SSAT DATCPI (FACTS)

Tool Qualification under DO-330 Case Studies

Deliverable 5
for
Contract NNL14AA07C

Contents

1 Introduction

This document reports on the results of the case study portion of our research effort that identifies qualification considerations for formal methods tools under DO-330 and DO-333 for use in certification within the DO-178C framework. For the case study portion of our work, we performed a notional tool qualification following DO-330 for two representative formal methods tools: "Tool A", Java PathFinder/Symbolic PathFinder for model checking, and "Tool B", CodeHawk for static analysis/abstract interpretation.

After reviewing the full set of objectives in DO-178C and DO-330 required for an actual qualification effort, we organized our case study around the DO-330 objectives tables, enumerating what subsets of these objectives would be in scope for the case studies, and what outputs we would need to satisfy them. The results of this effort are a selection of qualification documents that show how existing state-of-the-art formal methods tools may be qualified using a safety-driven approach toward qualification within the DO-330 framework.

Initially, we anticipated changes to the way the formal methods tools are developed, how they are used, what claims they can make and support, and how they are integrated into life cycle tool chains may have to be changed from current practices. Our high-level experience thus far is that qualification of formal methods tools differs little from qualification of any other tool used in the development of safety-critical systems. However, we did identify potential issues and areas of special concern, and have reported on those in our respective case study reports.

Regardless of any required changes, DO-330 qualification and use of formal methods tools for DO-178C certification has great potential benefits for safety-critical system software, and these benefits will offer compelling justification for the broader adoption of formal methods tools into software and system engineering life cycles.

The remainder of this document is structured as follows:

- Section 2 discusses the motivation behind our case study effort, and the rationale for selecting JPF and CodeHawk as representative formal methods tools.
- Section 3 discusses the complementary nature of the two tools, and how they might be used within the broader context of a tool chain used in certification under DO-178C.
- Section 4 describes the scope of the case studies, and enumerates the set of documents produced for each study. Note that each case study effort for both tools produced a set of documents, all of which are referenced from this overview, but are each independent, mimicking the collection of documents constructed for DO-330 qualification.
- Section 5 provides an overview of the main issues and points of interest that we identified when writing the various case study documents.
- Section 6 provides some commentary on the ability of the case study tools to meet DO-330 qualification objectives that were previously identified in our case study preparation document.

- Section 7 provides a brief discussion of the FMEA analysis that was performed for each tool.
- Finally, Section 8 offers a summary of conclusions.

2 Motivation for Tool Qualification Case Studies

To better understand issues that may be encountered when formal methods tools are qualified for specific applications under DO-178C, DO-330, and DO-333, we have undertaken two Tool Qualification case studies for representative formal methods tools. One of these, Java PathFinder (JPF) is a model checker which examines reachability properties and assertions in algorithms and requirements expressed in the Java programming language. The second, the CodeHawk C Analyzer, is a standalone static analysis tool for proving (or disproving) memory safety properties of C programs via abstract interpretation. This exercise was intended to reveal issues that would become apparent during an actual qualification exercise, but might otherwise be hidden from a top-level analysis of the tools and the process. For example, during the qualification exercise, it became clear that JPF has many features suitable to an iterative test and repair approach to development, but that those features should be treated carefully when JPF is used to contribute results to a final certification for a finished software product. This issue came to light during the case study and inspired recommendations related to error reporting and internal tool limits (*e.g.*, search depth limits).

Case studies are valuable in that they have the potential to identify more issues in tool combinations and potential mitigations. For this effort, we designed our case studies to apply appropriate limitations on software design and tool usage, and to utilize patterns of argumentation for qualification and safety case support. Using this approach, we wrote the case study documents to illustrate techniques for satisfying the tool qualification objectives.

3 Toolchain Context

As shown in Figure 1, the two formal methods tools selected for our Tool Qualification Case Studies serve complementary roles in the software certification process. JPF will verify the correctness of LLRs encoded in Java with respect to HLRs. CodeHawk will validate that the source C code used to produce the object code is free of memory safety issues.

