Technical Report

Improvement of the Applicability of Formal Methods of the Functional Safety Base Standard IEC 61508-3 (DIN EN 61508-3, VDE 0803-3)

Dr.-Ing. Bernd Sieker

Table of Contents

1 Motivation

There is a wealth of literature saying that the requirements are where most errors arise in critical systems. It is therefore appropriate that this study focus on improving specification and analysis of requirements. In particular, since there is very little on requirements analysis in the current IEC 61508:2010. The 914.0.3 committee has observed the lack of traceability assurance for safety requirements from preliminary hazard analysis through to software safety requirements in the current IEC 61508:2010. The authors agree that consistency and completeness are important. Any reservations are confined to practical matters of whether tools are adequate and what to recommend. The former is checkable; the latter is relative to one's choice of completeness criteria (such as Heimdahl/Leveson and Ladkin). Neither criterion is required to be checked in IEC 61508:2010. We propose that clauses be added to IEC 61508 in the forthcoming maintenance cycle to require checking of safety requirements for consistency and relative completeness, along with an informational part (for Part 7) which indicates how this may be done, with aid of available tools.

According to [Ladkin FDLaffidavits] there seems to be almost unanimity amongst discussion contributors to the System Safety List that use at some point or other of a system description with an unambiguous semantics is essential in the development of safety-critical system software. Most of the contributors also think that such a description is most appropriately formulated in a formal language with such a semantics (there are, of course, formal languages without, such as full Ada, C, Java). But even those who said "Natural English" require an unambiguous interpretation, that is, a form of Controlled English. German rail law requires that requirements are expressed in natural language - as far as I know there is no express legal requirement for non-ambiguity. We recommend that 914.0.3 argue for requiring use of a formal description language with unambiguous semantics (Controlled Natural Language is one such) in the MT.

2 The Current State of IEC 61508-3

IEC 61508 is the base standard for the functional safety of electric, electronic and programmable electronic systems (E/E/PES). it covers the life cycle of these systems. Today the major part of the functionality of such systems is implemented in software. Part 3 of the IEC 61508 deals with software requirements.

Annex A (normative) and B (informative) contain tables listing various techniques and measures, and is supposed to provide a guide to the selection of such techniques for different safety integrity levels (SIL). It lists general categories and gives different levels of recommendation for these, such as "not recommended", "recommended" or "highly recommended". In the following sections it lists more specific techniques for various phases of software development.

These "techniques and measures" are a mix of overly-generic and overly-specific labels. "Formal Methods" on the one hand is a very broad term referring to the use of mathematical and logical methods for assessing, designing and verifying software. There are a large number of such formal methods, aimed at various phases of the development. Methods are available for the specifying requirements, for the assessment of the design, for checking source code and object code and the derivation of test suites.

On the other hand, a measure such as "Limited use of pointers" or "Time Petri nets" refer to the avoidance or use of very specific and sometimes programming-language dependent techniques.

Annex C contains a much longer list of techniques and measures and details the effectiveness of these in achieving certain properties, to an informally defined rank of "Rigour" (R1 being the least, and R3 being the most rigorous), and gives guidance which rigour should be targeted at which SIL.

According to the current definitions, Software-SILs are measures of the rate of dangerous failures caused by software behaviour. It is believed by many members of the national committees that IEC 61508-3 sets no requirements on achievement, or demonstration of achievement, of specific software SILs, because it is not possible to evaluate through testing SW SIL requirement above SIL 1. This leaves the use of recommended methods during development as the only requirement.

It is not initially clear what the connection is between the specific methods listed in Annexes A and B and the quality of the software. Some companies have robust data that clearly demonstrates the delivered code quality, and the improvement of that code quality. Besides the methods used for creating such code quality metrics, a certain corporate culture is also used as an input, as well as personnel continuity. While the methods used can certainly contribute to the quality of the software, using them alone cannot be used to consistently achieve a high confidence of very high software quality.

As such, Part 3 software requirements, by recommending methods and applying various levels of rigour do not generally achieve the objective of Software SILs. Since it does not require any demonstration of software system properties, it is not optimal in assuring high quality software. It may lead to a "box-ticking" mentality when developing software, nominally using as many of the "highly recommended" methods as possible and then delivering software that pro forma fulfills the requirements for the SIL for which it is going to be used.

3 Alternative Suggestion

According to Document DKE-914_0.3_2010_0001, we propose that instead of the extensive Tables in Annexes A through C, there be only the recommendation to use "Formal Methods" for all SILs in Annex A.

For Annex B we propose a supplementary list of the following 26 "Formal Methods".

We will here list these 26 entries to existing and known methods, techniques and tools that are in use in the industry and are considered best practice. The known methods will be described shortly and for each we shall list the software properties (1-26) that can be achieved by using the methods. As well as a generic description of such industry techniques we will also list known specific tools that have been shown to achieve high quality software.

An alternative form of the tables in Annex B of the DIN EN 61508 (VDE 0803) could be similar to the suggestion in Annexes A and B of this document. This is a preliminary suggestion and subject to discussion in the standardisation committees, as to how and in what form which parts of these tables should be modified / extended and which parts should be suggested by the national committee for a new revision of DIN EN 61508 (VDE 0803) via the normal commenting method.

