DEFENCE RESEARCH AGENCY

RSRE Memorandum 4585

Title: Languages for Requirements Specifications

Author: G P Randell

Date: March 1992

Abstract

This memorandum discusses 11 desirable features that make a requirements specification language a "good" language. Current languages are assessed to determine which of these features they possess, and where they are lacking. Conclusions are drawn as to the current state of requirements specification languages, and topics for further research are proposed.

Contents

1. Introduction........1

1.1 The Requirements Specification........1

1.2 Requirements Specification Languages........2

1.3 Other Factors........3

1.4 Structure of the Memorandum........4

2. Desirable Features........5

3. How Good are Existing Languages?........10

4. Conclusions and Further Work........14

References........16

1. Introduction

1.1 The Requirements Specification

In any model of procurement, whether it be the traditional "waterfall" model or an incremental approach (the "build a little, field a little, test a little" philosophy), the requirements specification plays an important role: it defines what the supplier is to build and what the customer, on behalf of the ultimate user of the system, is prepared to accept, and often forms the basis of a legal contract. If the requirements specification is inadequate, then the system delivered is unlikely to satisfy the users' real needs.

The requirements specification acts as the interface between the customer and supplier, and this role has inherent difficulties. For if the customer and the supplier have different technical backgrounds, each will have different interpretations of the specification. This inevitably leads to communication problems between them, contradictory to the major purpose of the specification: to communicate requirements.

The most important feature of a requirements specification is that it must be understandable, to both customer and supplier. In addition, the STARTS Guide [STARTS] suggests that it should also have the following characteristics:

a) unambiguous - every requirement should have only one interpretation;

b) complete - all requirements should be stated, there should be no undeclared assumptions;

c) testable or demonstrable - each requirement should be stated in such a way that the system can be checked to show that the requirement has been met;

d) consistent - there should be no conflict between individual requirements;

e) modifiable - it is a fact of life that customers change their requirements, so the specification must be structured in such a way that this can be achieved in a straightforward manner;

f) traceable - proper referencing is required to allow requirements to be traced back to their originator and forwards to the design;

g) modular - it should be possible to produce and understand parts of the requirements specification independently.

Producing a requirements specification with all these desired characteristics is a difficult task. The purpose of this memorandum is to concentrate on one aspect of specification, namely the language used to express the requirements.

1.2 Requirements Specification Languages

Currently, requirements specifications tend to be written in natural languages like English on the grounds that English requires very little training and is understandable to all parties. In fact English does require training to use well. Poorly written English specifications may be verbose, and their meanings obscure. English is also inherently ambiguous: the same statement may be interpreted in more than one way. An English specification is also very difficult to validate: the only thing that can be done with such a specification is to read it. It is very difficult to ensure consistency and completeness of the specification.

In an attempt to overcome the difficulties associated with the use of English, formal languages have been proposed for requirements specification. These languages have a sound mathematical or logical basis, usually in set theory or algebra. Examples of formal languages are Z [Spiveya, Spiveyb], VDM [Jones] and OBJ [Gerrard]. Because they are mathematically based, these languages are unambiguous and, in principle, analysable. That is, propositions about the specification can be formulated and proved. For example, a specification may be proved to be internally consistent. However, little guidance currently exists on how to analyse specifications or indeed what to analyse for. The major disadvantage with current formal languages is that they are difficult to learn, use and understand. They use mathematical notations which may, at first sight, be daunting.

English and mathematics may be considered to be at opposite ends of a spectrum. In between lie diagrams and semi-formal languages. Many of the structured methods of system development, such as SSADM [SSADM], use diagrammatic techniques. Data flow diagrams, for example are informal in that they do not have an agreed semantics and thus may be interpreted in more than one way. However they use a notation which is relatively easy to learn and use. Diagrams do not have to be informal; it is possible to define mathematically diagrams which gives them a formal semantics and thus makes them formal objects. However, most of the diagrams used in systems development today do not have a mathematical base.

Thus it may be seen that there is a variety of languages available, each with its own advantages and disadvantages. However, there are factors other than understandability and the lack of ambiguity that affect the choice of a language for requirements specification, and these are discussed below.

