SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 96-17

Formal Methods Pilot Project
Final Report

CSC Australia and SVRC
Pilot Project Team

July 1996

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533

Note: Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

Formal Methods Pilot Project Final Report *

Tracey Hart, Fiona Linn, Roberto Morello, Greg Royle
Trusted Systems Group, CSC Australia Pty Ltd,
Module 7, Endeavour House, Technology Park, The Levels, SA 5095
(thart,flinn,rmo@csaadel.adl.csa.oz.au)

Peter Kearney, Peter Lindsay, Kelvin Ross, Owen Traynor
Software Verification Research Centre, Dept of Computer Science,
The University of Queensland, St Lucia, Queensland 4072
(pk,pal,kjross,owen@cs.uq.edu.au)

1 INTRODUCTION

This paper reports on a collaborative project between industry, academia and defence to pilot the use of formal methods in the development of safety related software. The project was intended primarily as a technology transfer exercise, to the mutual benefit of the collaborating organizations.

1.1 Background

Formal methods are expected to be useful in the development of software for highly critical systems, in which incorrect operation may lead to significant risk to personnel, the environment, national security or finance. An increasing number of international standards for critical systems recommend the use of such methods [6].

Formal methods are mathematically based, providing a high degree of precision in the specification of requirements for the system, and proof of correct implementation with respect to the specifications. These qualities are particularly desirable in critical systems, to minimise risk during operation. In Australia, there is little published experience of formal methods being applied in industry.

At the project's outset, the main difficulties perceived in using formal methods in industry were:

- The methods do not scale to industrial-sized applications.
- Tool support is inadequate or non-existent.
- A significant amount of training is required to be able to read and understand formal specifications.
- The developers need significant mathematical skills.
- The process is manually intensive.

The pilot project set out to address some of these perceptions by applying formal methods in a safety-critical application.

Specifically, the project aimed to produce a complete and validated set of formal specifications for a safety-critical application, and to exercise all aspects of the SVRC's evolving `Cogito` formal development methodology [4]. This approach meant that CSC acquired experience with most aspects of formal development and the SVRC gained valuable feedback on the usability of its methodology and tools.

1.2 The Application

The application chosen for development was a safety-related test unit for the Nulka Project's

*Appeared in: *Industry Experience Track Proceedings, The Ninth Australian Software Engineering Conference (ASWEC'96)*, Melbourne, July 1996.

anti-missile decoy system, jointly developed by the US Navy and the Australian Government's Department of Defence (DoD). At the project outset, the test unit was perceived as being safety-critical, because if it did not function according to its specifications it could accidentally activate the equipment, thereby endangering the test operator and anyone else nearby. Also, if the test unit incorrectly passed a faulty piece of equipment, or left the equipment in a faulty state, then subsequent operation of that equipment could have disastrous consequences.

For the purposes of the project, the SVRC's evolving `Cogito` methodology and supporting tools were integrated into CSC's hazard analysis and software development life-cycle. The resulting methodology was designed to conform to the principles of the Australian Ordnance Council's Pillar Proceeding 223.93[1] for the highest integrity level (S4) software. Software documentation was designed to conform to the US's DoD STD-2167a [2].

The application was chosen because of its interest to the industrial partner and their client (defence). It was recognized from the outset that the chosen application was far from ideal as a demonstration of the capabilities of the `Cogito` methodology.

1.3 Participants

The Pilot Project was carried out by staff from CSC Australia's Trusted Systems Group in Adelaide. The Brisbane-based SVRC provided training, tools and consultancy support in the use of `Cogito`, and reviewed the outputs that CSC staff produced. Staff from DSTO's Trusted Computer Systems Group acted as Independent Safety Auditors. A Steering Committee was formed with membership comprising representatives of the client (the DoD), CSC, the SVRC and DSTO. Project partners contributed to the project on an own-costs basis.

