# Guidelines for the Use of Theorem Proving in the Certification of Critical Systems

88x31.png

This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](http://creativecommons.org/licenses/by-sa/4.0/).

This document is a collective effort originating from the [Workshop on Theorem Proving in Certification](http://www.cl.cam.ac.uk/~mjcg/FMStandardsWorkshop/). This is a draft. You can contribute by sending feedback or suggestions of improvement on the public mailing-list [tpcert@googlegroups.com](mailto:tpcert@googlegroups.com). You can also participate more actively in the discussions by joining the googlegroup, or by participating to the next edition of the Workshop.

Feel free to share with others the link to this document:

<https://docs.google.com/document/d/1nTXzxHLXg9s97mxtrEOS8rfgUNGK3dnW7L43WwU7Lpo>

# Introduction

Formal methods are mathematical techniques used in the specification, design, implementation or verification of computer systems. We retain the definition of a formal method given in the “*Formal Methods Supplement to DO-178C and DO-278A*” document, a companion document to the aeronautic certification standards:

* *“A formal model combined with a formal analysis constitutes a formal method.” [[1]](#footnote-0)*

where

*“to be formal, a model should have an unambiguous, mathematically defined syntax and semantics” 1*

and

*“an analysis method can only be regarded as formal analysis if its determination of a property is sound. Sound analysis means that the method never asserts a property to be true when it is not true.” 1*

Formal methods have long been considered as a means of compliance to satisfy verification objectives in critical software development for some certification domains, for example in railway (EN 50128) and industrial processes (IEC 51508). The aeronautic standard has recently recognized formal methods as a means of compliance on par with the dominant technique of testing (DO-178C, 2012). Other certification standards in the domains of automotive (ISO 26262), nuclear (IEC 60880) and space (ECSS-QST-80C) also recognize some uses of formal methods as verification techniques. The English military certification standard *“Safety Management Requirements for Defence Systems”* (DEFSTAN 0056) puts a strong emphasis on formal methods as a means of compliance.

Theorem proving is a branch of formal methods characterized by the use of (possibly intermediate) logical formulas to express properties over the system. These formulas expressed in a mathematical logic, typically first-order propositional or predicate logic, type theory or set theory, are generated from the system so that their validity implies that the desired properties hold, and proved valid either automatically or with human guidance. In both cases, the validity of formulas should be checked mechanically.

The purpose of this document is to give a set of recommendations for the application of theorem proving to address verification objectives in critical systems development, so that certification credit can be claimed from the use of theorem proving. This document attempts to describe how theorem proving should be applied to address the concerns of certification, and what information is required about this application in order to support such claims.

Other formal methods not considered directly in this document are model checking and abstract interpretation, although they might be addressed indirectly regarding their use of theorem proving techniques. For example, software model checking is usually based on theorem proving for deciding the feasibility of execution paths.

We use as basis the report written by John Rushby for the NASA in 1993 on *“Formal Methods and the Certification of Critical Systems”*. Although this report is 20 years old, its analysis of the benefits and fallibilities of theorem proving for certification is still very much valid. We will reuse from Rushby’s report the classification of theorem proving techniques in three levels of rigor:[[2]](#footnote-1)

1. *Use of concepts and notation from discrete mathematics*
2. *Use of formalized specification languages with some mechanized support tools*
3. *Use of fully formal specification languages with comprehensive support environments, including mechanized theorem proving or proof checking*

and the distinction between theorem proving applied in the early lifecycle (on high level requirements, or HLR) and theorem proving applied in the late lifecycle (on code).[[3]](#footnote-2)

Although Rushby’s report talks about “formal methods”, this mostly corresponds to what we call here “theorem proving”: model checking techniques are mentioned en passant, and nothing is said about abstract interpretation techniques, which did not have then the recognition that they have today. Here is how Rushby defines formal methods:[[4]](#footnote-3)

*"What I am concerned with is the extent to which a method is “truly formal”: that is, formulates specifications in an axiomatic style, explicitly enumerates all assumptions and reduces proofs to explicit applications of elementary rules of inference."*

We will follow Rushby in calling the combination of a formal method and a tool implementing it as a formal system, or system for short.

