ANIMATION OF Z SPECIFICATIONS BY TRANSLATION TO PROLOG

Omar Salman
Department of Computer Science (ISSR)
Cairo University (Egypt) §

Abstract: Formal methods of software development rely on the validation of the specification of the software. Such specification is normally expressed in a formal language such as Z. However, in order to be validated the Z specification must be tested, and to achieve this it has to be transformed into a form that can be executed or animated. Prolog was one of the languages used for animation of Z specifications. This paper explains the techniques used for translating Z schemas into Prolog predicates. It also examines some of this translation shortcomings and unreliable features.

Keywords: *formal methods, Z specification, animation, Z schema.*

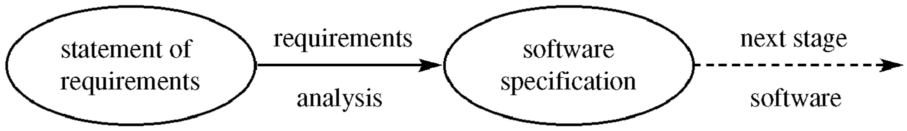
Özet: Yazılım geliştirebilmenin formal metodları o yazılım tanımlamasının geçerliliğine bağlıdır. Böyle bir tanımlama genelde 'Z' gibi bir formal dilde ifade edilir. Ancak, geçerli olması için, 'Z' tanımlaması test edilmeli, bunu yapabilmek için de animasyon yapılabilecek ve icra edilebilecek bir forma transfer edilebilmelidir. 'Z' tanımlamalarının animasyonları için kullanılan dillerden birisi Prolog'dur. Bu makalede 'Z' şemalarını Prolog'a çeviren teknikler açıklanmaktadır. Aynı zamanda bu tür bir çevirmenin eksikleri ve belirsizlikleri üzerinde durulacaktır.

Anahtar kelimeler: *formal metodlar, 'Z' tanımlama dili, animasyon, 'Z' şeması*

(§) This work was done in co-operation with Doğuş University, Istanbul, Turkey.

1. INTRODUCTION

The starting point of a software project is the customer statement of requirements. This document is an informal description of the properties of a new software or the changes to an existing one. The next step in the software life-cycle is to process and analyse the statement of requirements to produce another document called the software specification. The software specification then becomes the major reference document for the development of the software and is used as a basis for validation and verification of the software product. This means that it must be expressed in a clear, consistent and unambiguous manner.



Much current software engineering research centres on the use of mathematics for specifying and developing software systems. Since the middle of the 1970's many formal specification languages, both algebraic (Futatsugi 85, Guttag 78) and model based (such as Z) (Jones 86, Spivy 89), have been proposed, and the term formal methods in software engineering embraces the utilisation of such language. Basic aims of formal methods applied to software engineering are the precise and complete specification of what is required of a software system and the exclusion of errors from a developing software system through rigorous verification of each step of the development life-cycle. Thus, faithfulness to the stated software requirements and software reliability are assured, and the costs of removing errors later in the life-cycle are avoided. The well-defined semantics and syntax of formal specification languages make them appropriate for expressing a precise description of the requirements of the system being built. Moreover, it make it possible to formally reason about that specification, and to prove that the subsequent design and implementation of the system conforms to meet those requirements. Indeed the use of mathematics has introduced a new methodology in software development called the formal software development methodology. In this methodology, the stages for the development of a software product can be described as (Berg 82, Wordsworth 96):

(a) The specification of the software product is described by using formal specification techniques.

(b) The specification is then validated to ensure that it is well-formed and reflects the user requirements.

(c) The formal specification is then refined until it can be transformed into a program in a high-level language.

(d) The complete program is then verified to ensure its quality.

As we can see having a valid formal specification (step (b)) is very important because the formal specification is taken as the basis for the next steps in software development (steps (c) and (d)).

The necessity of formal methods for the design and implementation of software systems, particularly large-scale safety-critical ones, is established by study and experimentation made by several researchers. For example, Boehm (Boehm 79) has reported that over 60% of errors uncovered in several operational software systems were due to shortcomings in the specifications. Gladden (Gladdden 82) reports that 35% of delivered software is not used because of the gap between it and the user's concept of the system. Bloomfield (Bloomfield 86) also addressed this problem and concluded that where formal methods have not been used, retrospective formal proof of developed software is not feasible. This is because of the reliance of formal methods on software structuring which is amenable to easy correctness argument.