As discussed in other documents, the use of tools to automate verification activities has direct impact on the workflow to satisfy certification objectives. For example, a verification tool may directly automate particular verification objectives/activities specified in DO-178C, and can be used to claim certification credit for those. Additionally, the use of a tool (e.g. JPF) for one objective may allow one to skip other objectives or may establish certain properties/constraints to be true such that a subsequent tool (e.g. CodeHawk) might be used in a more flexible way, taking advantage of those assumptions. An extended discussion of this may be found in the FMEA portion of our results.

However, for our case study, each tool is treated is separately and stand-alone. As such, we developed two independent sets of case study documents: one set for tool A (JPF), and a separate set for tool B (CodeHawk).

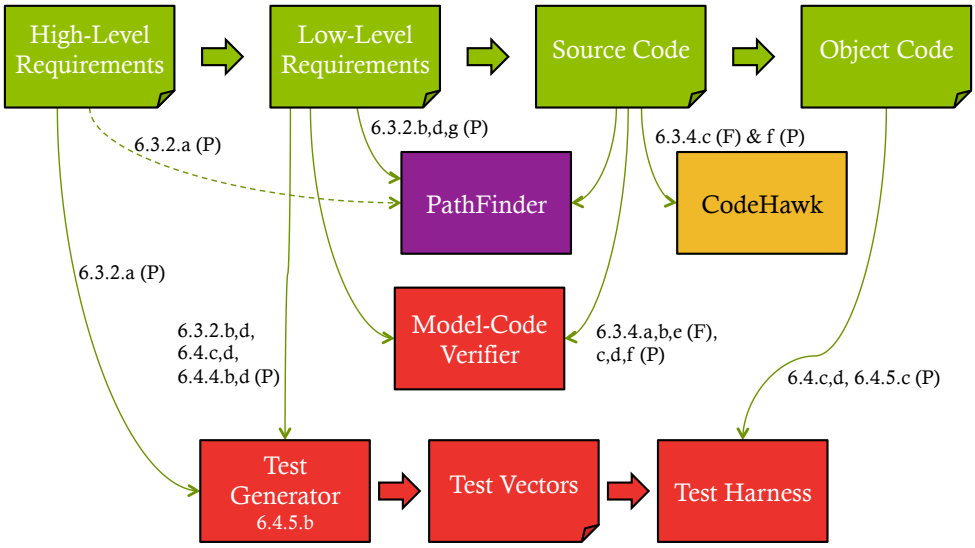

Figure 1: The case studies assume that JPF and CodeHawk play complementary roles in the development process. JPF will verify the correctness of LLRs encoded in Java with respect to HLRs. CodeHawk will validate that the source C code used to produce the object code is free of memory safety issues.

4 Scope of Case Studies

The scope of the case studies was focused on achieving the most insight into the actual tool qualification process for formal methods tools, while reflecting the limitations of the project. The most important restrictions were the absence of an actual flight software certification effort to inform the tool qualification, our position outside the tool development process, and the realities of project funding.

In practice, tool qualification under DO-178C is a collaboration between a tool developer and a tool user that is developing a flight software component, even if the tool developer and the tool user are part of the same organization. Since we did not have access to the tool developer for JPF, we

have limited our treatment of some of the documents associated with tool design and development. Also, since our work was not motivated by a specific flight software application, we have eliminated a number of specific elements which require reference to an overarching flight certification effort. For those aspects of qualification that we did complete, we sometimes limited ourselves to a few representative examples. For example, we tried to consider a few tool requirements to a useful depth, rather than cover the full breadth of requirements necessary for a full, real-world qualification.

