SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 98-25

A Tutorial Introduction to Formal Methods

Peter A. Lindsay

October 1998

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

This paper appeared in P. Lindsay (ed), *Proceedings 3rd Australian Workshop on Industrial Experience with Safety Critical Systems and Software*, 13 November 1998, Australian Computer Society, pages 29-37.

A Tutorial Introduction to Formal Methods

Peter A. Lindsay

Abstract

This paper gives details of a small example to illustrate the use of Formal Methods of system and software development, including modelling, specification, validation and design verification. The example concerns part of a simple hypothetical Air Traffic Control system. Z notation and the `Cogito` *methodology are used.*

A Suggested Reading List is included for readers wishing to know more about the use of formal methods.

1 Introduction

Formal Methods are methods with a sound mathematical foundation and which support reasoning about properties of systems. There is a large diversity of formal methods for system, software and hardware development – more than could possibly be covered in a short introduction. The Suggested Reading List at the end of the paper gives some pointers to the relevant literature, including web sites, tool support, experience reports and comparative studies.

This short paper focusses on one particular family of formal methods – the so-called *model-based specification methods* – which have achieved widespread usage for system and software development, and which include the well-known Z [33] and VDM [21] notations. A small example is used to illustrate the approach. The example is adapted from a more complete example originally developed by the author in VDM [3] and fully verified using the `mural` support environment [22]. The current example is given here in the `Sum` dialect of Z, as supported by the `Cogito` methodology and tool-set [5]. [1]

The example concerns a simple (hypothetical) systems management tool such as might be used to oversee communications between aircraft pilots and air-traffic controllers in an Air Traffic Control region. Note however that the purpose of the example is to illustrate the use of formal methods; hence the example is kept simple and context information is included only sufficiently to understand the methods. No particular assumptions are made about how the different parts of the system would be implemented, nor about what other systems and procedures would be involved. In practice of course, the system's development would take place in the larger context of requirements analysis, hazard and risk analysis, integrity allocation, testing and so on, as required by the appropriate standards: formal methods are intended to supplement and improve these techniques, not to replace them.

The paper illustrates the use of a particular Formal Method on the following aspects of critical system development:

- modelling of "static" features of high-level system design
- modelling of system functionality and safety requirements
- validation of design by verifying logical consistency
- modelling of (part of) the system architecture, and verification that the architecture meets high-level design requirements

To show how the method can be used to identify errors and oversights early in the development life-cycle, the example purposefully includes a number of false starts, including some subtle errors which are brought to light by Verification and Validation (V&V) techniques associated with the method.

The paper is organised as follows: Section 2 introduces the mathematical notation used in the paper. Section 3 defines a high-level "data model" for the system, including the main types of entities involved, the different possible states of the system, and formalisation of some system terminology and safety requirements; use of formal validation is illustrated on the model. Section 4 illustrates formal specification of system functionality and validation of the functional model. Section 5 illustrates a possible design step in which part of the system architecture is modelled and shown (in part) to satisfy the requirements of the high-level system specification. The paper concludes with a list of Suggested Reading for readers wishing to know more about the use of formal methods.

[1]The `Sum` front-end tools and type-checker are ftp-able from `//svrc.it.uq.edu.au/pub/software/SumTypechecker.tar`

2 Z notation

This section introduces the Z mathematical notation [33] which will be used in the examples below; in what follows, underscores are used as placeholders for arguments to functions.

$\mathbb{N}$	natural numbers (0,1,2,. . .)
$\mathbb{P}\_$	sets of
$\_ \in \_$	is an element of
$\#\_$	size of
$\_ \subseteq \_$	is a subset of
$\_ \setminus \_$	set subtract
$\mathrm{seq}\,\_$	sequences (ordered lists) of
$\_ \frown \_$	sequence concatenation
$\_ \nrightarrow \_$	mapping (partial function)
$\_ \rightarrowtail\!\!\!\!\!\nrightarrow \_$	one-one mapping
$\_ \mapsto \_$	maplet (single argument-result component of a mapping)
$\mathrm{dom}\,\_$	'domain' of a mapping (the set of all its possible inputs)
$\mathrm{ran}\,\_$	'range' of a mapping (the set of all its outputs)
$\_ \oplus \_$	function overwrite
$\forall$	for all

More complex expressions can be built from these: e.g. $\{n : \mathbb{N} \mid n^2 \leq 9\} = \{0, 1, 2, 3\}$.

