JPF-Core-X:
Plan for Software Aspects of Certification (PSAC)

Contents

a Identification of the tool **1**

b Details of certification credit sought **3**

c Substantiation of maturity and technical background **3**
c.1 Maturity 5
c.2 Technical Background 5

d TQL proposed **6**

e Tool source **7**

f Stakeholders **7**

g Description of TOR, TOIP, and TOV processes **8**

h Operational environment **8**

i Previous Applications **8**

j TQP **9**

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 "Plan for Software Aspects of Certification" (PSAC) for JPF-Core-X, and is written according to the guidelines in DO-330 Section 10.1.1.

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 PSAC will not be relevant. This document is thus a Framework for an actual PSAC, 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 PSAC; 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 j) of Section 10.1.1 in DO-330. In each section, we attempt to provide representative content of what should appear in an actual PSAC 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 Identification of the tool

DO-330

"Identification of the tool and its intended use, including its impact in the software life cycle process." [DO330-10.1.1-a]

The tool to be qualified for certification is "JPF-Core-X". We obtained the core module of JPF (JPF-Core) from the JPF mercurial repository maintained by NASA on March 1, 2016. We refer to this version of the software as JPF-Core-X. 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/jpf-core`. The last changeset included in JPF-Core-X is `29:820b89dd6c97` committed on October 16, 2015.

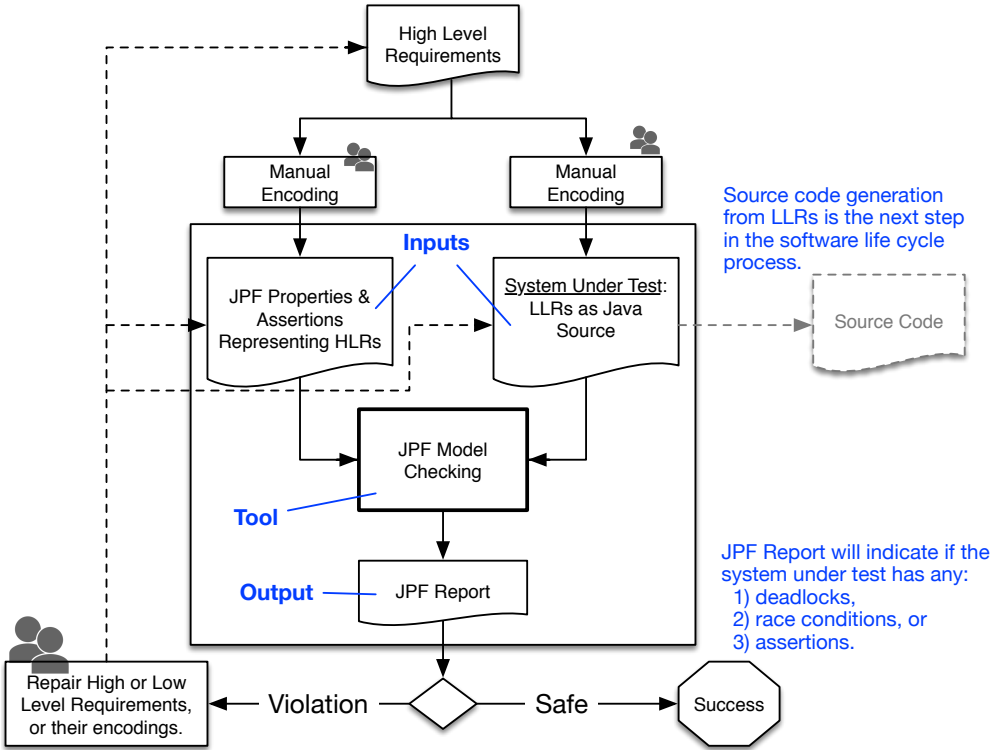

Figure 1: Flow of HLRs and LLRs for JPF-Core-X model checking.

The JPF-Core-X tool, a model-checking tool, will be used to verify the absence of some types of faults from the Low Level Requirements produced during the software development process. By using JPF, we will effectively replace some elements of the code review process (*e.g.*, inspection for potential system deadlocks) with a more rigorous, formal analysis.

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 [19].