3.1 List of "Formal Methods"

The development according to the waterfall model, which is still useful for conceptual purposes, can be shown roughly as follows:

Requirements Specification → Design → Coding → Testing → Deployment

This process can be refined, and quite often this will be useful, such as separate unit and integration testing, or more steps in software development, which this progression does not show. This does also not show partitioning of software components in the development of simultaneously active components of a software-based system. The proposed following methods and techniques roughly follow a more refined version of the above progression.

1. **Formal functional requirements specification (FRS)**

It is important that such requirements specifications fulfil certain characteristics, which are much easier to ascertain when they are formulated in a formal language as opposed to natural language. These requirements specialisations (FRS) have to be consistent (free of contradictions), (relatively) complete.

Known methods for this include:

- Formal Description Language
- Controlled Natural Language
- Semi-Formal Graphical Language
- Formal Graphical Language
- Formal Logic

2. **Formal FRS analysis**

When the requirements specialisations are in a formal language, they can be formally analysed for consistency, and completeness.

Known methods for this include:

- Automated Consistency Checker
- Theorem Prover

3. **Formal safety requirements specification (FSRS)**

It is important, that in addition to the safety requirements, the safety requirements also be specified formally, so that in later stages of the development, their fulfilment can be proven mathematically. Unlike the functional requirements ("what is the software supposed to do?") the safety requirements in many cases specify what the software must not do. Methods include:

- Formal Description Language
- Formal Notation Language
- Controlled Natural Language
- Formal Logic
- Ontological Hazard Analysis

4. **Formal FSRS analysis**

Just like the functional requirements, the safety requirements also have to be checked with formal methods for their consistency and completeness. Methods to do this include

- Automated Consistency Checking
- Theorem Prover / Proof Checkers

5. **Automated proving/proof checking of properties (consistency, completeness of certain types) of FRS and FSRS**

This is partially covered by 2. and 4., but includes the relative consistency of both functional and safety requirements specifications, so that no parts of either contradict any parts of the other. Methods for this include mostly the same techniques as listed under 2. and 4.:

- Automated Consystency Checking
- Theorem Prover / Proof Checkers

6. **Formal modelling, model checking, and model exploration of FRS, FSRS**

This is another way for analysing and checking the formal specifications, but in addition to checking for consistency and completeness, model checking and/or model exploration can also give information of possible liveliness problems with the requirements specifications. For instance, a system for air traffic control where no flights at all will fly, can be consistent and complete, but it will not be satisfactory.

- Modelling Language
- Formal modelling of functions
- Hierarchical Modelling
- Model Checker
- Simulation

7. **Formal design specification (FDS)**

Unlike the functional and safety requirements specifications, the design specifications describe how the system is going to be designed in order to implement ("fulfil") the requirements (FRS and FSRS). Methods for this include

- Graphical Design Language
- Formal Description Language
- Formal Logic
- Formal Refinement

8. **Formal analysis of FDS**

The resulting formal design specification has to be checked, just like the requirements specifications and the safety requirements specifications. Similar tools can be used for these analyses:

- Automated Consistency Checking
- Theorem Prover / Proof Checkers

9. **Automated proving/proof checking of fulfilment of the FRS/ FSRS by FDS**

To enable a complete traceability of the requirements, in order to be confident that the final product fulfils the initial requirements it has to be proven formally that the formal design specification does indeed implement those requirements. Various formal tools and methods can be used to achieve this, including

- Formal Refinement
- Theorem Prover
- Proof Checker
- Model Checking

10. **Formal modelling, model checking, and model exploration of FDS**

As with the higher level previous specifications, besides the fulfilment of the functional requirements and the safety requirements, the formal design specification needs to be shown to fulfil liveness requirements. The following methods can be used to achieve this

- Modelling of Functions
- Hierarchical State Machines
- Modelling using petri-nets
- Model Checking

- Model State Exploration

11. Formal determininistic static analysis of FDS (information flow, data flow, possibilities of run-time error)

A static analysis of the formal design specification at this point can already show problems with data flow and information flow or possibilities of some times of run-time errors. Methods include

- Static data flow analysis
- Static information flow analysis
- Static Analysis for runtime errors

12. Codevelopment of FDS with Executable Source-Code Level (ESCL)

Source-Code can be developed together with the formal design specification if a suitable language is used. The design specification can have the form of annotations in the source-code

- Codevelopment Tools for software source-code
- Languages allowing simultaneous development of source code and formal design specifications

13. Automated source-code generation from FDS or intermediate specification (IS)

By using source code automatically generated from a formal specification, multiple error sources are eliminated. Manual coding errors. The validation of the resulting software against the specification becomes much easier. Tools for performing this include:

- Automatic Code Generators from textual formal design specifications
- Automatic Code Generators from graphical design specifications

14. Automated proving/proof checking of fulfilment of FDS by IS

Automated proving or assisted checking of proofs will ensure that the intermediate specification (if any) implements (fulfils) the formal design specification. Methods for this include

- Automated Proof Checker
- Automated Theorom Prover

15. Automated verification-condition generation from/with ESCL

When using a language with design specifications, verification conditions can be generated automatically by using appropriate tools, such as

- Automated Theorem Prover
- Proof Checker

16.Rigorous semantics of ESCL

It is important to use a programming language for writing the source code that has anambiguous semantics. This is not true for all languages; in many languages, the semantics depend on the compiler or the operating systems or other parameters of the environment. Sometimes a programming language subset with rigorous semantics can be used.