3 Data model for the system

This section defines a mathematical model of the main static concepts for the system. This is done by

1. defining the "basic" types of entities with which the system is concerned
2. defining the variables ("states") of the system and what possible values they can take
3. formalising concepts relevant to system requirements and defining terminology
4. formalising system properties (including safety requirements) which can be stated as relationships between state variables.

The logical consistency of the model is checked as a validation step.

3.1 Basic types

The following types of entities will be used without further analysis or definition in the model:

Controller – the set of all possible air-traffic controllers

Space – the set of all possible airspaces

Aircraft – the set of all possible aircraft

The entities are treated abstractly in this high-level model, and need not represent particular physical entities: e.g. airspaces are regarded here as logical entities only, and no assumption are made about what 3-dimensional volumes of air they represent, nor whether they can overlap or co-exist, etc. (Such attributes could of course be formalised if desired – but here we abstract away from detail so as to better focus on the important logical relationships between the above entities.)

3.2 State variables

The state of the system is modelled here by declaring four variables (the "state variables") whose values determine the main relationships between the basic entities:

1. the controllers who are currently on duty, although not necessarily in control of an airspace

   $$on\_duty : \mathbb{P}\ Controller$$

2. which controller is assigned to which airspace

   $$control : Space \rightarrowtail\!\!\!\!\!\nrightarrow Controller$$

3. how many aircraft an airspace can accommodate safely

   $$capacity : Space \nrightarrow \mathbb{N}$$

4. which airspace an aircraft currently occupies

   $$location : Aircraft \nrightarrow Space$$

Note that certain constraints on the state are already implicit in the type declarations above. For example, at most one controller is assigned to any airspace at any time, since *control* is a one-one mapping. Note also that *control* is a partial function, which means that some airspaces may not have a controller assigned to them at all times (e.g. when the airspace is closed down for the night).

3.3 Defined concepts

Other state-dependent concepts can now be defined in terms of the values of the state variables: e.g.

- The "active controllers": ran *control*

- The "activated airspaces": dom $\mathit{control}$
- The "utilized airspaces" (i.e., those in which aircraft are currently subject to air-traffic control): ran $\mathit{location}$

We can also define functions to extract information which is implicit in the model: e.g. the number of aircraft assigned to airspace s is given by

$$\begin{aligned}&\mathit{num\_aircraft}(s, \mathit{location}) = \\ &\quad \#\{a : \mathit{Aircraft} \mid \mathit{location}(a) = s\}\end{aligned}$$

Note that this function's dependence on the value of the $\mathit{location}$ state variable is noted explicitly.

3.4 Some safety requirements

In this section we formalize some of the system safety requirements as "state invariants" – relationships between the state variables which should be preserved by all system functions:

1. An airspace can be activated only if its capacity has been set:

   $$\mathrm{dom}\ \mathit{control} \subseteq \mathrm{dom}\ \mathit{capacity}$$

2. Only on-duty controllers can control airspaces:

   $$\mathrm{ran}\ \mathit{control} \subseteq \mathit{on\_duty}$$

3. All utilized airspaces have controllers:

   $$\mathrm{ran}\ \mathit{location} \subseteq \mathrm{dom}\ \mathit{control}$$

4. The capacity of each utilized airspace is not exceeded

   $$\begin{aligned}&\forall\, s : \mathrm{ran}\ \mathit{location} \bullet \\ &\quad \mathit{num\_aircraft}(s, \mathit{location}) \leq \mathit{capacity}(s)\end{aligned}$$

We assume such properties have been derived from a system hazard analysis or similiar. Use of formal notation then adds value by making requirements precise and mathematically verifiable.

3.5 Validation of the data model

Having formalised the data model, automated support is available for checking ("validating") the model, including:

- Type-checking – to check that definitions are being used type-consistently throughout the model. This check is fully automated for `Sum` specifications.
- Semantic checking – to check full mathematical consistency: e.g. that partial functions are applied only to values within their domain, no division by zero, etc. This check is partially automated in `Cogito` using theorem provers.

