Development of Practical Verification Tools[†]

D.J. King
R.D. Arthan
ICL Winnersh

Abstract

Current best practice for high-assurance security and safety-critical software often requires the use of machine-checked rigorous mathematical techniques. Unfortunately, while there have been some notable successes, the provision of software tools that adequately support such techniques is a hard research problem, albeit one that is slowly being solved. This paper describes some contributions to this area of research and development in ICL in recent years. This research builds both on ICL's own experiences in building and using tools to support security applications and on work carried out by the Defence Research Agency into notations and methods for program verification.

1. Introduction

Writing in a recent volume of this journal, B.A. Wichmann asks the question "is quality assurance impossible?" [Wichmann, 1995]. The answer is probably "yes", in that we can never have certain knowledge of the correctness of an operational system under all possible conditions. Nonetheless, as Wichmann points out, there are techniques which can give genuine added confidence in the fitness-for-purpose of a system. One such technique, generally referred to as "formal methods", involves rigorous mathematical modelling to validate the critical aspects of a design or implementation. Machine-checked mathematical proof can then serve as a powerful adjunct to conventional checking and testing. This paper reports on recent work within ICL in developing practical tools for specifying and verifying formal mathematical models.

[†] Published in The ICL Systems Journal. Volume 11. Issue 1. May 1996.

For the highest levels of assurance in security applications, UK and US government standards demand that security requirements be mathematically described and that mathematical modelling and proof be used to validate system architectures. For many years the High Assurance Team (HAT) at ICL Winnersh has been involved in developing and applying mathematical methods and tools for specification and verification of high assurance secure systems [Jones, 1992].

When we first began work in this area in the mid 1980s, the only tools available to support such work were academic research tools. We had some success in deploying one of these tools, namely Mike Gordon's Higher Order Logic (HOL) [Gordon, 1988] on several developments, including the UK's first product certified at UK Level 6. From the beginning, we found that we had to become tool-builders as well as tool-users in order to meet our customers' requirements. In particular, the customers generally mandated that work be presented to them using the Z notation [Spivey, 1992]. This made us devote a great deal of time to customising and extending the HOL tool to support Z-like notations.

In 1990, we began a three year research project, in collaboration with Program Validation Ltd. and the Universities of Kent and Cambridge. Our role in this project was to design and implement a completely re-engineered tool supporting specification and proof in HOL. In particular, we wanted a version of HOL which made it easier to provide support for other notations, such as Z. This project was very successful; by early 1993 we had a tool called ProofPower supporting both HOL and Z, which was sufficiently mature for us to make it available to other organisations and to give courses in its use. Since then, ProofPower has been used for applications concerned with security and on case studies for other types of application.

Z was primarily designed as a language for specification, and so expressive power was of more interest to its designers than the requirement to do completely rigorous proof. The language consequently has many features that provide hard problems for the implementor of automated or semi-

automated mechanical theorem-proving tools. We have continued to do research into these problems and to enhance the tool as opportunities have arisen.

Our main applications have been in high assurance secure systems. There is also considerable interest in Z and similar notations in the safety-critical community. Standards for safety-critical software [e.g., MOD, 1991] require the developer to provide an evaluator with convincing evidence that the software meets its specification. In safety-critical applications, e.g., avionics software, it has traditionally been an expensive and difficult task for both parties to prepare and maintain these so-called compliance arguments. The compliance argument for a system such as an engine controller might comprise scores of pages of narrative and mathematical material. Without tool support, it is difficult for the evaluator to relate the compliance argument to the program code and for the developer to keep it in step with the evolving requirements on the software.

As an outcome of some three years of research into tools for software integrity, the Systems Integrity Research Team at the Defence Research Agency, Malvern (DRA), have designed a notation for presenting compliance arguments [Sennett, 1992, O'Halloran et al., 1994]. This notation permits programs written in the SPARK subset of Ada [Carré et al. 1992] to be specified and verified against Z specifications. Unlike most other such formalisms, DRA's approach gives the user considerable flexibility over the level of rigour to be applied. In 1994, DRA let a contract to ICL to develop a tool supporting their notation. This tool is implemented on top of ProofPower and was delivered at the beginning of 1995. The notation and the tool are currently being assessed on a variety of case studies and being enhanced in the light of the experience gained.

