EXPLORING THE BARRIERS TO FORMAL SPECIFICATION

By

Colin Frank Snook

A thesis submitted for the degree of Doctor of Philosophy

Declarative Systems and Software Engineering,
Department of Electronics and Computer Science,
Faculty of Engineering and Applied Sciences,
University of Southampton
United Kingdom.

November 2001

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCES

ELECTRONICS AND COMPUTER SCIENCE DEPARTMENT

Doctor of Philosophy

EXPLORING THE BARRIERS TO FORMAL SPECIFICATION

by Colin Frank Snook

This thesis explores barriers to using formal specification for software development in industry. Empirical assessment techniques are used initially in an exploratory stage and subsequently in testing a hypothesis arising from the first stage. A second hypothesis is investigated by construction of a method and tool with subjective assessment of its effect. The first stage consists of a survey of experienced industrial formal methods users via a questionnaire-based interview. The interviews explore the practicalities of using formal methods in an industrial setting. From the many findings in this stage, two hypotheses are selected for further investigation. The first hypothesis is that formal specifications are no more difficult to understand than code. This is tested by formal experiment. The subject's ability to understand the functionality of a formal specification is compared with their ability to understand its implementation in program code. The second hypothesis is derived from observations, during the survey stage, that formal specifications are difficult to write. In particular, choosing appropriate abstractions is difficult. We consider what might make formal specification difficult and compare the process with that of programming. The second hypothesis is that a tool supported, graphical modelling notation would be of benefit in the process of writing a formal specification. Such a notation is devised by adapting the UML and augmenting it with a formal text notation. A tool that converts this graphical formal specification into the formal notation, B is described and examples of its use are analysed.

Acknowledgements

Thanks must go foremost to my two superb supervisors; firstly Rachel Harrison, now at Reading University, who taught me how to start a PhD and supervised the first part (up to Chapter 4), and latterly Michael Butler who supervised the second part and taught me how to finish a PhD. Thanks also to Manoranjan Satpathy and everyone else at Reading University for their input and encouragement during collaborative work. Thanks to those who contributed to the survey work in Chapter 3: Paul Krause, John Wordsworth, Anthony Hall, Jonathan Draper and Interviewee A. Thanks to Gwil Edmunds who helped organize the experiment in Chapter 4 and to all the CS2 students who participated in it. Thanks to all who expressed an interest in U2B: particularly Marina Waldén, at Åbo Akademi and Muan Ng here at Southampton. For help, suggestions, general encouragement or inspiration at various points, thanks to Andy Gravell, Simon Cox, the Social Statistics department, Barbara Kitchenham, Martin Shepperd, Jim Davies and Christie Bolton. For financial support I would like to thank my funding body, the EPSRC.

I would also like to thank my main confidants: my sister, Alison, who was always able to assuage any guilt and Moira for endless dissections of studentship and its supervision. Thanks to Bean, for making me determined enough to start; my parents, Brenda and Brian, for engendering a belief in the importance of learning; Grandma Snook, for her unconditional love and house and my daughters, Louise and Ruth, for an endless supply of grandchildren and enthusiasm.

CFS

Contents

Chapter 1

Introduction

Formal methods have long held the promise of providing a much-needed solid engineering foundation for the 'art' of programming computers. Proponents have countered popular myths that dubious practitioners raised to dismiss them (Hall, 1990; Bowen and Hinchey, 1995). Experiential reports of their use have invariably been favourable and yet still the adoption of formal methods has been limited. Academic interest in formal methods has been lively with many active research groups throughout the world and plenty of conferences dedicated to their discussion. Despite this interest, uptake within industry has mainly been limited to safety critical applications (some due to mandate by regulatory authorities) and experimentation by a few pioneering market leaders. It seems that practitioners, in their constant search for an edge in productivity and quality are keeping an eye on formal methods but judge them to be insufficiently beneficial to outweigh pragmatic problems. Formal specification is the first step to using formal methods and is, in itself, a useful activity even if the formal specifications are not subsequently used in a full formal development. However, even this first step is not being adopted to any great degree within the industry. Perhaps academia is not prioritising the problems it researches to the greatest effect. Targeting the pragmatic problems that practitioners initially face would lead to increased interest and funding from industry, and a more widespread take-up of formal specification would later lead to faster development of subsequent research in academically appealing areas of formal methods.