- Programming language with rigorous semantics
- Programming language subset with rigorous semantics

17.Automated ESCL-level proving /proof checking of properties (such as freedom from susceptibility to certain kinds of run-time error)

By using automated static code analysis tools, freedom from certain kinds of runtime errors and data-flow and informatio-flow problems can be proven. Tools for achieving this include

- Static Code analysis tools

18.Automated proving/proof checking of fulfilment of FDS by ESCL

By using a programming language which allows a definition of pre-and post conditions for each function in the software source code, automatic tools can be used to prove that the source code fulfils (implements) the formal design specification. Such tools include

- Automated Proof Checker

19.Formal test generation from FRS

Formal tests can be generated from the functional requirements specification, against which the executable sotware can be tested. These serve to validate that the development process works, but are not supposed to prove correctness.

- Automatic test generators

20.Formal test generation from FSRS

Similarly to the test generation from the functional requirements specifications, tests can be generated automatically from the formal safety requirements specifications. As with the functional requirements specifications, these tests are not supposed to find errors, but to demonstrate that the software works in some cases. A development using formal methods ensuring traceability of the requirements will then ensure that it works correctly in all cases.

- Automatic test generators

21.Formal test generation from FDS

The same is true for format test generations from the formal design specifications, against which the executable code can then be tested to validate the development.

- Automatic test generators

22. Formal test generation from IS

If an intermediate specification was used, this can also be used to generate tests for validating the executable software.

- Automatic test generators

23. Formal test generation from ESCL

To validate the compiler and linker and the execution environment, tests can be generated from the source code.

- Automatic test generators

24. Formal coding-standards analysis (SPARK, MISRA C, etc)

Formal codings standards should be used. If it is impractical or impossible to use a language with unambiguous semantics (see point 16), adhering to formal (and ideally verifiable) coding standards can assure avoidance of many problematic software constructions.

- Coding Standards
- Coding Standards Analysis

25. Worst-Case Execution Time (WCET) analysis

For timing-critical software, an analysis of the worst-case execution time should be performed to ensure that time limits in a real-time application will never be exceeded.

- WCET-analysis tools

26. Monitor synthesis/runtime verification

In order to verify correct function of the software, formal requirements can be used to create monitors and include them into software as an additional safeguard against malfunctions.

- Monitor synthesis / runtime verification

4 Abstract Methods and Industrial Implementations

The methods listed under each of the 26 points in section 3.1 are abstract methods that do not mention any specific implementation. This section will list known variants that are in use in the industry with some short descriptions for each.

1 Formal Description Language

For the creation of requirements specifications that will enable the two essential properties of being formally checkable for consistency and completeness, as well as enable proofs of later implementation stages fulfilling the requirements, it is essential that the requirements specification be written in a formal language.

A number of industrial formal description languages have been in use which enable such a formal notation of specifications.

1. Z notation

The Z notation is a formal specification notation based on set theory and predicate calculus.

It is a formal (i.e., mathematical) specification notation used in high-integrity systems as part of the software development process. Among others it was used in the following projects:

- The development of the software for the Radio Therapy Machine of the Department of Radiation Oncology at the University of Washington used Z-notation: http://staff.washington.edu/jon/cnts/cnts-report.pdf
- Z was also used in the specification of parts of the IBM Customer Information Control System (CICS): King, Steve (1993). "The Use of Z in the Restructure of IBM CICS". In Hayes, Ian. Specification Case Studies (2nd ed.). New York: Prentice Hall. pp. 202–213. ISBN 0-13-832544-8.
- The Z-notation was used for the development of a large part of safety-critical software of the Airbus A330/A340 series. : ZUM '95: The Z Formal Specification Notation: 9th International Conference of Z Users, Limerick, Ireland, September 7 - 9, 1995. Proceedings, p. 162.

A number of software support tools are available for creating a specification in the Z-notation:

- The Z Word Tools allow Z specifications to be written in Microsoft Word.
- Moby/OZ, a graphical editor to build Z and Object-Z specifications.

2. Abstract Machine Notation (B Method)

Created by the author of the Z notation, the Abstract Machine Notation is formal description language. Its best-known application:

- Météor, the first driverless Metro line in Paris: http://www.tcs.hut.fi/Studies/T-79.5303/2008SPR/paris_metro.pdf

Tools are also available for using an formal description using this method:

- Atélier B, the original support toolkit by the authors of the B method: http://www.atelierb.eu/
- Event-B and the Rodin platform offers free software developed in the DEPLOY EU project: http://www.event-b.org/

3. VDM-SL / VDM++

The Vienna Development Model Specification Language is another formal specification language that has been used and found to be useful in a number of cases:

- http://www2.imm.dtu.dk/courses/02263/F07/Files/DTU%20VDM%20Technology%20in%20Industry%202007.ppt lists several examples of industrial use, e. g. British Aerospace (Security: Gateway), Adelard (Safety: Dust Explosives), CS-CI, France for Space

A number of tools are also available to create specifications in VDM-SL/VDM++:

- The Overture Tool is a free development toolkit for the entire VDM, including tools for writing in VDM-SL and VDM++

4. CASL

CASL is the specification language of the Common Framework Initiative (CoFI) for algebraic specification and development. CASL is an expressive, simple, pragmatic language suitable for specifying requirements and design for software:

- http://www.informatik.uni-bremen.de/cofi/index.php/CASL

2 Controlled Natural Language

It is tempting to write requirements in natural language; in fact some public authorities may require the use of natural language. However it is not precise enough to write a specification on which to base a formal development of a software design and implementation. Natural language always contains a large number of ambiguities. One possible way to attempt to get the best of both worlds may be to create rigorous semantics for a language based on natural language words and a grammar that will allow sentences that resembles natural language. If carefully created, such a controlled natural language may be precise enough to support formal development, and yet be understandable with minimal training by human practitioners. Examples of a controlled natural language are few.

