SSAT DATCPI (FACTS)

Risk Mitigation Strategies Report

Deliverable 6
for
Contract NNL14AA07C

Contents

1 Introduction **2**

2 Redundant Tools **4**
2.1 Similar Redundant Tools 5
2.1.1 Multiple back-end SMT solvers 7
2.1.1.1 SMT Solver compatibility via SMT-LIB 7
2.2 Dissimilar Redundant Tools 9
2.2.1 Static analysis on the model and source code. 10

3 Tool Self-Tests and Benchmark Propagation **11**
3.1 Tool Self Tests 11
3.1.1 Example: Deadlock with Varying Number of Threads 12
3.1.1.1 Inputs 12
3.1.1.2 Expected Results 13
3.1.1.3 Use of Redundant Tools to Supplement Test Failures 13
3.2 Benchmark Propagation 13

4 Tool Life Cycle Enhancements **16**
4.1 Tool Input/Output/Execution Checking and Proof Checking Approaches . . 16
4.1.1 Employing Input/Output and Execution Checkers 17
4.1.2 Use of Proof Checkers 19
4.2 Tool Life Cycle and Qualification Considerations 21

5 Safety Case Argument Templates **22**

6 Conclusion **27**

1 Introduction

While the use of formal tools has gained significant presence during the early development phases of avionic software, to claim that the software is indeed safe and works as expected to the Federal Aviation Administration (FAA) using the results from such tools is a challenge since one has to establish trustworthiness in those tools. According to the FAA, the trustworthiness of a tool can be established by adequately satisfying all the objectives listed in a qualification supplement guidance document (DO-330). Unfortunately, when it comes to qualifying formal tools there is a lack of intellectual clarity since formal tools make use of mathematically rigorous analyses and typically include sophisticated heuristic search algorithms that pose unique qualification challenges and considerations in satisfying all the DO-330 objectives. In this context, we explored the risks and challenges associated with qualification using two formal verification tools as case examples, namely Java PathFinder (JPF)—a model checking tool for JAVA and CodeHawk—a static analyzer for C. In this report, we present various mitigation strategies that are not explicitly suggested or described in DO-330 to address risks such as adequately identifying and mitigating tool failures that may hinder establishing trustworthiness of formal tools.

We broadly classify all possible failures that can occur in verification tools into three categories: 1) false negatives, 2) false positives, and 3) no result. False positives occur when the tool incorrectly reports a safety violation in the software, and false negatives occur when the tool fails to detect the presence of a safety violation. Errors such as memory violation or a segmentation fault may prevent the tool from completing its analysis, producing no output. Though false positive and no result errors are costly in terms of the additional time and resources required to resolve the issue, they are generally considered acceptable failures from a safety point of view because net impact is greater conservatism. False negatives, on the other hand, are fundamentally unacceptable. If a false negative were to occur in a tool used in the certification process, it would risk the possibility of faulty, unsafe software quietly passing through certification undetected.

Most verification tools propose to find the defects they can find, absence of evidence is not evidence of absence. Formal Methods tools, on the other hand, claim a much more ambitious task—to prove the absence of defects. When JPF finds no deadlocks, the code is proven deadlock free; and when CodeHawk satisfies all its proof obligations, the code is proven memory safe. The risk of an incorrect proof, however it is generated, is therefore much higher. The causes are often not inherent in the tool. Overly abstract formal models, improperly formalized requirements, and unverified assumptions all undermine sound reasoning. That said, the decision to use formal methods tools in certification must start with assurance that the tool is sound when the inputs are correct and the output is properly interpreted. Unfortunately, it is not feasible to exhaustively test any tool, so there will always be some doubt in the veracity of its output.

It is therefore extremely important to identify reliable approaches to mitigate the risk of using a tool to support certification. One way to mitigate the risk associated with a single

tool failing is to introduce redundancy. The use of multiple different tools to accomplish the same verification task increases confidence when the tools agree, and helps to identify specific tool misuse or internal failures when they do not agree. Redundant tool sets can come in a variety of forms. In general, for two tools to be redundant, they are used to accomplish the same task. In Section 2 we explore the use of *dissimilar redundant tools* that use different methodologies to identify the same concerns, as well as *similar redundant tools* that independently implement the same fundamental approach.

Multiple tools are often assembled into a tool chain to break up the verification task into smaller and simpler sub-tasks, each performed by a separate tool. In this case they are not cross-checking each other as redundant tools, they must work together to complete an assurance argument. When considering a tool chain we can extend the notion of tests and benchmarks to apply to the full chain as opposed to just the individual tools. To this end, in Section 3, we describe tool self-tests and the propagation of benchmarks through a notional tool chain.

In Section 4, we assess the effectiveness of our proposed risk mitigation strategies in addressing the Tool Life Cycle Impacts. A class of soundness issues occur due to users providing inputs that are not within the tools' operational scope and incorrectly interpreting the outputs from tools. While this is partly mitigated by using dissimilar tools that require inputs and produce outputs in different forms, we propose the use of specialized tools, that we call *input and output checkers*, to augment the tools. In addition, we also propose the use of tools that use artifacts developed at various stages of the software life cycle in order to provide the necessary confidence in the overall properties of software.

Finally, in Section 5, we present a sample *assurance case* that argues the satisfaction the DO-330 objectives to qualify tools and tool chains. An assurance case is a structured argument that systematically establishes certain claims about the system in a certain context through adequate evidence and justifications. An assurance case, although not categorized as a mitigation strategy, is instrumental in finding areas of weakness in the way the tools are used and perceived to establish certification credits. While there are many notations to capture safety cases, we have chosen the domain specific language, **L**, from the CertWare II Safety Case Workbench [5] since it appears to be the most appropriate formalism for capturing "templates or patterns" of these augmented arguments.

2 Redundant Tools

Section 11.5 of DO-330 lists "dissimilar tools" as an alternative method for qualification. We propose this approach as a strategy in assembling a tool chain to mitigate possible errors in tools or their usage. In this scenario two or more tools are used that each individually claim the same certification objective to increase the confidence in the result. We call such tools *redundant tools*, and we consider two categories: *similar redundant tools* that have similar or the same input artifacts and underlying formal methods and *dissimilar redundant tools* that approach the objective with different methods and inputs. For instance, model checking tools that typically have similar search strategies can be considered similar redundant; while model checkers, static analyzers, and testing tools are considered dissimilar.

The primary reason to categorize these tools is to analyze the difference in level of confidence the combinations of tools can provide in the correctness of the results. While we discuss the details of each tool category in detail in the sections below, we found that agreement in results from dissimilar tools yields higher overall confidence than agreement in similar tool results. For one, it is less likely that two different tools with different approaches have a common mode of failure. In addition, the different tools and methods tend to involve different inputs, which creates an opportunity for cross-verification simply in the construction of their respective input artifacts. Similar tools, on the other hand, are likely to have similar limitations and failures and hence may result in a medium level confidence, as illustrated in Figure 1. It is also important to consider the difference in costs in setting up, re-running verification, and comparing results in similar versus dissimilar arrangements.