1.3 Other Factors

A requirements specification contains not only functional requirements, detailing what the system is to do, but also non-functional requirements, which are constraints on the operation, development and maintenance of the system. The former are relatively well-understood, but the latter are often seen as the "poor relation" and considered only as an after-thought. Non-functional requirements may be of three types (after [STARTS]):

a) Performance, including processing speed, volume of data, storage usage;

b) Dependability, including safety, reliability, availability and security;

c) Quality, including ease of change, interoperability, intelligibility and ease of use.

In addition the environment in which the system will be expected to operate must be described, including physical characteristics like heat, light and power, as well as any design constraints, such as time-scales or the mandatory use of a particular programming language or other standard.

Not only are different types of requirement present, but they may be at varying levels of detail from the abstract to the specific. For example, interface protocols defining how the system communicates with others may need to be very detailed, whereas behavioural properties may leave the designer much more freedom in their implementation. The language used for a specification should ideally be suitable for expressing all requirements usefully, and at all necessary levels of detail.

The choice of language is further affected by the necessity for validating a specification, that is, the process of demonstrating its completeness and consistency with respect to the initial ideas of what the system can do. Informally, validation may be thought of as answering the question "are we building the right system?". Validation is a social process by which confidence is gained that the specification accurately reflects the real needs. Techniques for validation are:

a) Reviews - in which the specification is read and appraised;

b) Walk-throughs - in which the author presents the specification to a group of reviewers and invites comment;

c) Prototyping - in which a model of all or part of the system is produced for trials and tests;

d) Executable specifications (a sort of prototyping) - in which the specification itself is written in an executable language and tested;

e) Static analysis - in which the specification, written in a suitable language, is analysed symbolically, for example for data usage or control flow;

f) Reasoning - a special form of static analysis in which propositions about a formal (mathematical) specification are formulated and proved.

It is apparent that the desire to validate affects the choice of language: if techniques other then reviews and walk-throughs are to be used, then the specification must be written in a language other than English. For example, English is neither executable nor amenable to static analysis. On the other hand, a mathematical specification is amenable to reasoning, but is likely to be difficult to review because the language will be difficult for non-experts to understand.

In a similar vein, the need to verify that the design and implementation meet the specification, and the need ultimately to accept the final system, has an impact on the choice of language. If the specification is mathematical, there is the possibility of proving that a design meets the specification, whereas only an informal argument can be used if the specification is written in English. For further discussion on validation and verification see [STARTS] and [C3ISSC].

From considering the "ideal" specification and the need for validation and verification, it is possible to extract those features which are desirable in any language used for requirements specification.

1.4 Structure of the Memorandum

The remainder of this memorandum is structured as follows. Section 2 proposes 11 features that are desired of a language for the specification of requirements, while section 3 discusses four current languages with respect to these features. The memorandum concludes with a summary of the current situation and proposals for further research.

2. Desirable Features

This section discusses the following 11 features which, it is proposed, are desirable in a language for requirements specification:

a. Understandable
b. Analysable
c. Animateable
d. Wide spectrum
e. Unambiguous
f. Able to express as many requirements as possible
g. Easy to learn and use
h. Modular
i. Allows abstraction
j. Machine processable
k. Concise

This list is in order of importance, the most important first. Each of the features will be discussed in turn.

a. Understandable

Understandability is the key to a successful requirements specification. The meaning of the requirements specification must be apparent to the users whose requirements it embodies and to the designers whose baseline for development it is. Misunderstanding a requirement will lead to a flawed system, and potentially to time and cost overruns or, in extreme cases, to complete failure of the project.

A difficult specification will not be understood if the time and effort required to gain that understanding is not available (as will usually be the case). This applies to both the user on whose behalf the customer is presenting the specification and to the designer who will build the system it describes. Confidence that the specification is right will be missing, as a specification that cannot be understood cannot be validated.