1. ASD Simplified Technical English

This is a standard by The AeroSpace and Defence Industries Association of Europe (ASD) to standardise the language of technical documentations, and remove ambiguities. This is achieved by creating a simplified grammar and style rule and defining only a limited vocabulary with narrowly defined meaning:

- http://www.asd-ste100.org/

It is supported by the Boeing Simplified English Checker to check the document for violations of the STE standard:

- http://www.boeing.com/boeing/phantom/sechecker/

3 Formal and Semiformal Graphical Languages

If they are backed up by rigorous semantics, using a graphical design language can be helpful to make the requirements specification, or parts of, easier to understand. A number of such methods have been developed.

1. Petri-Nets

This technique has been developed to model systems with concurrency. It has been used successfully in the industry:

- Ericsson has used it to validate IPv6 protocol implementation: L.M. Kristensen, J.B. Jørgensen and K. Jensen: Application of Coloured Petri Nets in System Development. Lectures on Concurrency and Petri Nets - Advanced in Petri Nets. Proc. of 4th Advanced Course on Petri Nets. Vol. 3098 of Lecture Notes in Computer Science, pp. 626-685. Springer-Verlag, 2004.
- It was used to develop software to supervise electronic funds bank transfers: V.O. Pinci, R.M. Shapiro: An Integrated Software Development Methodology Based on Hierarchical Colored Petri Nets. In: G. Rozenberg (ed.): Advances in Petri Nets 1991, Lecture Notes in Computer Science Vol. 524, Springer-Verlag 1991, 227-252. Also in K. Jensen, G. Rozenberg (eds.): High-level Petri Nets. Theory and Application. Springer-Verlag, 1991, 649-667.

A range of software tool support is also available. http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html has a large overview, including:

- CPN Tools is a tool for editing, simulating, and analyzing Colored Petri nets: http://cpntools.org/
- The ORIS graphical petri net editor: http://www.oris-tool.org/
- Mathwork's Petri Net Toolbox: http://de.mathworks.com/products/connections/product_detail/product_35741.html

2. UML

While not really a formal method in itself, the Unified Modeling Language offers the possibility of visualising a design graphically, but it lacks unambiguous semantics. It can be a useful addition, if used in conjunction with other tools and a rigorous definition of the semantics of the diagram components and their combination. UML has been used in one form or another in many projects. Success has been reported, e. g. when combining UML with a more rigorous notation:

- http://eprints.soton.ac.uk/260169/2/UMLBeprint.pdf
- http://www.scirp.org/journal/PaperDownload.aspx?paperID=19732

Tool support is also widely available

- Sparx Systems Enterprise Architect includes a UML editor: http://www.sparxsystems.com/products/ea/index.html
- Altova UModel is another industrial UML editor: http://www.altova.com/umodel.html
- Software Ideas Modeler: http://www.softwareideas.net/
- OpenModelSphere is free software that also offers support for editing UML: http://www.modelsphere.org/index.html

4 Formal Logic

This is not really a method in itself, but the underpinning of most formal methods. However, as well as being the foundation of many formal description languages, logic expressions in predicate logic or linear temporal logic can also be used to describe requirements. These are then easily checked for consistency.

5 Finite State Machines

Although also a mathematical concept, state machines lend themselves to visualisation more than many other techniques. There have also a rigorous semantics available for a long time, and their construction is well understood. Many commercial enterprises have used state machines in early phases of their design. Almost all software development methodologies these days include some sort of finite state machine description at some point, as do most successful development stories. There is widespread tool support, both free software, stand-alone commercial software and as part of integrated development systems.

6 Automated Consistency Checking

Specifications written in a formal language can be checked automatically for consistency. Consitency is very important, because it is impossible to create a correctly functioning program from contradictory requirements. Some research has been done on the practical applications:

- Constance L. Heitmeyer, Ralph D. Jeffords, And Bruce G. Labaw: Automated Consistency Checking of Requirements Specifications, Part of the "Software Cost Reduction" Program by the US Naval Research Laboratory—Washington, D.C.
- Martin Ouimet, Kristina Lundqvist: Automated Verification of Completeness and Consistency of Abstract State Machine Specifications using a SAT Solver in *Electronic Notes in Theoretical Computer Science* 190 (2007) 85–97

Almost all development environments and tools which support a formal description language will also offer tools to check such formally specified specifications with automated tools.

Examples of a Z checker are:

- FUZZ: http://spivey.oriel.ox.ac.uk/mike/fuzz/

There is a formal consistency checker for CASL, the specification language of the CoFI:

- CASL consistency checker: http://www.informatik.uni-bremen.de/cofi/ccc/

7 Automated Therom Prover / Proof Checker

Automated theorem provers / proof checkers can be used in many phases during software development. They can be used to prove that the formal design specification implements the requirements specification, or that the source code implements the design specification. This requires formal definitions on variosu levels and the refinement of a higher-level specification into a lower-level specification or implementation to be done carefully. A formal proof can then find problems with the implementation that are sometimes too subtle or too complex to be found reliably by humans.