Even when specification is formal, the correspondence between the formal specifications and the requirements expressed by the customer cannot be proved. Tools which help towards validating the latter correspondence, are therefore of potential value to both the software designer/developer and the recipient of delivered software product.

Animation has been proposed as one of the methods for partly solving this key problem of ascertaining that a formal specification corresponds to the expressed requirements of the customer.

2. VALIDATION OF THE FORMAL SPECIFICATION

There are several approaches to demonstrate the validity of the formal specification. The two most important ones are: *formal reasoning* and *testing* or *animation*. A fully formal proof gives a much higher level of confidence in the specification, but is also much more expensive. The effort needed to prove a specification formally is far greater than that for writing it. Experimentation using formal reasoning in software development conducted by Fields et al (Fields 92) indicates that formally verifying specifications and development is extremely tedious and time consuming due to the level of detail at which one is forced to work when doing fully "formal development". On the other hand the testing technique is easier to use and well understood by many people. Testing to validate a formal specification means executing the specification against test data. This idea has been proposed by many people, for example by Kemmerer (Kemmerer 85) and Jalote (Jalote 89). Thus for the testing to be done, the formal specification must be executable. However, most formal specification notations are non-procedural and thus cannot be executed directly (Hayes 91). So before this type of formal specification can be tested, it must be transformed into a procedural form that can be executed or animated.

2.1. Animation

Animation is valuable for the following reasons (Barden 94)

- Mistakes in the specification, either owing to errors in the mathematics, or to errors in the requirements capture process, may be made apparent by animation.

- For many systems the client might not really know what is required until the system is working. By showing him (or her) the specification in a working executable form, he (or she) can try out the system to examine its behaviour. Unforeseen emergent properties of the specification that correctly captures the stated requirements may be exposed. If these properties are not desirable, then the stated requirements fail to capture what is really wanted and have to be modified. Alternatively, if these properties are desirable, they may be formulated as `theorems' about the global behaviour of the system.

- Animation can provide more confidence that the specification does correctly capture the requirements, both for the specifiers and for clients who experiment with the animation.

The disadvantage of animation as a testing process can be found in Dijkstra's (Dijkstra 79) well-known aphorism, *program testing can be used to show the presence of bugs, but never their absence*, which applies equally to specification validation. Nevertheless, the increased confidence and understanding animation provides makes it a worthwhile validation exercise. Indeed, animation is now mandated as a validation technique in the UK Ministry of Defence's Interim Standard (MoD UK 91) concerned with safety-critical software.

There are two approaches which have been proposed for animating a formal specification. These are direct execution and rapid prototyping. Direct execution means that the formal specification statements are executed directly, normally by interpretation. Whereas rapid prototyping is a technique in which a formal specification is translated into a program in a high-level language (Kemmerer 85). Direct execution is the most convenient method for animating formal specification statements. But as stated above, in order to do it, the formal specification language must be executable. Examples of this type of languages are "me too" (Henderson 85), EPROL (Hekmatpor 88) and OBJ (Goguen 84). However, since Z is a nonprocedural language, we have to employ the rapid prototyping technique.

Many implementations of the rapid prototyping technique have used high level procedural languages, for example Pascal, C or Ada as the object languages. But the approach taken for Z is to use either a rule-based language, for example CRYSTAL, (Andrew 90), a functional language like Miranda (Turner 86, Diller 90), or Prolog (West 88). Most researchers, however, preferred Prolog as the object language for a number of reasons. Firstly, Prolog is more widely available on many computers both as compilers and interpreters. Secondly, Prolog offers a very flexible and simple method of asking queries. Finally, since both Z and Prolog are based on the first order predicate logic, it is possible for automatic translation to be done from Z schemas into Prolog clauses. Statements in Prolog are written in terms of Horn clauses. A Horn clause is a clause with at most one unnegated literal. So Horn clauses are either single predicates or single negated predicates or implies-statements in which the conclusion is a single predicate rather than a disjunction of them (Ross 89). It has been shown that a normal first order predicate logic statement can be translated into Horn clauses (Clocksin 94), i.e. Prolog.

3. Z TO PROLOG TRANSLATION

Generally, there are two Z/Prolog translation strategies (West 92), *formal program synthesis* and *structure simulation*.