The remainder of this paper is organised as follows:

Section 2 describes the ProofPower tool and shows it at work on a very simple security example using the Z notation.

Section 3 describes the Compliance Notation and gives a simple example of its use.

Section 4 gives some concluding remarks.

A detailed knowledge of Z is not required to read sections 2 and 3, but some familiarity with the logical symbols generally used in discrete mathematics will be helpful to understand all the technical details.

2. ProofPower

2.1 Overview

The heart of ProofPower is essentially a system for managing a database of logical theories and conducting proofs about the objects described by those theories. This part of the system follows the LCF paradigm [Gordon et al., 1979] and is implemented in the powerful functional programming language Standard ML [Milner et al. 1990]. Since ML is an interactive language, it serves not only as the implementation language but also as the command language for the system.

ProofPower follows the tradition of notations such as Z in supporting a document-based paradigm for developing and managing specifications and proofs. To write a specification is to produce a document in which the mathematics is interleaved with explanatory narrative. Fonts providing the necessary symbols are used to enable a near WYSIWYG appearance for the mathematical material when edited with a suitable editor.

Specifications and proofs are generally incrementally checked by ProofPower as they are being developed. When a fragment of the document being developed is to be checked, the user selects it with the mouse and then “executes” the fragment. This causes an appropriate ML command to be executed which will check the fragment and, if it is correct, cause the state of the ProofPower system to be updated appropriately. E.g., when a logical definition is executed, the logical content of the definition is added to the theory database.

2.2 A Simple Example

We are now going to present in the Z notation the information-flow security properties of a very simple system. The treatment will follow in microcosm parts of the approach recommended by the UK regulatory authorities for high-assurance secure systems [CESG, 1991]. Further examples and a more detailed discussion of the approach and of alternatives may be found in [Jones, 1992].

There are three parts to the example; first of all we identify the salient characteristics of the systems of interest and express a *security policy* by defining the subclass of systems which meet the desired information-flow constraints. We then describe a particular system which we will call an *architectural model.* Finally, we prove that the architectural model does indeed conform to the security policy.

2.2.1 Security Policy

We are going to consider systems whose state contains data at two sensitivity levels: unclassified data and classified data, and we will be concerned with security properties which restrict the data flows between the two levels. The internal structure of the data will not be relevant to our considerations. We can therefore use the following Z paragraphs to model the states of the systems of interest. The first paragraph introduces a type *DATA;* the second paragraph defines the set *STATE* to comprise pairs of *DATA* values labelled unclassified and classified.

z

$$[DATA]$$

z

$$STATE$$

$$classified : DATA;$$
$$unclassified : DATA$$

We will model individual systems by their state-transition functions. The set of all systems of interest is then modelled by the following Z paragraph which defines SYSTEM to comprise all functions from the set *STATE* to itself. Note that *SYSTEM* includes all possible transition functions, not just the secure ones.

z
$$SYSTEM \mathrel{\widehat=} STATE \to STATE$$

Now we want to distinguish within the set *SYSTEM* those transition functions which satisfy the required constraints on data flows. Informally, let us assume that what is required is that no transition shall cause information to flow from the classified component to the unclassified component. Another way of saying this is that observation of the unclassified component in the state after a transition does not allow us to distinguish two before-states which have the same unclassified component. We can therefore capture our informal notion of security in the following definition of the set *SECURE* comprising transition functions, *trf,* which have the desired property:

z
$$\begin{array}{l} SECURE \mathrel{\widehat=} \\ \{ \quad trf : SYSTEM \\ \mid \quad \forall st1, st2 : STATE \\ \bullet \quad st1.unclassified = st2.unclassified \\ \Rightarrow \quad (trf\ st1).unclassified = (trf\ st2).unclassified \ \} \end{array}$$