The project had an elapsed time of 12 months (not including a 3 month period about halfway through the project when the project was suspended in the expectation that requirements would stabilize. During this period of suspension, CSC staff were seconded to other activities and the SVRC undertook additional tool enhancement and population). It involved approximately 1.8 person-years of CSC effort (one full-time technical officer, and two others providing management and review support). Before commencing the project, the industrial participants' only training in formal methods was a one week course in the Z specification language, held two months prior to commencement. The team members had some prior experience of hazard analysis and software development. The technical officer's familiarity with mathematics was very limited: basically, first-year undergraduate applied mathematics.

In order to demonstrate transferability of SVRC techniques to industry after a suitable period of training and with minimal support thereafter, SVRC staff did not undertake any of the software analysis or development tasks directly.

1.4 The Development Process

The project was originally intended to start from a Software Requirements Specification (SRS) for the test unit, to be supplied by the client. In the event, production of the SRS was delayed until well into the project, so we went ahead – on a tentative basis at first – with a home-brewed version of the SRS, extracted from the Operational Requirements Document and (later) the Operational Concepts Document for the test unit. Because of the uncertainty involved, and the corresponding lack of impetus, 6 months into the project a decision was made to proceed with development on the basis of the working SRS.

The software development tasks undertaken during the project included the following:

1. Hazard analysis, to determine the safety-critical functions of the software and the safety criteria. Fault Tree Analysis [3] was used for the hazard analysis.
2. Formal specification in `Sum` (the specification notation of `Cogito`) of the required functionality of the test unit's software.
3. Formal "validation" of the formal specification, to show that it is mathematically consistent.

Month		Topic/activity	#days
1	T	introduction to specification & validation using `Sum`	6
	W	top level specification workshopping	4
5	T	introduction to formal reasoning	4
	T	reasoning about formal specifications	3
	W	specification & formalization of safety properties	3
7	T	tool support for formal reasoning	2
	T	introduction to data refinement	1
	W	specification & proof of safety properties	2
8	T	data refinement	1
	W	validation & use of Ergo theorem prover	4
10	W	data refinement & use of Ergo	5
11	T	introduction to algorithm refinement	1
	W	validation, data refinement	4

Table 1: Pilot project training schedule (T=training, W=workshopping)

4. Formalization of the identified safety criteria, where possible.

5. Proof that the formalized safety criteria are logical consequences of the formal specification. The proofs were carried out rigorously by hand initially, and later checked using the `Ergo` interactive theorem prover.

6. Detailed design and formal development of part of the application, using data refinement.

7. Verification using mathematical proof that the design is correct against the corresponding part of the top-level specification.

Algorithm refinement was covered in the training but not applied in the project, and as a result no Ada code was generated.

1.5 Training and Consultancy Support

Table 1 outlines the project training schedule. (Training dates are shown in terms of project elapse time over 12 months.) The training was supplemented with workshopping of the tools and techniques on parts of the application. In addition, project technical meetings were held approximately every seven weeks at the SVRC for 1-2 days each. All other SVRC support was provided by phone, fax or email.

2 ABOUT COGITO

`Cogito` [4] is an integrated methodology and toolset supporting formal program development. The `Cogito` methodology addresses specification, design and development, and construction of implementations and verification. Verification is seen as a crucial activity, carried out as an integral part of all development phases.

2.1 The `Cogito` Methodology

Figure 1 shows an abstract model of the `Cogito` development process. The phases of the model which are addressed by the `Cogito` development methodology are specification, development, implementation and (to a certain extent) evolution.

At the heart of the `Cogito` methodology is the specification language Sum [13]. The `Cogito` development system revolves around the processing and analysis of Sum specifications.