3.1. Formal program synthesis

This is the most obvious strategy, arising from the fact that Z and Prolog are related in a mathematical way, and this relationship can be realised by a formal synthesis of the Prolog program code via a direct translation of the Z schema. The method of formal program synthesis is detailed by Hogger (Hogger 84), who identifies as the most important advantage the fact that, when a program is derived logically from a specification, its correctness with regard to the specification is assured. This method relies on devising an algorithmic procedure for converting Z into clausal form, and hence into Prolog. The method breaks down into the following steps (West 92):

- step 1: re-express the higher order theory of Z, as first-order predicate logic
- step 2: turn the resulting first-order formulae into Prolog.

In principle, step 1 could be accomplished formally by a method discussed by Hatcher (Hatcher 82), and step 2 by a method presented by Kowalski (Kowalski 79). The latter is a method whereby programs are deduced from specifications using rules of inference such as resolution, combined with clausal form transformation. An attempt of formal program synthesis was initially tried by West and Eaglestone (West 92) but then abandoned. The problem is, program synthesis relies on human intelligence to determine which clauses are most suitable for a resolution step, and for this there is no algorithmic method available yet.

3.2. Structure simulation.

In this approach, instead of a formal transformation, characteristics of the Z schema were identified and adapted so that the logical structure of the specification would be preserved as far as possible in the resulting model.

Several researchers have developed Z to Prolog translation systems using this method. We describe here the most important work in that direction.

3.2.1. Manual translation system

An early account of animating a Z specification using translation to Prolog was given by Stepney et al (Stepney 87) in 1987. There, a medium-sized Z specification was hand-translated into Prolog, and a simple user interface was also written. The hand-translation involved writing predicates to handle Z constructs such as sets, including relations and functions, and toolkit operations, then performing an almost line-for-line translation of the Z schemas into their Prolog counterparts. More effort was spent on writing the user interface than in translating the Z specification itself. This was partly because the user interface was not formally specified, and so was slower to write and debug, and partly because the mismatch between the language and the 'graphical' interface. (Prolog being used to generate VT100 control codes for block graphics characters). Although the graphics were crude they were sufficient to make understandable what was happening. More importantly, users could interact directly with the animation, watch the consequences of their requests appearing on the screen.

The advantages of this animation were quite substantial (Barden 94). The specifiers became happier that they had captured the requirements correctly in Z, and the clients were satisfied that requirements that had been captured really were those wanted. Also, the clients found that they could subsequently understand the Z itself much better. However, the obvious disadvantage of this system is that the translation was done manually.

3.2.2. Early automatic system

Other work that looked at automatic translation of Z specifications into Prolog was done by Dick et al 1989 (Dick 90). Automation is valuable, since it reduces possible transcription errors. Effort has been spent on performing valid optimisations of the resulting Prolog, since naive automatic translations can often result in highly inefficient generate-and-test algorithms. But sequencing operations and building a sensible user interface still has to be done by hand. (The system described by Dick et al requires the user to type Prolog queries, and hence to know the translation from Z.)The Z schema was mapped to a form from which the set-theory operations could be called; the latter were contained in a separate collection or library of Prolog rules, based on the one developed by Knott et al 1988 (Knott 88), which had been created for the purpose. This library consists of Prolog recursive predicates that model set-theory operations and, by implication, type constructors such as partial function. Sets are represented by lists, where the order of elements in the list is irrelevant to set equality. In addition some control features such as 'cut' were utilised. These Prolog control features have no axiomatic representation (Lloy 84), but they increase the efficiency of the code for the set operations.

3.2.3. West's Z/Prolog translator

West 1988 (West 88), and West and Eaglestone 1992 (West 92) described a Z to Prolog translation using the structured method. By examining the characteristics of the Z notation they established eight rules on which the general translation method is based. These were stated in details in their work. The first five rules relate to translation within a schema and the last three to schema calculus which allows the construction of a new schema from one or more others. For example rule 9 states that the operation of schema piping

$$C \pm A >> B$$

is captured in Prolog by conjunction of schema predicates as follows
schema_type(L_3,c):-
schema_type(L_1,a), schema_type(L_2,b).

The signature variables of C contained in L_3 are obtained from those of A and B, with the exception of the common identifier which is both the output of A and the input to B, and the rest of L_1, L_2 are merged.

3.2.4. zp translator

Zin (Zin 93) described a Z to Prolog translator as a part of a formal development support system called ZFDSS. His work in principle is similar to the work done by Dick et al (Dick 90) and West and Eaglestone (West 92). He noted, however, that although Z and Prolog are mainly based on predicate logic, both representations are not purely first order predicate logic. This means that there are issues to be resolved first in order for the translation process to be done correctly. These were

- A simple schema can be viewed as a collection of predicates. But the Z schema allows many more complex schema statements, e.g. nested and quantified schema.