2.2.2 Architectural Model

We are now going to produce a model of the architecture of a very simple system. This model will be an element of the set *SYSTEM* defined in the previous section. We will define the transition function of the model in terms of two auxiliary functions, one to compute the classified part of the state, and one to compute the

unclassified part. The actual computation carried out by these functions will be irrelevant to the security of the system. We introduce the two functions, *c_trf* and *u_trf* using the following Z paragraph which simply asserts that *c_trf* and *u_trf* are some functions with the specified signatures.

z

$$c\_trf : DATA \times DATA \rightarrow DATA;$$
$$u\_trf : DATA \rightarrow DATA$$

We now use these functions to construct the architectural model of the system. The state transitions carried out by the system use *u_trf* and *c_trf* to compute their respective components of the state. The architectural decision embodied here is that while *c_trf* has both components of the state available to it, *u_trf* can only work with the unclassified component. This is a very simple model of the sort of separation which might be achieved in practice using hardware or software access control mechanisms of some sort.

z

$$system : SYSTEM$$

$$\forall st : STATE \bullet$$
$$(system\ st).classified = c\_trf\ (st.classified,\ st.unclassified)$$
$$\wedge \quad (system\ st).unclassified = u\_trf\ (st.unclassified)$$

2.2.3 Proof Of Security

In section 2.2.1, we have given a mathematical specification of information-flow security for a simple class of systems, and in section 2.2.2, we have given a mathematical construction of a member of this class which we believe to be secure. We can now test this belief by attempting to prove it mathematically. In this section we will show the steps of a ProofPower session in which this proof is conducted. We will suppress the Standard ML commands which control the proof, and just show the output of the interactive subsystem of ProofPower used to find proofs.

We begin by telling ProofPower the goal we want to prove, namely that the architectural model *system* is a member of the set *SECURE* of secure systems. ProofPower acknowledges this by echoing back our goal, together with a little bit of decoration designed to make later steps more readable.

ProofPower Output

$$(* \ ?\vdash \ *) \ \ulcorner_{Z} system \in SECURE \urcorner$$

Our first step is essentially to expand the definition of *SECURE,* resulting in the following goal. Note that ProofPower has replaced *SECURE* in the goal with its definition and then simplified away all the set notation.

ProofPower Output

$$\begin{array}{l} (* \ ?\vdash \ *) \ \ulcorner_{Z} \forall \ st1, \ st2 : STATE \\ \qquad\qquad \bullet \ st1.unclassified = st2.unclassified \\ \qquad\qquad \Rightarrow (system \ st1).unclassified = (system \ st2).unclassified \urcorner \end{array}$$

We now need to attack the universal quantifier which has appeared in our goal. This is done with a standard logical transformation which results in a goal with three assumptions. The idea is that we may now take the facts labelled 1, 2 and 3 as given and must deduce the bottom line from them.

ProofPower Output

$$\begin{array}{l} (* \ \ 3 \ *) \ \ulcorner_{Z} st1 \in STATE \urcorner \\ (* \ \ 2 \ *) \ \ulcorner_{Z} st2 \in STATE \urcorner \\ (* \ \ 1 \ *) \ \ulcorner_{Z} st1.unclassified = st2.unclassified \urcorner \\ \\ (* \ ?\vdash \ *) \ \ulcorner_{Z} (system \ st1).unclassified = (system \ st2).unclassified \urcorner \end{array}$$

We must now make use of the definition of *system*. After rewriting the goal with that definition, we arrive at the following goal.

ProofPower Output

$$
\begin{array}{l}
(*\ \ 3\ *)\ \ \ulcorner_{z} st1 \in STATE \urcorner \\
(*\ \ 2\ *)\ \ \ulcorner_{z} st2 \in STATE \urcorner \\
(*\ \ 1\ *)\ \ \ulcorner_{z} st1.unclassified = st2.unclassified \urcorner \\
\\
(*\ ?\vdash\ *)\ \ \ulcorner_{z} u\_trf\ (st1.unclassified) = u\_trf\ (st2.unclassified) \urcorner
\end{array}
$$

It is now apparent that the equation in the assumptions implies the truth of the bottom line. Using the assumptions as rewrite rules should complete the proof, and ProofPower does indeed tell us we are done. We then execute a command to save the newly proved theorem away in the database and it responds with:

ProofPower Output

$$
\begin{array}{l}
Now\ 0\ goals\ on\ the\ main\ goal\ stack \\
val\ system\_secure\_thm = \vdash system \in SECURE : THM
\end{array}
$$

3. The Compliance Tool

3.1 The Compliance Notation

The target application of DRA's Compliance Notation [Sennett, 1992, O'Halloran et al., 1994] is the provision of convincing arguments that a piece of code complies with a detailed technical requirement. In typical safety-critical applications, some parts of the requirements are likely to be of a mathematical nature, albeit informally expressed. The compliance argument will often be prepared quite late in the development life-cycle since much of it will depend on the fine details of the implementation; the argument will also need to be amenable to change as the software evolves. Since pragmatic constraints will prohibit full mathematical rigour at every stage of the argument, the notation must allow the developer to choose the level of formality at which to pitch the argument.

The Compliance Notation combines two key ideas:

- Literate programming [Knuth, 1984], whereby the design of a program is presented in a readable document in which the program text is embedded. In literate programming, the order of presentation of the program text is governed by the designer's preference for explaining the design rather than the programming language's rules for sequencing of program constructs;

- Refinement calculus [Morgan, 1990], whereby the designer can use mathematical predicates as a kind of abstract program statement and can assert that one such program statement "refines" (i.e., correctly implements) another. At one extreme, a predicate expressing the state changes to be achieved by the program as a whole constitutes a rigorous mathematical specification of the program. At the other extreme, an actual programming language command can be viewed as a predicate. A sequence of commands which refine the specification of the program as a whole is a correct implementation of that specification.

The Compliance Notation exploits these ideas taking a subset of Ada called SPARK [Carré et al. 1992] as the programming language, and using the specification language Z [Spivey, 1992] to express the mathematical predicates — these being amongst the more important languages of choice in military safety-critical applications. The ideas are applicable to most imperative programming languages, although, typically, a rigorous mathematical treatment would only be available for a "safe" subset of the language. The popularity of SPARK for safety-critical work suggests that using a small subset of a complex language is less of a problem than is often alleged. A discussion of programming language design issues is outside the scope of this document. [Carré et al. 1992] discusses the design philosophy for SPARK.

The literate script idea is realised by identifying places in the SPARK syntax where the user can use a place-holder instead of a programming language construct. This enables a compliance argument to be presented in a readable

document in which the order of presentation is not constrained by the programming language syntax. The construct to fill in a place-holder can be given in the document at the point where it is relevant to the argument. For example, the internal details of an intermediate data structure need not be revealed until they are necessary. Where appropriate, the syntax for the place-holders permits the user to specify program behaviour mathematically by means of Z pre- and post-conditions.

Fragments of program which are mathematically specified give rise to verification conditions. A verification condition (VC) is a mathematical proposition which, if true, guarantee the compliance of that program fragment with its specification. These VCs can be automatically generated from the Compliance Notation presentation of the program. The designer can then give either informal or rigorous justifications of the VCs to the evaluator as part of the compliance argument. If very high assurance is required, the VCs are amenable to fully machine-checked proof. A simple example of the Compliance Notation is given later in this paper.

The notation allows the user to choose how much of the program is to be treated with full rigour. Fragments of program which are not critical can be entered without any mathematics. For example, the user can elect simply provide the code for a subprogram body without having to specify it in any way. Where deisred, the user can even circumvent the rather stringent syntactic restrictions imposed by SPARK and insert arbitrary Ada code, for example to access library packages which do not conform to the SPARK subset. The notation makes it clear where these shortcuts have been taken, and the literate programming style allows clear documentation of what has been done and why.

The SPARK source of a program can be mechanically extracted from the document containing its compliance argument. Thus, in the interests of maintainability and traceability, the source document of the argument can serve as the master copy of the program source for future development purposes. If machine-checked proof techniques are used, then this can provide a very rigorous approach to impact analysis as software evolves.

3.2 The Compliance Tool

At the beginning of 1994, we began a project under contract to DRA to construct a tool to support the Compliance Notation. The basic requirement was for a tool to read a Compliance Notation script, perform syntax and type checking, and then output the SPARK program and the VCs as a Z document. Outputting the SPARK program is a relatively simple task, although not completely trivial because the program constructs in the script have to be assembled into the right order. The generation of VCs is a much more complex algorithm and DRA had devoted considerable effort to specifying all aspects of the tool that are critical to the soundness of this approach. This specification provided an objective and precise record of DRA's technical requirement.