The language used to express the requirement obviously has a significant impact on the understandability of the specification. A language which uses unfamiliar symbols or terminology may result in a specification which is hard to understand. Diagrams and pictures have a role to play here, as they are often relatively easy to understand. Understandability is the most important feature of a requirements specification language.

b. Analysable

The analysis of requirements is an important part of the requirements definition activity, but one that is often ignored The purpose of analysis is to examine the information elicited from the users to detect inconsistencies and incompleteness. However, the rigour of the analyses possible depends on the rigour of the language used to express the requirements. For example, in the case of a formal language, mathematical proof may be used, but, in the case of English, analysis is limited.

Analysis may be either static (such as proof), or dynamic (testing). In order to carry out dynamic analysis, the specification must be capable of being exercised. This is discussed further in the following section (on the third feature: animateable). From now on, the term "analysis" is used to refer to static analysis.

Analysis may be used as a means of validation, by showing that desired properties exist and that unwanted properties are absent. This sort of analysis is common for secure systems to show, for example, the absence of covert channels. Again the language used has a significant impact on the analyses that are possible.

The benefits to be gained from analysis are such that any language used for requirements specifications must be capable of being analysed.

c. Animateable

A major part of the requirements definition process is validating the requirements specification, as discussed in section 1.3 above. One promising validation technique is animation, a form of rapid prototyping. Animation provides an indication of dynamic behaviour and illustrates the meaning of a specification. Tests can be run and trade-offs and potential bottlenecks investigated. The major advantage of rapid prototyping is that an actual "thing" can be given to users to experiment with to see if it is likely to meet their needs. It should be remembered that testing a specification, like testing a computer program, will only show the presence of errors, not their absence.

Animation may be achieved directly, by writing the specification in a language which is executable, or indirectly by translating (preferably automatically) the specification into an executable language. Whichever route is chosen, a suitable language must be used.

It is unlikely that a rapid prototype generated in this way will be a suitable implementation, often because of inefficiency. However, there is the possibility of transforming a prototype to optimise it. The prototype may also be used for the purpose of static analysis (see the discussion on analysis above).

Because of the benefits to be gained from rapid prototyping, particularly in the validation of a specification, it is highly desirable that a requirements specification language should be animateable.

d. Wide spectrum

Different languages are usually used throughout the development of a system, from the requirements specification through to the implementation. As a consequence, those involved in the development need to understand at least two languages, the language of the level they are working from (for example, the requirements specification) and the language of the level they are producing (for example, the architectural design). Using different languages for different levels leads to problems when trying to demonstrate that one level conforms to another.

A possible solution is to use a wide-spectrum language, that is, a single language which may be used at several levels in the development process. Not only does this alleviate the training burden and reduce the need for translations, but it also has a knock-on effect on the tool support required. Fewer language editors will be needed and there will no longer be the need to integrate many different tools to support many different languages. "Wide-spectrum" should not be confused with "able to express all requirements". The former refers to different stages in the development cycle whilst the latter refers to coverage at a single stage.

Thus a wide-spectrum language which is useful for requirements specifications and other stages (but not necessarily all) of the development cycle would have significant benefits.

e. Unambiguous

Not only should the specification be understandable, as discussed above, but it should be understood as it was written. That is, each statement should mean only one thing to the reader, and that one thing should be the meaning that the writer intended. The writer will then know that the reader will understand correctly what has been written.

Ambiguity in a specification can lead to the wrong interpretation being placed on a statement, and subsequently the design and implementation may be not what was expected. An ambiguous specification should not be used in a contract because if a statement can be interpreted in many ways and the system satisfies one of them, there is no basis for complaining that the system is wrong, although it may not match the intended interpretation.

A form of ambiguity, called underspecification, should be allowed. Underspecification may be thought of as "ambiguity that does not matter", and means that several interpretations are possible, any one of which is acceptable. This is a valuable tool of abstraction and is used to give design freedom.

The language used to express the requirements has an obvious impact here. If the language has a defined semantics (meaning) then it is less likely to be misinterpreted. Indeed, if the language has a formal semantics then each statement only has one meaning, although interpreting the formal meaning in the real world offers the potential for confusion. Languages with no defined semantics, such as English, invite misinterpretation unless they are used with particular care.