- The technique used in Prolog programming considers any query to be false if it cannot be derived from the program. This is known as "negation as failure" or the "closed-world assumption" (Moore 82, Burke 96). This type of reasoning causes Prolog to give up two of the main features of the first order logic: reasoning with incomplete knowledge, and being able to distinguish between that a statement is false and not knowing that it is true.

- The third problem is to ensure that the semantics of Z statements are properly preserved by the equivalent Prolog statements. Z is based on declarative semantics, whereas Prolog supports both types of operational semantics: declarative and procedural semantics. Although in theory, both types of semantics are similar, in practice, they are not (Deville 90).

He then examined in detail the three main aspects addressed by his translation model

- Data representation: which covers the translation of simple and power sets, tuples, functions, relations, and sequences

- Operations representation: which covers the translation of functions and relational statements, comprehensive sets, quantifiers and expression quantifiers, and arithmetic and logical operations

- Z Prolog library: this consists of predicates to define the behaviour of each Z function, relation or statement. The predicates in the library were divided into four categories; managing the data base, handling set elements and basic set operations, defining Z functions and operations, and meta-predicates to handle if $\Rightarrow$, for all $\forall$, exists $\exists$, and the other quantifiers.

The translation of simple Z schema was straightforward. Since a schema is a collection of Z statements which can be represented as predicates in Prolog, so a Z schema can also be represented as a predicate. However, for more complicated Z schema that can include other schemas it was not straightforward. In order for the translation to Prolog to be correct schemas were expanded first to remove all included schemas, if any, and this expanded schema is then translated to Prolog.

3.2.5. PiZA, a Prolog Z animator

PiZA (Hewitt 97) is a program which is able to typeset Z specifications using the Latex typesetter. It accepts input in an ASCII format which is able to convert into many flavours of LaTeX. PiZA also converts restricted forms of the Z specification language into Prolog and execute them. Currently this aspect of PiZAis very poorly documented. The current release (beta release) suffers from two main problems. Firstly, It requires more documentation. Specifically nothing has been written explaining how to convert a declarative Z specification to an executable Z specification. This lack of documentation greatly limits the current tool for use as a Z animator. Also it does not have an integrated type checker. Type checking at present is done by interfacing with external type checkers.

4. CASE STUDY

A demonstration of the application of the translation process from Z to Prolog is shown in the appendix. The translator used was zp. The example is a Z specification of a simple directory that records room numbers and staff names who occupy these rooms in a particular institution.

As the output for this simple Z example is many lines of Prolog, only the predicates for the schemas 'NewStaffrm' and 'FullNewStaffrm' are shown. It is worth mentioning that the translator manipulates data by using a list and store it by using structure. The two predicates 'update(A,L)' and 'memberof(A,L)' are used for manipulating the data base. The predicate 'update(A,L)' updates the "after" variables into "before" variables and transforms them from the structure representation into its list representation. Whereas the predicate 'memberof(A,L)' will assign the list Lto A.

The translation of the schema expression 'FullNewStaffrm', on the other hand, has an error. The goals V712, V317, V716 and V732 are not known at the time of the output. This is not correct Prolog syntax as it cannot be replaced by the correct built-in predicate 'call(_)'.

5. CONCLUSION

Animating Z specification is still in its infancy. Although it is developing, only few tools are commercially available to support it. A technique for animating Z requires translation of Z to Prolog. Some success was achieved by using the structured method and several translators were designed. Though the translation approach was simple and the Prolog implementation was fairly general, two limitations can be noticed. The first is that because a general relationship between Z and Prolog was not found, the simulation depends on the characteristics of a specification being within the bounds of the translation rules used. The second lies in the nature of the Prolog logic. A Z specification gives a logical relationship, whereas Prolog, although in theory a declarative language, in practice does rely on the textual sequence of the code. The lack of data types also means that Prolog sets have to be implemented by lists. These factors could limit the subset of Z that is capable of translation by this method. Moreover animation is still lacking full automation, which can be costly. The effort needed to animate a specification may be more than that of writing it. Hence more work is needed to fully automate the animation process.