The Compliance Tool, which was delivered at the beginning of 1995, is implemented on top of ProofPower. In addition to its support for specification and proof in Z that are described in section 2 of this paper, ProofPower provides some additional advantages for implementing this type of tool, including:

- Very high level implementation language: ProofPower is programmed using Standard ML a powerful functional programming language allowing complex algorithms for manipulating syntax to be coded easily and concisely.

- Library code for manipulating Z: the support for Z in ProofPower includes an extensive programming interface for manipulating Z syntax. DRA were particularly concerned that the implementation could be easily traced back to the specifications. An experiment to code parts of their VC generation algorithm in ML showed that it was straightforward to write the code so that implementation details do not obscure the connection with the specifications.

- Document preparation: ProofPower supports presentation of specifications and proofs in good quality documents via an interface to LaTeX. This would be easy to extend to handle the document preparation requirements for the new tool.

The development of the tool was very successful, both because of the good fit of ProofPower to the problem and because of the considerable efforts DRA had taken to specify their requirements on the tool.

3.3 An Example

To give a flavour of how the Compliance Notation is used, here is a simple example. The example is a very small one, involving only a few lines of Ada code, and is designed just to give the flavour of the Compliance Notation approach. Case studies have been carried out on life-size examples involving non-trivial algorithms and exploiting Ada's features for programming in the large.

Let us assume that we have to specify and implement an Ada procedure to carry out some simple calculations. The procedure has two input/output parameters X and Y; the procedure is required to set X to the sum of the squares of the input values of X and Y and to set Y to twice the sum of their product. The procedure might be specified in the Compliance Notation as follows.

Compliance Notation

$$
\begin{array}{l}
\mathit{procedure}\ EG\ (X : \mathit{in\ out}\ \mathit{INTEGER};\ Y : \mathit{in\ out}\ \mathit{INTEGER}) \\
\mathit{is} \\
\langle\ \mathit{space\ for\ local\ declarations}\ \rangle \qquad (1) \\
\mathit{begin} \\
\quad \Delta\ X,\ Y\ [\ X = X_0 * X_0 + Y_0 * Y_0 \wedge Y = 2 * X_0 * Y_0] \qquad (2) \\
\mathit{end}\ EG;
\end{array}
$$

Here, what we have is a regular Ada procedure declaration with certain parts replaced by place-holders to be filled in later. The first place-holder

(labelled (1)) makes room for the declarations of any local variables which are required, and includes a comment to that effect. The second place-holder (labelled (2)) stands in for the statements forming the body of the procedure and includes a mathematical specification of what the body must achieve. This specification statement is introduced by a Δ followed by a list of the variables which the body may change and a mathematical predicate in square brackets specifying the desired state change. In this case, the predicate requires that the output values of X and Y should be equal to the indicated expressions involving their input values (written X_0 and Y_0).

Note that apart from the decisions it records about the interface (e.g., the order of the parameters), there is no implementation bias in this specification of the procedure. Note also that the notation lets us use any mathematical assertions about the code as a specification statement, and so the tool can be used as a general means for reasoning about Ada programs. For example, one can have a specification statement which asserts that the computations to be carried out do not cause arithmetic overflow, or that the value assigned to X is not negative.

To implement the procedure, we must decide on an algorithm, and fill in the place-holders accordingly. By way of example, we are going to give a rather contrived solution (which might just be justified in practice by a desire to minimise the number of multiplications to be performed). First of all, we decide we need a local variable. The literate programming approach allows us to introduce local variables at the point where they are need in the explanation of the algorithm. In this case, we need the local variable straight away, and we ask for it as follows:

```
Compliance Notation
| (1) ≡
|         A: INTEGER;
```

Now we use some special syntax provided by the Compliance Notation to carry out what is called a *refinement step*. In this case, the refinement step supplies Ada code to implement the algorithm specified mathematically at label (2) above.