Figure 1 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 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;

Discussion

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 maintainers or the addition of any end user extensions, such as custom `Listener` classes. Accordingly, our verification is limited to only the core module of JPF Ẇe do not use or qualify any modules or custom extensions in this stufy.

b Details of certification credit sought

DO-330

"Details of the certification credit sought through tool use for eliminating, reducing, or automating the process(es)." [DO330-10.1.1-b]

The certification credit sought by the use of JPF as a verification tool is to automate a portion of DO-178C certification activities. This includes verifying the following: compliance of low level requirements (LLRs) with high level requirements (HLRs); the accuracy and consistency of the LLRs; the verifiability of the LLRs; and algorithm accuracy in the LLRs.

Figure b shows the Certification Credit Matrix, which lists the specific credits sought for using the JPF tool to automate requirements verification.

c Substantiation of maturity and technical background

DO-330

"Substantiation of the maturity and technical background of any technology or theory (for example, mathematical theory) implemented in the tool to show its applicability." [DO330-10.1.1-c]

DO-178C Table/ Item	Verification Activities	DO- 178C Ref.	Certification Credit	Justification	Additional Activities Required
A-4/1	Low-level requirements comply with high-level requirements.	6.3.2.a	Partial	JPF-Core ingests LLRs embodied in Java code to create an exhaustive, dynamic simulation of system behavior. A subset of HLRs can be to encoded as JPF properties to be verified though checking. Specifically, JPF-Core can check the compliance of LLRs to HLRs that are captured as application-specific assertions within the Java source code used to express the LLRs.	• Additional testing must be performed to verify HLRs that cannot be expressed as JPF-Core properties. • Additional engineer inspection may be required to validate the correctness of assertions that represent HLR. For instance, the requirement might be formalized incorrectly. • Additional engineer effort may be required to assess whether the HLRs expressed as JPF assertions provide adequate coverage of the HLRs. • To justify full credit the tool must be configured as per the TOR.
A-4/2	Low-level requirements are accurate and consistent.	6.3.2.b	Partial	JPF-Core's compiliation and execution of the the Java code representing the LLRs checks internal consistency of the requirements. (see FM6.3 b & c)	• Additional testing/inspection must be performed to verify the accuracy of the LLRs with respect to the HLRs, for example when JPF cannot positively verify an assertion. • To justify full credit the tool must be configured as per the TOR.
A-4/4	Low-level requirements are verifiable.	6.3.2.d	Partial	LLRs that can be expressed as Java code are verifiable in that they can be tested by JPF. "Life cycle data items that are formally expressed can be checked for consistency (the absence of conflicts). If a set of formal statements is found to be logically inconsistent, then the inconsistencies should be addressed before any subsequent analysis is conducted." (FM6.3 d)	• Additional activities must address the verifiability of LLRs which cannot be expressed as Java source code. • To justify full credit the tool must be configured as per the TOR.
A-4/7	Algorithms are accurate.	6.3.2.g	Partial	LLRs represented as Java code capture the algorithmic behavior of the design to be implemented in flight source code. JPF-Core exhaustively covers the execution of the LLRs representing these algorithms and verifies the absence of common errors in the implementation of algorithms such as loop conditions and logic errors, such as: • Deadlocks • Race Conditions • Application-Specific Assertions "If low-level requirements are formally modeled, then they can be checked for accuracy and behavior by using formal analysis." (FM.6.3.2 g)	None.

Figure 2: Certification Credit Matrix for requirements verification using Java PathFinder.

c.1 Maturity

The model-checking technology underlying JPF was developed over 35 years ago [4, 5] and is now embodied in a range of tools, including JPF [20, 16, 11], Spin [24, 12], NuSMV [21, 2, 3], Kind2 [14, 9], and Uppaal [25, 15].

One of the earliest applications of JPF was the after the fact verification of the Remote Agent intelligent satellite control system [10, 26], where it successfully detected a deadlock condition that had been encountered during the mission. JPF has also been used to detect a subtle error in the DEOS operating system developed by Honeywell for business aircraft [26, 7, 8, 22]. JPF has also been applied to check parts of the Shuttle Abort Flight Management (SAFM) developed as part of the Shuttle Cockpit Avionics Upgrade (CAU) program [17] and tested on seeded errors from Martian Rover autonomy software [1]. JPF has also been used on early prototypes of the Onboard Abort Executive (OAE) for the Crew Exploration Vehicle (CEV) for which it discovered a significant design error [23]. Recent work has extended JPF applications to UML modeling, using a workflow similar to the one defined here in which LLRs are represented in Java code for verification purposes [18].

c.2 Technical Background

JPF belongs to the class of model checkers sometimes called "execution-based model checker" [13] which were first described in 1997 [6]. 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 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 ListenerAdapter 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.