APPENDIX

An example of translating a Z specification to Prolog using a Z/Prolog translator (zp)

[NAME, ROOMS]
REPORT ::= ok | already_known | not_known

RoomBook
known : **P** *NAME*
staffrmno : *NAME* $\rightarrow\!\!\!|$ *ROOMS*

---

known=**dom** *staffrmno*

NewStaffrm
Δ*RoomBook*
name? : *NAME*
roomno? : *ROOMS*

---

name? known
staffrmno¢ =staffrmno $\cup$ *{name?* $\mapsto$ *roomno?}*

FindRoomno
Ξ*RoomBook*
name? : *NAME*
roomno! : *ROOMS*

---

name? ⌊ *known*
roomno!=staffrmno name?

WhoinRoomno
Ξ*RoomBook*
roomno? : *ROOMS*
names! : **P** *NAME*

---

names!={n : known | *staffrmno n=roomno?}*

Success
result! : *REPORT*

---

result!=ok

AlreadyKnown
Ξ*RoomBook*
name? : *NAME*
result!=REPORT

---

name? ⌊ *known*
result!=already_known

NotKnown
Ξ*RoomBook*
name? : *NAME*
result!=REPORT

---

name? | *known*
result!=not_known

FullNewStaffrm ± (NewStaffrm $\wedge$ Success) $\vee$ AlreadyKnown
FullFindRoomno ± (FindRoomno $\wedge$ Success) $\vee$ NotKnown
FullWhoinRoomno ± WhoinRoomno $\wedge$ Success

```
pNewStaffrm(V310,V311):-
          update(vknown,V307),
          update(vstaffrmno,V308),!,
          dom(V308,V700),
          equalset(V307,V700,true),
          !,
          notelement(V307,V310,true),
          map(V310,V311,V704),
          makeset(V704,V705),
          union(V308,V705,V706),
          equalset(V308P,V706,true),
          dom(V308P,V708),
          equalset(V307P,V708,true),
          !,
          memberof(vknown,V307),
          memberof(vstaffrmno,V308),
          memberof(vknownP,V307P),
          memberof(vstaffrmnoP,V308P).

pFullNewStaffrm(V310,V311,V317):-
          update(vknown,V307),
          update(vstaffrmno,V308),!,
          dom(V308,V700),
          equalset(V307,V700,V701),
          !,
          notelement(V307,V310,V703),
          and(V701,V703,V704),
          map(V310,V311,V705),
          makeset(V705,V706),
          union(V308,V706,V707),
          equalset(V308P,V707,V708),
          and(V704,V708,V709),
          dom(V308P,V710),
          equalset(V307P,V710,V711),
          and(V709,V711,V712),
          !,
          V712,
          V317,
          and(V714,V715,V716),
          V716,
          dom(V308,V718),
          equalset(V307,V718,V719),
          !,
          element(V307,V310,V721),
          and(V719,V721,V722),
          equalset(V317,valready_known,V723),
          and(V722,V723,V724),
          equalset(V307,V307P,V725),
          and(V724,V725,V726),
          equalset(V308,V308P,V727),
          and(V726,V727,V728),
          !,
          dom(V308P,V730),
          equalset(V307P,V730,V731),
          and(V728,V731,V732),
          !,
          V732,
          or(V717,V734,true),
          memberof(vknown,V307),
          memberof(vstaffrmno,V308),
          memberof(vknownP,V307P),
          memberof(vstaffrmnoP,V308P).
```

REFERENCES

1. ANDREW, S. and Norcliffe, A. (1990), "A CASE Tool for Demonstrating Z specifications", *Proc. IEE Colloquium on Application of CASE Tools*, IEE, London.
2. BARDEN, R.; Stepney, S. and Cooper, D. (1994), *Z in Practice*, Prentice-Hall.
3. BERG, H. K.; Boebert W. E.; Franta, W. R. and Moher, T. G. (1982), *Formal methods of program Verification and specification*, Prentice-Hall Inc.
4. BLOOMFIELD, R. E. and Froome, P. K. D. (1986), "The Application of Formal Methods to the Assessment of High Integrity Software", *IEEE Trans.*, SE-12(9), 988-993.
5. BOEHM, B. K. (1979), "Software engineering: R & D trends and defence needs", *Research Directions in software Technology*, M.I.T Press, 44-86
6. BURKE, E. and Foxley ,E. (1996), *Logic and its Applications*, Prentice Hall International Series in Computer Science.
7. CLOCKSIN, W. F. and Mellish, C. S. (1994), *Programming in Prolog* , Springer-Verlag.
8. DEVILLE, Y. (1990), *Logic Programming - Systematic Program Development*, Addison-Wesley Pub. Co.
9. DICK, A. J.; Kraus, P. J. and Cozens, J. (1990), "Computer Aided

Transformation of Z into Prolog", *Proc. Fourth Annual Z Users Meetings 1989*, Workshops in Computing: Springer-Verlag, Oxford, 71-85.