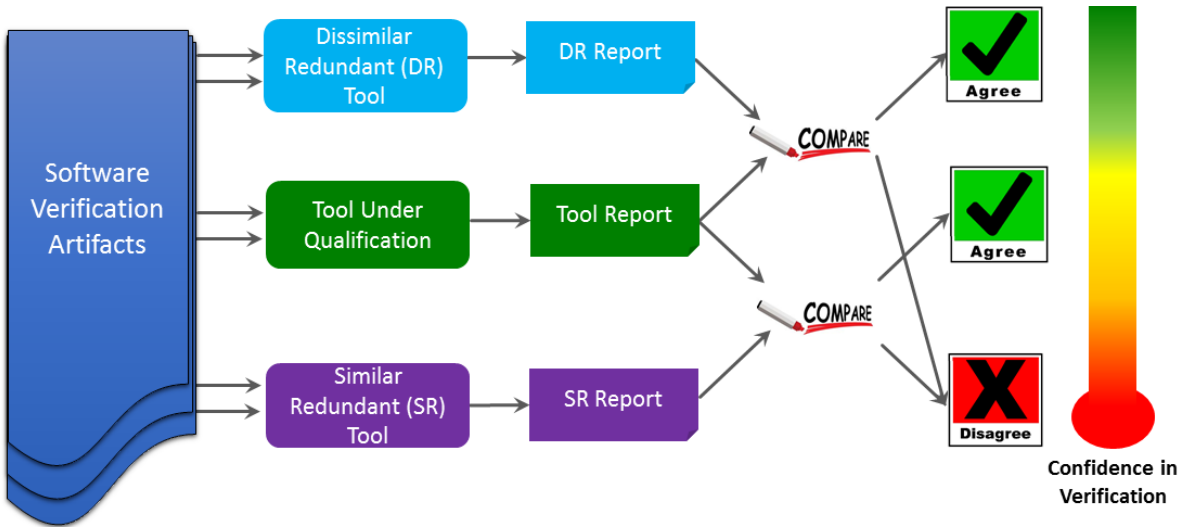

Figure 1: Confidence Levels in Verification Results using Redundant Tools

2.1 Similar Redundant Tools

Capability or feature of a tool is *similar redundant* to a capability of the tool under qualification, if both tools:

- accept the same inputs such as tools accepting source code written in the same programming language.
- produce similar outputs (i.e., the tools' outputs answer the same question) such as tools detecting deadlocks in source code.
- design and implementation are not vastly different such as model checkers that have similar search strategy to explore the state space of the input.
- are sufficiently independent from the another, i.e. one tool does not depend upon the other to produce results.
- are qualifiable for that capability

The use of similar redundant tools have several benefits. First of all, it is much cheaper and easier to use than an dissimilar tool. Since they are closer in design and implementation setting up the verification environment, running the tool, inferring results and qualifying the tool (if necessary) will be similar to the main tool. Since we consider tools that are sufficiently independent in producing outputs, soundness issues in the results of the main tool can be validated with the results produced by the redundant tool. However, on the flip side, they may have common limitations and failure modes that may result in no additional information provided by the redundancy. In the rest of this section, we elaborate different ways similar redundant tools can be used for CodeHawk, JPF and its extensions.

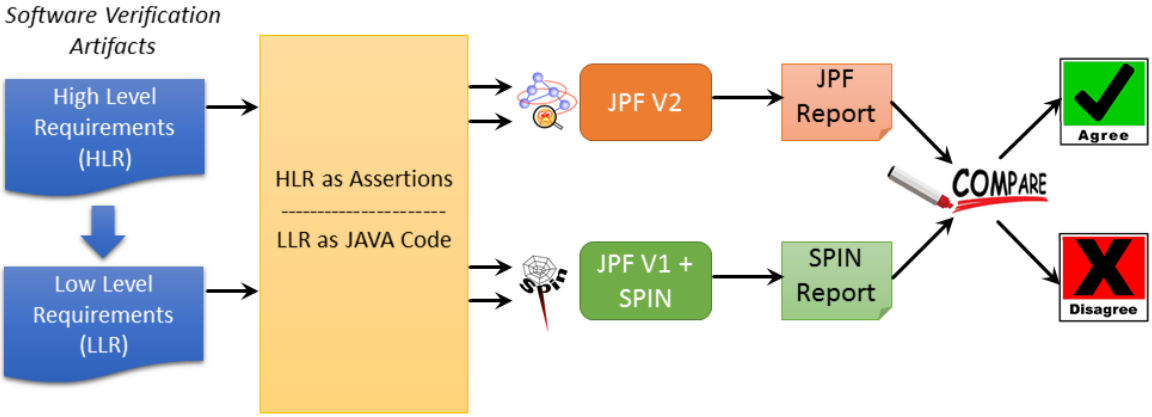

Figure 2: Similar redundancy for JPF

Figure 2 illustrates the way in which we examined redundancy for JPF V2 (the tool under qualification)[1]. The verification artifacts (the blue boxes on the left side of the figure) are some of the major verification artifacts outlined in software verification process in DO-330 (see DO-330, Figure FM.6-1, Level A Software Verification Processes).

We evaluated the use of JPF V1 [10] and SPIN [11] model checker together as a similar redundant tool for JPF V2. SPIN is a model checker that is capable of efficiently identifying deadlocks, race conditions and certain classes of property violations by exploring all paths of the source program using an optimized depth-first graph traversal method. JPF V1, the earlier version of JPF, translates a given Java program into a PROMELA model, the programming language for SPIN model checker, to be model checked using the SPIN tool. In order to deal with large or infinite state spaces, JPF V2 was custom developed to detect deadlock, race condition and property violations without having to use SPIN. JPF V2 includes a model checking engine that calls a custom Java Virtual Machine to interpret bytecode generated by a Java compiler. Both JPF V2 and V1 are explicit state model checkers and search all paths in the state space using depth first search technique. Considering all these similarities yet independence between the two tools, we believe that JPF V1 + SPIN can be used to improve confidence on results produced by JPF V2.

We also examined the applicability of Symbolic PathFinder (SPF) [14] as another similar redundant tool for JPF. SPF also model checks programs but on symbolic inputs representing multiple concrete inputs that helps address the state space explosion problem of explicit state model checkers such as JPF V2. While it may appear that SPF and JPF V2 are similar redundant, SPF internally uses the analysis engine of the JPF V2 inducing dependency and increasing the chance of common failures occurring in both their outputs. While SPF can be used to explore state spaces that are practically inefficient or infeasible for JPF V2 to search, they can not be used to increase confidence in each other's outputs.

For CodeHawk, we considered the use of Coverity Static Analysis tool for C [1]. Coverity is a commercial tool that applies static analysis techniques to identify some of the memory related issues in C code, that are listed as Common Weakness Enumeration [13]. Since CodeHawk has similar capabilities (statically analyses source code) and is sufficiently independent from Coverity's tool, the results from both tools executed on the same source code to check the same objectives can be used to validate each others' outputs.

The commonality with the tool variations is that, because they are based on similar approaches, they can be used to perform the same types of verification tasks. In this sense, they can be used concurrently as a means of providing redundancy (and therefore greater conservatism) in the overall verification process. As discussed previously, the introduction of redundancy is motivated by the need to mitigate the risk of relying too heavily on any single tool as well as cross-checking the soundness of the results. It is also important, though, to consider the differences that exist between each of the alternate tools, and the associated

[1]To differentiate the earlier version from JPF from the current version, we explicitly specify the version number only in this section. However, in the rest of the document wherever we mention JPF we refer to its current version, i.e. V2

impacts that those differences may have on the broader verification process.

One of the key differences associated with the introduction of alternate tools is the expression of inputs and interpretation of outputs. For instance, although the both JPF versions accept LLR coded in JAVA, there are slight variations to the way HLR are specified in each tool. JPF V2 expects HLR encoded as assertions, whereas in JPF V1 accepts HLRs formalized using linear temporal logic notation. Although there is some guidance to help transform HLRs from one notation to the other, it is often a manual activity. Hence, care has to be taken to ensure that no errors are introduced in the process. Along the same lines, the outputs from these tools, although help answer the same question, must be interpreted by a human to conclude whether or not it was satisfied. Certainly, the reporting of a pass/fail condition can easily be expressed and interpreted, regardless of the presentation format. The full reports provided by the model checker, however, include much more detail, especially when violations are detected. Without standards to unify them, the presentation format of different tools' outputs are bound to take on significant differences. In general, redundant tools may use slightly different languages to express models, which creates an additional burden to ensure input equivalency and output equivalency between the tools.