The specifier uses the Sum notation to build mathematical models of the software to be developed, at various levels of abstraction. The

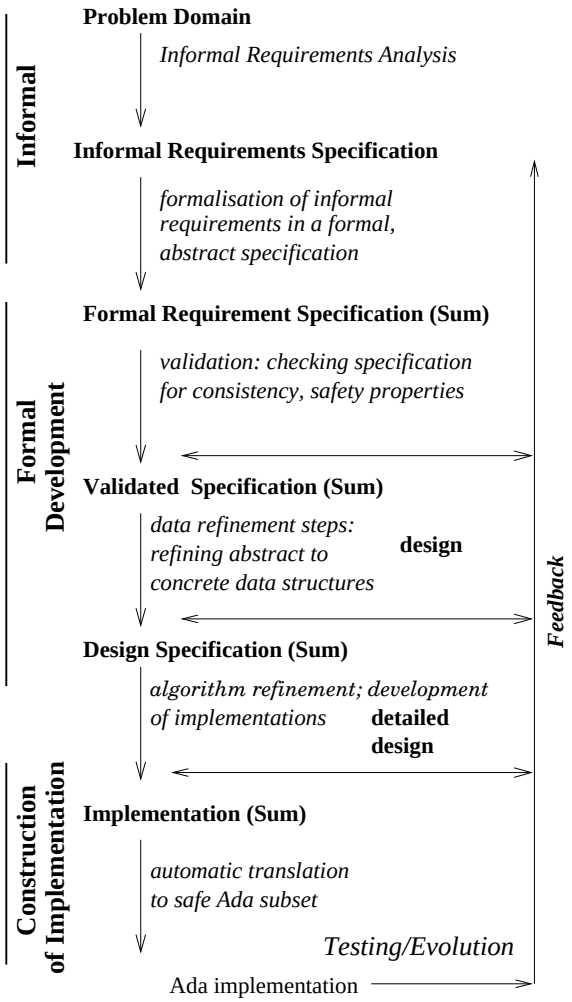

Figure 1: A model of the Cogito development process.

models describe the application's data types, its "state" and its "functions" (written as input/output operations that change the state of the system). The Sum language extends traditional logical notation (predicate calculus) with constructs for many commonly occuring mathematical abstractions such as lists, sets and relations.

Sum extends Z [10] by adding: a module mechanism; explicit preconditions; distinguished state, initialisation and operation schemas; and boolean and character (including string) types.

Briefly, the provision of high level structuring mechanisms as part of the specification language allows Cogito to address more effectively issues of complexity management; separation of concerns; abstraction; reuse; and increased coherency (particularly for tool support).

2.2 The Cogito System

Cogito exploits many existing tools and methods. Central to Cogito's architectural design is the ability to accommodate existing tools and development subsystems. For this reason, the central component of the Cogito toolset architecture is a tool integration harness which allows the integration, coordination and control of existing tools. The integration harness [12] (from here on referred to as the repository manager) also provides mechanisms for configuration and version control. It reflects the overall state of a development and coordinates (in accordance with the Cogito methodology) the user's activities between the various tools of the Cogito system. More details on the repository manager can be found in [11] and [12].

The overall structure of the Cogito development environment is shown in Figure 2. The Cogito tool architecture is described in more detail in [11].

The reasoning needs of Cogito are serviced by the Ergo proof tool [14]. Within a Cogito development, Ergo is used in the following ways:

- as a theorem prover, providing facilities for undertaking formal proof in the context of a Sum specification;
- as a proof obligation generator, generating

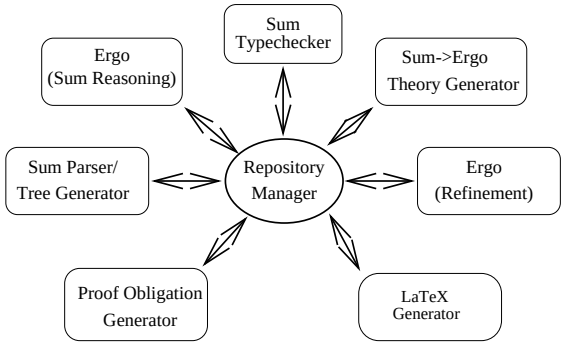

Figure 2: The Conceptual Architecture of the `Cogito` Development Environment.

obligations whose proofs ensure the feasibility and consistency of a Sum specification; and

- as a refinement tool, providing an integrated environment for undertaking refinement and demonstrating the correctness of those refinements.

