UNLIMITED

Report No. 89005

Report No. 89005

ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

TOOL SUPPORT FOR THE PRODUCTION OF HIGH INTEGRITY SOFTWARE

Author: C T Sennett

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

April 1989

0045474

CONDITIONS OF RELEASE

BR-110738

.... U

.... Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 89005

Title: Tool support for the production of high integrity software

Author: C T Sennett

Date: April 1989

Summary

This report was commissioned by the UK computer security policy authority. It discusses the software tools required for the production of trusted software, following the guidelines given by the UK systems security confidence levels. Recommendations are given for the development of new tools and techniques where appropriate.

Contents

1 Introduction

High integrity software is that software which has been produced to a given level of *assurance*: that is, the confidence that the trustworthiness of the software is, if not exactly quantifiable, at least known to the extent that one can make reasonable judgements about the expectation of error. These judgements are based on assurance criteria which specify such things as means of production and documentation. An *evaluation* assesses an implementation to see whether the criteria are satisfied. If so the software may be certified as meeting a given assurance level, where the level is determined by the set of criteria satisfied. The assurance levels are arranged in an order giving progressively increasing confidence in the trustworthiness of the product, but also requiring increasing effort in implementation. Assurance levels are essential for secure systems which must be certified and accredited before they are allowed to be used. They are also of interest to all producers of software who are concerned about the quality of the product. The lower assurance levels are applicable to high quality software in general and can be justified on cost grounds if the whole life-cycle is taken into consideration. The higher assurance levels are applicable to safety critical software.

Assurance criteria are concerned with all aspects of the production of a system and many of them may be supported by suitable tools. This report is concerned with the various ways in which tool use can support the satisfaction of assurance criteria and gives both long and short term recommendations for the development of tools which would meet this need. As security critical software is the immediate concern we shall be starting with the existing position within the security community where there is some emphasis on the use of the Z specification language [Spivey 1989] and software analysis tools such as MALPAS [Rex, Thompson & Partners, 1987]. Other techniques and languages are necessary and these will also be discussed. Two factors affect the context. The first is that high integrity software is produced to contract. It is important that the techniques and tools recommended should be such that they may be specified in a contractually binding procurement specification and that one can gain a reasonable understanding of the costs of their use. The other factor is that high integrity software requires evaluation. This means that assurance tools are directed towards the timely production of evaluation evidence, rather than towards the development of the software making up the operational system. The evaluation process is an impediment to the production of software and so the usability of the tool must be assessed from the point of view of the cost to the implementors as well as from the point of view of the usefulness to the evaluators.

To give a shape to this discussion it will be related to the six aspects of assurance given in the UK confidence levels [CESG 1989]. These are as follows:

1. Specification of security requirements
2. Architectural definition
3. Implementation
4. Evaluation
5. Documentation
6. Configuration control

A few words on each of these aspects is in order. The *Specification of security requirements* is concerned with the degree of precision with which the trusted functionality and trusted requirements have been specified. This can range from loose specifications in natural language, through precise definition to fully precise mathematical descriptions.

The *Architectural definition* is concerned with the way in which the requirements are satisfied by the high level design. It deals with the specification of the trusted components, software or major system components such as one-way regulators, and the way in which they depend on each other to meet the requirements.

Implementation is concerned with the methods of production of module specifications and the actual software which meets these specifications. This is an area which has been given most attention in the academic community and there are several techniques which will add to the assurance of this process.

Evaluation is concerned with two aspects: establishing the quality of the implementor's methods of production and with direct assessment of the evaluation evidence within the context of the operational system. Tools may usefully be used to support assessment.

Documentation is an essential element in any trusted system. Apart from the usual design documentation, trusted software requires additional evidence in the form of a rationale for trusted and untrusted components, trusted paths and covert channel analysis. Documentation is the most important deliverable for evaluation and its production may be assisted by a range of tools. For the most part these correspond to standard office automation techniques, but the problem is to integrate them into the software development process.

Configuration control is necessary for the maintenance of the status of the evaluated software after hand over of the system. It is an essential requirement during development if the system is to be capable of being evaluated. During development the evaluators must be certain that what they are evaluating is up to date and cannot be corrupted after it has been assessed.

2 Specification of security requirements

Security requirements must be stated in natural language but extra assurance is gained by the use of a formal specification language. The extra precision gained by the formality is simply not available in natural language. Precise, legalistic English enables the precision of natural language to be improved, but for anything other than trivial concepts the improvement is gained at the expense of understandability. A formal specification language, well supported by tools, encourages the use of concise definitions with mathematical precision. Indeed, it can be argued that formal specifications are cost-effective for all software in that they allow implementations to proceed more rapidly and the act of specifying enables the requirement to be made more precise.