2.1.1 Multiple back-end SMT solvers

Symbolic model checkers, such as SPF, frequently use off-the-shelf decision procedures or constraint solvers to verify satisfiability of properties. For instance, SPF uses the Choco (pure Java) for linear/non-linear integer/real constraints, IASolver (pure Java) the Brandeis Interval Arithmetic Constraint Solver and CVC3 for real and integer linear arithmetic and also for bit vector operations [2]. SPF also supports specialized solvers such as Green Solver, dReal and Coral. In addition to superior performance and reliability that is achieved by using these third party tools, one of the main advantages is being able to interchangeably use multiple solvers to gain higher level confidence in the verification results. Figure 3 illustrates the use of different SMT solvers and decision procedures that could be used as redundant tools to verify output of SPF.

2.1.1.1 SMT Solver compatibility via SMT-LIB The SMT-LIB project [4] offers a critical technology for the future use of redundant SMT solvers and increasing confidence in the correctness of available SMT solvers. To enable the comparison and evaluation of existing SMT solvers and active research projects, the SMT-LIB project has created key capabilities, including:

- a standard language for the input and output of SMT solvers,
- a formalism for the expression of each theory supported by a SMT solver, and
- a large collection of benchmark problems.

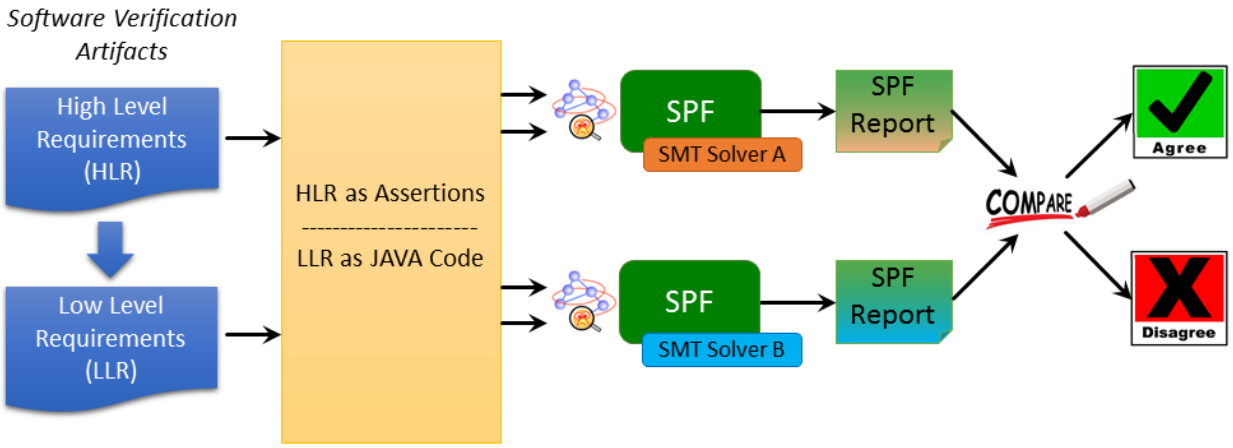

Figure 3: Similar redundancy for SPF

The combination of these capabilities aids the qualification of symbolic model checkers in two important ways. First, by using the standards established by SMT-LIB to define the theories required by a model checker, its input queries, and the result provided by the SMT solver, a model checker can increase confidence in its overall solution by using multiple SMT solvers to check path conditions. This benefit comes at very little cost. There is a small cost in tool development overhead required to correctly handle the tasking of multiple SMT solvers and the comparison of their results. There is also a small runtime cost in CPU time to use multiple solvers. However, there is no algorithmic obstacle to parallelizing multiple SMT solvers, which would reduce the additional runtime cost to the difference between the maximum runtime of the multiple solvers and the runtime of the solver with the best expected performance.

The use of the SMT-LIB standards and benchmarks to test the performance of the chosen SMT solvers is the second source of increased confidence. Since 2005, the SMT Competition (SMT-COMP) has been held annually to encourage research advances in the implementation of SMT solvers. The benchmarks cover many combinations of a range of possible theories extending SAT (*e.g.*, a combination uninterpreted functions and nonlinear arithmetic). In particular, the benchmarks include logics of interest to model checking such as array and bit-vector logics.

2.2 Dissimilar Redundant Tools

A capability or feature of a tool is *dissimilar redundant* to a capability of another, if:

- the objective of using the tools is similar (i.e., the tools' outputs can be analyzed to infer the answer a common question),
- the underlying mathematical formalisms are distinct, such as model checkers and static analyzers,
- the tools are independent from the other (i.e., one tool does not depend upon the other to produce results), and
- both tools are qualifiable for that objective.

Using dissimilar redundant tools have several benefits. The results from a dissimilar tool determines the confidence level one could have in the main tool's output in a rigorous way, since they verify if an objective is met using distinct approaches. This dissimilarity reduces the chance of common mode failures among tools (both of them reporting an unsound result for the same objective). In fact, it is often the case that they are developed by different teams using different approaches and hence even subdued errors/limitations in one can be identified when using the other.

Introducing a dissimilar tool, however, may be expensive. The tools often have different verification environments and hence the setting up the environment and running the verification task adds up to the cost. Also, due to the dissimilarity among the tools, it is often the case that the inputs fed to the tools and output from the tools may slightly vary from one another requiring additional manual effort and expertise to prepare them as well as interpret them. In cases where the dissimilar tools' input and output may not exactly match, there is an overhead of ensuring the translation to the expected formats is accurate and sufficiently detailed.

Testing, in general, can be considered a dissimilar approach to increase confidence on formal verification results. By using test suites that specifically check the software for certain objectives, it is possible to cross check the results obtained from verification. While the exhaustiveness and guarantee provided by testing is much lesser than verification, their practicality and ease of use makes it one of the most commonly used alternate means to ensure least defects in software.

In addition to testing, we also explored Coverity Static code analyzer for JAVA [1] as a dissimilar redundant tool for JPF. Coverity accepts Java code and is capable of detecting deadlock and race conditions by examining all possible code paths like JPF. However, it is a static analysis tool that is commercially developed Coverity that makes its significantly different from JPF. The dissimilarity in the underlying formal method and approach provides higher confidence in the JPF results when it matches with the result from the static analysis tool.

2.2.1 Static analysis on the model and source code.

In model-based development, certain properties can be proven on both the development model and the code. For example, a model analyzer/checker could show that the model has no inherent overflow. This does not automatically mean that the code generated from the model does not have overflow, but it increases the confidence in a proof of absence of overflow from a code analyzer. In this case the objectives are not the same (6.3.2.g algorithms are accurate and 6.3.4 source code is accurate and consistent); but verification (preferably by a qualified tool) that the code implements the model means they reinforce each other, which is why they were both included in DO-178. With different inputs and underlying formalisms, there can be no common mode failures; so both tools reporting an absence of overflow provides high confidence in the result.

3 Tool Self-Tests and Benchmark Propagation

The development of test cases and procedures for the formal methods tool is a significant part of the overall tool verification process. Even though testing alone cannot provide a complete proof of correctness, the establishment of a comprehensive test suite still represents an important strategy for risk mitigation. In essence, the more we test the tool and demonstrate its correct behavior, the more confident we can be that it will not produce a fault.

In the test cases and procedures document developed for JPF as part of our case study exercise, we identified two different test categories:

1. Full end-to-end tests of JPF-Core-X to verify that it appropriately finds (or confirms the absence of) certain property violations for specific examples of SUT's. Each of these tests represents a unique benchmark, and supports high-level usage requirements that are listed in the TOR document, such as the requirement to find and report all deadlocks.

2. A suite of low-level unit tests to verify correct implementation of distinct methods in JPF-Core-X. Each of these tests support one or more lower-level functional requirements that are identified in the TR document.

The purpose of the end-to-end benchmark tests is to demonstrate that the tool correctly exhibits the required behavior as described in the Tool Operational Requirements (TOR) document. Successful completion of each test demonstrates that the external interface requirements and the verification operational requirements listed in the TOR are satisfied for that test case.

The purpose of the unit tests is to verify that each method used within the tool is implemented correctly. Each unit test verification is based upon the expectation that, for a given input, the method being tested has a corresponding correct output. Tests pass when the output (or some part or property of it) matches the expected correct output.