While the focus of Rushby’s report is on the use of level 3 methods applied early in the lifecycle to the hardest problems (concurrency, asynchrony, real time, redundancy management and fault tolerance), the 20 years since the report have seen evolutions in the use of theorem proving, mostly due to the many-orders-of-magnitude increase in automation of some theorem proving methods. In particular, use of theorem proving in the late lifecycle may be more common today than its use in the early lifecycle.

# 

# 

Relation to Certification Standards

*This section will describe how the guidelines provided in this document relate to the certification standards for various domains. In particular, a mapping from guidelines to certification standards objectives would be useful, to show how these guidelines apply in practice.*

The certification standards for critical software of many domains have been recently updated. Many of them recognize the use of formal methods as a means of compliance for development or verification of the software specification or implementation. The following table[[5]](#footnote-4) gives an overview of the current situation:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **domain** | **standard name** | **last version** | **use of FM for**  **specification** | **use of FM for implementation** |
| avionics | DO-178 | 2012 | applicable | partially applicable |
| automotive | ISO 26262 | 2011 | applicable | applicable |
| medical | IEC 62304 | 2006 | recommended | recommended |
| nuclear | IEC 60880 | 2006 | encouraged | applicable |
| process | IEC 61508 | 2010 | recommended | recommended |
| railway | EN 50128 | 2011 | recommended | recommended |
| space | ECSS-Q-ST-80C | 2009 | applicable | applicable |

Although most standards allow or even recommend the use of formal methods, they are not prescriptive in how formal methods should be used. The goal of these guidelines is to help projects using theorem proving at either the specification or the implementation level to present adequate evidence related to theorem proving activities for certification.

# 

# 

Choice of Methods and Tools

*This section will describe how the choice of methods and tools have an impact on the work required for justifying their use in certification.*

We are interested in this document in the factors that matter for certification. These are the factors that are relevant for assessing if a formal system is *“correctly defined, justified, and appropriate to meet [its] verification objective” [[6]](#footnote-5)*.

Nontechnical factors may influence the selection of methods and tools. Rushby’s report lists *“popularity, availability of training courses and textbooks, and standardization and endorsement by various national and international bodies”* among these factors. These factors are not relevant in the sense given above.

Many technical factors contribute only to the effectiveness of the application of a given formal system in an industrial context, and are also not relevant in the sense given above. For example, the facility to learn a formal system bears little importance for certification, provided users of the system had adequate knowledge for applying it. Rushby’s report lists 48 issues technical factors for the choice of formal system, 16 of which are relevant for certification. We list those in Annex 1.

**Choice of Methods**

A method should comply with some general requirements in order to provide adequate guarantees that it is a bona fide instance of theorem proving. These requirements should ensure in particular that the methods has an adequate mathematical foundation.

The “*Formal Methods Supplement to DO-178C and DO-278A*” document states it with two precise objectives:[[7]](#footnote-6)

*“All notations used for formal analysis should be verified to have precise, unambiguous, mathematically defined syntax and semantics; that is, they are formal notations.”*

*“The soundness of each formal analysis method should be justified. A sound method never asserts that a property is true when it may not be true.”*

We detail below guidelines to address these two objectives.

Is the development model analyzed expressed in a formal notation?

The model (whether a program model or a program source code) should be expressed in a language that has an unambiguous interpretation in logic. In particular, special care should be taken with methods whose model is defined by a natural language description, such as a language reference manual, to make sure that the model is mathematically defined.

The complexity of the model language, the complexity of the logic used, or the gap between the model language and the logic used, can all be sources of ambiguity and inconsistency. When the model language was not primarily designed for formal analysis, it should be restricted to a subset suitable for formal analysis. If the logic is not based on classical foundations (first-order logic, set theory, type theory), its basis should be described formally. If the logic supports partial functions, the mechanisms to ensure soundness should be described formally. When the model language allows expressing operations on state (as in imperative programming languages), and the mechanism used to generate formulas is not a classical calculus (Dijkstra’s weakest preconditions or strongest postconditions), its basis should be described formally.