For example, in this case the semantic check on invariant 4 involves checking that $\mathit{capacity}(s)$ is defined for all airspaces s in ran $\mathit{location}$ – or in other words, that capacities have been set for all utilized airspaces. This check would have revealed the necessity for something like invariants 1 and 3 above, had they been missing.

4 Modelling system functionality

We next turn to modelling dynamic aspects of the system: i.e., system functions and how they affect the state of the system.

4.1 Notation

In state-based specification languages such as Z and VDM, system functions are expressed as "operations" which change the state of the system; they may also take inputs and/or return outputs. By Z convention, operation inputs have suffix ?, outputs have suffix !, and the new value of a changed state variable has suffix $'$. The `Sum` dialect also supports "preconditions" – statements which define the conditions under which the operation can be invoked – written as a clause pre(_). Also in `Sum`, a $\mathit{changes\_only}\{\_\}$ clause can be used to indicate which state variables may be affected by the operation; all other state variables are left unchanged.

4.2 Some system functions

For our example, here is a specification of an operation for handing over aircraft $a?$ from airspace $\mathit{from}?$ to airspace $\mathit{to}?$:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Handover}} \\ \hline
a? : \mathit{Aircraft} \\
\mathit{from}?, \mathit{to}? : \mathit{Space} \\ \hline
\mathrm{pre}(a? \in \mathit{location}(\mathit{from}?)) \\
\mathit{changes\_only}\{\mathit{location}\} \\
\mathit{location}' = \mathit{location} \oplus \{a? \mapsto \mathit{to}?\} \\ \hline
\end{array}$$

The specification says that $a?$ must be assigned to airspace $\mathit{from}?$ at the time the operation is invoked, and will be reassigned to airspace $\mathit{to}?$ as a result of the operation; note that $\mathit{location}$ is the only state variable affected by this operation.

As a second example, here is the operation for controller $c?$ going off duty:

$$
\begin{array}{|l}
\textit{ClockOff} \\
\hline
c? : \textit{Controller} \\
\hline
\mathrm{pre}(c? \in (\textit{on\_duty} \setminus \mathrm{ran}\ \textit{control})) \\
\textit{changes\_only}\{\textit{on\_duty}\} \\
\textit{on\_duty}' = \textit{on\_duty} \setminus \{c?\} \\
\hline
\end{array}
$$

The precondition says that $c?$ should be on-duty but not in control of an airspace. If $c?$ does currently control an airspace, that airspace must be reassigned to another on-duty controller before $c?$ can be clocked off.

$$
\begin{array}{|l}
\textit{Reassign} \\
\hline
s? : \textit{Space} \\
c?, c! : \textit{Controller} \\
\hline
\mathrm{pre}(\textit{control}(s?) = c? \land \mathrm{ran}\ \textit{control} \neq \textit{on\_duty}) \\
\textit{changes\_only}\{\textit{control}\} \\
\textit{control}' = \textit{control} \oplus \{s? \mapsto c!\} \\
c! \in \textit{on\_duty} \setminus \mathrm{ran}\ \textit{control} \\
\hline
\end{array}
$$

The specification of *Reassign* says that the operation should reassign control of airspace $s?$ to an on-duty controller $c!$ who is not currently controlling an airspace. This high-level model gives no further details of exactly how $c!$ is selected; that's left to lower-level specifications.

4.3 Validation of the functional model

In addition to type- and semantic-checking as for the data model above, automated support is available for further checks on the functional specification:

- Feasibility checking – to check that for any state in which the operation can be invoked (i.e., its precondition is true), the system can make a transition to a state in which the operation specification is satisfied and the state invariants are preserved. This check can be partially automated.
- Animation – to check that the system-as-modelled behaves as expected. The `Possum` tool [20] has a wide variety of modes for animating specifications from "animation scripts" written by system analysts. For example, it can step through operational scenarios by applying operations one by one under interactive control, or it can perform complete searches of function spaces checking for user-defined conditions.