The test categories discussed above are focused on the testing of an individual tool. In the case where multiple tools are to be used in concert as part of a risk mitigation strategy, then an additional layer of testing for the combined tool suite is appropriate. In general, any positive attributes that one expects (or claims) to achieve through the combined use of multiple tools – whether they be applied in serial as complementary tools, or in parallel for redundancy – must be confirmed through targeted benchmark tests that cover the full chain and are repeatable.

3.1 Tool Self Tests

At qualification time, the test suite for an individual tool is designed to provide full coverage over the requirements of that tool. In general, the presence of a failed test indicates that the

tool does not satisfy the requirement associated with that test. Consider now the addition of a second redundant tool that *is* capable of satisfying that requirement. In this way, redundant tool sets may be used to fill in gaps in the overall set of qualification requirements that might not be satisfied by a single tool on its own.

After tool qualification, the tools are used repeatedly in the process of certifying software. The use of alternative tools helps to protect the integrity of the overall certification process because, if one tool fails to detect a fault, there is still an opportunity for it to be caught by one or more other tools.

It is possible (in fact, it is preferred) for this type of discrepancy in tool output to occur in tool self tests. This raises an interesting question. When two alternative (redundant) tools provide fundamentally different outputs for an equivalent input, what should be done? Assuming that both results cannot be true, it may not be evident which result should be trusted as the correct one. In such a case, further investigation is required on the part of the user to make the determination. This will likely lead to a failure to be declared in at least one of the tools.

3.1.1 Example: Deadlock with Varying Number of Threads

JPF provides an example benchmark problem that allows us to vary 3 parameters. The purpose of the test is to verify that the tool properly detects the presence of a deadlock that is known to exist in the system under test.

The system under test includes a bounded buffer of size B, N_P producer threads and N_C consumer threads. The bounded buffer consists of B slots. Producer threads put an object into the buffer, and consumer threads remove an object from the buffer. The deadlock depends on a notification choice between a consumer and a producer in a context where only threads of the notifier type are still runnable. A deadlock results if the number of producer or consumer threads is at least 2x the size of the buffer.

This particular test is adapted from "Concurrency: State Models & Java Programs"[9].

We can run this test with different values of B, N_P, N_C to demonstrate that the tool correctly identifies the presence or absence of deadlocks for each unique scenario. In general, increasing the total number of threads ($N_P + N_C$) leads to a combinatorial growth in the size of the state space to be searched. Therefore, this benchmark provides a convenient mechanism for evaluating the tool across varying levels of program complexity.

3.1.1.1 Inputs

- SUT: jpf-core/src/examples/BoundedBuffer.java
- JPF Properties: jpf-core/src/examples/BoundedBuffer<X>.jpf

where <X> is a unique identifier to distinguish between different variations of the test.

The JPF properties file for this test specifies "BoundedBuffer" as the target, and defines the following target arguments: `target_args = 1,2,1`. This specifies buffer of size 1, 2 producer threads, and 1 consumer thread.

3.1.1.2 Expected Results In this test, we expect a deadlock to be encountered whenever $N_C \geq 2B$ or $N_P \geq 2B$. If either is satisfied for a particular test case, we expect the printout to the JPF console to include an error message indicating the occurrence of a deadlock, a snapshot showing the current status of each thread when the deadlock is found, and a summary results message. Otherwise, the printout to the console should display "no errors found" and indicate that the search has finished.

Buffer Size	Producers	Consumers	Total Threads	Expected Result	JPF Result	Duration (min)
1	2	1	3	Deadlock	Deadlock	0.05
2	1	4	5	Deadlock	Deadlock	0.05
2	4	1	5	Deadlock	Deadlock	0.08
3	1	6	7	Deadlock	Deadlock	0.25
3	6	1	7	Deadlock	Deadlock	1.50
4	1	4	5	Safe	Safe	10.00
4	4	1	5	Safe	Safe	2.00
4	8	1	9	Deadlock	?	>30

3.1.1.3 Use of Redundant Tools to Supplement Test Failures All but the last test case are shown to pass because they terminate with the expected result. If we elected to time-out each test case at 30 minutes, then the final test would have to be marked as a fail. In such a case, we might elect to use a different model checker, such as SPIN, or perhaps just a different configuration of JPF, if it is known to reach completion faster. For example, changing the search heuristic from a depth-first search to one with a random policy may cause the deadlock to be found faster, and retaining the seed for that random search would make the test repeatable.

3.2 Benchmark Propagation

Figure 4 shows the Level A Software Verification Process from DO-178C, and illustrates how multiple tools may be used within that process. In this example, we show the redundant model-checking and code analyzers for JPF and CodeHawk respectively to form a tool chain. In the figure, the blue arrows indicate the general process of generating LLR from HLR and source code from the verified HLRs and LLRs. The red arrows points to the specific software verification artifact identified in DO-178C process, that serve as inputs to the tool chain. JPF model is used to check the high and low level requirements. Once that process is

complete, the source code developed using the (verified) requirements provides the input for the CodeHawk C Analyzer, which becomes a separate link in the overall tool chain.

When considering a tool-chain, it is important to identify test cases which span the sequential use of multiple tools in the chain. For the example shown in Figure 4, we would have a test suite for JPF Core, and another test suite for CodeHawk. Taken in isolation, each test suite is designed to provide evidence that the respective tool satisfies all requirements that were identified for qualification. The redundant tools for each tool will share test cases among themselves, since they have similar operational requirements. In practice, however, the tools are meant to be used in conjunction with one another as part of the overall certification process. In this context, it is appropriate to identify test cases that exercise each tool using data that propagates through the tool chain.

