach a possible system state because the incorrect input improperly makes some parts of the execution space unreachable. In the latter case, a safety property could be a poor translation of the expressed requirement.

As in the case of false positives, syntax and style checking could make errors of this type less likely by identifying errors in the system models and possibly vacuous safety properties. In addition, a strategy should be employed to provide some checking of the completeness of the execution simulation. For example, tool users should specify "sentinel" states and properties that the tool should reach during simulated execution. These states and properties are separate from the software requirements since they are specific to a particular embodiment of the software requirements and do not trace directly to any HLR. In fact, although some of these states and properties should sample routine nominal behavior of the software, many of them should identify "corner cases" in the particular design which software engineers have considered during design. The use of these sentinels may also benefit the design process by identifying which of these corner cases will not be encountered. In that case, tool users and software engineers should consider whether the inputs correctly specify all operating conditions and/or whether there is unreachable code in the design.

- *User error:* It is possible that tool users may misconstrue the output of the tool. In the case that the user incorrectly concludes that the tool has found an error, we expect that the error will be quickly corrected by close inspection of the tool output. However, in the event that the user incorrectly concludes that a test has passed, an error may persist until code inspection or testing.

  To mitigate the possibility that a user may believe a test has passed when it has actually failed, we require the tool to provide clear and unambiguous output when it finds an error in the SUT. Further, the tool should return an exit signal of one to its caller to support error reporting by any automated build or test process.

- *Incomplete test:* Depending on the complexity of the SUT and the computing resources provided to the tool, model checking may run for a relatively long time. This creates the possibility that an incomplete test may be interpreted as a passing test. Since an incomplete test will not completely explore the possible executions of the software, a safety violation may exist in the unexplored portion of the execution spare and remain

undiscovered. Any undiscovered safety violation might persist until code inspection or even system testing, incurring enormous costs to redesign and retest the system to overcome the error.

To mitigate this possibility the tool must be required to clearly log and signal an error if it is interrupted before completing its exploration of the execution space. For example, any handlers in the tool that terminate execution based on deadlines or resource allocation failures must output an unambiguous error message and return an exit signal of one. Similarly, when an external signal terminates the tool, it must clearly indicate that the test did not pass. Furthermore, tool users must be trained to recognize the difference between a successful test and an incomplete test. Anyone who writes a script to automate the use of the tool must take care that their script does not muffle error messages or exit signals.

f.1. Incorrect Inputs

JPF-Core-X must signal an unambiguous error when it encounters syntactically incorrect inputs. Correct deployment of JPF-Core-X requires a correct representation of the LLRs as Java source code. The HLRs (properties with which the LLRs must comply) are encoded by reference to a Listener in the JPF properties file.

Discussion

Representative requirements for input handling are defined in the TOR.

See Req. TOR-f..1 in the TOR.

See Req. TOR-f..2 in the TOR.

f.2. Errors When Internal Limits Exceeded

JPF-Core-X must signal an unambiguous error when it exceeds internal limits such as running out of memory or exceeding its internal depth limit.

Requirement TR-f..1: Error for exceeding memory limit JPF-Core-X shall exit with an error and clearly indicate to the user the type of error when it runs out of memory during state-space exploration.

Discussion

We discussed the tool's response to reaching a depth limit in Section d. In the actual implementation of the depth-first search in JPF-Core, the response is to notify the report listeners, backtrack, and continue the search. Perhaps a more appropriate response for a verification tool would be to exit with an error, since reaching this condition means that

part of the state space would not be explored, and as a result, the SUT could not be verified. If this approach were to be used, then the following requirement would be appropriate.

Requirement TR-f..2: Error for reaching depth limit JPF-Core-X shall exit with an error and clearly indicate to the user the type of error when it meets or exceeds its internal limit on depth of search.

f.3. Errors In Internal Logic

Requirement TR-f..3: Error for internal logic failure JPF-Core-X shall exit with an error and clearly indicate to the user the type of error when it fails an internal assertion indicating an error in internal logic such as "transition without choice generator."

g. Responses Under Abnormal Operating Conditions

DO-330

"The expected responses of the tool under abnormal operating conditions." [DO330-10.2.1-g]

JPF-Core-X must signal an unambiguous error when it is invoked in an incorrectly configured operating environment. Qualified operation requires that Java paths (environment variables) refer to the correct JPF class libraries and that the JPF properties file refers to qualified Listener objects.

Discussion

Representative requirements for the operational environment.

See Req. TOR-f..3 in the TOR.

See Req. TOR-f..4 in the TOR.

h. Internal Interface Requirements

DO-330

"For a collection of tools, interface requirements between the tools within the collection. " [DO330-10.2.1-h]

JPF-Core-X is a single tool, not a collection of tools, and there are no internal interfaces to define.

References

[1] P. C. Mehlitz. *JavaPathfinder Tutorial 02/2010*. http://babelfish.arc.nasa.gov/trac/jpf/raw-attachment/wiki/presentations/start/JPF-Tutorial.pdf.

[2] S. Russell and P. Norvig, *Artificial Intelligence : A Modern Approach*, Prentice-Hall, second edition, 2003.