How is the soundness of the analysis ensured?

guideline: The method should be independently assessed by recognized experts to show that it is an appropriate formal analysis applied on a formal model.

For well-known methods having a historical track record of applicability in theorem proving (for example, B method), the existing peer reviewed scientific and technical literature may be sufficient to perform this assessment. For lesser known methods, or newer methods, a mathematician or computer scientist should be assigned to this assessment.

Note that this does not exclude methods relying on heuristics parts, as far as the analysis is sound. For example, a proof can be derived heuristically by an unsound technique, as long as it is later checked by a sound one.

guideline: The adequation of the method with the verification objective should be assessed.

Some methods have a built-in model of computation, for example synchronous communications. This bias in the method should be consistent with the verification objective, for example a method based on synchronous communications is not adequate for analyzing synchronization algorithms.

guideline: The possibilities for introducing inconsistencies in the method should be identified.

A potential weakness of theorem proving methods is that a single source of inconsistency in the background logical theory used to prove some results may falsify all results. That’s the case when the inconsistent background logical theory allows a “proof of False”: if the formula “False” can be proved, then any other formula “F” can probably be proved by application of the contradiction rule.

Strong typing is a powerful feature to avoid classes of inconsistencies. From [Rushby 1995]:

*“In my opinion, strong typechecking (the stronger the better) should always be required for formal specifications offered in support of certification for safety-critical systems.”*

Even with strong typing, both definitions and axioms introduced by users may lead to an inconsistency.

guideline: The possibilities for validating formal models should be identified.

A formal model is only a representation of the reality. When possible, this representation should be reality-checked, for example by execution or animation of models.

**Choice of Tools**

Each certification standard defines the selection process by which a verification tool can be accepted as a means of compliance with objectives of the standard. For example, this process is called “tool validation” in the railway certification standard EN 50128 and “tool qualification” in the aeronautic certification standard DO-178C. Note that most standards have different requirements for tools that:

1. are only used as verification tools, or
2. are used both as development and verification tools.

Specific issues are relevant when applying this selection process to theorem proving tools.

Limitations of a tool should be taken into account. A tool may be appropriate for the formal analysis of some properties of some projects, and inappropriate for slightly different properties or slightly different projects. For example, a tool that models floating-point computations as computations over mathematical real numbers is not appropriate for any property that depends on floating-point values, if the project uses such values.

guideline: Due diligence should have been performed on the tool to mitigate the risk of inconsistencies.

Rushby’s report enumerates four dimensions along which a given method or tool can be evaluated to measure its potential vulnerability to inconsistencies:[[8]](#footnote-7)

1. Strong typechecking
2. Definitional principle
3. Exhibition of models
4. Modules and parameters

While 1 and 2 lay more with the choice of method, the tool may be more or less effective at detecting inconsistencies.

Elements of this due diligence might include:

* independent auditing, testing and review
* availability of a historical database of bug reports related to inconsistencies found in the tool
* existence of a significant testsuite of both succeeding and failing proofs, on examples that are representative of actual usage

Public availability of the source code for the tool (open-source access) or the binaries of the tool (free access) may provide arguments of independent testing and review.

guideline: Axioms introduced by users of the tool should be subject to additional verifications to ensure that they are consistent with (1) the background logical theory of the tool, (2) other axioms introduced by users, and (3) the reality that they model.

Every axiom introduced adds some risk that an inconsistency arise as a result of the interaction of this new axiom and the (usually large) set of existing axioms from either the tool background logical theory or previous user-defined axioms. Therefore, the availability of this capability in a tool should trigger additional verification.

guideline: Assumptions underlying the validity of the results should be clearly identified.

Theorem proving is always operating at the level of a model of reality. This model relies on assumptions about the reality (for example, the size of machine integers, or restrictions on the instants at which memory can be updated by the outside world) that need to be validated for the results of proof to be representative of reality.