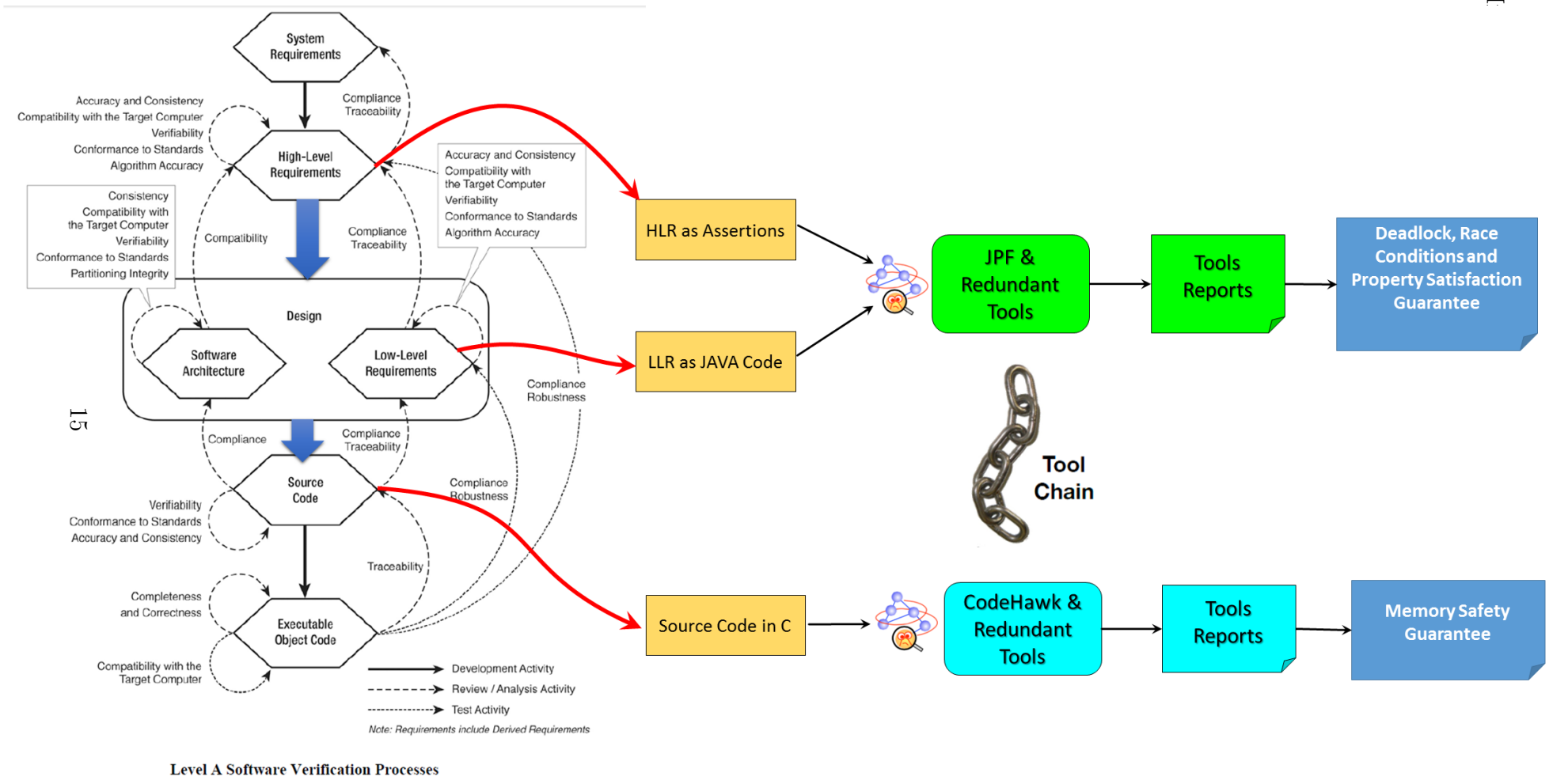

Figure 4: Tool chain in Software Verification Process.

4 Tool Life Cycle Enhancements

In usage of static analysis and model checker tools, errors or ambiguities can arise in the user interface (inputs, outputs) or internal tool functions. These may result in one or more of the following erroneous situations:

- False positives—e.g., a defect reported by overly conservative abstractions in the static analysis but the defect doesn't exist; this is not a tool soundness issue but a usability problem.
- False negatives—e.g., a property proven incorrectly by a model checker. This is a tool soundness issue.
- An ambiguous report to the user that doesn't state a property properly or doesn't provide a clear indication of the results of property check.
- Not implementing graceful handling of execution time and memory explosion (due to large model state spaces)—e.g., can partial results be trusted if the tool hangs?

In this section we examine risk mitigation strategies that impact the tool life cycle—driven by its usage context with the tool inputs and outputs. In particular, we will consider strategies for checking of tool inputs and outputs, and tool execution process. The next subsection describes input/output checking approaches relevant to our tool case studies. We also consider implementation, qualification issues and additional consideration for usage in a software project subject to certification.

4.1 Tool Input/Output/Execution Checking and Proof Checking Approaches

The application of formal tools to complex system verification involves the steps of translating the system/software designs or code artifacts into *intermediate representations* (models and the desired properties into a formal specification) as inputs to the tools, as well as correctly interpreting the outputs of the tool. Errors, ambiguities, or lack of precision in any one of these may lead to unsound results and ultimately misplaced confidence in the verification.

In our investigations of theoretical soundness issues in the previous year of this effort, we described several soundness issues can occur due to errors in the intermediate representations and tool implementations [3]. In the context of the case studies on the JPF model checking tool and CodeHawk static analysis tool, we have analyzed the specific failure modes of theses tools that are summarized in separate documents produced in this effort. In this section, we address the mitigation of errors in the intermediate representations—the tool inputs and outputs. We examine input checking, output checking, execution checking, and use of proof checkers. Figure 5 illustrates the flow of information from the software development

artifacts through the tools to the satisfaction of the DO-178C objectives to verify the software development process.

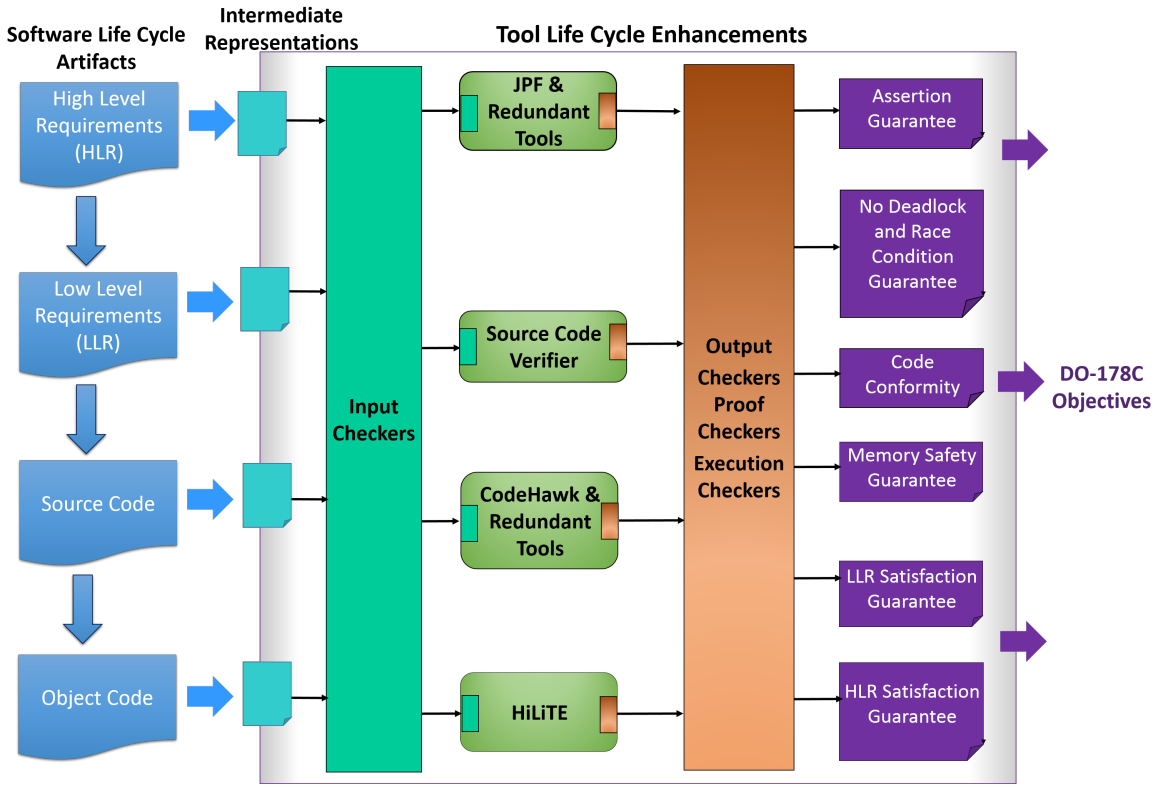

Figure 5: Life Cycle Enhancements to Verification Tools

4.1.1 Employing Input/Output and Execution Checkers

Input Checkers. While most tools will report syntax related issues in their inputs, formal verification tools often have a complex input model where the use of abstractions, assumptions, and correct semantic interpretation is necessary to guarantee sound results. For example, the consistency and validity of the model that is input to the formal tool is very important. The abstractions—e.g., concurrency relationships, data types, and abstractions dealing with time—shouldn't be incorrect or lossy. Furthermore there can be tool limitations that can put restrictions on the inputs, e.g.: allowable primitives and connections and interfaces between the primitives, maximum number of primitives, array sizes, states, threads, nesting of control constructs, cyclomatic complexity, recursive complexity, etc. For certain critical tools such as qualified code generators, the certification guidance requires establishing such limits based upon what maximum limits were reached in qualification tests, even though the tool may not inherently have any such limitations. Thus, for most tools, it becomes crucial to ensure that the input provided to the tools are well within the scope of

acceptable inputs to the tool. Limits consideration should be properly guided by the tool FMEA and any specific certification guidance applicable to this class of tools.