Since formal specification is the first step to using formal methods it is also the first barrier that must be overcome if the benefits of full formal methods including refinement and verification is to be achieved. To limit the area of research and make it more manageable, this thesis concentrates on the barriers to formal specification.

1.1 Aims of Research

This thesis explores some of the barriers to the widespread use of formal specification in industry. While we cannot hope to explain all such barriers, the aim is to make some progress in understanding what some of the barriers are and to evaluate them. A further aim is to suggest

possible ways to overcome the identified barriers and to demonstrate that the suggested methods are effective in this respect.

Formal specification bears many similarities with program design. It is convenient and useful when thinking about barriers to formal specification, to think about whether similar barriers exist in programming; and if so, how they have been overcome. The comparison with programming is useful because programming is a more developed and researched area. It is also the main activity and primary goal of the people that we would like to help overcome the barriers to formal specification. These people have a good intuitive 'feel' for attributes of programming, making comparisons meaningful in a practical sense. A parallel, or more concrete, aim therefore is to compare the activity of formal specification with that of programming.

1.2 Outline of Research

Initially the research is wide and exploratory in nature. The thesis explores the main issues in using formal methods as perceived by experienced practitioners. The practitioners were interviewed using a questionnaire as a basis for the discussions. The interviewees were encouraged and prompted to expand on topics of interest in keeping with the exploratory nature of this stage of the research.

From the many findings of this first stage, two topics that are relevant to the aims of the thesis were selected for more detailed investigation. The first topic, comprehension of formal specifications, was selected because it might be thought to be a barrier to formal specification. The interviewees' opinion, however, was that comprehension is not a barrier to software designers and programmers. Anthony Hall, the interviewee from Praxis Critical Systems, made this point most directly. The other interviewees generally indicated that they didn't see comprehension as a problem. The second stage of the research focuses on this issue. A formal experiment was conducted to investigate the hypothesis that formal specifications are no more difficult to understand than code. The subjects' ability to understand the functionality of a formal specification is compared with their ability to understand its implementation in program code. The experimental results support the hypothesis indicating that comprehension of formal specifications by programmers is not a barrier to their use.

The second topic that was selected for further investigation, that writing formal specifications is difficult, was selected because there was a consensus amongst the interviewees that this is a significant problem. In order to explore this topic the thesis compares the activity of writing a formal specification with that of designing software. This leads to the hypothesis that a tool

supported, graphical modelling notation similar to those used in program design would be of benefit in the process of writing a formal specification. In order to explore this hypothesis, such a notation is devised by adapting the UML and augmenting it with a formal textual notation. A tool that converts this graphical formal specification into the formal notation, B, is described and examples of its use are analysed. The examples illustrate the effectiveness of using a semi-graphical formal notation with tool support for the exploratory design activities involved in formal specification. Hence, this may be a route to overcoming a major barrier to the use of formal specifications.

1.3 Structure of Thesis

The rest of the thesis is structured as follows:

Chapter 2 provides a background to the thesis. It summarises the empirical assessment techniques used in the thesis giving examples of their use elsewhere. It introduces the notations discussed in the thesis and the concept of integrating formal and semi-formal notations.

Chapter 3 describes a survey of practitioners using formal methods leading to the selection of two issues for further investigation. This chapter is based on Snook and Harrison (2001a).

Chapter 4 describes an experiment comparing the comprehensibility of a formal specification with its implementation. The chapter investigates the first of the two issues selected in Chapter 3. This chapter is based on Snook and Harrison (2001b).

Chapter 5 discusses the nature of formal specifications and the process of writing them. The similarities between the process of writing a formal specification and that of designing a program are discussed. The difficulties of writing a formal specification are analysed and contrasted with the situation in program design. The chapter provides a theoretical under-pinning for the adaptation of a program design notation and tool to formal specification.