All assumptions should be clearly identified, either at the tool level (for general assumptions) or at the level of the individual use of the tool (for assumptions specific to a specific use).

As Rushby’s report says it:[[9]](#footnote-8)

*“The only difference made by formal, as opposed to informal verification, is that formal verification makes all the assumptions and the requirements explicit, and provides a very strong guarantee that the design satisfies the requirements, subject to the assumptions.”*

The “*Formal Methods Supplement to DO-178C and DO-278A*” document states it very clearly as well:[[10]](#footnote-9)

*“All assumptions related to each formal analysis should be described and justified; for example, assumptions associated with the target computer or about the data range limits.”*

guideline: Artifacts should be generated that support manual review of the proofs.

Ultimately, a mechanical proof of the property of an abstract model of the reality should only be trusted if a human reviewer may convince herself that the proof is “correct” in the sense of a valid use of deduction rules of the system, based on reasonable assumptions (in particular, not based on an inconsistent set of axioms), in a way that corresponds to reality.

From [Rushby 1995] goes even further:

*“In my opinion, the analysis developed with the aid of a theorem prover should also be rendered into a clear and compelling semiformal (i.e., Level 1) argument that is subjected to intense human review, and it is the combination of stringent mechanical and human scrutiny (and other evidence, such as tests) that should be considered in certification.”*

The level of review that Rushby described is more adapted however to proof of algorithms and protocols than to proofs of program models and program source code.

# 

# 

Verification Objectives

*This section will describe common verification objectives that can be assigned to theorem proving, and for each, the usual ways that theorem proving is used to achieve these objectives, as well as the known pitfalls that should be avoided.*

**Absence of Certain Errors**

A classical verification objective is the guarantee that a certain class of errors is absent from the model. Errors are described as a negative property of the model, and theorem proving is used to show that this property does not hold. In a stateful model with a notion of actions that modify the state, errors are typically represented as specific states, or a property of states, and theorem proving is used to show that these states are never reached.

If the state space is huge, and the error states are numerous, it may be difficult to ensure that all possible errors have been considered.

**Refinement**

Theorem proving is well suited to show that a given model is a refinement of another model, in the sense that the more refined model can be considered as a special case of the more general model. Refinement of a specification, A, with respect to the relevant aspects of a higher level of specification, B, can be demonstrated using theorem proving by generating a theorem of the form, Ⱶ Context ∩ Abstractions → (A → B) or a logically equivalent variant of this schema.

For example, theorem proving can be used to demonstrate that the low-level requirements for a system (typically at software level) are a correct refinement of the high-level requirements for the system (typically at the software/system boundary), so that any mixed software/hardware system that respects the low-level requirements automatically respects the high-level requirements.

As another example, theorem proving can be used to demonstrate that a software implementation in a programming language is a correct refinement of the software low-level requirements, typically expressed in a software specification language.

Such a relationship between different levels of specification is likely to depend on certain contextual details such as platform behaviors and intended use. As well, the two different levels of specification may be formally represented at different level of abstractions. For example, the formal representation of the system level behavior might model numerical values as natural numbers (with an infinite range) while the formal representation of the software level behavior might model numerical values as 32-bit values (with a finite range). When the two different levels of specification at represented at different levels of abstraction, it will be necessary to define a correspondence between these two levels of abstraction when using theorem proving to demonstrate refinement of one level of specification with respect to a higher level of specification.

**Exhaustivity of Decomposition**

Theorem proving can be used to demonstrate that a specification is complete in the sense that the specification explicitly specifies the behavior of the system for all possible logical combinations of inputs and internal states, except for combinations that are explicitly excluded. Suppose that each requirement of a specification is formally represented by a logical expression of the form, S → R, where S is the “stimulus” part of a requirement and R is the “response” part of a requirement. Theorem proving could then be used to show that S1 ∩ S2 ∩ … Sn is a tautology. This particular notion of “truth table” completeness is limited in the sense that it only takes account of stimulus part of each requirement. For example, a specification might be complete in this sense even if it fail to specify any requirements for a particular response.