Ergo provides sophisticated theory construction and structuring mechanisms [9]. These facilities are used to structure the theories generated from Sum specifications. The translation of Sum specification to Ergo theories is supported by an extensive modelling of the Sum mathematical toolkit, the type system of the Sum language and the notion of a Sum schema and the associated schema calculus operations.

Over 50 theories and upwards of 3000 postulates and theorems form the basis of the Ergo environment in which reasoning about Sum specification is carried out. About three person years of effort have been invested in the development of the Ergo theory base [5] in support of reasoning about Sum specifications.

Ergo also offers comprehensive support for the definition of strategies and tactics which are indispensable when reasoning. For example, there are tactics supporting reasoning about integers, sets, types and schemas.

2.3 Specification Validation

Specification validations required by the `Cogito` 1 methodology ensure:

1. The existence of an initial state.
2. The adequacy of the stated precondition, for each operation.

The first of these means that a state with the required initial characteristics exists as a mathematical object. Note that this subsumes a check on the consistency of the state specification, since if an initial state exists, clearly the state specification is satisfiable. The second validation check requires that the stated precondition of an operation, together with the state invariant, are sufficient to ensure that a final result is possible as required by the operation specification.

In addition to the validation conditions that are predetermined by the `Cogito` methodology, safety criteria may also be specified. These properties can be formulated directly using the high level schema operations available within Ergo. They are then embedded in a Sum specification as formal safety comments. In doing this they automatically become part of the set of obligations required to be proved in order for a specification to be considered validated.

2.4 Refinement

The notion of a Sum module is modelled within Ergo and is exploited in order to enable Ergo to perform data refinements. In this sense, refinements are expressed in Ergo as inference rules over these module structures. Algorithm refinement is also be modelled in Ergo in a similar way, but is carried out at the schema level rather than the module level.

Intuitively, a module A is data refined by a module C if the externally visible effects of C's operations are the same as A's, although C's internal state may be different to A's. In `Cogito` 1, this is formalised by the notion of data refinement between 'state machines' (see also [7] and [15]). A 'state machine' comprises a state schema, and initialisation and operation schemas.

A state machine A is data-refined by a state machine C, with respect to a representation relation R, and a correspondence between the operations of A and C, if R is a schema which relates the state of A to the state of C. A more detailed description of the data refinement process can be found in [4].

3 PROJECT ACTIVITIES AND OUTCOMES

This section makes some more detailed observations on project activities and outcomes.

3.1 Training

In general training in a particular task was received just prior to it being applied. We chose to schedule the training this way because it was thought that learning about the tools and techniques too far in advance of use would not be effective – there was a large volume of information to be absorbed that included unfamiliar and relatively complex concepts. However, this did mean that at any point in time there was little knowledge about what would be required in future tasks.

3.2 Safety Analysis

Fault Tree Analysis (FTA) was used to identify system and design level hazards in two stages. First, system level fault trees were developed, in which the test unit was considered as just one part of the overall system. The fault trees were refined until the level of detail matched that contained in the Operational Requirements Document. In the second stage, the fault trees were extended to consider the failure modes of the test unit as identified in the design.

From the leaf hazards identified in each FTA, a number of Safety Criteria were derived. At Formal Requirement Specification level, the main safety criterion we identified concerned completeness of coverage of the test unit: namely, that the software should perform all test assigned to it before pronouncing the equipment 'passed'.

3.3 Formal Specification

The specification of the system was structured into three levels - Top Level, Design Level, and Implementation Level. Capturing requirements in Sum was fairly straightforward.

However, achieving a level of abstraction appropriate to the specification level (top level, design level, implementation level) was found to be very difficult. Difficulty in working at an appropriate level of abstraction has been noted on a number of formal methods projects (for example, [8]). The skill of abstraction improves with experience.