10. DIJKSTRA, E. G. (1979), "Structured Programming", *Classics in Software Engineering*, Yourdon Press.
11. DILLER, A. (1990), Z: *An Introduction to Formal Methods*, John Wiley, UK.
12. FIELDS, B. and Elovang-Goransson, M. (1992), "AVDM Case Study in mural", *IEEE Trans. Software Eng.*, 18(4), 279-295.
13. Futatsugi, K.; Goguen, J. A.; Jouannaud, J. P. and Meseguer, J. (1985), "Principles of OBJ2", *Proc. 12th ACM Symposium on Principles of Programming Languages*, New Orleans, 52-66.
14. Gladden, G. R. (1982), "Stop the Life-cycle, I Want to Get Off", *ACM SIGSOFT Soft. Eng. Notes*, 7(2), 35-39.
15. Goguen, J. A. (1984), "Parameterized Programming", *IEEE Trans. Software Eng.*, 10(5), 528-543.
16. Guttag, J. V. and Horning, J. J. (1978), "The Algebraic Specification of Abstract Data Types", *Acta Inform.*, 10, 27-52.
17. Hatcher, W. S. (1982), *The logical Foundations of Mathematics*, Pergamon Press, Canada.
18. Hayes, I. J. and Jones, C. B. (1991), "Specifications are not (Necessarily) Executable", *IEE Software Engineering J.*, 330-338.
19. Hekmatpor, S. and Ince, D. (1988), *Software Prototyping, Formal Methods and VDM*, Addison-Wesley.
20. Henderson, P. and Minkowitz, C. (1985), "The 'me too' method of software design", *Technical Report FPN-10*, University of Stirling, Dept. of Computer Science.
21. Hewitt, M. A.; O'Halloran, C. M. and Sennett, C. T. (*1997*),"Experiences with PiZA, an Animator for Z", *Proc. 11th Annual Z Users Meetings*, Workshops in Computing: Springer-Verlag, LNCS 1212.
22. Hogger, C. (1984), *Introduction to logic programming*, Academic Press, London,
23. Jalote, P. (1989), "Testing the completeness of Specifications", *IEEE Transaction on software engineering* , 15(5), 526-531.
24. Jones, C. B. (1986), *Systematic Software Development Using VDM* Prentice-Hall, London.
25. Kemmerer, R. A. (1985), "Testing Formal Specifications to Detect Design Errors", *IEEE Trans. Software Eng.*, 11(1), 32-43.
26. Knott, R. D. and Kraus, P. J. (1988), "An Approach to Animating Z Using Prolog", *Report* A1.1, Alvey Project SE/065, University of Surrey.

27. Kowalski, R. A. (1979), *Logic for problem solving,* North-Holland, New York.

28. Lloyd, J. W. (1984), *Foundations of Logic Programming*, Springer-Verlag, New York.

29. MoD UK (1991), "The Procurement of Safety Critical Software in Defence Equipment", *Defence Standard 00-55/Issue1*, UK Ministry of Defence.

30. Moore, R. C. (1982), *The Role of Logic in Intelligent Systems*, SRI International.

31. Ross, P. (1989), Advanced Prolog, Addison-Wesley Pub. Co.

32. Spivey, J. M. (1989), *The Z Notation: a Reference Manual*, Prentice-Hall.

33. Stepney, S. and Lord, S. P. (1987), "Formal Specification of an Access Control System", *Software-Practice and Experience*, 17(9), 575-593.

34. Turner, D. (1986), "An Overview of Miranda", *ACM SIGPLAN Notices*, 21(12), 158-166.

35. West, M. M. and Eaglestone, B. M. (1992), "Software development: two approaches to animation of Z specification using Prolog", *Software Engineering*, 7(4), 264-276.

36. West, M. M. (1988), *Z/PROLOG Translator*, M.Sc. Dissertation, University of Bradford.

37. Wordsworth, J. B. (1996), *Software Engineering with B*, Addison-Wesley.

38. Zin, A. M. (1993), ZFDSS: *A Formal Development Support System Based on the Liberal Approach*, Ph.D. Thesis, Dept. of Comp. Science, University of Nottingham, UK.