For certain kinds of specifications, it might be possible to use theorem proving to demonstrate more specialized notions of completeness. For example, if the specification includes a formal representation of a finite state machine, theorem proving could be used to prove that every reachable state has a next state for any possible input.

For low-level specifications of software, specifications given as Parnas tables or a set of non-overlapping and complete behaviors are particularly well suited for this kind of formal analysis.

**Non-Functional Properties**

Theorem proving may be applied to compute or verify non-functional properties of software, like Worst-Case Execution Time, stack usage, memory or spatial separation. The main difficulty here is in matching the model of the host platform with the actual behavior of the processor / operating system.

# 

Verification of Programs

*This section will describe how theorem proving can be applied for verifying properties of programs, given either in a textual programming language, or using a textual or graphical modeling language.*

Application of theorem proving to programs requires special attention, due to the usually large language for models on one side, and the possible semantic variations for execution on different platforms on the other side. Additionally, properties proved at the level of a program abstraction such as source code or modeling design does not necessarily hold at the level of the executable code. Special care is required to ensure that the code generator (for modeling languages) and the compiler preserve the semantics of the program.

**Addressing the Language Issue**

It is important to demonstrate that the language of models is unambiguous. it should be easier in the cases where the primary purpose of the language of models is formal analysis (for example, the B Method). In other cases, a subset of the language suitable for formal analysis should be defined.

Common pitfalls to address that may introduce ambiguity are: pointer aliasing, access to uninitialized data, read/write or write/write interferences during expression evaluation.

**Addressing the Execution Platform Issue**

The semantics of programs is likely dependent on specificities of the execution platform, such as the size of the standard numeric types (integers, floating-point types, etc.) or endianness (big-endian vs. little-endian).

One way to address this variability is to make sure that formal analysis does not depend on these specificities, which restricts formal analysis to those properties that are invariant across execution platforms.

Another way to address this variability is to document precisely the assumptions on which formal analysis depends that are related to the execution platform, so that these assumptions can be independently validated.

**Addressing the Compilation Issue**

The natural way to is to demonstrate that properties established at the source level still hold at the binary level is to show that requirements at source-code level are traceable down to the object-code level. Demonstrating traceability between source and object code is greatly facilitated by using qualified tools for purposes such as enforcing coding restrictions against features that would complicate traceability, by applying appropriate compiler options to preserve control flow, and by using code traceability analyses prepared by compiler vendors.

Assuring the correctness of the compiler’s translation of source code into object code is, of course, important. Trust can be based on examination of the compiler itself (the tool qualification process) or the compiler’s output. The former approach (qualifying the compiler) is rare because of the effort involved. The latter approach provides the relevant degree of assurance through the multiple and overlapping activities required by certification standards, including for example the hardware/software integration testing and the verification of untraceable object code.

**Alternative Objectives to Coverage**

To increase confidence in the comprehensiveness of testing-based verification activities, some certification standards requires coverage analysis. Test coverage analysis is a two-step process that involves requirements-based and structural coverage analyses. Requirements-based coverage establishes that verification evidence exists for all of the software’s requirements — that is, that all the requirements have been met. This also applies to formal verification. Structural coverage analysis during testing (for example, statement coverage) aims to detect shortcomings in test cases, inadequacies in requirements, or extraneous code.

Structural coverage analysis doesn’t apply to formal verification. Instead, alternative objectives should be defined to achieve the same three goals as structural coverage (substituting verification cases for test cases in the first one). For example, DO-178C’s supplement on formal methods, DO-333, defines four alternative activities to reach the structural coverage goals when using formal verification:

1. Cover: Detect Missing Verification Evidence

Unlike testing, formal verification can provide complete coverage with respect to a given requirement: it ensures that each requirement has been sufficiently — in other words, mathematically — verified. But unlike testing, formal verification results depend on assumptions, typically constraints on the running environment, such as the range of values from a sensor. Thus, all assumptions should be known, understood, and justified.