Team members found the ASCII Sum notation more comfortable to work with than the mathematical (Z-style) notation, which involved a lot of work in translating unfamiliar symbols to corresponding concepts.

In producing the specification, a number of errors were detected in the requirements document. One example of this involved the discovery of incompleteness in the specification of what the test unit reports to the operator.

3.4 Specification V&V

3.4.1 Proof processes

Two classes of proof activity were carried out to assist in verification and validation of the specifications.

The first of these involved carrying out proofs recommended by the `Cogito` methodology to check the internal consistency of the specifications.

The second involved formalising the safety criteria derived from the Safety Analysis, and carrying out proofs to show that the formal specification entailed the safety criteria. The informal safety criteria required interpretation and formalisation before this could be carried out. In this process an number of different possible formalisations were considered, each of which required different proofs.

In this formalisation process, it was decided that it was better to produce a larger number of formal safety requirements that were easy to understand and prove rather than fewer formal

requirements that were difficult to understand and prove. As a result of this, after a few proof templates were worked out, relatively unskilled personnel were seconded to the project and carried out many of the proofs. The alternative approach (fewer, more complicated proofs) requires personnel with a higher skill level.

3.4.2 Error Discovery

Nine errors in the specification and in the formalisation of the safety criteria were detected during the proof processes.

3.4.3 Reviewability of Formal Proofs

CSC's quality system requires each project product to be reviewed. It was found that review of the formal (machine checked) proofs was difficult. It is felt that this can be addressed by some extra facilities in the Repository Manager and in Ergo, to more efficiently check that proof obligations have been generated and to check that all postulates on which outstanding proof obligations depend have been proved.

3.4.4 Difficulty of Proof

Proof was found to be a more difficult process than specification. However, after training and some experience, team members were able to carry out formal proofs using Ergo with little or no assistance from the SVRC. In fact, once certain proof styles had been worked out even personnel who had received no direct training could carry out simple proofs.

3.5 Refinement

Refinement is the process whereby design decisions are introduced and shown formally to satisfy previous specifications.

Data refinement on the communications portion of the specification introduced data design in two stages, introducing the relation between abstract messages and the communications byte stream, and detailing how parity and overrun errors were handled.

In comparison with traditional design, formal refinement is more restrictive. It was found that the availability of applicable refinement techniques influenced the way specifications needed to be structured.

The proofs required for verifying the data refinement were more difficult than those done in the verification and validation work. Team members found this work significantly increased their theorem proving skills.

Team members feel they need more experience with refinement techniques and that they would like to learn about different refinement methods so that the refinement effort could be optimised by selecting techniques appropriate to the problem. On the research side, the SVRC is interested in researching more flexible refinement methods.

3.6 Configuration Management

3.6.1 Version Management Policies

Changes to the specification were managed by the Repository Manager, which provides version control of specifications at the module level.

However, the user needs to develop a policy to control when new versions of the specifications are created and when the current version is modified. This issue needs to be looked at further because the simple policy used for this project would be inadequate for larger projects. If a general policy for version control had been available at the start of the project, then this could have been implemented within the Repository Manager.

3.6.2 Fine-grained Management of Proofs and Theories

The process followed introduces a number of points of iteration, in which previous stages need to be revisited. This process would be assisted by a finer grain of configuration management of theories and proofs.

For example, in the process of performing the formal proofs, changes and improvements were made to the specification. This required new

theories to be generated and existing proofs to be re-run. A finer grain of configuration management would cut down on the number of proofs which need to be re-run.

Again, in the process of developing design level fault trees, it was thought that changes to the system level fault trees were appropriate. Unfortunately by this time the informal safety criteria had been formalised and a large number of proofs completed. In this project then completed proofs were not updated to cater for the changes to the fault trees. Again, finer grain management of proofs would alleviate the problem.

Future work at the SVRC will address the issue of finer-grained configuration management of proofs and theories.