In our case study we investigated the need of specialized tools to check input of JPF and CodeHawk. We identified that the version of JPF is qualified (tested) to verify JAVA programs with no more than 8 threads. While this does not necessarily mean JPF fails if there are more threads, such an output is not guaranteed to be sound and can not be used to claim certification credit. So, there is an additional responsibility to ensure that the application that JPF will verify will not have more than 8 threads. CodeHawk, on the other hand, is built in with a specific version of compiler (gcc) and any program that is successfully compiled by that compiler is acceptable for the tool. Hence, there was no need for any special input checkers for CodeHawk. However, it is necessary to compile the code with the specific version of gcc before running CodeHawk to verify there are no compile errors or warnings, in order to guarantee sound results from CodeHawk. This is similar to the input checking required for the usage of Honeywell's HiLiTE tool—the input Simulink model must be simulatable and must pass a model compliance check to check limits on allowed primitives and parameters.

Output Checkers. Another class of checking tools, that we call *output checkers*, can be used to check for certain properties in outputs of verification tools. Model checkers such as JPF return counterexamples to illustrate how a property was falsified. In many cases many spurious and trivial counter examples will be returned that results in spending enormous amount of time to avoid them. Alternatively, one could implement tools that checks the validity of counterexamples so that obviously false conjectures are eliminated. However, when the model checker does satisfy the specification it does simply return a positive but boolean answer. While the positive answer does guarantee that property is satisfied, it may have been satisfied in an unintended fashion. For instance, if one indented to check a property that the system issued an acknowledgment whenever a request was sensed, the property will be trivially satisfied if there was never a request to the system due to an incorrect assumption. This is called *vacuity* and it many cases it has been found that detecting vacuities have uncovered unintended specifications and system behaviors [12]. One can implement such a vacuity checker in a model checking tool.

In other cases, specialized checkers that parse the tool logs in addition to the actual result from the tool can report if the output is trustworthy from a specific perspective. For instance, JPF searches the application state space until a specified depth and reports property violations and satisfactions within that depth. However, if the depth of the software under test is more than the set depth, there is a possibility that JPF does not report the error found beyond the set depth. To avoid this problem, one could implement a special checkers that can parse JPF logs to identify if there were such unexplored paths and report the result appropriately. Other approaches, such as proof checkers, to check more complex/comprehensive aspects of a property proof are described in the next subsection.

Tool Execution Checks. An important criteria for all tools is to provide a set of complete and accurate results, with clear indication in the results if some error was detected

by the incomplete tool execution that would compromise the soundness of the results. For example, in our case studies, we determined that there is the need to assure CodeHawk's component scripts are executed properly. In the old CodeHawk implementation, the user invoking each CodeHawk component separately put the burden on the user to ensure the consistency between these component runs. The CodeHawk implementation was actually modified as part of the case study to mitigate this type of error—there is a master script now that runs the individual components.

Another important consideration for formal methods tools is to provide graceful handling of execution time and memory explosion (due to large model state spaces). This requires that the results should clearly indicate that tool execution is erroneous—thus invalidating all results. From the usability perspective, however, it may be practically necessary to identify validity of partial results (e.g., individual properties correctly proven). Our experience of using HiLiTE and model checkers suggests that such tools should implement stages of execution (e.g., one property proven in one stage) and return a successful exit code to the calling script if the tool terminates normally. A method should be provided, as permitted by the operating system, to terminate the tool in case of unacceptable execution time or memory usage that records in the overall execution results that the tool did not terminate normally.

In our case studies, both CodeHawk and JPF did not include any input/output checkers. In our failure mode analysis of these tools, we found that the risk of soundness issues by incorrect inputs and outputs of the tools were high and hence, one of our proposed mitigation strategy is to consider using relevant checkers outside the tool. These companion tool sets will help ensure that inputs and outputs not within the bounds of the designed limitations and required behavior of the verification or analysis tool.

4.1.2 Use of Proof Checkers

Recently, there have been a few research prototypes of *proof checkers* for formal analysis tools. In general, proof checkers are tools that are specifically constructed to validate or check the formal workings of proofs that were developed either manually, automatically by tools, or semi-automatically through human assisted techniques [6] (also called interactive theorem provers). These checkers are intended to check each step of the proof by direct application of a strategy or rule in the logic either automatically or through human-assisted means.

Wetzler et al. [8] describe the *DRAT-trim* proof checker tool for satisfiability (SAT) solvers. It accepts the SAT input formulas and the proof produced by the SAT solver in the DRAT notation, and then validates the proof using knowledge of all existing proof techniques used by state-of-the-art SAT solvers. Tinelli et al. [7] have implemented a proof checker for the *Kind 2* model checker that generates proof certificates. The proof certificates used as input to the CVC4 SMT Solver that generates LFSC proofs of safety properties that are checked by an LFSC checker. The proofs contain details about translation from model input

language Lustre to internal representation inside Kind 2 and invariance by k-induction. From the user documentation of Kind 2, the independence of the proof checking capabilities from the model checking is unclear or incomplete. For example, the Lustre model input is not independently checked by the LFSC proof checker to verify the correctness of translation of Lustre model to the Kind 2 internal implementation. Furthermore, it is unclear whether the SMT solving proof techniques employed by CVC4 are also checked in a manner similar to how DRAT-trim checks for proof techniques employed by SAT solvers.

In general, the use of a proof checker in conjunction with a model checker would require the proof checker to have access to the same inputs as the model checker—including the model, properties to be prove, and any options or command-line parameters as illustrated in Figure 6. All these would be needed to interpret and verify the proof details and the property proof results of the model checker. Furthermore, the proof contents and proof checking techniques will need to be specialized to the state-space exploration, invariance definition, and other proof techniques used by a model checker.

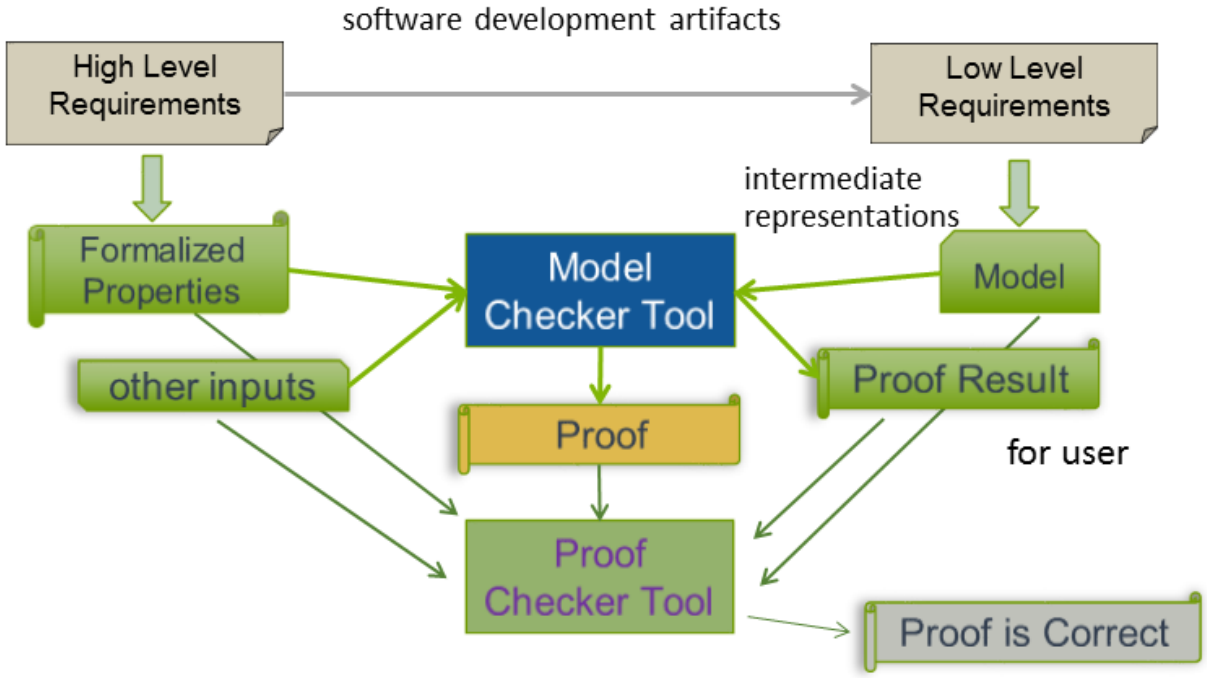

Figure 6: The General Context of using a Proof Checker for a Model Checker

Similarly, checking the proofs produced by an SMT solver (used as a back-end of a model checker) would need to be specialized. Different SMT solvers provide different capabilities (e.g. some might handle some non-linear expressions, while others do not), and may use different proof techniques. Use of different SMT solvers by a model checker would practically require qualification of the overall model checker tools with each separate SMT solver plugged

in. Furthermore, a particular model checker may use an SMT solver for a specific sub-problem of model checking and the SMT solver may just provide output results rather than part of the overall proof. Thus checking the output of the SMT solver itself using a proof checker may not make sense. For example, in our case study, the use of SMT in JPF is limited to pruning the state space during search, and as such is not normally part of the results of the model checker (JPF) itself.