The archetypal security model, namely that of Bell and LaPadula [1976], is a formal model, but written in a notation which is informal in the sense of not having a formal syntax or semantics. The notation used is that of standard mathematics, following conventions adhered to by most mathematicians. This means there is no way of checking the model except by reading it. Currently, many specification languages are defined including Z, VDM, CSP, LOTOS and the various languages associated with the US verification tools such as Gypsy and EHDM. It seems reasonable therefore to require that high integrity software should be specified using a language with a well-defined syntax and semantics and the specifications produced should be checked mechanically.

Given a security requirement expressed, say, in 10 or 20 pages of Z, how is it possible to assess the quality of what has been written? The first technical consideration is that the specification should be *consistent*. This will ensure that it contains no contradictions and represents the specification of something which could actually be implemented. Tools to support this aspect of processing a specification language are at a rather primitive state of development. What is required is a specification of the proof obligations incurred by using the various structures within the specification language and a tool to compute them and possibly prove them. Consistency checking has little tool support at the moment. In default of automatic checking a carefully presented specification should be required to address these issues and present the various theorems which need to be proved in order to establish consistency.

Technical consistency is not the most important aspect of quality, certainly as far as security is concerned. An inconsistent specification cannot be satisfied so no implementation is possible. Conversely, if the specification is implementable, it is consistent, so it could be argued that this will be checked anyway. An evaluation of a security model will be more concerned with the *validity* of the model and with the absence of *vulnerabilities*. Validity is concerned with the question of whether the specification actually captures the user's requirements. In the case of security, the specification should accurately model the security properties required. Vulnerabilities can arise from at the specification stage when the user's requirements conflict with the desired security properties. A typical example is a covert channel

in a security model which may allow information flows using error codes and file names defined for the user functionality.

Validity is a difficult issue to address with a tool as it will almost certainly involve some human interaction. Bearing this in mind, one of the prime requirements for a security model is that it should be understandable. Z supports this issue very well *provided* the narrative between the Z text is always supplied and is of reasonable quality. Industrially, one would expect to see documentation standards reflecting this. Other aspects of understandability are more elusive. Generally, a model based specification, such as is the usual style with Z, will be more understandable than an algebraic specification, such as is obtained with OBJ [Goguen and Tardo 1979] for example, although the algebraic approach has advantages in the formal verification of the specification. The model must be in terms the user can understand and yet be reasonably realistic in terms of how the model may be mapped on to a computer. There is some conflict in these requirements although they are not entirely opposed: the user will usually have cast the requirements with machine oriented aspects in mind.

Vulnerability is more susceptible to analytical techniques than validity. A typical security model would specify access control policies. Fundamentally, the security properties required concern the control of information transfer, so it is useful to be able to reason about the information flow properties of the model. This reinforces the need to have a realistic model of execution which can be used, say, to specify non-interference properties between inputs and outputs and hence to highlight covert channels. This example indicates that quality in a security model can only be assessed within the context of a risk assessment. The quality criteria for a security model for a CCIS will be different from that of a key distribution centre because the perceived threats and vulnerabilities will differ. A covert channel analysis may be necessary in one case, but not in another. A good security model allows the property required to be expressed within the framework provided by the model. Consequently, it will be possible to demonstrate the absence of vulnerabilities, or to indicate their nature, by means of theorems requiring the properties to be present. Thus requirements for vulnerability analysis will often reduce to the requirement to be able to reason about the model.

Given this groundwork, one can make the following specific, tool-based, recommendations for development.

- Standards for specification languages should be supported. This will make tools more widely available and encourage the widespread adoption of formal specification.

- Proof obligations to establish the consistency of specifications should be established and tools developed to generate and prove them.

- Documentation standards for use with specification languages (and Z in particular) should be developed.

- Specification languages should provide the ability to write down theorems and tools should be provided to support their proof. Specifically a proof theory for Z needs to be developed.

3 Architectural definition

The architectural definition is concerned with identifying the trusted functionality and the major breakdown of the system into hardware and software components. The traditional view of this process is that the system will be centralised with the main security aspects controlled from a large central ADP component with essentially subservient workstations and networks. However, modern computer systems are nearly always distributed. Modern secure systems will often involve trusted network components and in this case one needs a description of the system which can be used to give a breakdown into components, with a clear definition of the trust requirements of each component. These two points of view require rather different approaches and so they will be treated separately.