Discussion

Development-time checking vs. verification Model checkers such as JPF can be useful at various times in the software life cycle. These different use cases have different objectives and, accordingly, the tools incorporate configurable features specialized for these objectives. Although this case study is focused on the qualification of a model checker for use in the verification process, we describe some features of JPF that should only be used

during other phases of the software life cycle.
Consider the differences between "development-time checking", by which we mean the iterative development and debugging of software artifacts such as requirements, models, or source code, and "certification-time verification," meaning the software verification process described in DO-178C, Section 6.0. During development-time checking, the software engineer often expects to find problems in an artifact that is a work in progress. In some cases, the objective is to find as many bugs as possible during a fixed period of time during which the tool is applied. In that case, the tool could sacrifice completeness for speed or expend computational effort on heuristics designed to guide the search toward execution paths more likely to contain errors. At other times, a team may be tasked with repairing a certain defect or class of defects, while ignoring other known defects (which may have been assigned to other teams or deferred as less urgent). In that case, the tool might either mask certain error reports or continue through certain internal error conditions that would halt execution at certification time.
When software verification processes as defined by DO-178C, Section 6.0 are undertaken, model checkers should be configured for their most conservative behavior, *e.g.*, when depth-limits are exceeded, errors must be signaled and it must be impossible to mask the presence of software errors by misconfiguration of error reporting.
The following table captures the differences between development-time and verification-time behavior for a set of exemplar behaviors for a model checker.

Feature	Development-time	Verification-time
Depth Limit	Allowing depth-limit makes possible discovery of bugs on different execution branches.	Depth limit allows incomplete search.
Configurable Error Reporting	Allows focus on errors of particular interest.	Risks verification of code with errors.
Fail-first heuristics	Attempts to quickly find errors in code base.	May expend unnecessary effort computing heuristic when complete absence of errors is only success criterion.

d TQL proposed

DO-330

The TQL proposed for the tool and supporting justification. [DO330-10.1.1-d]

We propose to qualify JPF-Core-X to TQL-4. Formal methods tools, such as JPF-Core-X, fall under "Criteria 2" defined by DO-178C under Section 12.2.2.b for determining the Tool Qualification Level because they are generally used to prove properties, analyze source code, and generate

tests. As such, they do not produce output that can be part of the system software and insert errors. This means the TQL depends on whether the certification approach will use formal analysis results to eliminate or reduce verification processes beyond those automated by the tool or development processes for the system software. For example, a formal method result that division by zero is not possible could justify absence of protection mechanisms in the system. This would be a reduction in a development process. For systems requiring Software Level (AL) A or B, JPF-Core-X must be qualified with TQL-4.

e Tool source

DO-330

"Tool source (for example, in-house, COTS, third party)." [DO330-10.1.1-e]

Discussion

JPF's status as an open source product raises many interesting issues. However, because this project is concerned with challenges to qualifying model checkers more generally, discussion of these particular issues is currently out-of-scope.

JPF-Core-X is an instantiation of an open source product that was created by researchers at NASA Ames Research Center and is now supported by a community of developers.

f Stakeholders

DO-330

"The stakeholders involved in the tool qualification and their specific roles and responsibilities, including who is responsible for satisfying specific objectives." [DO330-10.1.1-f]

Discussion

There's nothing interesting with respect to model checkers about this section.

THIS SPACE INTENTIONALLY LEFT BLANK.

g Description of TOR, TOIP, and TOV processes

DO-330

"Description of the Tool Operational Requirements definition process (see 5.1), tool operation integration process (see 5.3), and tool operational V&V process (see 6.2) (or a reference to where these processes will be described)." [DO330-10.1.1-g]

The JPF Tool Operational Requirements is a separate document.

Discussion

Not applicable to these case studies. Installation is only used for tool deployment in certification programs.

h Operational environment

DO-330

"Description of the tool operational environment in which the tool will be used." [DO330-10.1.1-h]

Discussion

There's nothing interesting with respect to model checkers about this section.

THIS SPACE INTENTIONALLY LEFT BLANK.

i Previous Applications

DO-330

"If the tool qualification data is reused, identify previous applications of the tool, the strategy and justification for reuse, and any applicable re-qualification activities. In the case of a third party tool or a COTS tool, information about previous application can be provided by the supplier since it may not be available from the users of the tool. See sections 11.2 and 11.3 for reuse of previously qualified tools and COTS tools." [DO330-10.1.1-i]

Discussion

There's nothing interesting with respect to model checkers about this section.