Chapter 6 describes B-UML and U2B. B-UML is an adaptation of UML class diagrams and statecharts with annotations in a B like textual format. B-UML is a semi-graphical formal specification notation based on UML. U2B is a program that converts B-UML specifications into B. This chapter is based on Snook and Butler (2001)

Chapter 7 describes examples of specifications written in B-UML. The examples demonstrate the use of B-UML and illustrate some problems with the current version. The first two examples are from Snook and Butler (2000) and Snook and Butler (2001) respectively. The third example

was written jointly with M. Satpathy of Reading University and is a simplified version of a case study (Satpathy, Harrison, Snook and Butler, 2001) based on a real application.

Chapter 8 describes related work on integrating formal and semi-formal notations comparing it with B-UML and U2B.

Chapter 9 draws conclusions from the thesis and describes further work that we hope to carry out.

Chapter 2

Background and Techniques

This chapter explains the importance of empirical evaluation in research and introduces the forms of evaluation and the techniques involved. A method that we use for assessing the cognitive aspects of a notation is introduced. The chapter introduces the formal methods and notations that are the subject of this investigation.

2.1 Empirical Assessment

The general lack of empirical validation of software engineering theories is described by Fenton (1993) and Glass (1994). Glass comments on the way research in software has become insular and 'academic', losing touch with practitioners and not validating theory with real scale evaluation. In response practitioners have lost faith in research results. This situation Glass says, has arisen from the, mathematical, university background of computer science that tends to view practical application issues with disdain and values pure theoretical research. This has been exacerbated by the practical difficulties of scale and expense in realistic evaluation and the industry's thirst for ideas (without waiting for evaluation) in the early years of computing. The mistrust between researchers and practitioners has been confounded by the researchers' habit of exaggerating the problems of software production as a 'software crisis'. Glass puts forward the Software Engineering Laboratory, SEL (which is a collaboration between academia, industry and government) as a model of how research should be organised. Research and development should go hand in hand so that research ideas are transferred into practice via an established process and bad ideas, which cannot be put into practice, are not kept alive purely by research advocates. Formal methods are cited as an example of an idea being kept alive purely by research. Glass ends by saying that we often make more progress out of our failures than our successes and suggests that the 'research crises' will in the end lead to the ideal co-operative of research organisations that he describes. Similarly, Fenton warns the research community that they should not be exasperated by the poor industrial acceptance of new methods when they lack empirical validation. Fenton discusses the lack of evidence to support formal methods, even for safety critical applications but recognises the difficulties inherent in measuring processes involving humans.

Zelkowitz and Wallace (1998) describe a classification of the possible types of validation methods for software engineering research theories. They point out the limitations of some (such as assertion, where the researcher has control over an example and can bias it) and the practical problems of more convincing methods (such as replicated experiments, which are expensive in most cases). They present the results of a review of past papers, showing the percentage of types of validation methods used. This shows that about a third had no validation, a third used assertion and the rest were distributed over the remaining types but favouring lessons learned, case study and simulation. In their more recent figures there appears to be a trend towards improvement with a fall in 'no validation' (assertion) papers and an increase in lessons learned, case studies and replicated experiments. Despite the improvements the current situation regarding validation of research is still poor.

2.1.1 Measurement

Any form of empirical assessment must be based on sound measurement and Fenton's book "Software Metrics" (1996) provides a theoretical basis to selecting measures and the types of, and relationships between, attributes as well as covering the prediction and measurement of specific external product attributes. Curtis (1980) provides an earlier description of many of these measurement issues and also covers issues in the design of experiments. Kitchenham, Pfleeger and Fenton (1995) define a structure model of measurement. This is followed up with models for the components of the structure model. The requirements for validating a measurement are then described in terms of these models. The structure model consists of entities, attributes, values, units, scales and measurement instruments. The concept of unit is extended from the classical meaning (applicable only to interval and ratio scales) to cover nominal and ordinal scales as well. Scales are associated with units not attributes, i.e. several different units, which could have different scale types could be used for a particular attribute, but the particular unit type is based on one scale type. For example, the attribute, temperature, can be measured using a ratio scale such as degrees Kelvin or an interval scale, such as Centigrade, or an ordinal scale such as cold-lukewarm-warm-hot. Indirect measures and compound units are discussed. The problems with creating a scalar value from a set of direct measures without having a valid underlying model of the relationships between these attributes are covered. It is suggested that in these cases it is preferable to leave the measure as a vector. Pfleeger, Jeffery, Curtis, and Kitchenham (1997) report on how practitioners are a long way behind the (measurement) theory and are making mistakes. Some views on what the research community needs to do to rectify things are suggested. Some of the areas in which practitioners are going wrong are: not keeping the goals in mind; relying on empirical evidence without regard to theoretical validity; not considering model validity; not distinguishing prediction from