The traditional view stems from the "Orange Book", the US DoD's Trusted Computer System Evaluation Criteria [DoD 1985]. Here, the architectural definition is called a "Descriptive Top-Level Specification" (DTLS) and for increased assurance this must be supplemented by a "Formal Top-Level Specification" (FTLS) which is related to the formal security model. Typically, a security model will be concerned with abstract entities called subjects and objects while the architectural design will be concerned with implementation entities such as processes and files. The key issue as far as security is concerned is maintaining traceability of the security requirements in the security model through the design and on to the implementation. In making the transition from security model to architectural design, many different entities in the design will correspond to objects and subjects, so the transition is akin to instantiation in this respect. However, certain minimum features of subjects and objects such as security labels must be present in the design, so in this respect the transition is rather like refinement (see section 4). For these reasons, the transition is usually done by a combination of formal and informal techniques, although there seems to be no reason why it should not be treated completely formally as a data refinement with the security model being repeatedly instantiated for each class of subject and object.

Tool requirements for this approach are no more than are needed for implementation, which will be described later. However, what is really required to be practically useful is a number of worked examples. In the open literature, there is only the Bell and LaPadula paper [1976] and the rather less available paper by Rushby [1985] on the basis of the MLS tool for Enhanced HDM. Both are valuable but they are unsatisfactory for practical use. The topic of writing a security model is covered in a general way in [Sennett 1989] but the development of security models into a FTLS tends only to be covered in project-specific documents.

A distributed system view of an architectural definition is more in keeping with the current approach to computing systems. The problem here is that a state machine description of the world becomes inappropriate. As many security models are expressed in these terms they provide an inadequate basis for expressing the architecture. It seems likely that process and trace oriented security models would more accurately portray distributed architectures, but apart from non-interference

assertions [Goguen and Meseguer 1982] and work at Oxford using CSP [Jacob 1988] little experience has been gained in their use.

From the tool point of view, the question is one of languages. Because of its rich structure in support of sequences it is possible to manipulate trace specifications in Z. Our report on separability [Sennett and Macdonald 1987] is one such use in which a security model, separability, is refined into an architectural definition suitable for use with SCP2 [Bottomley 1986]. While this approach was successful, nevertheless the feeling remained that a specifically process oriented language such as CSP would be more suitable for capturing the security requirements and for demonstrating conformity with a subsequent architectural definition. A particular problem in this area is concerned with the decomposition of a system into its components. The architectural design should provide a specification for the individual components and make visible the dependencies between them. Work by Neely and Freeman on trust domains [1985] is very relevant here and it is interesting that Jones [1987] was able to capture most of the formalisation of dependencies within the Z framework, but slightly extended.

A further comment on architectural definitions applies to both points of view. Security requirements are not concerned only with functional aspects. Associated with each function is the requirement that it should introduce no side-effects and that the controls provided should not be capable of being by-passed. At each stage in the transformation from security model, through architectural definition through to implementation, the non-functional security requirements change, because they now need to be expressed within the context of the refined definition. In the traditional approach, the FTLS should allow an estimate, for example, of the covert channel bandwidth which is now provided as a result of introducing error messages and return codes. If this is to be done formally, it is necessary to maintain the execution model at the architectural level, which may be cumbersome. For the moment, it seems to be the case that covert channel analysis at this level is best done informally.

A trust domain approach should, in theory, allow the non-functional aspects of security to be captured and hence should suffer less from these problems. However, this must remain a conjecture until the approach has been tried in practice.

The recommendations for architectural definition are thus more for the development of technique rather than for specific tools. They are as follows:

- Examples of the relation of access control security models to the formal design specifications need to be available.
- More experience is needed in the expression of security properties in process oriented languages.
- As a useful interim measure, it would be helpful to implement the work of Jones on trust domain operators for Z.

- Techniques for integrating process based specification in, for example, CSP with sequential specifications in, for example, Z need to be developed.

4 Implementation

Tools for implementation are normally concerned with assisting the implementation process: however, for high integrity software, tools are also required to demonstrate the trustworthiness of the implementation. For the most part these will be concerned with demonstrating compliance, namely that the implemented software satisfies its specification. Note that security requirements are different from the standard requirements for correctness and transformations preserving correctness will not necessarily preserve security properties. Indeed there are some theoretical reasons for believing that the security properties must be stated anew for each step in the transformation of the specification into an implementation. The maintenance of the link between specification and implementation is a most important aspect of assurance. Errors actually manifest themselves at the implementation stage. Unless the specification has actually been traced through to the implementation, it has served little purpose except as a mind clearing exercise.