1. Complete: Detect Missing or Incomplete Requirements

Formal verification is complete with respect to any given requirement. However, additional activities are necessary to ensure that all requirements have been expressed — that is, all admissible behaviors of the software have been specified. This activity states that the

completeness of the set of requirements should be demonstrated with respect to the intended function:

* *“For all input conditions, the required output has been specified.”*
* *“For all outputs, the required input conditions have been specified.”*

Checking that the cases don’t overlap and that they cover all input conditions is sufficient for demonstrating the first bullet point. Furthermore, it’s easy to detect obvious violations of the

second point by checking syntactically that each case explicitly mentions each output. A manual review completes this verification. Note that formal methods can’t handle the more general problem of detecting all missing requirements.

1. Dataflow: Detect Unintended Dataflow

To show that the coding phase didn’t introduce undesired functionality, the absence of unintended dependencies between the source code’s inputs and outputs must be demonstrated. You can use formal analysis to achieve this objective. Formal notations exist to

specify dataflows, such as the SPARK dataflow contracts or the Fan-C notation in Frama-C, and associated tool automate the analysis.

1. Extraneous: Detect Code That Doesn’t Correspond to a Requirement

DO-178C requires demonstrating the absence of “extraneous code”: any code that can’t be traced to a requirement. This includes “dead code” as defined in DO-178C: code that’s present by error and unreachable. The relevant section of DO-333 explicitly states that

detection of extraneous code should be achieved by “review or analysis (other than formal).” Although formal analysis might detect some such code, computability theory tells us that any practical formal analysis tool (which doesn’t generate so many false alarms that it’s useless in practice) will be unsound, meaning it will fail to detect some instances of extraneous code. DO-178C doesn’t allow unsound tools. The effort required by this review or analysis depends chiefly on the degree of confidence obtained after completing the previous activities (cover, complete, and dataflow). Testing detects extraneous code as code that isn’t executed at runtime. This step detects both unreachable code that can never be executed and unintended functionalities — those that could be executed but aren’t triggered by the tests derived from requirements. When you use formal analysis, the previous activities give some degree of confidence that unintended functionalities can be detected. It only remains to detect by review or analysis the unreachable code. Because this is a manual activity, its details vary from project to project.

**Combining Evidence from Theorem Proving and Test**

Formal analysis is almost never applied to the entire program, posing a potential soundness risk if some assumptions are not verified. Typically, parts of a program that are not formally analyzed are verified through testing, with special care to ensure that the assumptions of formal verification are met.

Such a combination of static and dynamic analyses make most sense when the boundaries between parts of the program that are formally analyzed and parts of the program that are tested are precisely defined, and when assumptions are managed automatically to ensure none is lost in the combination.

Different kinds of combinations can be envisioned:

* modular - on different parts (prove module A, test module B)
* cumulative - on same parts (prove property A, test property B)

# 

# 

Artifacts for Certification

*This section will describe what artifacts from theorem proving are expected for certification.*

**Limitations of the Approach**

As a model is only a representation of the reality, the limitations of the model should be precisely defined, in a way that an auditor could easily detect in which cases the model does not apply.

**Evidence from Theorem Proving**

There should be a clear correspondence between requirements of the software and verification results. In particular, evidence is required that all requirements have indeed been verified, and that each verification result corresponds to a requirement. It should be avoided to have many requirements correspond to a single verification result.

Any additional assumptions used for the verification must be justified and their impact assessed. A possible way to detect incorrect assumptions is to slightly modify requirements so that they should not be verifiable any more, and check the verification output.

A full deductive proof is a possible form of verification result, but it is not required if the adequateness of the employed formal method and tool has been demonstrated.

**Validation of Evidence**

The tool documentation should make clear what exactly the tool can verify and the verification results should make clear what exactly has been verified. The specifications should be understandable by the certification authority - natural language comments may help to provide explanations/rationales to formal requirements.

Possible ways to get more confidence in verification results are to express the same property in different but equivalent ways, or to try to derive expected properties from the stated requirements.