A precisely defined language therefore offers benefits by avoiding ambiguities of meaning.

f. Able to express as many requirements as possible

As discussed in section 1.3, a requirements specification must contain a variety of requirements, both functional and non-functional. At present, different languages are often used to express different sorts of requirements. For example, for a secure system the security policy may be expressed in English and the security policy model (a more precise statement of the important parts of the policy) expressed in mathematics. Using different languages in this way leads to problems when trying to ensure that all the requirements are consistent: how can an English and a mathematical statement be checked for consistency in any meaningful way? The same problem will arise no matter what the languages whenever more than one is used.

It is unlikely that any one language will be capable of expressing all requirements, but it would be beneficial to use as few languages as possible. Each should be capable of expressing requirements in a way which is understandable and useful as a basis for devising acceptance tests.

g. Easy to learn and use

A requirements specification is used by a variety of people, from the user whose requirements it embodies through to the designer who designs the system based on it. All the people involved have to understand it, and thus have to learn and use the language in which it is written. In order to learn a new language, training will be needed. But training can be expensive in terms of money and time, and these may not be available.

In addition to increased training costs, a difficult language will have more problems in gaining user acceptance, thus the specification language should be easy to learn and to use.

h. Modular

Specifications of large and complex systems (such as CIS) need to be able to be broken up into manageable-sized pieces, so that no one piece is bigger than a "headful". This enables the specification to be understood more easily. Modularisation also allows several people to work on one specification at a time, both when writing it and when designing a system from it.

Thus a requirements specification language should allow modularisation, for example, sentences, paragraphs and chapters (as in English), or procedures and modules (as in some programming languages).

i. Allows abstraction

Requirements specifications contain information at many levels of detail. Abstraction allows the writer of the specification to concentrate at a level of detail appropriate to each requirement, and thus aids understandability and conciseness.

Thus the language used should be capable of abstraction, that is, of ignoring unnecessary detail, where this is appropriate.

j. Machine processable

Requirements specifications are often written using word-processors and other computer-based editors. The advantages in using these tools over a type-writer are well known. Thus a specification language should be capable of being written on a computer. And not merely written, but also checked by computer. That is, the computer should be able to check that the language has been used correctly according to the rules of that language: that the syntax (the grammar) is correct. This will enable many minor errors to be automatically detected. However, such checks will not ensure that the specification is a sensible one, and should not be interpreted as giving more confidence than is justified.

k. Concise

The size of a specification has a psychological impact on the reader. A large, weighty volume is off-putting, and also requires more time and effort to read. On the other hand, a concise statement is more readable and thus is more likely to be read. A powerful language will be able to express ideas concisely, but that very power means that the language must be used carefully.

Conciseness, while important, should not be pursued at the expense of clarity (and hence understandability). It is more important that the specification is understood. It is often all too easy to devote time and effort to stating some requirement in as short a way as possible while forgetting that it needs to be understood by others. All requirements should be made explicit, and should not have to be inferred.

Thus a requirements specification language should allow for a concise statement of the requirements and not require large amounts of words to express simple concepts.

3. How Good are Existing Languages?

Eleven desirable features for a language for requirements specification have been proposed in the previous section. Currently, many languages exist for specification, but the question is: how good are these languages? Problems have been encountered with all of them. The purpose of this section is to discuss how many desirable features each of the following languages possesses:

a. English
b. Z
c. Ada[1]
d. Data Flow Diagrams (DFDs)

These cover a broad spectrum of current languages, from the informal English, through diagrams (e.g. DFDs) and programming languages (e.g. Ada), to the formal, mathematical language Z. There are many other languages in use, but there is not room to discuss them all here.

a. English

One of the most common ways of writing a requirements specification (in this country) is to use English. The English language is very informal: it has no explicitly prescribed rules but depends on current usage. As a consequence, English is often ambiguous. An obvious advantage with using English is that it is taught to everyone at school: no additional training should be necessary.

But poorly written English can be difficult to follow. However, this criticism applies to any language - all languages can be written badly. It does not require any extra special effort to understand English. The specification will look familiar with no particular symbols or graphics. The lack of graphics can mean that an English specification may not be as concise as one would like, but it will be modular, using sentences, paragraphs, sections and chapters.