The relation of a specification to an implementation may be established either formally or informally, depending on the degree of assurance required. The demonstration should follow the mathematical principles involved, and for a formal implementation the term *refinement* is used. In other words, an implementation is correct if it can be shown that it is a refinement of the specification. Note that it is important to distinguish this assurance aspect of demonstrating correctness from the process of producing the implementation in the first instance. Refinement may or may not be of use in the production of software, but it is essential for demonstrating correctness.

The refinement techniques depend upon the formalism used for both the specification and the implementation. For example the state based specifications typified by normal Z or VDM usage follow the simulation notions proposed by Milner [1971] and developed by Jones [1980]. Algebraic specifications use the concept of homomorphisms while trace based specifications must be unwound. All are probably equivalent at some deep level, but most experience has been gained with the state based approach, which is well suited to sequential software. Algebraic methods are particularly suited to relatively low-level properties, so they will be discussed under analysis and test in the section on evaluation. Process based refinement is likely to become important with the emphasis on trusted networks and components and as this technique is not well understood, it is important that it should be properly researched.

State based refinement on the other hand is a well understood technique. The barriers to its use in practice are largely questions of lack of tools and notations. It is usually divided into two categories: operation refinement and data refinement. Operation refinement is relatively easily understood as it is related to the approaches of the traditional verification systems such as Gypsy and VDM. Briefly, it is concerned with relating the target language's control structure to the logical structure of the specification. For example if the specification requires the

establishment of a state satsifying the predicate $G \wedge P \vee \neg G \wedge Q$ then operation refinement tells us that this may be achieved by the program

IF G
THEN establish P
ELSE establish Q
FI

Data refinement on the other hand is concerned with changes of representation. The abstract variables of the specification, which may be expressed in terms of sets, must be represented by the concrete variables of the program, such as lists or arrays. To carry out refinement it is simply necessary to provide an *abstraction invariant* which relates the concrete values to the abstract ones. Using this, the operations necessary in the implementation can be derived from the abstract operations specified.

So much for the theory. The practical problem is concerned with notation and tool support. The fundamental reason for doing refinement is to *demonstrate* that a given specification has been satisfied. Essentially, the requirement is to reason about fragments of program with respect to fragments of specification and neither specification languages nor implementation languages cater for this fragmentary approach. It is for this reason that the work of Morgan at Oxford [1987] is important. In a series of papers this author has developed a formalism which allows a Z-like specification language to be merged with an implementation language. The formalism allows an incremental style of presentation ideally suited to evaluation. This formalism provides the basis for a refinement language, or rather for a series of languages, one for each implementation language chosen.

Tool support for this methodology presents a series of problems, none particularly insuperable. First of all the notation has to be defined and the target languages chosen. The first among these would be Z itself as the development of a formal design at a high level can be carried out entirely within the Z notation. This can probably be achieved with a few minor additions to the Z syntax although the result would be a different language containing some meta-notions which are statements about specifications rather than specifications themselves. Other target languages may be chosen pragmatically, probably on the basis of the formality of their semantics. This is because the work involved in developing a target specific refinement notation is mainly concerned with developing the logical foundations. The actual tools to process the languages are fairly straightforward as there is no need to compile the refinement notation directly: instead, the implementation language may be filtered out of the refinement notation and compiled independently.

Given a notation, tools for syntax and type checking and programs to filter out the implementation are relatively easily provided. However, three problem areas remain. The first is that refinement steps incur proof obligations so one needs a tool to compute these and a tool to carry out the proofs. The requirements for this will be discussed under the heading of evaluation. The second problem area is that the refinement technique will only be of value if refinements may be modularised.

Suitable refinements for use with common functions and data structures need to be stored in libraries and a large part of the effort in making a refinement system will be incurred in defining useful refinements from more primitive elements, so that the reasoning process may be carried out at a reasonably high level. Finally, some refinements, notably from the mathematical integers to computer arithmetic, are probably best done semi-automatically by special purpose tools. Again the problem is to decide what form these tools should take and how they fit into the development process.

Clearly, there is a large amount of work to do in the production of a suitable toolset. In the meantime it is possible to follow the spirit of the refinement process without this being checked by machine. The minimum requirements here would be a document describing the refinement in an incremental style. The informality in the process arises from the fact that the notation is not checked by machine and the refinement steps are assessed informally. A high assurance would be gained by having many small refinement steps which could be checked manually, whereas for a normal level of assurance quite large steps could be made, which would at least give some guide as to what the correctness criteria were. A reasonable target level, giving a high level of assurance for a relatively modest cost, would be a requirement to exhibit all abstraction invariants and pre- and post-conditions for all procedures. This could then be used to derive compliancy conditions for program analysis. A case study of refinement carried out with a high level of formality is contained in [Macdonald *et al*, 1989].