Compliance Notation

$$
\begin{array}{l}
(2) \sqsubseteq \\
\quad A := X + Y; \\
\quad Y := X * Y; \\
\quad Y := Y + Y; \\
\quad X := A * A - Y;
\end{array}
$$

Here we have implemented the procedure body in one refinement step. For a more complex algorithm, one can also use further specification statements in the refinement step, and so present the salient features of the algorithm in a structured way. A more extended example may be found on the original paper on the Compliance Notation [Sennett, 1992].

Now, it would be quite easy to make a mistake in the above coding. However, by analysing the statements in the refinement step against the specification statement being refined, the VC generator component of the tool allows us to check the coding by purely mathematical means. In this case, the only VC produced is the following conjecture (which is recorded in the Z document produced by processing the above script).

z

$$
\begin{array}{l}
?\vdash \ \forall X : INTEGER; Y : INTEGER \bullet \\
\quad (X + Y) * (X + Y) - (X * Y + X * Y) = X * X + Y * Y \\
\wedge \ X * Y + X * Y = 2 * X * Y
\end{array}
$$

This conjecture is easy to check by algebra, either on paper, or, even more easily, by using the automated proof facilities provided with ProofPower, which include a decision procedure covering a useful range of arithmetic facts. This decision procedure will give a completely automatic proof of this conjecture, and, if it were to fail in a simple case like this one, then the most likely cause would be an error in the coding.

Note that this type of formal verification is essentially complementary to testing. Testing on the one hand generally deals with the actual object code in a situation as close as possible to the real target environment; while

problems arising from interaction with the environment can be detected by testing, time constraints almost invariably prohibit complete test coverage. Formal verification as discussed here is concerned with the algorithm as expressed in the source language. While verification will not normally detect problems with the environment, it does offer complete coverage at the algorithmic level. In a sense, mathematical reasoning allows us to consider an infinite set of test cases in finite time.

To conclude the example, we can show the Ada code generated by processing our script:

```
SPARK Program
PROCEDURE EG (X, Y : IN OUT INTEGER)
IS
  A : INTEGER;
BEGIN
  A := X + Y;
  Y := X * Y;
  Y := Y + Y;
  X := A * A - Y;
END EG;
```

We have presented a very small example. This of course raises the question of whether the methods and tools will scale up to real-world problems. Work on case studies to date has shown that both the method and the tool will handle significant examples. For instance, an Ada package taken from the safety-critical parts of the operator interface in an actual development has been specified and verified using the notation. The specification is about 25 pages long including a comprehensive narrative relating the mathematical treatment to the natural language statement of requirements. Processing the specification produces about 300 lines of Ada and 61 VCs. 28 of the VCs can be proved automatically. The proofs of the remaining VCs vary in difficulty from relatively simple checks on the correctness of control logic to more complex reasoning to unwind the structure of the code and its specification and then to verify the mathematical facts

underlying the algorithms used. Carrying out the proofs took about 15 days including 5 days of familiarisation and analysis.

4. Conclusions

Research into methods and tools for high assurance security at ICL and at DRA for safety-critical applications has resulted in novel tools. These tools facilitate the use formal methods for securing increased confidence in fitness-for-purpose. We have illustrated the use of the tools on toy examples taken from opposite ends of the development lifecycle. Indeed, requirements validation on the one hand and code verification on the other seem to be the niches where rigorous mathematical methods have much to offer.

Readers familiar with the “European” tradition of formal methods will note that both the examples we have presented differ in many respects from the use of notations such as Z and VDM as design and development methods as generally described in the literature. Too often, formal methods have been oversold as a panacea, when in fact, their main benefit would seem to be in particular niches where the details of the method have to be adapted to fit the special characteristics of the problem at hand.

In the case of the Compliance Notation, the Z notation and the SPARK subset of Ada are brought together to combine the expressive power of Z with the respected software engineering advantages of SPARK in the safety-critical domain. The capability to develop fully machine-checked proofs using ProofPower offers a very high degree of assurance, while the flexibility inherent in the Compliance Notation permits the user to choose an appropriate level of formality.