Did you spot the deliberate error in the *Handover* operation above? The feasibilty check will reveal that the given specification's precondition is not sufficient, since it cannot guarantee that the capacity of airspace $to?$ will not be exceeded. To correct the oversight, the following constraint should be added to the precondition:

$$\textit{num\_aircraft}(to?, \textit{location}) < \textit{capacity}(to?)$$

5 Modelling system design details

The last topic treated in this short introduction to `Cogito` concerns support for verification of system designs at lower levels of detail. Here we illustrate how part of a system architecture for the ATC example could be modelled and shown to meet higher-level requirements.

5.1 A distributed architecture

Let us suppose that a distributed architecture is chosen for the system, whereby each airspace is assigned its own particular equipment (such as a monitor, radar, radio communications, etc). Suppose that at this new level of design it is decided to associate a sequence ("queue") of aircraft with each airspace – by analogy with the ranks of flight-progress strips often used in ATC centres, say.

The system model at this new level of detail will have state variables *on_duty*, *control* and *capacity* as before, but the *location* variable will be replaced by a new variable for "assignment queues" as follows:

$$\textit{assigQueue} : \textit{Space} \rightarrow\!\!\!\!\!+\ \mathrm{seq}\ \textit{Aircraft}$$

In the modified data model, the definition of utilized airspaces becomes

$$\{s : \textit{Space} \mid \#\textit{assigQueue}(s) \neq 0\}$$

and the number of aircraft in a utilized airspace s is $\#\textit{assigQueue}(s)$. State invariants 1 and 2 will remain as above, but invariants 3 and 4 will be replaced by the following:[2]

- Airspaces are assigned queues if and only if they are activated

$$\mathrm{dom}\ \textit{assigQueue} = \mathrm{dom}\ \textit{control}$$

[2]Note that in Z mathematical semantics, a sequence is regarded as a mapping from a sequence of numbers (the sequence indexes) to sequence elements:

$$\langle x_1, x_2, \ldots, x_n\rangle = \{1 \mapsto x_1, 2 \mapsto x_2, \ldots, n \mapsto x_n\}$$

Thus, $\#s$ gives the length of a sequence s and ran s converts the sequence into a set with the same elements.

- The capacity of activated airspaces is not exceeded

$$\begin{array}{l} \forall\, s : \mathrm{dom}\, \mathit{assigQueue} \bullet \\ \quad \#\mathit{assigQueue}(s) \leq \mathit{capacity}(s) \end{array}$$

The operations from the high-level model need to be adapted to the new model: e.g.

$$\begin{array}{|l} \hline \mathit{Handover} \\ a? : \mathit{Aircraft};\ \mathit{from}?, \mathit{to}? : \mathit{Space} \\ \hline \mathrm{pre}(a? \in \mathrm{ran}\, \mathit{assigQueue}(\mathit{from}?) \wedge \\ \quad \#\mathit{assigQueue}(\mathit{to}?) < \mathit{capacity}(\mathit{to}?)) \\ \mathit{changes\_only}\{\mathit{assigQueue}\} \\ \mathit{assigQueue}'(\mathit{from}?) = \\ \qquad \mathit{remove}(a?, \mathit{assigQueue}(\mathit{from}?)) \\ \mathit{assigQueue}'(\mathit{to}?) = \mathit{assigQueue}(\mathit{to}?) \frown \langle a? \rangle \\ \{\mathit{from}?, \mathit{to}?\} ⩤ \mathit{assigQueue}' = \\ \qquad \{\mathit{from}?, \mathit{to}?\} ⩤ \mathit{assigQueue} \\ \hline \end{array}$$

where *remove* is a function which removes an element from a list and $\_ ⩤ \_$ is a Z function which removes sets of elements from the domain of a mapping [33].