Finally, it is necessary to return to the point made at the beginning of this section that the formal methods must be integrated into the tools and techniques used normally for the production of software. This will have an impact on the derivation of the specification in the first instance and on the relation of the refinement to the design methodology chosen.

In summary, the recommendations for tool development are as follows:

- A notation needs to be developed to express refinement. This notation should incorporate a module system and a number of case studies need to be undertaken to prove its worth.
- A tool structure needs to be developed which will allow proof obligations to be computed and verified and target languages to be produced.
- Implementation languages need to be assessed with a view to their use with refinement.
- The formal methods and tools need to be integrated within normal structured programming development methods.

5 Evaluation

The activity of evaluation consists of the assessment of the developer's methods, system documentation and proofs, and the analysis and test of the delivered system. Tool aspects of documentation are considered later under a separate heading; this section will be concerned with proof, analysis and test tools.

5.1 Proof

The purpose of a proof is to convince an evaluator that the implementation is correct. Although a skilled professional programmer might like to convince *himself* that his software is correct by outlining a proof, for high integrity software the evaluator will normally be a third party. Given this, the presentation of the proof, which will be called the proof document here, is all-important. On general grounds, a proof ought to lead to a greater understanding of the problem. A proof document should contain formal and informal, natural language, statements and the proof should have a structure which can be grasped as a whole. These requirements also apply to mechanically checked proofs. The view taken here is that the proof document is the main deliverable required for evaluation of a proof. An informal proof, required for rigorous levels of assurance, would simply consist of this document. The evaluator has to assess whether the proof steps are clear enough to form an acceptable proof. For higher levels of assurance the proof steps may be checked mechanically. The details of the mechanical proof need to be available to the evaluator, rather as the compiled code needs to be available when assessing the implementation. But generally, evaluators will be working with the high level description, that is, the proof document, just as with the implementation they will normally be using the source text.

The proof document needs to be in the same terms as the proof statement. Proof obligations may arise from the specification itself, to express consistency or security properties, and from the refinement, to express the correctness of the refinement. Given the use of Z for specification, both of these may be expressed as Z theorems. In writing a proof down, the natural way to do it is backwards. You write down that which is to be proved as the first step and then break this goal down into smaller goals, the proof of which would entail the overall goal. The process is repeated until the goals are axioms or previously proved theorems. This style of proof calls for a notation rather like the refinement notation, in which parts of a greater whole are named and then treated in turn within a context implied by the position of the part within the whole. Note that the production of the proof document is a rather different process from the production of the proof itself. The latter will normally be done by some interactive process, possibly a proof editor or some such tool. The proof document is a *record* of the proof when it is in its final acceptable state.

For informal proofs therefore, one needs a notation developed from the Z notation, to express the proof. For mechanically checked proofs one needs in addition a formal basis for manipulating Z theorems. This is a much wider topic than any which has

been touched upon so far. This is not the place to give a survey of formal methods for proof, but a way forward based on UK technology is as follows.

First of all, higher order logic seems to be the best foundation logic to express the proof theory. This logic provides a foundational system in which one may hope to express other logics, for example, those associated with CSP. It further provides a "meta-logic" approach in which one can express meta-level statements such as the induction principle for the natural numbers. This allows proof statements which would otherwise have to be expressed in the proof theoretic meta language to be expressed within the logic itself, which is a more consistent approach. Note that the variety based approach adopted by Spivey [1988] in his semantics for Z cannot be used for a proof theory directly, because theorems consist of statements about Z and semantics is concerned with what a Z specification actually means. It is not possible to give a semantics for a Z theorem, because the theorem is a statement about a Z specification. However, a Z proof theory has to be validated against the Z semantics, to ensure for example that $\wedge$ and $\vee$ have the same properties in the specification and the proof theory.