3.7 General Comments on the Process

It was recognized from the outset that the chosen application was far from ideal as a demonstration of the capabilities of the `Cogito` methodology: in particular, it was process-oriented and had a number of real-time requirements. (`Cogito` had hitherto been applied mainly to data-driven applications and this version of `Cogito` was not specifically designed to support development of real-time systems.) As a result, some 'process-oriented' safety criteria were more difficult to express and to prove, and some safety criteria could not be modelled at all due to their 'real-time' nature. Also, because the application was a test unit for an existing system, its communications interfaces were tightly constrained, and this meant the developer had very little or no design freedom.

Nevertheless, the process of hazard analysis, specification, proof and refinement proved to be a good framework for the development of safety critical software.

- Performing the hazard analysis, writing the top-level formal specification and formalising and proving the safety criteria provided enormous insight into the application.
- Writing the design level formal specification and doing the rigorous consistency and refinement proofs on this design was also helpful for giving insights into the application.
- Formal proofs were useful in reinforcing and adding to the understanding of the data types used in the specification, and in providing increased assurance.

To address some of the problems mentioned above, the SVRC is now investigating the addition of process-oriented and real-time capabilities to the `Cogito` methodology.

4 CONCLUSIONS

The pilot project has successfully demonstrated the feasibility of transferring formal methods to industry. The project threw light on some of the initial industrial perceptions of formal methods:

Industrial scale: The project showed that the methods are feasible, but productivity remains an issue. A full formal specification of the application was completed. Formal proof of consistency was performed to the point where it was clear it could be completed with little extra effort. Formalization of the safety criteria resulted in a large number of proof obligations; each individual proof obligation presented no significant difficulty (in fact, many could be proven automatically), but the sheer number was a problem. Only part of the application was refined through to detailed design, due to time pressures not entirely due to the methodology (see below).

The formal specifications for this application are comparable in size to the SRS, but are more precise. `Cogito` currently has no support for real-time aspects, and more direct support for process-oriented aspects would have been useful.

Tool support: The `Cogito` tool-set allows large specifications to be constructed, checked and managed, and reports to be generated. The provision of a modularity mechanism in the specification language

enabled a coherent and manageable specification to be constructed. Fine-grained configuration management of proofs and theories needs addressing, to reduce the amount of rework required each time changes are made to the specification or design. To increase efficiency of the proof tools, further population work is required.

Full tool support for development through to Ada statement-level code was not available during the project. Since then, however, tool support has been completed and is currently being evaluated in the SVRC by developing code for part of the test unit.

Training: The pilot project had 18 days of training and 22 days of workshopping involving SVRC staff. The learning curve was steep, but notwithstanding the novelty and complexity of the processes involved, and the lack of prior training, the development team members were able to successfully apply the methods with little extra support from the SVRC. The project experience underlined the importance of workshopping as a follow-up to initial training.

Mathematical skills: The principal CSC developer had only two weeks prior experience of formal mathematics. With respect to formal proof, SVRC staff were sometimes called upon to help with proofs, but generally once they had demonstrated how to tackle a certain kind of proof obligation, CSC staff were able to complete all other proofs of the same kind without further help.

Effort involved: Certainly there is work involved in applying formal methods. The additional work, beyond that required by traditional approaches, is involved with formalizing requirements and designs and proving properties of them; the result is improved assurance in the correctness of the final product. Also, the products are fully traceable and can be re-checked by tools after changes have been made, thereby substantially reducing the need for technical reviews.

A number of factors impacted on progress in developing the application through to code:

- The project was hampered by the delay of an SRS for the application being developed, which compounded the other delays introduced into the project.
- Substantial effort was invested in the course of the project in evolving and maturing the `Cogito` methodology and associated support tools (primarily based on feedback from the project). This had an adverse affect on overall progress. Subsequent projects would suffer much less from this impediment and would therefore progress substantially faster.
- Initial interaction between CSC and the SVRC was relatively intensive, but hampered by the remote nature of the communication, which introduced delays in many parts of the project. It was found that follow-up intensive workshopping for several days at a time was more effective. Note however that the need for on-the-spot support in subsequent projects, pursued by the same CSC staff, would be significantly reduced. Again, this would be a significant factor in improving the productivity of subsequent formal methods projects.