Based upon our analysis of current proof checking techniques, they are not simple checker tools but rather redundant (if overlapping domain) or complementary tools whose the selection would be based upon the model checker's specific capabilities and proof techniques employed. The fundamental problem is that the proof checker would have to do the same analysis of the model and properties that the model checker would do and therefore the same complexity issues that foil model checkers would apply (e.g., space and time complexity). Thus the qualification of just the proof checker instead of the model checker may be as complex as qualifying the model checker directly and thus rendering it impractical. We recommend treating a proof checker the same as a redundant or dissimilar tool and apply the same considerations as described in Section 2.1 of this report.

4.2 Tool Life Cycle and Qualification Considerations

There are different way to introduce checkers in the verification life cycle: either building them internally in the tool set itself or developing them outside the scope of the tool for the purpose of verifying the software under certification. There are pros and cons in each approach.

When such checking is built into the tools, it makes it convenient to users since they do not have to deal with ensuring correctness of inputs and outputs. It eliminates the class of soundness issues due incorrect inputs and outputs for all tool users with the users having to employ another tool and track it in their certification process. However, this implies additional development, error reporting, testing and maintenance for the tool developers. In fact, it adds to the complexity of building the tool. Users may not have access to details of the input/output is checked, so any tool errors in the input checker is difficult to detect. Furthermore, a built-in input/output checker may share common failure modes with the tool itself.

On the other hand, if the tool does not handle such checks, then it becomes the responsibility of the user to ensure that the inputs are good and the output is sound. But, If users implement such a checker, although it might be an overhead in terms of development and ensuring consistency with the tool, they have the flexibility to restrict the input/output checking to only those that are relevant for that application and could be efficiently done. This can be done independently from the verification task.

5 Safety Case Argument Templates

The overall form of a safety case argument derived from the use of formal methods tools in the certification of avionics software is that the various tools in the tool chain verify that the resulting executable software does or does not have certain safety related properties. Since this argument relies on a high degree of confidence in the soundness of the formal methods tools verifying these properties, the safety case can be improved by additional arguments that account for possible soundness risks in the use of these tools. Following the suggested analogy above with the use of failure modes and effects analysis (FMEA) for assessing the effects of individual component failures on the overall system behavior, we have looked for the effects of "soundness failures" in individual tools on the overall verification by the tool chain. And as with the FMEA exercise of recommending mitigation strategies for point-component failures, we have enumerated strategies to mitigate the risks of point-tool "soundness failures" on the final verification verdict.

The safety case argument DSL, "L", from the CertWare II Safety Case Workbench appears to be the most appropriate formalism for capturing "templates or patterns" of these augmented arguments, so we have chosen it for this subtask. L is an English-like dialect of a standard Answer Set Programming (ASP) language. Safety case arguments captured in L can be automatically translated to the standard ASP language and submitted to external ASP solvers for assessing the logical satisfiability and validity of the arguments. Because this is a formal logic language, we can capture "patterns or templates" of argumentation as general rules over predicates with variables. More concrete specializations of these general predicates represent individual safety cases that conform to the overall argument structure.

We capture the most general form of the (augmented) safety case argument with the following L rule:

```
invalidityRisk(certification(property P, software S)) if
      reliesOn(certification(P, S), evidence E) and
      produces(toolChain TC, E, P, S) and
      verificationRisk(risk R, P, S, TC) and
      not mitigated(R, P, S, TC).
```

This says that there is a risk that the certification of any software S having any property P is invalid if the certification relies on some evidence for S having P, produced by some tool chain, and there is a verification risk for this component of the tool chain that has not been mitigated. This project has enumerated various kinds of these verification risks for formal methods (FM) tools, but we have concentrated on undetected soundness risks for individual tools. We will formalize three of these below, for the three classes of FM tools we have been considering, by refining the extension of the `verificationRisk` predicate:

```
verificationRisk(unsound(staticAnalyzer), property P, software S, toolChain T)  if
```

```
        executes(staticAnalyzer SA, input(P, S), output(verified), T) and
        produces(T, evidence E, P, S) and
        includes(E, verified(P, S, SA)).

verificationRisk(unsound(modelChecker), property P, software S, toolChain T)  if
        modelOf(S, model M) and
        executes(modelChecker MC, input(P, M), output(verified), T) and
        produces(T, evidence E, P, S) and
        includes(E, verified(P, M, MC)).

verificationRisk(unsound(codeGenerator), property P, software S, toolChain T)  if
        modelOf(S, model M) and
        executes(codeGenerator CG, input(P, M), output(S), T) and
        produces(T, evidence E, P, S) and
        includes(E, generated(S, M, CG)).
```

Note that the verification risk accrues in each case simply because an FM tool's output is included in the verification evidence. This represents the generic risk that any FM tool may be unsound in ways that have not yet been detected. We will refine the `mitigated` predicate by enumerating mostly the cases dealing with potential soundness failures:

```
% redundant static analyzers
mitigated(unsound(staticAnalyzer), property P, software S, toolChain T)  if
       executes(staticAnalyzer SA1, input(P, S), output(verified), T) and
       executes(staticAnalyzer SA2, input(P, S), output(verified), T) and
       SA1 != SA2.

% redundant model checkers
mitigated(unsound(modelChecker), property P, software S, toolChain T)  if
       modelOf(S, model M) and
       executes(modelChecker MC1, input(P, M), output(verified), T) and
       executes(modelChecker MC2, input(P, M), output(verified), T) and
       MC1 != MC2.

% verify same property with model checker and static analyzer
mitigated(unsound(codeGenerator), property P, software S, toolChain T)  if
       modelOf(S, model M) and
       executes(codeGenerator CG, input(P, M), output(S), T) and
       executes(modelChecker MC, input(P, M), output(verified), T) and
       executes(staticAnalyzer SA, input(P, S), output(verified), T).

% multiple backend SMT solvers
```

```
mitigated(unsound(modelChecker), property P, software S, toolChain T)  if
       modelOf(S, model M) and
       executes(modelChecker MC, input(P, M, smtSolver SMT1), output(verified), T) and
       executes(modelChecker MC, input(P, M, smtSolver SMT2), output(verified), T) and
       SMT1 != SMT2.
```

In both standard logic programming and the ASP paradigm, a "logic program" is a formal theory in the first-order predicate calculus. Standard logic programming restricts the theory to the definite clause subset of full first-order logic and attempts to find all instantiations for all variables in a conjecture over one or more of the theory's predicates. ASP, by contrast, uses the full first-order logic, and attempts to find stable models for the entire theory/program. Programs with models are satisfiable; those without models are unsatisfiable (implying contradictions). To make an ASP evaluation computationally feasible, all variables in the program are first grounded against a stated universe of primitive facts. An ASP program thus typically consists of general rules, as above, and some ground facts which represent a typical concrete scenario for the rules to be evaluated over. Our initial ground universe will consist of a sample qualification tool chain scenario.