Given the use of higher order logic, the natural tool to use is Gordon's HOL [1987] and indeed this seems to be the best approach in the short term. Blackburn and Jones [1987] give a suitable translation scheme for a subset of Z which seems to be a viable approach to the immediate problem of attacking Z proofs with a mechanical tool. However there are a number of drawbacks, for example the incompatibility of the type systems, which make one want to adopt a different theorem proving basis. Given this, one can question the basic theorem proving approach which has been adopted in HOL, which is closely based on that of LCF. In both of these systems inference rules are represented by functions in the meta-language. This means that derived rules correspond to composition of functions and leads to some awkwardness in the handling of backwards proof. In fact the way of doing a backwards proof in these systems is to accumulate the forwards proof, which must surely lead to inefficiencies. Both Isabelle [Paulson 1986, 1988] and Genesis [Harwood 1987] provide alternative approaches in which the rule has a representation and derived rules are, in effect, data. Both of these approaches would seem to have advantages over HOL, particularly for Z where derived rules are certain to play an important part.

Apart from this, the LCF paradigm of theorem proving seems to offer the best way forward. In other words, the development of a proof should be seen as the development of a program in the meta language to carry out the proof. This carries with it the idea that commonly required proof tactics may be kept in modules which would have the effect of raising the level of automatic proof checking.

To summarise, recommendations for proof tools are as follows:

- A notation for Z proof documents needs to be defined.
- A suitable proof theory for Z needs to be developed.

- Tools to support the interactive development of proofs for Z need to be implemented.

5.2 Analysis and test

Analysis and test may be carried out both at the specification and implementation levels. For specifications, most forms of analysis will be carried out during the course of generating proof obligations, in other words, analysing for consistency. It is possible that simplified forms of analysis, such as computing dependencies, may be of use as an intermediate step on the way to proving full consistency. Testing of specifications is linked with the idea of animation. At its simplest, this might involve checking that a theorem is in fact true given some values for the variables or checking the effects of an operation in a given state. Both of these involve execution of the specification which implies some restriction on the type of specification to which this would be applicable.

At the implementation level, analysis and test may be carried out for simple programs using techniques which are currently available. The immediate problem, therefore, is the integration of the existing tools with the new formal verification ones proposed. In particular, it may well be useful to use the refinement notation to provide assertion annotations for compliance analysis. In the longer term, sophisticated programs will call for a more advanced computational model than is provided by the current analysis systems. Work on advanced algebraic techniques which would support the analysis of the full range of software is being undertaken in the Ten15 programme [Foster 1989] and this will need to be supported by the provision of suitable tools.

Thus, for analysis and test:

- A study needs to be made into the benefits of specification testing, over and above the analysis that may be necessary for computing proof obligations.

- A study also needs to be made into the integration of implementation language analysis tools into a formal verification environment.

- Work on extending analysis methods to the full range of software will need to be supported.

6 Documentation

Documentation is one of the most important aspects of a high integrity system. It is also one of the deliverables most likely to be missing as it is always the documentation which suffers when deadlines grow imminent. While there is no substitute for actual thought during the production of doumentation, nevertheless tool support can transform the production process from a tedious chore to a pleasurable occupation. The tools required are similar to those found with standard office automation, but the necessity to interact with the formal methods tools and the software development process in general brings in some extra requirements. These are listed below:

The need to prepare mathematical texts requires suitable fonts, including the mathematical symbols and the ability to subscript and superscript.

The formal languages require indexing and layout tools.

The evaluation deliverables will be oriented towards one view of the system. The implementors tend to need other views of the system which requires the ability to produce differing documents and cross-refer or quote out of them. The problem here is to maintain the integrity of the cross-references when the various documents change, as they will do during the course of development.

The documentation items need to be maintained under configuration control.

The way in which document preparation is carried out is dependent upon the implementation of the tools and the representations used. Consequently only one recommendation is made as follows:

- Tools for formal methods should support the necessary features for document preparation and maintenance.

7 Configuration control

Configuration control is frequently considered only in terms of management structures, configuration control boards and so on. These are clearly necessary, but if the control aspects are to be taken at all seriously there is a strong impact on the functionality of the tools and the features of the development environment to support them. As far as the development environment is concerned the various control objectives required have been covered elsewhere [Sennett 1987]. To summarise that report the main controls required are the ability to separate trusted and untrusted data and to enforce a mandatory control of access over the over-writing of data. This type of access control gives a high degree of assurance that trusted data has not been accidentally or deliberately over-written. It is the basis of the integrity policy being developed for PCTE+ and appears in the NATO requirements document for development environments. However, apart from the requirement to populate the trusted development environment with tools, the functionality required should not be greatly constrained by the *integrity* considerations.