Many available toolkits allow automated or guided theorem proving and proof checking, but to use it for source code level requires the use of a sufficiently rigorous language with unambiguous semantics (see 15).

Examples of theorem provers are

1. **Isabelle,**
   a highly configurable proof assistant, free under a BSD-style license:
   - http://isabelle.in.tum.de/
2. **PVS**
   The Prototype Verification System
   - http://www.csl.sri.com/projects/pvs/

   PVS has been used, among other things to verify the design of a commercial microprocessor
   - http://www.csl.sri.com/papers/chdl95-ms/

8 Formalised Hazard and Risk Analysis

To create a formal safety requirements specification, it is necessary to have an at least somewhat formalised way of doing a hazard analysis to come up with the safety requirements. Requirements elicitation is a challenging process, and all aspects of it are beyond the scope of this document, but various techniques such as HAZOP, FME(C)A, Fault Tree Analysis, Even Tree Analysis and others may be used single or in combination to help finding the hazards and thus safey requirements. A formalised version of starting. One technique is based on very high-level abstract specifications and using formal refinement to elicit safety requirements:

1. **Ontological Hazard Analysis**
   This method has been shown to be useful to elicit safety requirements, prove relative correctness and ensure traceability down to object code.
   - Dissertation by Dr. Bernd Sieker: http://pub.uni-bielefeld.de/publication/2306191
   - OHA of a communications bus: http://www.causalis.com/90-publications/OHA-CommBus.pdf

   There are common standards for other techniques:
   - HAZOP: http://webstore.iec.ch/webstore/webstore.nsf/Artnum_PK/26991
   - FMEA: http://webstore.iec.ch/webstore/webstore.nsf/ArtNum_PK/35494

9 Formal Modelling Language / Model Checking

To do model checking of a formal description it needs to be described in a suitable language. Most model checkers and model exploration software have their own model description language.

1. ProMeLa / SPIN

The Process Modelling Language of the model checker SPIN, which allows the modelling of concurrent programs.

- http://spinroot.com/spin/whatispin.html

SPIN has been used to verify a number of mission-critical software projects, such as:

- Call processing of a commercial data switch: http://spinroot.com/gerard/pdf/bltj2000.pdf
- Verification of medical device data transmission protocols: http://www.rug.nl/news/2011/01/emergingtechnologyaward

2. Alloy

is a language for describing structures and a tool for exploring them. It has been used in a wide range of applications from finding holes in security mechanisms to designing telephone switching networks.

The language and the tool support software can be found

- http://alloy.mit.edu/alloy/

It has been used successfully, among other things,

- to analyse a Flash file system: http://link.springer.com/chapter/10.1007%2F978-3-540-87603-8_23?LI=true
- To analyse a properties of the Border Gateway Protocol:
- Matvey Arye, Rob Harrison, Richard Wang, Pamela Zave. Jennifer Rexford: Toward a Lightweight Model of BGP Safety, http://wripe11.cis.upenn.edu/program/papers/wripe11-paper14.pdf

10 Simulation

In cases where a full exhaustive state-space search is impractical, or where the satisfaction of requirements has been proven, it can be useful to perform a simulation of the software to validate certain properties. Some model checkers offer the possibility to run in a simulation mode, where not the whole state space is checked, but only a random exploration.

Supporting tools that allow a simulation model to be created and a simulation to be performed are

1. Simulink for MathWorks

This allows complex systems to be simulated

- http://de.mathworks.com/products/simulink/

2. **SPIN**
   which is primarily a model checker, but will also allow simulation of the multi-threaded software model to be performed, when an exhaustive state-space search is not possible

   - http://spinroot.com/spin/whatispin.html

11 Formal Refinement

Formal refinement is a technique where, starting from more abstract specifications, the description is refined to achieve a description that is closer to the actual system to be implemented. These refinements need to be described in a formal language and require an ontology and unambiguous definitions of how the refined descriptions relate to the more abstract versions. Formal refinement is an essential part of the Ontological Hazard Analysis. Formal Refinement is supported by SysML, and various tools exist to support it.

12 Model State Exploration

The exploration of all states of a model is similar to model checking or, if not all possible states are checked, simulation.

13 Codevelopment Tools for software source-code

In some cases it can be useful to develop the source code alongside the functional and safety requirements specifications. This is often the case in languages which allow the requirements to be specified in the source code, such as in the form of annotations.

14 Languages allowing simultaneous development of source code and formal design specifications

Languages that allow for the specification of requirements in the source code can facilitat formal correctness checking. These requirements specifications in the source code can be derived from more abstract specifications by formal refinement and usually take the form of pre-conditions, post-conditions, invariants and restrictions on data flow and information flow. These can then be checked with theorem provers. Such languages include

1. **SPARK**
   Altran UK has been developing and using SPARK for many years and has provable very low defect rates in the resulting software

   - UK: http://intelligent-systems.altran.com/technologies/software-engineering/spark.html

Similar "Design-by-Contract" tools are available for Java, PHP, C++ and other languages

15 Programming language (subset) with rigorous semantics

It may seem obvious that a programming language have rigorous semantics, as the same compiler/linker will always create the same executable from the same source code. But since there are different compilers and different machine architectures, some programming languages (most notably C) may produce differently working executable code when compiled by different compilers, even though they may all work according to the C language specifications. The reason is that these language specifications are not rigorous. Languages have been developed to have rigorous semantics, among them

1. **Ada**
   - http://www.adacore.com/
2. **SPARK**

   SPARK is a specialised programming language for high-integrity systems that uses a subset of Ada with annotations.