For each tool we produced the following set of documents in accordance with the DO-178C regime, to a level of fidelity consistent with the lack of an actual software component to be certified and our funding:

- The tool-specific sections of the Plan for Software Aspects of Certification (PSAC), including DO-178C objectives, proposed product TQL, and the tools impact on the software lifecycle.
- The Tool Qualification Plan (TQP) sections related to the means of qualification compliance.
- The Tool & Tool Operational Requirements (TR and TOR) sections containing requirements that support the claimed objectives.
- A slim Verification & Validation Cases and Procedures document containing a few representative test cases traced to requirements.
- A similar sample of Verification & Validation Results tracing test execution results to the test procedures we defined.

We did not produce the following documents which would be required by an actual tool qualification exercise associated with an actual flight software component subject to DO-178C:

- The Verification Plan, for which the relevant information was covered in the TQP and the Test Cases & Procedures documents.
- The Development, Configuration Management, and Quality Assurance Plans, for which the relevant objectives are not applicable to TQL 4.
- The Design Description including design standards were considered to be out of scope. We did, however, provide detailed a description of the tool architectures and algorithms in other documents.
- Source Code, for which the relevant objectives are not applicable to TQL 4 & 5.
- Configuration Management Records, Configuration Index, Problem Reports, Life Cycle Environment Configuration Index were deemed out of scope since they are only relevant to actual software deployments and do not have any unique impacts on formal methods tools.
- Quality Assurance Records, for which there are no relevant research objectives for the case study.

5 Issues Uncovered

As the case study process unfolded, we uncovered many issues related to the qualification of the two representative formal methods tools. The most in-depth treatment of each individual topic is embedded in the case study artifacts themselves. For example, in section c.1 of the JPF Tool Operational Requirements document, we include a lengthy discussion of "Development-time checking vs. verification". That discussion will not be repeated here, but it does provide background for the conclusion that some accommodation should be made for developers to take advantage of both qualified and *u*nqualified tool features during early stages of software development. In the JPF case study, there are over sixty separate discussions covering a range of categories, including subtleties in tool identification and configuration, problems with JPF documentation and testing, details in error handling, treatment of non-determinism, and the scalability of model-checking. Similarly, the "Tool Operational Verification and Validation Cases and Procedures Framework for CodeHawk C Analyzer (Tool B)" included in the CodeHawk case study includes extensive discussion of the relationship between the results of the recent "Gold Standard" project funded by the Department of Homeland Security and testing required for DO-330 tool qualification.

For JPF the issues can be grouped into four high-level categories:

- Tool identification and configuration: as the product of an active research project, JPF does not follow a release schedule or protocol. We identified a specific JPF "release" for treatment by the case study by reference to a date and source code repository commit tag. It can also be difficult to determine which modules associated with JPF are active and well-maintained. (Further discussion can be found in Section A of "JPF Core: Tool Qualification Plan (TQP)".)

- Documentation: JPF documentation is in the form of an on-line wiki which covers many aspects of the JPF research endeavor beyond the core functionality that would be qualified. (Further discussed in Section G of "JPF Core: Tool Operational Requirements (TOR)".)

- Testing: Although JPF does have a set of tests, the JPF test suite seems ad hoc, including full system tests providing uneven coverage of representative program flaws. It does not include unit tests. (Further discussed in Section B of "JPF Core: Tool Verification Cases and Procedures (TVCAP)".)

- Error handling: our case study discusses at length the necessity for clear error messages to avoid operator error during the certification process. (Further discussed in Section F of "JPF Core: Tool Requirements (TR)".)

- Scalability: although model checking techniques constantly improve (for example, by incorporating symbolic execution techniques), an exhaustive check of all software aboard a modern airliner is beyond the capabilities of current model checking tools. To compensate, organizations using model checkers must carefully scope the system under test, possibly by checking abstractions of the final executable (such as low level requirements) or critical subsystems. (Further discussed in Section C.2 of "JPF Core: Tool Operational Requirements (TOR)".)