Rushby’s report says:

*Proofs of significant theorems should be subjected to human review as well as mechanical proof checking. [...] Not all theorem-proving systems work in ways that resemble the steps of a journal-style proof, and for some purposes may be considered unacceptable on that account. Note that what are required are steps that resemble those of traditional mathematical demonstrations, not the primitive steps of textbook*

*treatments of formal deduction.*

An important distinction here is between “significant theorems” and those that are not, like the many theorems required to proof absence of run-time errors in a program.

# 

# 

# Example Applications

*This section will describe example applications of theorem proving, both toy-like (for example, the Nose Gear Challenge) and real ones, and show how the guidelines given in this document were followed.*

**Nose Gear Challenge**

The Nose Gear challenge is a toy system description meant to serve as a vehicle for comparing formal analysis approaches in a certification context. The system describes the computation of the velocity of the front wheel of a plane. Two high level requirements are stated for the system:

* When the computed velocity is available, it shall  be  within  3  km/hr  of  the  true  velocity  of  the  aircraft  at  some  moment  within  the  past  3  seconds.
* Under normal conditions, the computed velocity should be available.

The description of the challenge can be found here:

<http://www.cl.cam.ac.uk/~mjcg/FMStandardsWorkshop/NoseGear.html>

Partial solutions were presented during the 2nd and 3rd editions of the Workshop on Theorem Proving in Certification.

**Pacemaker Formal Methods Challenge**

The medical device company Boston Scientific, one of the leading provider of pacemakers, has released into the public domain the system specification for a previous generation pacemaker. A major reason for publishing this specification is to have it serve as the basis for a challenge to the formal methods community.

The description of the challenge can be found here:

<http://sqrl.mcmaster.ca/pacemaker.htm>

# 

# Conclusion

# 

# 

# Bibliography

[Bowen 1993] Jonathan Bowen, *Formal Methods in Safety-Critical Standards*, Software Engineering Standards Symposium, 1993

[Brown 2010] Duncan Brown et al., *Guidance for Using Formal Methods in a Certification Context*, presented at Embedded Real-Time Systems and Software, 2010

[Heitmeyer 2009] Constance L. Heitmeyer, *On the Role of Formal Methods in Software Certification: An Experience Report*, Electronic Notes in Theoretical Computer Science, Volume 238 Issue 4, September, 2009

[Ledinot 2014] Emmanuel Ledinot et al., *Joint use of static and dynamic software verification techniques: a cross-domain view in safety critical system industries*, presented at Embedded Real-Time Systems and Software, 2014

[Moy 2013] Yannick Moy et al., *Testing or Formal Verification: DO-178C Alternatives and Industrial Experience*, IEEE Software 2013

[Rushby 1993] John Rushby, *Formal Methods and the Certification of Critical Systems*, Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, 1993

[Rushby 1995] John Rushby, *Formal Methods and their Role in the Certification of Critical Systems*, Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, 1995

Annex 1: Issues in Tool Selection

Rushby’s report gives in 3.5 *Selection of Formal Methods, Levels and Tools* the following list of issues to consider in the selection of a formal system, presented as questions, organized in categories. We only list here the questions, see the report for the accompanying rationale and explanations. Issues that are relevant for certification are displayed in **bold** font.

Verification System

* Does the system have adequate documentation and examples?
* Has the system been used on real problems?
* Is it easy to learn? And does it provide effective support for experienced users?
* Does the system support the selected formal methods and analyses effectively?
* Is the system generic or specific to a particular logic and language?
* Does the system support the implementation language concerned?