English cannot be animated or analysed in any meaningful way, and is only machine-processable in that word-processors exist, which often have facilities for checking spelling, but not for checking grammar. Thus validating an English specification is difficult - the only thing that can be done with it is to read it.

Any English specification should have a glossary attached, explaining words which are used with a particular meaning. Jargon, which is only understood by those "in the know", should be avoided as it may lead to misinterpretation and misunderstanding. English does allow unnecessary detail to be hidden (abstraction), and all requirements can be expressed using it (although not always usefully). Designers and implementors do not usually use English, as it is too imprecise for their purposes, so English is not really a wide-spectrum language. English is,

[1] Ada is a registered trademark of the US Government - Ada Joint Program Office

however, used throughout all the stages in the development process to illustrate the more precise designs and implementation, all the way down to comments in the actual code (in the case of software).

To summarize, English has great advantages in terms of its familiarity to everyone - we think in English, and we explain what we mean in English - but it has little else to offer.

b. Z

Z is a formal language based on set theory and predicate calculus. It has been developed at Oxford University, based on the work of Abrial. The language has two parts: the mathematical toolkit (also called the Z library) and the schema calculus. The former is used to construct the mathematical text of the specification while the latter is used to structure the descriptions written in the mathematical language. The Z notation is particularly suitable for a model based approach to specification. With this approach, an abstract mathematical model of the system's state is constructed, and operations are defined on that state. Schemas are used to define both the state and the operations. While Z has not yet been standardised, a reference manual [Spiveya] and semantics for Z [Spiveyb] have been published.

As Z has a formal semantics, it is unambiguous. However, it is difficult to understand and requires a significant amount of training to write it well. Reading Z is easier than writing it. Z can be extremely concise, but often at the expense of clarity. Every Z specification should contain as much English as mathematics. This "commentary" explains the mathematics and describes how it should be interpreted in the real world. Interpreting (understanding) Z is made easier by meaningful names being used for objects in the specification, in much the same way that a program is more understandable if meaningful names are used (and comments used to explain what is going on).

Z encourages abstraction, and allows detail to be ignored where this is wanted. Because it is formal, Z is also, in principle, analysable. However, the analysis of Z specifications has attracted little interest to date, except in the areas of syntax and type-checking. Consequently few techniques, methods or tools for analysis (other than syntax and type-checkers) are available. One area of work which has attracted some interest is that of animating Z using the Prolog language [Dick], although this technique is not yet widely available. Z is extremely good at modelling systems which comprise states and operations on states, but is not very good at expressing sequences of operations, histories, and other aspects of time. Thus Z is useful for only some sorts of requirements.

To summarize, Z is a formal language and is concise and unambiguous. It encourages abstraction, and can be used for both specifications and detailed designs. It is machine processable, and, in principle, analysable and possibly animateable. There is limited modularity (using the schema calculus). However, it is difficult to understand, and hard to learn how to use. Z is not appropriate for all requirements.

c. Ada

Although Ada is a high level programming language, it can also be used for requirements specifications. The idea of using a programming language may appear unnatural at first, but many of the constructs found in programming languages, such as if-then-else, are very natural ways of explaining a requirement (for example, if the incoming aircraft is hostile then shoot at it, else let it land). Ada allows the specification of procedures, functions and packages to be written and compiled separately from their bodies (the detail).

As for any other programming language, Ada does require training to use. It can be readily understood, with the help of meaningful names and comments, as for Z (discussed above). Ada has a defined semantics, which, however, is not formal. But a particular compiler will give a single meaning to an Ada program. Because Ada is a programming language it can be compiled and executed, and also analysed using program analysis tools such as MALPAS (the Malvern Program Analysis Suite) [RTP] and SPADE [Carré].

Ada can be used for specification, and also for implementation, thus it can be wide-spectrum. It is also modular, using constructs such as procedures, functions and packages. It is reasonably concise, and is machine-processable - a compiler checks syntax and for various other errors. Ada also has constructs for parallelism, unlike many other high level programming languages, but is not applicable for specifying other non-functional requirements. Using Ada in a top-down fashion allows detail to be added later, thus abstraction is supported.