Configuration control on the other hand gives rise to a strong requirement for consistency which does require tool support. In order to talk about the requirements in a concrete fashion, an example of one particular tool structure and view of the world will be described. Others are possible, but lead to similar requirements. The first element of the requirement for consistency is the need for modularity. Large systems are built by many programmers and it is important to have the ability to break down a specification, a proof or an implementation into modules and work on them independently. It is possible to think of these modules as generalisations of implementation language modules, built up from a text (the source code), the compiled object code and a language specification. In the module structure under consideration three different types of module for specification, refinement and proof may be envisaged.

A *specification module* is built from a specification document which is, say, a Z text which refers to other specification modules. The object code depends on the nature of the processing done on the specification but it will probably be a representation of the abstract syntax of the specification together with the functions necessary to carry out the proofs of the proof obligations generated in the specification. The language specification of the specification module will be mainly concerned with the Z types of the objects exported, but may also include theorems for use in other specification modules. "Execution" of this module carries out the proofs; a proof failure raises an exception.

The text for a *refinement module* expresses the fact that a given implementation is a refinement of a given specification. It uses specification and refinement modules and exports the implementation module. Execution again carries out the proofs required and the rather complex specification expresses the type constraints for the specification and implementation language variables.

A *proof module* is built from the proof document delivers the proof function which

will be obeyed when the specification or refinement module calls for it. It will probably call on meta proof modules (say, programmed in ML if an LCF type of system were used) to carry out the proof steps and these will be ordinary implementation language modules.

This is a hypothetical module structure but the intention is to make the point that in formal verification a number of things such as the proof and refinement documents, which might be thought of as internal to a verification system, must come under configuration control and hence be treated as external objects. The consistency of these objects is crucial to the integrity of the whole verification process and the programmer must be provided with tools to maintain them. The prime requirement is to be able to archive and restore the individual modules and to re-build them anew when specifications change, a process akin to recompilation on change of specification. Without tools in this area, life becomes unbearable.

Apart from consistency, there are the standard requirements for configuration control. It is necessary to maintain versions and variants and provide for traceability. It is necessary to be able to associate other documents, such as test plans and descriptions, with modules and it is probably necessary to form other associations between computer generated items and the source of data used to generate them. Thus the requirement is for configuration control to be applied to objects of various forms, including structured data and functions. This has an impact on the development environment. Traditionally, an operating system provides for files of unstructured data as the means of communication between tools. For advanced verification systems this is no longer appropriate, or if it is attempted leads to gross inefficiencies as source text is re-interpreted to form the objects it is actually required to communicate. The interface between verification tools needs to be much more flexible than this, requiring, for example, the use of the strongly-typed algebraic approach adopted by Ten15 [Foster 1989].

Because of the dependency of this topic on the particular form of tools chosen, it is not possible to make specific recommendations for a configuration management tool. Instead

- The tool and module structure for formal specification and implementation should be studied taking into account the impact on the development environment.

8 Conclusions

The detailed recommendations for tool development have been highlighted in each of the previous sections, and are repeated in the table below.

Specification of security requirements

- Standards for specification languages should be supported. This will make tools more widely available and encourage the widespread adoption of formal specification.
- Proof obligations to establish the consistency of specifications should be established and tools developed to generate and prove them.
- Documentation standards for use with specification languages (and Z in particular) should be developed.
- Specification languages should provide the ability to write down theorems and tools should be provided to support their proof. Specifically a proof theory for Z needs to be developed.

Architectural definition

- Examples of the relation of access control security models to the formal design specifications need to be available.
- More experience is needed in the expression of security properties in process oriented languages.
- As a useful interim measure, it would be helpful to implement the work of Jones on trust domain operators for Z.
- Techniques for integrating process based specification in, for example, CSP with sequential specifications in, for example, Z need to be developed.

Implementation

- A notation needs to be developed to express refinement. This notation should incorporate a module system and a number of case studies need to be undertaken to prove its worth.
- A tool structure needs to be developed which will allow proof obligations to be computed and verified and target languages to be produced.
- Implementation languages need to be assessed with a view to their use with refinement.
- The formal methods and tools need to be integrated within normal structured programming development methods.

Proof

- A notation for Z proof documents needs to be defined.
- A suitable proof theory for Z needs to be developed.
- Tools to support the interactive development of proofs for Z need to be implemented.

Analysis and test

- A study needs to be made into the benefits of specification testing, over and above the analysis that may be necessary for computing proof obligations.
- A study also needs to be made into the integration of implementation language analysis tools into a formal verification environment.
- Work on extending analysis methods to the full range of software will need to be supported.

Documentation

- Tools for formal methods should support the necessary features for document preparation and maintenance.

Configuration control

- The tool and module structure for formal specification and implementation should be studied taking into account the impact on the development environment.