Specification Language

* **Does the language have an explicit semantics?**
* Is the logic underlying the specification language at least as expressive as first- order predicate calculus, or is it a more specialized logic?
* Does the language have computer-science constructions (such as records, tuples, enumerations, updates)?
* Does the language have familiar and convenient syntax (e.g., infix + etc.)?
* **Is the specification language strongly typed? How rich is the type-system, and how stringent is the error-checking performed by the typechecker?**
* **How does the logic deal with partial functions?**
* **Does the specification language allow formulas to be introduced as axioms?**
* **Does the specification language have a definitional principle that guarantees conservative extension? Is it mechanically checked?**
* Does the system support definition of recursively defined abstract datatypes?
* Are specifications required to be model-oriented, or property-oriented, or can both styles be accommodated?
* Does the language have overloading and type inference?
* Are specifications purely functional, or expressed in terms of pre- and post- conditions on a state, or can both styles be accommodated?
* Does the specification language have an encapsulation or modularization mechanism? Can modules be parameterized? **Can semantic constraints be placed on the instantiations of parameters? How are the constraints enforced?**
* **Does the language have a built-in model of computation?**
* **Is the specification language executable, or does it have an executable subset, or are there some capabilities for animating specifications?**
* Is there support for state exploration, model checking, and related methods?

Utilities

* Does the Formal Method have a comprehensive library of standard types, functions, and other constructions? **How well validated is the library?**
* Does the system provide editing tools? Are they free-form or structure based, or both?
* **Does the system support reviews and formal inspections?**
* **Does the system have facilities for producing nicely typeset specifications, or for presenting specification in the form of tables, or diagrams?**
* **Does the system provide facilities for cross-referencing, browsing, and requirements tracing?**
* Does the system record the state of a development (including proofs) from one session to the next, so that work can pick up where it left off?
* Does the system support mechanisms for change control and version management?
* **Does the system propagate the consequences of changes to the specification appropriately?** At what granularity are changes recognized? Are updates performed incrementally?

Theorem Prover

* Does the system allow lemmas to be used before they are proved?
* Does the system allow new definitions to be introduced during proof? Does it allow existing definitions to be modified?
* Does the theorem prover allow cases to be postponed, and to be tackled in any order?
* Does the theorem prover provide automated support for proofs by induction?
* Does the theorem prover provide facilities for comprehending the overall structure of a proof?
* **Does the system identify all the axioms, definitions, assumptions and lemmas used in the proof of a formula (and so on recursively, for all the lemmas used in the proof)?**
* **Does the theorem prover provide information that will help construct a human- readable journal-style proof?**
* Is it easy to reverify a theorem following slight changes to the specification?
* How is the theorem prover controlled and guided?
* Does the theorem prover perform rewriting?
* Does the theorem prover provide automated support for arithmetic reasoning?
* Can the theorem prover handle large propositional expressions efficiently? Does it employ BDDs?
* Does the theorem prover present users with their own formulas or with canon- ical representations? Are quantifiers retained?
* Does the theorem prover minimize the quantity and maximize the relevance of information presented to the user?
* Does the theorem prover provide automated support for instantiation of quantified variables?
* Can proofs be “cut and pasted” from one formula to another?
* Does the theorem prover allow the user to compose proof steps into larger ones?
* Can users extend the capabilities of the theorem prover with their own code? **Does the theorem prover have a “safe” mode that performs only the built-in inference procedures?**

1. FM.1.6 Characteristics of Formal Methods [↑](#footnote-ref-0)
2. Rushby’s report, section 2.1 Level of Rigor in Formal Methods [↑](#footnote-ref-1)
3. Rushby’s report, section 2.2.1 Formal Methods in the Lifecycle [↑](#footnote-ref-2)
4. Rushby’s report, section 2.1 Level of Rigor in Formal Methods [↑](#footnote-ref-3)
5. Mostly taken from [Ledinot 2014]

   [↑](#footnote-ref-4)
6. FM.6.2.1 Considerations for Formal Methods [↑](#footnote-ref-5)
7. FM.6.2.1 Considerations for Formal Methods [↑](#footnote-ref-6)
8. Rushby’s report, section 2.3.1 Internal Consistency [↑](#footnote-ref-7)
9. Rushby’s report, section 2.5.2 Fallibilities of Formal Verifications [↑](#footnote-ref-8)
10. FM.6.2.1 Considerations for Formal Methods [↑](#footnote-ref-9)