assessment; unwillingness to commit resource to process measurements; use of published model parameters that are only relevant to a particular case. Researchers need to "fashion results into tools and techniques that practitioners can easily understand and apply" and focus on the areas that practitioners and customers desire most (early measures – requirements, costs). Pfleeger, Jeffery, Curtis, and Kitchenham end with a warning from a statistician not to become like the statistics community, which is segregated from the people using the methods. Software Metrics research must produce methods that are useful to and useable by the software engineering practitioners.

2.1.2 Types of Empirical Assessment

Most authors of general empirical assessment literature classify assessments into three general forms. These are Surveys (systematic post-hoc data collection from a known population), Formal Experiments (controlled and replicated treatments on a number of subjects) and Case studies (intensive interpretation of a small sample). For example Wynekoop and Russo (1997) classify published assessments of software development methods into these (and other) categories. (Their other empirical assessment categories could all be considered sub categories of case studies under a looser definition). Kitchenham (1996) attempts to identify a method for selection of validation techniques for evaluating software engineering methods and tools. She starts by defining a classification of validation methods and this is based on surveys, formal experiments and case studies. As part of the same, DESMET, project Kitchenham, Linkman and Law (1994) provide a critical review of past quantitative assessments and base this around a classification into surveys, formal experiments and case studies. They recommend case studies as being particularly effective from an industry point of view. Daly (1996) points out the value of using all three forms of empirical assessment to support each other in establishing an hypothesis. The Survey contributes to the formulation of the hypothesis and increases the likelihood that it is relevant, the formal experiments establish that a relationship exists and the case study demonstrates that the results can be generalised to real life situations.

Surveys

Surveys rely on individual's memories of their experiences. Because of this, they can be limited in accuracy. Pannell and Pannell (1999) give an informative discussion on the problems of extracting the truth via surveys and how to maximise the chances of getting valid answers. Some of the problems include incorrect answers (an estimated 5-17% of answers are incorrect), misinformation, changing opinions, wording of questions, misinterpretation and ordering of questions. Nevertheless, surveys provide a powerful method to get an initial indication of the

properties of a topic from a wide subject base. Survey data can lead to the formulation of relevant, and widely held, hypotheses.

A survey based on a distributed questionnaire relies on the questions asked and the way they are phrased. This implies that a prior knowledge of the interesting issues and a possible outcome. A structured interview consists of an interview based around a predefined set of questions. The questions provide a consistent structure for the interviews but the interviewer can discover knowledge by seeking confirmatory evidence as necessary. The interviewer can also explore the experience and language of the interviewee to put answers in context. Thus many of the shortcomings of an independent survey are overcome. Structured interviews are limited to a small selected set of experienced subjects but enable a wider exploration of the subject to be performed and a higher level of confidence in the answers. However, the results will be a reflection of the opinions and prejudices of a small set of subjects. The selection of these subjects may ensure that they are the best-placed individuals to give an accurate opinion. On the other hand other empirical assessment techniques should be used to test the results of the structured interviews. Our structured interview is reported in Chapter 3 of this thesis.

A survey of formal methods usage in industry and academia was carried out by Austin and Parkin (1993) of the National Physical Laboratory. The industrial survey was performed by sending out questionnaires to both formal methods users and non-users. (The author participated in this survey as a non-user). The most popular benefits of formal methods were their clear and unambiguous specifications, their early detection of errors. The ability to prove properties, build the software and prove its correctness and the ability to demonstrate the specification to clients were less popular but also strongly represented. The main limitations were that clients cannot understand them and that some aspects of modelling are difficult or even impossible (e.g. timing, maintainability etc.). Other limitations that were strongly supported were, the lack of experienced staff, the high costs of performing proofs and the possibility that the formal specification may contain mistakes. The main barriers to the use of formal specifications were considered to be, the lack of tool support and the high costs. Other barriers that were identified were, the need for training, the fact that they are difficult to use, the lack of objective evidence of the benefits and a perception that they are not mature enough. Interestingly, the results indicated a general agreement between formal methods users and non-users, dispelling to some extent the notion that there is a false prejudice against formal methods. The results of this survey do not contradict the results of our survey and in some areas, such as 'early detection of problems', our findings are in agreement. However, they do not support our findings very strongly either. In particular, the NPL survey makes little reference to the hypotheses we selected for further investigation, which were strongly suggested from our interviews with