We would not claim that the tools and methods we have described are currently appropriate for mainstream, low-to-medium assurance software development, or, indeed, that they ever will be. It seems

likely that the costs in terms of human skills and elapsed time of rigorous mathematical methods will rarely be justified in cases where cost and quality can simply be traded off against one another. Nonetheless, as software plays an ever-increasing role in safety-critical applications, we believe that such methods will become an important part of the developer's battery of validation and verification techniques.

Acknowledgements

We would like to thank Colin O'Halloran, Chris Sennett and Alf Smith of DRA for their painstaking technical work in designing and specifying the Compliance Notation, for their assistance and advice with the development of the tool and for providing comments during the preparation of this paper.

The ProofPower tool on which the implementation was based has benefited from contributions from all of the other past and present members of the High Assurance Team: Kevin Blackburn, Adrian Hammon, Andrew Hayward, Barry Homer, Roger Jones, Gill Prout, Geoff Scullard and Roger Stokes.

References

CARRÉ, B.A., JENNINGS, T.J., MACLENNAN, F.J., FARROW, P.F., GARNSOWRTHY, P.F., "SPARK - The SPADE Ada Kernel". Program Validation Limited. 1992.

CESG. "Computer Security Manual F — A Formal Development Methodology for High Confidence Systems". GCHQ. 1991.

GORDON, M.J.C., MILNER, A.J, WADSWORTH, C.P. "Edinburgh LCF". LNCS volume 78. Springer. 1979.

GORDON, M.J.C., "HOL: A Proof Generating System for Higher Order Logic". In Proceedings of the 1989 Banff Conference on Hardware Verification. G. Birtwistle and P.A. Subrahmanyam (Eds.). Springer 1988.

JONES, R.B. "Methods and Tools for the Verification of Critical Properties". In Proceedings of the 5th Refinement Workshop. Cliff B. Jones, Roger C. Shaw and Tim Denvir (Eds.) Springer 1992.

KNUTH, D.E. "Literate Programming". Donald E. Knuth. The Computer Journal. 27(2):97-111, 1984.

MILNER, R., TOFTE, M., HARPER R. "The definition of Standard ML". MIT Press. 1990.

MOD. Interim Defence Standard 00-55. "The Procurement of Safety-critical Software in Defence Equipment". Ministry of Defence. April 1991.

MORGAN, C.C. Programming from Specifications. Prentice-Hall, 1990.

O'HALLORAN, C.M., SENNETT, C.T., SMITH, A. "Specification of the Compliance Notation for SPARK and Z". DRA Technical Report DRA/CIS/CSE3/TR/94/27. In three volumes. 1994.

SENNETT, C.T. "Demonstrating the Compliance of Ada Programs with Z Specifications". In Proceedings of the 5th Refinement Workshop. Cliff B. Jones, Roger C. Shaw and Tim Denvir (Eds.) Springer 1992.

SPIVEY, J.M. "The Z Notation: A Reference Manual" Second Edition. Prentice-Hall. 1992.

WICHMANN, B.A. "Why is it difficult producing safety-critical software". Ingenuity: the ICL Technical Journal. Vol. 10 Issue 1. May 1995.

Information about DRA's research into Software Integrity including the specification of the Compliance Notation [O' Halloran et al., 1994] may be found on the World Wide Web at URL:

http://daedalus.dra.hmg.gb/hewitt/swi/swi.html

Technical and marketing information on ProofPower and the Compliance Tool may be found on the World Wide Web at URL:

http://www**.to.icl.fi/ICLE/ProofPower/index.html**

Biography

David King

David King has a Ph.D. in Computer Science and joined ICL in 1988. He has been involved in the design of secure applications and in the development and application of ProofPower. He led the team that implemented the Compliance Tool and has subsequently contributed to work on enhancing it. David recently left ICL to join Hoskyns as a security consultant.

Rob Arthan

Rob Arthan has a Ph.D. in Pure Mathematics and has worked at ICL since 1985, first on compiler technology for ICL's Distributed Array Processor, and since 1989 on using formal methods on security applications and on developing tools for formal specification and proof. He led the team that designed and implemented ProofPower and was heavily involved in the development of the Compliance Tool.