To summarize, Ada is a programming language which requires some training to use, but is reasonably understandable and concise. It is animateable and analysable, supports abstraction and is modular. It can be considered to be wide-spectrum, and is also machine-processable. However, it is not appropriate for expressing all requirements.

d. Data Flow Diagrams (DFDs)

Data Flow Diagrams (DFDs) are used by many structured methods for systems analysis and design. They are used to show the movement of data to, from and around the system. They identify and summarize all processing within a system.

DFDs are an informal technique, they have an agreed syntax, but no defined semantics. Thus they can be ambiguous. Some methods, however, give tighter definitions of DFDs than others. They are concise, and are easy to learn, use and understand. They are useful for specifying functionality, but are not useful for other requirements. They can also be used at different stages in the development process, so can be considered wide-spectrum.

DFDs are machine-processable (syntax may be checked), and can be animated. They do not contain enough information to carry out detailed analysis, but some simple checks can be made. Decomposing a DFD through several levels, each adding more detail, supports abstraction. Modularity is also supported, to some extent, by the same mechanism.

To summarize, DFDs are simple and useful for a limited number of requirements. They can be ambiguous, are machine-processable and animateable. Little analysis can be carried out. They support modularity and abstraction, are concise, and can be used in several stages of development.

Overall Scores

From these discussions, it is apparent that no one language available today has all the desired features. The following table summarizes those features which each language possesses. Each language is given a score from 0 - 5 for each feature, indicating how well the language supports that feature (0 means not at all, 5 means excellent).

	a	b	c	d	e	f	g	h	i	j	k
English	5	0	0	1	0	4	5	5	4	1	3
Z	1	3	3	3	5	1	1	2	5	5	4
Ada	3	5	5	3	3	1	3	4	3	5	3
DFDs	4	4	1	2	1	1	4	3	4	4	4

Key to table:
Desirable features:

a.	Understandable
b.	Animateable
c.	Analysable
d.	Wide spectrum
e.	Unambiguous
f.	Able to express as many requirements as possible
g.	Easy to learn and use
h.	Modular
i.	Allows abstraction
j.	Machine processable
k.	Concise

NB. This table is a summary - the scores should not be read on their own, nor is it useful to add up the scores for each language.

4. Conclusions and Further Work

There are many features which we would like to have in a language for requirements specification. However, no currently available language has all of these features to the degree we would wish.

It is apparent from the discussions in section 3, that English cannot entirely be replaced in the requirements definition stage, or any other stage of the development cycle. However, it is also apparent that English on its own is not sufficient. The lack of animation and analyses are serious drawbacks. It is the author's contention that a more precise language, with these facilities, should be used, with English (and possibly graphical) commentary to improve understandability. But it must be remembered that the English part is merely commentary, and is not the definitive part of the specification.

Another problem with English is its ambiguity. However, it may be possible to lessen the problems arising from this by using a form of structured or constrained English. This restricts the writers' freedom. It is then possible to carry out some checks on the specification, to ensure that the syntax of the structured English has been followed correctly. This leads to the first recommendation for further research:

> To investigate current forms of structured English, and develop a form suitable for use in requirements.

It is also possible, with requirements expressed concisely in a structured way, that database technology may be used to manage requirements and maintain traceability through the development of the system (see [Randell] for such a proposal).

Formal languages are quite different. Their biggest problem is understandability, although sensible use of commentary does help. Another way of helping to overcome this is to generate diagrams from the formal text, in order to help explain it in a user-friendly way. This is the second area where further research is needed:

> To develop the techniques and tools for the production of diagrams from formal specifications.

Further work is also needed on the analysis and animation of specifications. These are both important topics which, potentially, will lead to much better requirements specifications. Languages which have these features, such as programming languages, and hardware description languages, for example ELLA, should be investigated to determine their suitability for requirements and improved as necessary. Further research is needed in these areas:

> To develop techniques and tools for analysis, for a range of languages.

To carry out further work to determine the usefulness and feasibility of animating specifications.