   - http://www.adaic.org/advantages/spark-ada/
3. **Lustre**

   Lustre is a programming language originally developed by mostly French academics in the 1980s to resolve the lack of suitable programming languages for safety-critical reactive systems. It has been used for developing the flight control systems of the Airbus A320, nuclear power station emergency shutdown systems (SCRAM) and railway systems. It is supported by commercially available tools provided among others by

   - Verimag: http://www-verimag.imag.fr/SYNCHRONE/
   - Esterel, which supports Lustre in its SCADE suite: http://www.esterel-technologies.com/products/scade-suite/

16 Static Code Analysis Tools

Static code analysis can detect classes of errors before the code is compiled into executable format. There are static code analysis tools available for a number of programming languages that can prove the absence of errors such as division by zero, array bounds violations, data type range violations, and others. Many of these are based on a subset of a programming language to remove features that are much harder or even impossible to check automatically. Support tools include

1. **SPARK toolkit**
   - available as a free version: http://libre.adacore.com/tools/spark-gpl-edition/ and as a
   - commercially supported toolset: http://www.adacore.com/sparkpro/
2. **C/++/C# language toolkits**

   The C programming language and its derivatives are still in widespread use because compilers and development environments have been available for a long time and are often established at companies. Examples include

   - The Escher C verifier
     http://www.eschertech.com/products/ecv.php,

- Microsoft Research's VCC for concurrent C programs http://research.microsoft.com/en-us/projects/vcc/
- Microsoft Research's SLAM project, which is now the standard way for verifying drivers for the Windows operating system family http://research.microsoft.com/en-us/projects/slam/
- Parasoft's static code analysis tools: http://www.parasoft.com/static-analysis

3. Ada

There are also a number of static code analysis tools for Ada.

- CoderPeer is a recent static code analysis tool for Ada 2012 that makes use of Ada 2012's contract-based programming capabilities: http://www.adacore.com/codepeer

17 Automatic Test Generators

In many cases it is possible and useful to create test cases automatically from the higher levels of the specification, against which to test the executable code. It should be noted that statistic testing alone can not guarantee very high reliability of software, but testing is nonetheless an essential validation of the development process.

Automatic test generators are available for a number of languages, and some of the tools include

- MIDOAN Mika Test Data Generator for Ada: http://www.midoan.com/index.html
- Microsoft Research's ATG: http://research.microsoft.com/en-us/projects/atg/

18 Coding Standards

There are a number of coding standards, the set out guidelines of best practices when writing source code. They usually contain some structures and constructs to be avoided, such as dynamic memory allocation and pointer, and also to avoid side-effects of functions, obscure and poorly readable code.

1. C/C++

C is still a widely used language for the development of software for E/E/PE systems, and there are a number of standards and guidelines for writing C code for high-integrity systems:

- MISRA-C is one of the best-known coding standards for the C programming language http://www.misra-c.com/Activities/MISRAC/tabid/160/Default.aspx
- IEC 61508 lists some techniques and constructs to be avoided in the Appendices in the tables of methods
- Parasoft provides coding standards for C/C++ among other languages: http://www.parasoft.com/solutions/cpp_solution.jsp?itemId=340

2. Ada / SPARK

The SPARK subsetting of Ada and the annotations enforce verifiable coding standards, and DO-

178B subscribes coding standards for development software for military avionics.

19 Coding Standards Analysis

Various tools exist for checking the conformance to coding standards.

1. MISRA-C

- Parasoft offers tools to check against MISRA-C rules
  http://www.parasoft.com/cpptest?itemId=47
- Cosmic Software also offers a checker for MISRA-C
  http://www.cosmicsoftware.com/misra.php

2. Ada / SPARK

AdaCore offers GNATCHeck as a coding standard checker as part of its GNATPro tool suite

- http://www.adacore.com/gnatpro/toolsuite/gnatcheck/

20 WCET-analysis Tools

For realt-time systems it is essential that functions never take longer than the allowable time frame to finish and return the correct result. Making sure that this is always the case, even in the worst case can be assured by performing a worst-case execution time analysis. If static analysis is not possible or not feasible, automated measurements of actual timing can be performed. There are tools to support such WCET analysis

- AbsInt offers the aiT WCET analyzers
  http://www.absint.com/ait/index.htm
- Esterel's SCADE suite offers tools for WCET analysis
  http://www.esterel-technologies.com/products/scade-suite/verify/scade-suite-timing-and-stack-verifiers/
- RAPITA systems offers tools for automated execution time measurements
  http://www.rapitasystems.com/products/rapitime

21 Monitor Synthesis / Runtime Verification

Up to a degree, software can be monitored at runtime while it is performing its intended functions. Such monitors can be synthesized from formal functional requirements specifications or formal safety requirements specifications. These monitors can then verify the correct execution of critical funtions. Available tools for monitor synthesis and runtime verification include