The CodeHawk case study discusses similar issues. The CodeHawk C Analyzer that is the object of the case study was produced by Kestrel Technology as part of a DHS research project. It is not a publicly released product, and has no user documentation at present. Also, because it is a research implementation, it does not yet have an externally identified accounting of error messages. There are also some issues of possible interest to the application of formal methods tools to DO-330 certified software treated only in the CodeHawk study:

- Third-party tools: CodeHawk depends on the third-party tools gcc [3] and CIL [2, 1] which introduces the issues of identification and configuration of these tools, as well as the possible need to conduct similar qualification exercises for the tools. (Further discussed in Section C.7 of "Tool Qualification Plan (TQP) Framework for CodeHawk C Analyzer (Tool B)".)
- Coverage of library code: like JPF and other formal methods tools that process programming language source, CodeHawk requires extensions to handle library calls for system services for which source code may not be available for analyses. In the case of CodeHawk, there may be the need to extend the current catalog of "functional summaries" to cover libraries of interest to flight critical applications. (Further discussed in Section C.3 of "Tool Qualification Plan (TQP) Framework for CodeHawk C Analyzer (Tool B)".)

6 Tool Qualification Objectives

As discussed in our case study preparation document, we structured our simulated case studies around the DO-330 objectives tables. Section 3 of the case study preparation document enumerates the subsets of DO-330 objectives that we found to be in scope for this effort, and what outputs we would need to satisfy them.

Here, we analyze the results of the case studies to assess how the objectives in DO-330 Tables T0-T10 are addressed for each tool. Building on this assessment, we attempt to identify areas of outstanding need where there may be gaps or inadequacies in meeting the DO-330 objectives.

Of the 11 different tables of objectives, we found that only 7 of them contained objectives that were applicable to our case studies. We found that, through our case study document preparation, we were able to satisfy most of the objectives. In the subsections that follow, we provide some discussion for those objectives where we identified potential gaps or complexities. Here, a "gap" indicates that the outcome (or output) of the case study documents was found to be insufficient for supporting the corresponding objective. A "complexity", on the other hand, suggests that some characteristics of the tool(s) make it particularly challenging to satisfy the objective.

6.1 Table T-0, Tool Operational Process

Obj. 4 Tool Operational Requirements are complete, accurate, verifiable, and consistent.

For any tool, the traceability between TOR, test cases and test results is required to ensure if the TOR are verifiable, consistent and accurate. For the case of JPF, there were very

few existing test cases to draw from, and no corresponding requirements to trace those test results to. We therefore attempted to first develop an appropriate set of tool operational requirements, and then define a suite of test cases to map to those requirements. While we only covered a subset of the full functionality of the tool (and therefore a subset of the needed test cases), this process could be extended naturally to show full traceability.

Obj. 5 Tool operation complies with the Tool Operational Requirements.
In a full qualification effort, the tests that demonstrate this compliance would be captured in the tool accomplishment summary. If deviations of test results from the expected outputs are found, explanations of these test failures would be analyzed to determine whether they would cause soundness issues or difficulty in using the tool. This is true for any tool being qualified, though. We found nothing to suggest a gap or complexity in meeting this objective with the subject formal methods tools.

Obj. 6 Tool Operational Requirements are sufficient and correct.
The validation of the Tool Operational Requirements critically depends on the role of the tool within the software life cycle and the objectives for the tool described in the PSAC. Within the limits of the scope of the case studies, we did validate the requirements documents with respect to the role of the tools we defined in the software life cycle. We do not believe this aspect of the DO-330 process presents any unique challenges to formal methods tools.

6.2 Table T-1, Tool Planning Process

Obj. 4 Additional considerations are addressed.
Complexity: Determining whether a tool is a suitable alternate tool to improve confidence may be challenging. For instance, although JPF and SPF might look like suitable redundant tools, the common code between them makes them unsuitable to use one to improve confidence on the other.

6.3 Table T-2, Tool Development Process

Obj. 1 Tool requirements are developed.
Complexity: Many formal tools are research based and hence they may lack adequate documentation. In many cases, the research paper explaining high level implementation detail might be the only available input. In the case of JPF, all of the tool requirements had to be written independently for this case study, using the online documentation and the source code as a guide. For CodeHawk, we started with a well-defined baseline set of documentation, including requirements. However, we did have to adapt this documentation to fit the scope and context of DO-330 objectives for qualification.