To investigate a variety of languages, including hardware description languages, to determine their suitability for requirements specifications, and recommend improvements to these languages where appropriate.

Throughout the work on analysis and animation, attention must be paid to the way the specifier interacts with the specification. That is, the results of animation and analysis must be easily understood, possibly with the help of diagrams.

Requirements specifications contain a range of concepts, context-related explanations and complex logical relationships, as well as facts and definitions. No single language is likely to have all the attributes necessary to express this variety adequately and effectively. Rather, different languages will be used for specific purposes. This leads to problems with checking consistency, and with analyses and animation. Translating the different parts of requirements specification into a single underlying rigorous language will enable that specification to be analysed and consistency to be checked. This language will not be readily understandable, therefore cannot be an effective specification language in its own right. Instead, translations from the specification languages into this underlying language need to be developed. This allows a consistent underlying view of a system to be maintained and views of the system to be projected from it.

The final recommendation is therefore that:

Translations between languages, including between diagrams and formal languages, be developed.

The vision is that different languages will be used in a requirements specification, and these compiled into one underlying representation suitable for analysis and animation. This representation will not be manipulated directly, rather mechanisms will be needed to frame queries in a user-friendly way, and to give the results of those queries in the same manner. Different views on the system, expressed in different languages as appropriate, may also be automatically generated from it for validation purposes. The same underlying representation may also be suitable as a basis for design, with design information being added and different views of the system appropriate to the designer rather than the specifier, such as the data view or the process view, again being automatically generated. This will lead to a better link between the requirements specification and the subsequent design, and reduce the number of errors in system development.

References

[C3ISSC]	T A D White and C J Young (Editors), Systems Engineering Methods and Tools Working Group Report on C^3I Requirements Definition Issues, C^3ISSC, 1990
[Carré]	B Carré, "Program Analysis and Verification", in High-Integrity Software, Chris Sennett (Ed), Pitman, 1989
[Dick]	A J J Dick, P J Krause, J Cozens, Computer Aided Transformation of Z into Prolog, Proceedings of the Fourth Annual Z Users Meeting, Oxford University Computing Laboratory, 1989
[Gerrard]	C Gerrard, Gerrard Software Ltd
[Jones]	C B Jones, Software Analysis: a Rigorous Approach, Prentice-Hall International, 1980
[Randell]	G P Randell, A Proposal for a Requirements Database for the MoD, Working Paper, DRA Electronics Division, September 1991
[RTP]	MALPAS User Guide, Rex, Thompson and Partners
[Spiveya]	J M Spivey, The Z Notation - A Reference Manual, Prentice-Hall International, 1989
[Spiveyb]	J M Spivey, Understanding Z - A Specification Language and its Formal Semantics, Cambridge University Press, 1988
[SSADM]	G Longworth and D Nicholls, SSADM Manual, NCC Publications, 1986
[STARTS]	STARTS Purchasers' Handbook "Procuring Software-based Systems", Second Edition, NCC Publications, 1989

REPORT DOCUMENTATION PAGE DRIC Reference Number (if known) ....

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 field concerned must be marked to indicate the classification, eg (R), (C) or (S).

Originators Reference/Report No. MEMO 4585	Month MARCH	Year 1992
Originators Name and Location DRA, ST ANDREWS ROAD MALVERN, WORCS WR14 3PS		
Monitoring Agency Name and Location		
Title LANGUAGES FOR REQUIREMENTS SPECIFICATIONS		
Report Security Classification UNCLASSIFIED		Title Classification (U, R, C or S) U
Foreign Language Title (in the case of translations)		
Conference Details		
Agency Reference	Contract Number and Period	
Project Number	Other References	
Authors RANDELL, G P		Pagination and Ref 16

Abstract

This memorandum discusses 11 desirable features that make a requirements specification language a "good" language. Current languages are assessed to determine which of these features they possess, and where they are lacking. Conclusions are drawn as to the current state of requirements specification languages, and topics for further research are proposed.

	Abstract Classification (U, R, C or S) U
Descriptors	
Distribution Statement (Enter any limitations on the distribution of the document) UNLIMITED	S80/48