It is clear that much work remains to be done before the technology to establish high

integrity software is widely available in industry. Two general points are worth emphasising. The first is that tool development, and particularly the kind of tool development called for by the use of formal methods, is very dependent on the software development environment, in particular on the nature of the data which can be passed between tools. Secondly, tools for high integrity software are very much concerned with the production of evaluation evidence and the acceptability and benefit of a tool are determined by the extent to which evaluators can make use of them. This has important consequences for the human aspects of the tools which, particularly for formal proof, have tended to be neglected. Both of these problems need to be recognised and studied.

References

Bell, D E and LaPadula, L J (1976). Secure computer system: unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, Bedford, Massachussetts, USA, March 1976.

Blackburn K and Jones, R B (1987). Translating Z into HOL. ICL report DBC/RBJ/086, November 1987.

Bottomley P C (1986). Technical overview of SCP2 - a multi-level secure communications processor. IEE conference on secure communication systems, London, 1986.

CESG (1989). UK systems security confidence levels. CESG computer security memorandum number 3.

DoD (1985). Department of Defense trusted computer system evaluation criteria. DOD 5200.28-STD, December 1985.

Foster J M (1989). The algebraic specification of a target machine. In *High Integrity Software*, Sennett C T (ed), Pitmans (to be published).

Goguen, J A and Meseguer, J (1982). Security policies and security models. Proc 1982 Berkeley Conference on Computer Security, IEEE Computer Society Press, 1982.

Goguen, J A and Tardo J (1979). An introduction to OBJ: a language for writing and testing software specifications. In *Specification of reliable software*, M Zelkowitz (ed), IEEE Press. Reprinted in *Software specification techniques*, N Gehani and A McGettrick (eds), Addison-Wesley, 1985.

Gordon, M J C (1987). HOL: a proof generating system for higher-order logic. In *VLSI specification, verification and synthesis*, Birtwhistle, G and Subrahmanyam, P A (eds), Kluwer 1987.

Harwood, W T (1987). An overview of Genesis. IST report, August 1987.

Jacob, J (1988). Security theories: the state of the engineering discipline. Proc Computer Security Foundations Workshop, Franconia, New Hampshire, USA, June 1988.

Jones, C B (1980). Software development, a rigorous approach. Prentice Hall International, London 1980.

Jones G (1987). Private communication (of work undertaken for CESG).

Macdonald R, Randell G P and Sennett C T (1989). Pattern matching in ML - a case study in refinement. RSRE report 89004.

Milner, R (1971). An algebraic definition of simulation between programs. Second International Joint Conference on Artificial Intelligence, London 1971.

Morgan C C, Robinson K A (1987). Specification statements and refinement. IBM Journal of Research and Development, 31, 5.

Neely, R B and Freeman, J W (1985). Structuring systems for formal verification. Proc 1985 IEEE Symposium on Security and Privacy.

Paulson, L C (1986). Natural deduction as higher-order resolution. J Logic Programming, 3, pp 237 - 258.

Paulson, L C (1988). The foundation of a generic theorem prover. Technical report 130, University of Cambridge Computer Laboratory.

Rex, Thompson and Partners (1987). Malpas intermediate language manual. R, T & P, Newhams, West Street, Farnham, Surrey.

Rushby, J M (1985). Mathematical foundations of the MLS tool for Revised Special. SRI International.

Sennett, C T (1987). The development environment for secure software. RSRE Report 87015.

Sennett, C T and Macdonald, R (1987). Separability and security models. RSRE Report 87020.

Sennett, C T (1989). The contractual specification of reliable software. In *High Integrity Software* Sennett (ed), Pitmans (to be published).

Spivey, J M (1988). Understanding Z: a specification language and its formal semantics. Cambridge University Press.

Spivey, J M (1989). The Z notation: a reference manual. Prentice Hall International series in Computer Science.

DOCUMENT CONTROL SHEET

Overall security classification of sheet ....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	Report 89005		U/C

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
7784000	ROYAL SIGNALS & RADAR ESTABLISHMENT ST ANDREWS ROAD, GREAT MALVERN WORCESTERSHIRE WR14 3PS
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

Tool support for the production of high integrity software

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
Sennett C T			1989.04	22

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

UNLIMITED

Descriptors (or keywords)

continue on separate piece of paper

Abstract

This report was commissioned by the UK computer security pol. y ority. It discusses the software tools required for the production of trusted software, following the guidelines given by the UK systems security confidence levels. Recommendations are given for the development of new tools and techniques where appropriate.

SBO/48