Obj. 2 Derived tool requirements are defined.
Complexity: Without a detailed design of the tool, it may not be feasible to determine whether a requirement is derived or not. Again, since formal tools are research based it might lack such detailed documentation.

6.4 Table T-3, Verification of Outputs of Tool Requirements Process

Obj. 4 Tool Requirements define the behavior of the tool in response to error conditions.

Complexity: While documentation of formal tools to some extent provides reasonably detailed explanation to functional requirements, it can be challenging to adequately define and/or meet all error reporting requirements. For example, one common behavior observed in many formal tools is that they might take a very long time to run to completion. In such cases, it may not be possible for the tool to provide useful / required outputs within a reasonable amount of time.

Obj. 5 Tool Requirements define user instructions and error messages.

See Obj. 4.

Obj. 6 Tool requirements are verifiable.

Complexity: In general, many of the tool requirements are directly verifiable. One particular challenge for formal methods tools, however, is the fact that some requirements may be impractical to verify for large and/or complex input sets. For example, we can develop test cases for JPF that show the tool properly detecting property violations (such as a deadlock) for a relatively small / simple code-base. If the requirement is to detect the presence of a deadlock for *any* system under test, then we would have to consider arbitrarily large / complex code bases during testing. Because the code base can scale in this way, it is unclear how to adequately develop test cases that can be used to verify a such a requirement.

Obj. 9 Algorithms are accurate.

Gap: Ensuring if the formal method was correctly implemented might require exhaustive testing, which might be infeasible. At most, one can argue that it has been reasonably well tested. Even the gold standard test suite used for CodeHawk provides only a certain level of confidence on the implementation.

6.5 Table T-6, Testing of Outputs of Integration Process

Obj. 1 Tool Executable Object Code complies with Tool Requirements.

Gap: Guaranteeing that a single test case passed for a requirement may not be an indicator that the requirement will be met in *all* cases. For instance, if a test on JPF returns "no deadlock" on a simple program, it is not a guarantee that JPF is capable of reporting "no deadlock" correctly on all other (more complex) programs too.

Obj. 2 Tool Executable Object Code is robust with Tool Requirements.

Gap: The tool cannot be tested exhaustively with *all possible* inputs. This practical limitation applies to testing for all tools, not just formal methods tools. Robustness can only be demonstrated by proving that the tool continues to correctly identify failures modes through more testing, with diverse input sets.

Complexity: Determining failure modes systematically and robustness criterion for formal tools may require in-depth knowledge of the tool implementation as well as the underlying formal methods. For instance, CodeHawk uses abstract interpretation and the tool is qualified for 4 abstract domains. Ensuring that the tool's output correctly reports results if applications where those abstract domains were not suitable needs a detailed analysis that may be time consuming. In the case of JPF, exhaustively testing applications with different depths may be practically very expensive to achieve.

6.6 Table T-7, Verification of Outputs of Tool Testing

Obj. 2 Test results are correct and discrepancies explained.

Gap: In formal tools, evaluating the correctness of test results may involve more than looking for pass/fail of the tests. While almost all tools provide reasons for failed tests, most tools do not provide detailed information on why and how tests passed. This is particularly a problem in formal tools. For instance, vacuity is a problem in many model checking tools. Hence, when JPF, or any model checker for that matter, reports that there are no property violations, establishing that the result was non-trivially satisfied may be necessary to trust the results. The problem of ensuring that assertions and (temporal logic) expressions are properly formed is equivalent to the effort involved in validating the requirements for a complex system. Hence, in addition to explanation to FAILED tests, the passed tests may require additional explanation. On the other hand, CodeHawk used a gold standard test suite whose oracle was manually determined. In this case study, the correctness of the oracle was only assumed and not validated. The successful proofs generated from CodeHawk were not checked for correctness. These might induce additional risk in trusting the outputs. The Juliet test suite for CodeHawk has a trusted oracle, but they are simple tests and do not adequately test all features of the tool.