In more general terms, CSC now has a number of developers experienced in the use of formal methods who are in a position to offer the kind of support locally to CSC staff that the SVRC offered during the project. This encourages us to believe that subsequent projects, even with inexperienced staff, would also progress more smoothly, because of the availability of local expertise. CSC pilot project staff have subsequently made substantial contributions to another CSC project applying formal methods in V&V of safety-critical software.

The pilot project is one of the few documented, successful applications of formal methods in Australian industry. It will serve as a milestone against which we can judge subsequent industrial efforts in the application of formal methods.

The pilot project was the first attempt to apply the `Cogito` methodology and system to an industrially relevant application. At the end of the project, `Cogito` had evolved significantly from the initial versions available at the start of the project. The end result was a system that was substantially more usable and robust, with significantly improved coverage and efficiency. The next generation of the `Cogito` methodology is expected to profit greatly from the feedback received from this project.

Acknowledgments

The authors are indebted to other members of the pilot project group who have made significant contributions to the project: in particular, we thank Anthony Bloesch, Ed Kazmierczak and Mark Utting. Thanks also to the other members of the Pilot Project Steering Committee (Tony Cant, Chris Edwards and John Staples) for their encouragement throughout the project.

References

[1] Assessment of munition related safety critical computing systems. Australian Ordnance Council, August 1993. Pillar Proceeding 223.93.

[2] Defense system software development. U.S. Dept of Defense, February 1988. MIL-STD-2167A.

[3] Fault tree analysis (FTA). International Electrotechnical Commission, 1990. International Standard IEC 1025.

[4] A. Bloesch, E. Kazmierczak, P. Kearney, O. and Traynor, Cogito: A Methodology and System for Formal Software Development, *International Journal of Software Engineering and Knowledge Engineering*, Vol. 5, No. 4, December 1995, 599-617.

[5] A. Bloesch, E. Kazmierczak, P. Kearney, J. Staples, O. Traynor and M. Utting, A Formal Reasoning Environment for Sum – A Z Based Specification Language, *Australian Computer Science Communications*, **18**, 1, February 1996, 149-158.

[6] J.P. Bowen and M.G. Hinchey, Formal Methods and Safety-Critical Standards, *IEEE Computer*, **27**, 8, August 1994, 68-71.

[7] C.B. Jones, *Software Development using VDM*, Second edition, Prentice Hall, (1990).

[8] P.G. Larsen, J. Fitzgerald, T. Brookes, Lessons Learned from Applying Formal Specification in Industry, to appear in *IEEE Software*, May 1996.

[9] Ray Nickson, Owen Traynor and Mark Utting. Cogito Ergo Sum - Providing Structured Theorem Prover Support for Specification Formalisms. In *Australian Computer Science Communications*, Vol. 18:1, pages 149-158, 1996.

[10] J.M. Spivey, *The Z notation, a reference manual*, Prentice Hall, (1992).

[11] O. Traynor and A. Bloesch, The Cogito Tool Architecture, in *Australian Computer Science Communications*, Vol. 18:1, pages 97-106.

[12] O. Traynor and A. Bloesch, The Cogito Repository Manager. In *Proc. Asia Pacific Software Engineering Conf.*, Tokyo, IEEE Press (1994).

[13] O. Traynor, E. Karlsen, E. Kazmierczak, P. Kearney, Li Wang, Extending Z with Modules, in: *Australian Computer Science Communications*, **17**, 1, February 1995, 513-522 (1995).

[14] M. Utting and K. Whitwell, The Ergo Interactive Theorem Prover V4.0, Technical Report 94-14, Software Verification Research Centre, The University of Queensland (1994).

[15] J.B. Wordsworth, *Software Development with Z*, Addison-Wesley (1992).