5.2 Design verification

The new model can be shown to be a true refinement of the high-level model by providing a relationship between the states of the two models which shows how "concrete" states relate back to "abstract" states and verifying certain other relationships hold – the so-called *data refinement proof obligations* [21]. In this case we could try to define *location* in terms of the new model:

$$\begin{array}{l} \mathit{location} : \mathit{Aircraft} \twoheadrightarrow\!\!\!\!\!\!\,\shortmid\;\; \mathit{Space} \\ \mathit{location} = \{a \mapsto s \mid a \in \mathrm{ran}\, \mathit{assigQueue}(s)\} \end{array}$$

However, when we try to verify the mathematical consistency of this definition – in particular, when we try to show it is a true function, with a unique output for each input in its domain – we come across a problem: how do we know that each aircraft belongs to a unique airspace? The state invariant for the new model needs to be strengthened by some statement to this effect: e.g.

$$\begin{array}{l} \forall\, s_1, s_2 : \mathrm{dom}\, \mathit{assigQueue} \bullet s_1 \neq s_2 \Rightarrow \\ \quad \mathrm{ran}\, \mathit{assigQueue}(s_1) \cap \mathrm{ran}\, \mathit{assigQueue}(s_2) = \varnothing \end{array}$$

This is an example of how verification of lower-level designs against higher-level designs can reveal errors and oversights.

6 Summary and conclusions

In summary, the paper illustrates one particular formal method (`Cogito`) on a simple example. Other formal methods are available for other kinds of problem.

Some of the main advantages of formal methods are:

- Formal specifications are precise, concise and unambiguous, which makes them an excellent medium for communication between system designers, analysts, testers and evaluators. Again, they are intended to supplement and improve informal specifications, not to replace them.

- Because they use formal, machine-checkable notation, a wide variety of automated checks can be applied to them (including checks not illustrated above, such as for freedom from deadlock and freedom from livelock). The ability to model and validate high levels of design means that errors can be caught earlier in the design life-cycle, with consequent cost savings.

- The ability to reason formally about specifications means that the level of semantic checking is far deeper than that provided by program analysis tools, for example. The discipline it imposes means that subtle errors and oversights are more likely to be picked up during analysis.

- Formalising the relationship between different levels of design means that design steps can be verified, and requirements traceability can be automated and checked. This has particular value during incremental builds and maintenance.

- Formal specifications can be used as a precise basis for a range of other activities: e.g. they can be used as test oracles, or to derive test cases in a systematic manner [27], or as a basis for FMEAs and HAZOPs.

- Prototypes can be derived directly from specifications.

However, formal methods are intended to enhance existing development and assurance techniques and not to replace them. To use them effectively requires staff with a good deal of experience and some mathematical maturity, a good pre-existing design methodology, and good tool support. Most importantly, they do not replace the need for domain expertise (although they can make it easier to acquire!).

7 Suggested Reading

7.1 On the web

Formal Methods home page:
http://www.comlab.ox.ac.uk/archive/formal-methods.html

NASA Formal Methods Program:
http://eis.jpl.nasa.gov/quality/Formal_Methods/

Rushby report for US Federal Aviation Authority
"Formal Methods and their Role in the Certification of Critical Systems" :
http://www.csl.sri.com/reports/postscript/csl-95-1.ps.gz

7.2 Annotated bibliography

The following list is not intended to be definitive, but gives some pointers to useful literature on the use of formal methods:

- Use of formal methods on Safety-Critical Systems: [2, 6, 7]
- Standards calling up formal methods: [29, 30]
- Case studies of applications of formal methods: [8, 11, 19, 23]
- Industry experience reports: [14, 16, 17, 34]
- Reference books on particular formal methods:
  - Z [33]
  - VDM [21]
  - B [1, 24]
  - Object Z [12] for object-oriented specifications
  - refinement of specifications to code [28]
  - Cogito [5]
  - Spark Ada [9]
- Use of formal methods in aerospace: [32]
- Use of formal methods in Human-Computer Interface design: [18]
- Formal reasoning: [3, 4]
- Tool support: [5, 20, 22, 25, 26, 31]
- Recent developments and future directions: [10, 13, 15]

SVRC technical reports can be obtained electronically from `http://svrc.it.uq.edu.au`

References

[1] J-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] L.M. Barroca and J.A. McDermid. Formal methods: Use and relevance for the development of safety critical systems. *Computer Journal*, 35(5), 1992.

[3] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT Series. Springer-Verlag, 1994. ISBN no. 3-540-19813-X.

[4] J.C. Bicarregui, editor. *Proof in VDM: Case Studies*. FACIT. Springer Verlag, 1998. ISBN 3-540-76186-1.