THIS SPACE INTENTIONALLY LEFT BLANK.

j TQP

DO-330

Reference to the TQP, or if no TQP is generated (for TQL-5), reference the data to support tool qualification. [DO330-10.1.1-j]

The JPF-Core-X TQP is a separate document.

References

[1] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, W. Visser, and R. Washington, "Experimental Evaluation of Verification and Validation Tools on Martian Rover Software," *Form. Methods Syst. Des.*, vol. 25, no. 2-3, pp. 167–198, September 2004. URL: http://dx.doi.org/10.1023/B:FORM.0000040027.28662.a4, doi:10.1023/B:FORM.0000040027.28662.a4.

[2] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new Symbolic Model Verifier," in *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, N. Halbwachs and D. Peled, editors, number 1633 in Lecture Notes in Computer Science, pp. 495–499, Trento, Italy, July 1999, Springer.

[3] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *In Proceeding of International Conference on Computer-Aided Verification (CAV 2002)*, pp. 359–364. Springer, 2002.

[4] E. M. Clarke and E. A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic," in *Logic of Programs*, pp. 52–71, 1981.

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

[6] P. Godefroid, "Model Checking for Programming Languages Using VeriSoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pp. 174–186, New York, NY, USA, 1997, ACM. URL: http://doi.acm.org/10.1145/263699.263717, doi:10.1145/263699.263717.

[7] A. Groce and W. Visser, "Model Checking Java Programs Using Structural Heuristics," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pp. 12–21, New York, NY, USA, 2002, ACM. URL: http://doi.acm.org/10.1145/566172.566175, doi:10.1145/566172.566175.

[8] A. Groce and W. Visser, "Heuristics for model checking Java programs," *International Journal on Software Tools for Technology Transfer*, vol. 6, no. 4, pp. 260–276, 2004. URL: http://dx.doi.org/10.1007/s10009-003-0130-9, doi:10.1007/s10009-003-0130-9.

[9] G. Hagen and C. Tinelli, "Scaling up the formal verification of Lustre programs with SMT-based techniques," in *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (Portland, Oregon)*, A. Cimatti and R. Jones, editors, pp. 109–117. IEEE, 2008. URL: ftp://ftp.cs.uiowa.edu/pub/tinelli/papers/HagTin-FMCAD-08.pdf.

[10] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. L. White, "Formal Analysis of the Remote Agent Before and After Flight," in *Proc. of the 5th NASA Langley Formal Methods Workshop*, June 2000.

[11] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[12] G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[13] R. Jhala and R. Majumdar, "Software Model Checking," *ACM Computing Surveys*, vol. 41, no. 4, Article 21, , 2009.

[14] *Kind2 tool website*. http://kind2-mc.github.io/kind2/.

[15] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *Int. Journal on Software Tools for Technology Transfer*, vol. 1, no. 1–2, pp. 134–152, October 1997.

[16] F. Lerda and W. Visser, "Addressing Dynamic Issues of Program Model Checking," in *Proccedings of SPIN2001*, 2001.

[17] L. Z. Markosian, M. Mansouri-Samani, P. C. Mehlitz, and T. Pressburger, "Program Model Checking Using Design-for-Verification: NASA Flight Software Case Study," in *IEEE Aerospace Conference Proceedings*, April 2007.

[18] P. C. Mehlitz, "Trust Your Model - Verifying Aerospace System Models with Java Pathfinder," in *IEEE Aerospace Conference Proceedings*, 2008.

[19] 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.

[20] NASA. *Java PathFinder (JPF)*. http://babelfish.arc.nasa.gov/trac/jpf/wiki.

[21] *NuSMV tool website*. http://nusmv.fbk.eu/.

[22] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger, "Verification of Time Partitioning in the DEOS Scheduler Kernel," *Formal Methods in Systems Design Journal*, vol. 26, no. 2, , March 2005.

[23] T. T. Pressburger, M. Mansouri-Samani, P. C. Mehlitz, C. S. Pasareanu, L. Z. Markosian, J. J. Penix, G. P. Brat, and W. C. Visser, "Program Model Checking: A Practitioners Guide," Technical Report TM-2008-214577, NASA, 2008.

[24] Spin maintainers. *Spin tool website*. http://spinroot.com.

[25] UPPAAL *tool website*. http://www.uppaal.org.

[26] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model Checking Programs," *Automated Software Eng.*, vol. 10, no. 2, pp. 203–232, apr 2003.