Complexity: Developing adequate test cases with correct oracles when the tools do not have one already will be a challenging task. For instance, JPF did not have an existing test suite. As a result, only a portion of the full required set of test cases were designed and presented.

Obj. 3 Test coverage of Tool Requirements is achieved.

Gap:

Complexity: The Juliet test suite for CodeHawk had some direct traceability to checking some requirements, but because they were very simple, they may not (by themselves) adequately establish the absence of errors in the tool's output. The gold standard test suite for CodeHawk, while not tied to tool requirements explicitly, is expected to check some critical features of the tool In general, additional testing with complex inputs increases the confidence on the tool output and performance, and appears to be a good way to supplement the more basic tests that tend to be used for test coverage.

6.7 Table T-10, Certification Liaison Process

Obj. 4 Impact of known problems on the Tool Operation Requirements is identified and analyzed.

Complexity: JPF and CodeHawk, like most formal methods tools, are primarily used for research. Consequently, known problems with the Tool Operational Requirements are not easily identified or tracked. In general, research-based tools tend to be under active development and may not have sufficiently detailed documentation of error reports or list of known issues. Also, for tools that have not been in use for a long time or that have not been used on a broad range of systems, one may not be able to sufficiently capture this information and may lead to mis-placed confidence on the tool output. This is especially a challenge for JPF, which does not have a readily available regression test suite.

7 Failure Modes and Effects Analysis

We conducted independent failure modes and effects analyses (FMEA) on both CodeHawk and JPF as well as the composed tool chain. These are provided separately as an Excel spreadsheet with tabs that show the tool chain and DO-178C credits claimed by each tool, the tool chain FMEA, and finally the individual tool FMEAs. For each tool, we composed the FMEA by addressing the following questions:

Potential Failure Mode What might go wrong?

Potential Failure Effects What is the impact on the end-user?

Severity How severe is this failure? We use a scale of 1–10, with 10 being the highest severity with a potential safety effect.

Potential Causes What are the root causes of the failure mode?

Occurrence How likely is it that this root cause(s) will occur? We again use a 1–10 scale, where 10 is the highest likelihood.

Current Design Controls Prevention What are the existing processes that prevent either the cause or the failure mode before it can impact certification?

Current Design Controls Detection What are the existing processes that detect either the cause or the failure mode to assure it is properly handled during certification?

Detection How easy or difficult is it to detect the failure? We use a 1–10 scale with 10 being very difficult or not possible to detect.

Upon answering these questions for each identified failure mode, we provide one or more recommended actions to mitigate the risk. We then use the severity, occurrence, and detection scores

to compute an aggregate "risk priority number" (RPN). The RPN is the product of these three component scores. As each component has a range of 1–10, the RPN has a range of 1–1000. A low number suggests low risk, while a high number suggests great risk. For each identified failure mode, A potential key characteristic (pKC) is flagged if the severity falls between 5–8, and if the occurrence is 4 or higher. A *critical pKC* is flagged if the severity is 9–10, with an occurrence of 4 or higher. These characteristics must be addressed in the tool chain composition with a case made in the Plan for Software Aspects of Certification (PSAC) to justify how these problems will be avoided or identified and worked around in the final certification.

7.1 FMEA Results for JPF

Most of the critical considerations we identified relate to user errors, such as insufficient depth setting or other configuration errors or incorrect assertions that do not match the intended requirements. These usage errors are unavoidable in tool design and qualification. The risk mitigation report looks at life cycle enhancements, namely verification of intermediate artifacts (in this case, the assertions) and input/output checkers that would include checking the options used in a run.

The remaining concerns are considered unlikely because the qualification process should rule them out. Errors in the search or backtracking implementation that may cause JPF to miss an execution path are notable because they are highly severe and undetectable by users. Likewise, the model classes used as surrogates for native calls should be subject to the qualification process and are unlikely to go undetected. Not only is the likelihood low that an undetected error would manifest in certification, the odds that the only errors in the system are in unexplored paths or improperly abstracted native calls are even lower.