[5] A. Bloesch, E. Kazmierczak, P. Kearney, and O. Traynor. Cogito: A Methodology and System for Formal Software Development. *International Journal of Software Engineering and Knowledge Engineering*, 5(4), December 1995.

[6] J. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189 – 209, July 1993.

[7] J. P. Bowen and M . G. Hinchey. Formal methods and safety-critical standards. *IEEE Computer*, 27(8):68–71, August 1994.

[8] J.P. Bowen and M. Hinchey, editors. *Applications of Formal Methods*. Prentice Hall International, 1995.

[9] B. Carré, J. Garnsworthy, and W. Marsh. SPARK: a safety-related Ada subset. In *Proc. 1992 Ada UK Conference*, London Docklands, 1992.

[10] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[11] B. Cohen, W.T. Harwood, and M.I. Jackson. *The Specification of Complex Systems*. Addison-Wesley Publishing Company, Wokingham, England, 1986.

[12] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995. Also appears as SVRC TR 94-45.

[13] J. Fitzgerald, C.B. Jones, and P. Lucas, editors. *Proc. Formal Methods Europe (FME'97)*. Springer Verlag, July 1997. LNCS 1313.

[14] S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, pages 21–28, January 1994.

[15] L. Groves and S. Reeves, editors. *Proc. Formal Methods Pacific (FMP'97)*. Springer Verlag, July 1997.

[16] A. Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, Sept 1990.

[17] A. Hall. Using formal methods to develop an ATC information system. *IEEE Software*, pages 66–76, March 1996.

[18] M. Harrison and D. Duke. A review of formalisms for describing interactive behaviour. In *Software Engineering and Human-Computer Interaction*, pages 49–75. Springer Verlag, 1995. Proc. ICSE'94 Workshop on SE-HCI.

[19] I. Hayes, editor. *Specification Case Studies*. Prentice-Hall, second edition, 1993. First Edition published in 1987.

[20] D. Hazel, P. Strooper, and O. Traynor. Possum: An animator for the sum specification language. In *Proc. Asia-Pacific Software Engineering Conference*, pages 42–51. IEEE Computer Society, 1997. Also available as SVRC TR 97-10.

[21] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, second edition, 1990.

[22] C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.

[23] C.B. Jones and R. Shaw. *Case Studies in Systematic Software Development*. Prentice-Hall International, 1990.

[24] K. Lano. *The B Language and Method: A Guide to Practical Formal Development*. FACIT Series. Springer-Verlag, 1996.

[25] P. B. Lassen. IFAD VDM-SL toolbox report. In *FME'93: Industrial Strength Formal Methods*, page 681. Springer Verlag, 1993. Proc. First Internat. Symp. of Formal Methods Europe, Odense, Denmark, April 1993.

[26] P.A. Lindsay and D. Hemer. Using CARE to construct verified software. In M.G. Hinchey and S. Liu, editors, *Proc. 1st Int Conf on Formal Eng Methods*, pages 122–131. IEEE Computer Society Press, November 1997. Also appears as SVRC TR 97-40.

[27] J. McDonald, L. Murray, and P. Strooper. Translating Object-Z specifications to object-oriented test oracles. In *Proceedings Asia-Pacific Software Engineering Conference*, pages 414–423. IEEE Computer Society, December 1997.

[28] C. Morgan. *Programming from Specifications*. Prentice-Hall International, 1990.

[29] Australian Department of Defence. Procurement of computer-based safety critical systems. Australian Defence Standard DEF(AUST) 5679, July 1998.

[30] U.K. Ministry of Defence. The Procurement of Safety Critical Software in Defence Equipment. Defence Standard 00-55, August 1995.
http://www.dstan.mod.uk.

[31] ProofPower home page.
http://www.to.icl.fi/ICL/ProofPower/index.html.

[32] J. Rushby. Formal methods and their role in digital systems validation for airborne systems. NASA Contractor Report 4673, August 1995.

[33] J.M. Spivey. *The Z Notation: a Reference Manual*. Prentice-Hall, New York, second edition, 1992.
http://spivey.oriel.ox.ac.uk/ mike/zrm/index.html.

[34] V. Stavridou et al. Formal methods and safety critical systems in practice. *High Integrity Systems*, 1(5):423–445, 1996.