- Galois Copilot
  http://galois.com/project/copilot/
- Runtime Verification Inc.
  https://runtimeverification.com/

It is a relatively recent subject and research in this field is ongoing. Relevant publications include

- André de Matos Pedro, David Pereira, Luís Miguel Pinho, Jorge Sousa Pinto: Towards a Runtime Verification Framework for the Ada Programming Language: http://link.springer.com/chapter/10.1007/978-3-319-08311-7_6

A conference of active research in runtime verification is RV 2015

- The 15th International Conference on Runtime Verification http://rv2015.conf.tuwien.ac.at/

5 Appendices

5.1 Appendix A: Table of techniques and methods with related known formal methods

	Name of the Technique / Method	**Specific Formal Methods / Tools**
1	Formal functional requirements specification (FRS)	• Formal Description Language • Controlled Natural Language • Formal and Semiformal Graphical Languages • Formal Logic
2	Formal FRS analysis	• Automated Consistency Checker • Theorem Prover
3	Formal safety requirements specification (FSRS)	• Formal Description Language • Formal Notation Language • Controlled Natural Language • Formal Logic • Ontological Hazard Analysis
4	Formal FSRS analysis	• Automated Consistency Checking • Theorem Prover / Proof Checkers
5	5. Automated proving/proof checking of properties (consistency, completeness of certain types) of FRS and FSRS	• Automated Consystency Checking • Theorem Prover / Proof Checkers
6	Formal modelling, model checking, and model exploration of FRS, FSRS	• Modelling Language • Formal modelling of functions • Hierarchical Modelling • Model Checker • Simulation
7	Formal design specification (FDS)	• Graphical Design Language • Formal Description Language • Formal Logic • Formal Refinement
8	Formal analysis of FDS	• Automated Consistency Checking • Theorem Prover / Proof Checkers
9	Automated proving/proof checking of fulfilment of the FRS/ FSRS by FDS	• Formal Refinement

	Name of the Technique / Method	**Specific Formal Methods / Tools**
		• Theorem Prover • Proof Checker • Model Checking
10	Formal modelling, model checking, and model exploration of FDS	• Modelling of Functions • Hierarchical State Machines • Modelling using petri-nets • Model Checking • Model State Exploration
11	Formal determininistic static analysis of FDS (information flow, data flow, possibilities of run-time error)	• Static data flow analysis • Static information flow analysis • Static Analysis for runtime errors
12	Codevelopment of FDS with Executable Source-Code Level (ESCL)	• Codevelopment Tools for software source-code • Languages allowing simultaneous development of source code and formal design specifications
13	Automated source-code generation from FDS or intermediate specification (IS)	• Automatic Code Generators from textual formal design specifications • Automatic Code Generators from graphical design specifications
14	Automated proving/proof checking of fulfilment of FDS by IS	• Automated Proof Checker • Automated Theorom Prover
15	Automated verification-condition generation from/with ESCL	• Automated Proof Checker • Automated Theorom Prover
16	Rigorous semantics of ESCL	• Programming language with rigorous semantics • Programming language subset with rigorous semantics
17	Automated ESCL-level proving /proof checking of properties (such as freedom from susceptibility to certain kinds of run-time error)	• Static Code analysis tools
18	Automated proving/proof checking of fulfilment of FDS by ESCL	• Automated Proof Checker
19	Formal test generation from FRS	• Automatic test generators
20	Formal test generation from FSRS	• Automatic test generators
21	Formal test generation from FDS	• Automatic test generators
22	Formal test generation from IS	• Automatic test generators

	Name of the Technique / Method	**Specific Formal Methods / Tools**
23	Formal test generation from ESCL	• Automatic test generators
24	Formal coding-standards analysis (SPARK, MISRA C, etc)	• Coding Standards • Coding Standards Analysis
25	Worst-Case Execution Time (WCET) analysis	• WCET-analysis tools
26	Monitor synthesis/runtime verification	• Monitor Synthesis / Runtime Verification

5.2 Appendix B: Table of Abstract Methods and Industrial Implementations

	Specific Formal Methods / Tools	**Known Industrial Implementations / Tools**	**Assured Properties of resulting Software**
1	Formal Description Language	• Z Notation • Abstract Machine Notation (B Method) • VDM-SL / VDM++ • CASL	• Enabling formal verification of relative completeness • Enabling formal verification of consistency
2	Controlled Natural Language	• ASD Simplified Technical English	• Enabling formal / semiformal verification of relative completeness • Enabling formal / semiformal verification of consistency
3	Formal and Semiformal Graphical Languages	• Petri-Nets • UML	When backed by rigorous semantics: • Enabling formal verification of relative completeness • Enabling formal verification of consistency
4	Formal Logic		• Enabling formal verification of relative completeness • Enabling formal verification of consistency
5	Finite State Machines		• Enabling formal verification of relative completeness • Enabling formal verification of consistency • Enabling of formal verification / model checking of functional requirements
6	Automated Consistency Checking	• Z-Checker FUZZ: http://spivey.oriel.ox.ac.uk/mike/fuzz/ • CASL consistency checker: http://www.informatik.uni-bremen.de/cofi/ccc/	• Guaranteed consistency of Requirements Specifications
7	Automated Therom Prover / Proof Checker	• Configurable Proof Assistant **Isabelle**: http://isabelle.in.tum.de/ • The Prototype Verification System **PVS** http://www.csl.sri.com/projects/pvs/	• Guaranteed fulfilment of requirements specifications of the checked portions
8	Ontological Hazard Analysis		• Guaranteed fulfilment of abstract Safety Requirements down to object code level
9	Formal Modelling Language / Model Checking	• ProMeLa / SPIN http://spinroot.com/spin/whatispin.html	• Verification of fulfilment of functional and safety requirements by the system design.