7.2 FMEA Results for CodeHawk

The CodeHawk FMEA also shows some usage errors. These are predominantly related to the execution of multiple scripts, and the artifacts each develops. Again, there are life cycle enhancements that can remedy these relatively easily. The execution of the scripts should be automated. This is not necessarily as trivial as executing each in the proper sequence; but all software can be run through the same script, so the benefits of constructing the script should easily outweigh the cost.

There are more specific possible implementation errors listed for CodeHawk than JPF, but that is probably related to having the developer on the team. Kestrel is obviously intimately familiar with the design and implementation as well as the areas where coding errors could undermine the formal method most severely or undetectably. The qualification process should make it unlikely that there are such errors, and even more unlikely that they should manifest for any certifiable system. The risk mitigation strategies report recommends ways to minimize the impact of such errors on certification.

There is a medium-priority failure mode relating to long runtime/non-termination. Where JPF has a depth option to terminate the run perhaps before the search is complete, CodeHawk's runtime is a function of the precision of the domains and the complexity of the system. Fine-grained domains

can result in a state-space explosion that can cause it to run out of memory or run for weeks. This is not a safety issue as long a care is taken to check that old outputs are not mistaken for fresh results.

7.3 FMEA Results for Tool Chain

The Tool Chain FMEA shows that the key critical concerns can be mitigated by lowering their likelihood or raising the likelihood of detection. For example, with two tools cross-checking each other, the likelihood of unsound results on the same system regarding the same property remain catastrophic and impossible to detect, but also extremely unlikely. Disagreement (one tool produces no result) or contradiction (where a property is proven in one tool while another says it doesn't hold) are much more likely and trivial to detect.

The highest risk priority score is actually for an unmitigated tool. The next scores relate to usage: intermediate representation errors, which are addressed in the life cycle section of the risk mitigation report, and certification credits claimed in error, which is addressed in the assurance case section of the risk mitigation report. Great care is needed in determining what actions need to be taken and verifying that the tools comprising the tool chain do indeed perform all those actions, nothing will generally point to an activity that wasn't performed.

8 Conclusions

Considering the full set of issues encountered while performing the case studies, we can synthesize a set of conclusions that can inform future real world formal method tools qualification efforts.

8.1 Hurdles to the Adoption of Research Tools

Qualification of a research tool requires an initial effort to:

- Define the precise tool to be qualified.
- Limit the tools scope to just those features needed for qualification.
- Provide appropriate documentation and tests.

8.2 Importance of Error Handling

Error Handling in a testing tool is extremely important.

- Make extra effort to ensure test results are interpreted correctly.
- Provide procedures to help ensure test cases are valid.

8.3 Treatment of Incomplete Results

Users must consider how to handle incomplete results when model checkers dont scale to their models.

- Use complementary analyses to cover state space.
- Qualify techniques for automatically abstracting state space.
- Reformulate models at more abstract level.

8.4 Use of Unqualified Tool Capabilities

Developers should be able to use unqualified features for development activities outside the certification process. Should qualification process impose requirements to eliminate confusion?

8.5 Dependence on Third Party Tools

Formal methods tools are inherently complex systems. Those that analyze source code must parse, interpret, and possibly simulate the execution of that source. To do so, tool developers can use existing tools with reputable pedigrees, but many of these tools have not been qualified. These tools generally offer many capabilities beyond those necessary to support the formal methods tool. To facilitate their use for qualified tools, strategies should be developed to qualify only the limited set of capabilities required by the tool.

References

[1] CIL Developers. *CIL (C Intermediate Language)*. https://sourceforge.net/projects/cil/.

[2] CIL Team. *CIL - Infrastructure for C Program Analysis and Transformation (v. 1.3.7)*. https://people.eecs.berkeley.edu/~necula/cil/.

[3] Free Software Foundation. *GCC, the GNU Compiler Collection*. https://gcc.gnu.org/.