The following type declarations provide the minimal number of individual constants to instantiate the types:

```
type codeGenerator = {codegen1}.
type staticAnalyzer = {staticAnalyzer1, staticAnalyzer2}.
type modelChecker = {modelChecker1, modelChecker2}.
type testGenerator = {testGen1}.
type testHarness = {tester1}.
type evidence = {evidence1}.
type software = {software1}.
type property = {property1}.
type model = {model1}.
type toolChain = {toolchain1}.
type risk = {unsound(staticAnalyzer), unsound(modelChecker), unsound(codeGenerator)}.
```

The following facts relate an instance of software, its property, and its model to an instance of certification evidence:

```
model(software1, model1).
reliesOn(certification(property1, software1), evidence1).
produces(toolchain1, evidence1, property1, software1).
includes(evidence1, verified(property1, software1, staticAnalyzer1)).
includes(evidence1, verified(property1, model1, modelChecker1)).
includes(evidence1, generated(software1, model1, codegen1)).
```

The following facts describe the role of each FM tool in a tool chain scenario:

```
executes(codegen1, input(property1, model1), output(software1), toolchain1).
executes(staticAnalyzer1, input(property1, software1), output(verified), toolchain1).
executes(modelChecker1, input(property1, model1), output(verified), toolchain1).
executes(testgen1, input(model1), output(testCases1), toolchain1).
executes(tester1, input(software1, testcases1), output(passed), toolchain1).
```

Running this in an ASP solver yields the following output:

```
Solving...
Answer: 1
codeGenerator(codegen1)
staticAnalyzer(staticAnalyzer1)
staticAnalyzer(staticAnalyzer2)
modelChecker(modelChecker1)
modelChecker(modelChecker2)
testGenerator(testGen1)
testHarness(tester1)
evidence(evidence1)
software(software1)
property(property1)
model(model1)
modelOf(software1,model1)
toolChain(toolchain1)
reliesOn(certification(property1,software1),evidence1)
produces(toolchain1,evidence1,property1,software1)
includes(evidence1,verified(property1,software1,staticAnalyzer1))
includes(evidence1,verified(property1,model1,modelChecker1))
includes(evidence1,generated(software1,model1,codegen1))
executes(codegen1,input(property1,model1),output(software1),toolchain1)
executes(staticAnalyzer1,input(property1,software1),output(verified),toolchain1)
executes(modelChecker1,input(property1,model1),output(verified),toolchain1)
executes(testgen1,input(model1),output(testCases1),toolchain1)
executes(tester1,input(software1,testcases1),output(passed),toolchain1)
verificationRisk(unsound(staticAnalyzer),property1,software1,toolchain1)
verificationRisk(unsound(modelChecker),property1,software1,toolchain1)
verificationRisk(unsound(codeGenerator),property1,software1,toolchain1)
mitigated(unsound(codeGenerator),property1,software1,toolchain1)
invalidityRisk(certification(property1,software1))
SATISFIABLE
```

```
Models       : 1
Calls        : 1
Time         : 0.004s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

The stable model includes an inferred instance of an invalidityRisk for the certification of software1 having property1. This is because our example uses executions of FM tools without explicit mitigation. Actually, the codeGenerator risk is mitigated because its input and output have the same property verified by separate analyzers, but it takes only one unmitigated risk to infer an invalid-certification risk.

If we add mitigation execution steps for the static analyzer and the model checker to our scenario:

```
executes(staticAnalyzer2, input(property1, software1), output(verified), toolchain1).
executes(modelChecker2, input(property1, model1), output(verified), toolchain1).
```

and then re-solve, we get this:

```
Solving...
Answer: 1
codeGenerator(codegen1)
staticAnalyzer(staticAnalyzer1)
staticAnalyzer(staticAnalyzer2)
modelChecker(modelChecker1)
modelChecker(modelChecker2)
testGenerator(testGen1)
testHarness(tester1)
evidence(evidence1)
software(software1)
property(property1)
model(model1)
modelOf(software1,model1)
toolChain(toolchain1)
reliesOn(certification(property1,software1),evidence1)
produces(toolchain1,evidence1,property1,software1)
includes(evidence1,verified(property1,software1,staticAnalyzer1))
includes(evidence1,verified(property1,model1,modelChecker1))
includes(evidence1,generated(software1,model1,codegen1))
executes(codegen1,input(property1,model1),output(software1),toolchain1)
executes(staticAnalyzer1,input(property1,software1),output(verified),toolchain1)
executes(modelChecker1,input(property1,model1),output(verified),toolchain1)
executes(testgen1,input(model1),output(testCases1),toolchain1)
```

```
executes(tester1,input(software1,testcases1),output(passed),toolchain1)
executes(staticAnalyzer2,input(property1,software1),output(verified),toolchain1)
executes(modelChecker2,input(property1,model1),output(verified),toolchain1)
verificationRisk(unsound(staticAnalyzer),property1,software1,toolchain1)
verificationRisk(unsound(modelChecker),property1,software1,toolchain1)
verificationRisk(unsound(codeGenerator),property1,software1,toolchain1)
mitigated(unsound(staticAnalyzer),property1,software1,toolchain1)
mitigated(unsound(modelChecker),property1,software1,toolchain1)
mitigated(unsound(codeGenerator),property1,software1,toolchain1)
SATISFIABLE

Models       : 1
Calls        : 1
Time         : 0.005s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

In the new stable model, mitigation are now inferred for each of the three FM tools, and an invalidityRisk is no longer implied.

6 Conclusion

While the avionics industry's maturity level in adopting formal methods tools while developing safety critical software has increased, qualifying the tools as per DO-330 and demonstrating to the FAA that the use of such tools and techniques can indeed assure safety is a challenge. Key among the concerns is establishing the *soundness* of the formal tools—the characteristic of the tool to never report a valid conclusion when it is actually invalid about the system. Since formal tools make use of mathematically complex artifacts, complicated analyses and typically include sophisticated heuristic search algorithms, there are various threats to trusting the results produced by those tools (discussed in the theoretical soundness report).

While there is risk in completely trusting a tool that finds safety-critical software is defect-free, there is also risk in rejecting tools because there might be as-yet undiscovered errors. Humans are no more capable of identifying these issues, and tools provide rigor and consistency. Humans are prone to rationalization (especially with a deadline looming), overconfidence, and laziness. In this report, we proposed various mitigation strategies to encourage deployment of these tools, even while acknowledging they cannot be guaranteed absolutely correct.

References

[1] Coverity - static code analysis. http://www.synopsys.com/software/coverity/. Accessed: 2016-09-09.

[2] Symbolic PathFinder (SPF). http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc. Accessed: 2016-09-09.

[3] Ssat datcpi (facts) theoretical soundness issues report. Technical Report submitted to NASA under Contract NNL14AA07C, 2015.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[5] Matthew R Barry. Certware: A workbench for safety case production and analysis. In *2011 Aerospace Conference*, 2011.

[6] Woodrow W Bledsoe and Peter Bruell. A man-machine theorem-proving system. *Artificial Intelligence*, 5(1):51–72, 1974.

[7] Cesare Tinelli et al. Kind 2 user documentation, version v1.0.1. https://kind2-mc.github.io/kind2/doc/user_documentation.pdf, 2016.

[8] Wetzler et al. Drat-trim: Efficient checking and trimming using expressive clausal proofs. Volume 8561 of the series Lecture Notes in Computer Science pp 422-429, 2014.

[9] William F Gilreath. Concurrency state models and java programs. *Scalable Computing: Practice and Experience*, 3(4), 2001.

[10] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[11] Gerard J. Holzmann. The design and validation of computer protocols. 1991.

[12] Orna Kupferman. Sanity checks in formal verification. In *CONCUR 2006–Concurrency Theory*, pages 37–51. Springer, 2006.

[13] Robert A Martin. Common weakness enumeration. *Mitre Corporation*, 2007.

[14] Corina S Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 179–180. ACM, 2010.