	Specific Formal Methods / Tools	**Known Industrial Implementations / Tools**	**Assured Properties of resulting Software**
		• Alloy http://alloy.mit.edu/alloy/	
10	Simulation	• Simulink for MathWorks http://de.mathworks.com/products/simulink/ • ProMeLa / SPIN http://spinroot.com/spin/whatispin.html	• Increased confidence in fulfilment of functional and safety requirements. Weaker evidence than full state-space-search (model checking)
11	Formal Refinement		• Formally specified implementation of more abstract specifications by more concrete design
12	Model State Exploration	• ProMeLa / SPIN http://spinroot.com/spin/whatispin.html • Alloy http://alloy.mit.edu/alloy/ • Simulink for MathWorks http://de.mathworks.com/products/simulink/	• See Model Checking / and or Simulation, depending on the level of state space search exhaustion.
13	Codevelopment Tools for software source-code		• Early detection of problems with the specification in relation to the implementation
14	Languages allowing simultaneous development of source code and formal design specifications	• SPARK http://intelligent-systems.altran.com/technologies/software-engineering/spark.html • Numerous other tools exist for Java, PHP, C++, ...	• Early detection of problems with the specification in relation to the implementation
15	Programming language (subset) with rigorous semantics	• Ada http://www.adacore.com/ • SPARK http://intelligent-systems.altran.com/technologies/software-engineering/spark.html • Lustre http://www-verimag.imag.fr/SYNCHRONE/	• Guaranteed semantics of object code in relation to source code
16	Static Code Analysis Tools	• SPARK toolkithttp://libre.adacore.com/tools/spark-gpl-edition/ http://www.adacore.com/sparkpro/ • C/++/C# language toolkits	• Guaranteed absence of certain classes of runtime errors, such as out-of-bounds array access, data type value range violations, division by zero, ...

	<u>**Specific Formal Methods / Tools**</u>	<u>**Known Industrial Implementations / Tools**</u>	<u>**Assured Properties of resulting Software**</u>
		○ Escher C Verifier: http://www.eschertech.com/products/ecv.php ○ Microsoft Research's VCC for concurrent C programs http://research.microsoft.com/en-us/projects/vcc/ ○ Parasoft's static code analysis tools: http://www.parasoft.com/static-analysis • Ada AdaCore's CorePeer: http://www.adacore.com/codepeer	
17	Automatic Test Generators	• MIDOAN Mika Test Data Generator for Ada: http://www.midoan.com/index.html • Microsoft Research's ATG: http://research.microsoft.com/en-us/projects/atg/	• Test regimes and patterns matched to the specifications and usage patterns to achieve optimal test coverage for a given use. (For very high dependability systems, testing alone cannot suffice to guarantee
18	Coding Standards	• C/C++ ○ MISRA-Chttp://www.misra-c.com/Activities/MISRAC/tabid/160/Default.aspx ○ IEC 61508 lists techniques avoid in the Appendices in the tables of methods ○ Parasoft provides coding standards http://www.parasoft.com/solutions/cpp_solution.jsp?itemId=340 • Ada / SPARK ○ The SPARK subsetting of Ada enforce coding standards ○ DO-178B prescribes coding standards for military avionics	• Possibility for avoidance of typical classes of errors and runtime resource restrictions, such as dangling pointers, access to uninitialised memory.
19	Coding Standards Analysis	• MISRA-C ○ Parasoft MISRA-C rules checker http://www.parasoft.com/cpptest?itemId=47 ○ Cosmic Software's MISRA-C checker http://www.cosmicsoftware.com/misra.php • Ada / SPARK ○ http://www.adacore.com/gnatpro/toolsuite/gn	• Avoidance of typical classes of errors and runtime resource restrictions. Weaker confidence than formal verification because some standards cannot be checked automatically

	Specific Formal Methods / Tools	**Known Industrial Implementations / Tools**	**Assured Properties of resulting Software**
		atcheck/	
20	WCET-analysis Tools	• AbsInt's aiT WCET analyzer http://www.absint.com/ait/index.htm • Esterel's SCADE suite offers WCET analysis http://www.esterel-technologies.com/products/scade-suite/verify/scade-suite-timing-and- stack-verifiers/ • RAPITA automated execution time measurements http://www.rapitasystems.com/products/rapitime	• Guaranteed maximum execution time for timing-critical real-time applications
21	Monitor Synthesis / Runtime Verification	• Galois Copilot http://galois.com/project/copilot/ • Runtime Verification Inc. https://runtimeverification.com/	• Verification of certain functions during runtime.

6 Annexes

INS1234-TR-A: Peter B. Ladkin: Notes on Properties Needed in Software Safety Requirements

INS1234-TR-B: Peter B. Ladkin and Bernd Sieker (ed.): Professional Opinion on Skills with Formal Description Languages