practitioners. We suggest that this may be because of the remote, questionnaire method. Despite the authors' stated attempts to "not lead people to answer the questions in a particular way", we believe the written style of the communication and its lack of interaction with the subjects means that emphasis or underlying causes are often missed. For example the 'lack of tools', 'training' and 'difficult to use' barriers may well be related to our survey finding and hypothesis that formal specifications are difficult to write and would benefit from tools similar to those used for program design. Similarly the lack of any mention of comprehensibility problems as barriers to use could be interpreted as a strong indication that comprehension is not a problem. (A small number of respondents mentioned the need for mathematics as a barrier but this was mostly non-users and did not distinguish between creation and comprehension).

Formal Experiments

The purpose of a formal experiment is to test a relationship in a particular system. The effect of confounding factors must be minimised so that we are able to attribute changes in the dependent variable to changes in the independent variable. Ideally the experiment should be performed in a realistic setting, however, it is usually impossible to control confounding factors adequately in a realistic setting. The priority in a formal experiment is to isolate and demonstrate the relationship under test. Once the relationship has been established as likely to exist we may then consider to what degree it is relevant to real life scenarios.

Tichy (1998) makes a case for performing formally controlled experiments and refutes the 'fallacies' that are often held up as reasons for not performing experiments in computer science. Brooks (1980) gives a useful description of things that must be considered in formal experiments, covering subjects, materials and measures. When many possible relationships can be envisaged, there is a temptation to gather one set of data and then try many different relationships in a search for a correlation. However, when we analyse experimental results we are considering the probability of the measured data with respect to a possible distribution. The more relationships are sought, therefore, the higher the probability that one will be detected incorrectly. Courtney and Gustafson (1993) warn of this danger. A well thought out and often-cited experiment is described by Scanlon (1989). Care was taken over the design and implementation of the experiment with a high level of training in the experimental method and automation of measuring methods. Experiments to determine the effect of commenting, meaningful names and structure on the comprehensibility of formal specifications have been carried out by Finney, Rennolls, and Fedorec (1998) and Finney, Fenton, and Fedorec (1999). It was found that good commenting and naming improves comprehensibility. It was also found that there is an optimal level of structuring. The notation used was Z and the specification was broken down to various degrees with schemas. Too many small schemas are detrimental to

comprehensibility, as is a monolithic specification lacking any schema structuring. Experiments have also been performed by Vinter (1998) to investigate the propensity for people to misinterpret various forms of logic statements.

To be of use to practitioners and researchers empirical assessments must meet certain criteria and must be reported effectively. Sufficient information must be provided so that practitioners can judge to what extent the results are likely to apply to their environment. Other researchers need information about the experimental methods and tools in order to be able to assess and replicate the results. Kitchenham, Pfleeger, Pickard, Jones, Hoaglin, El-Emam, and Rosenborg (2001) provide comprehensive guidelines for performing and reporting software engineering research experiments.

Case Studies

Case studies lack the level of control that formal experiments have. The behaviour of interest is observed in a real life example. The many other environmental parameters are uncontrolled and may influence the dependent variable being observed. To alleviate this to some extent a typical baseline is used for comparison. However, a case study cannot be considered as rigorous an empirical investigation as a formal experiment. Nevertheless, case studies have an important role because they test whether a relationship is observed in real situations. This can support formal experiment results, either as an investigatory stage (establishing a hypothesis to test) or as a follow up stage (establishing the generality of experimental results).

An interesting retrospective case study in the use of formal methods is described by Pfleeger and Hatton (1997). This case study was hampered by the fact that it was not planned in advance. Hence the authors found limitations in the data that had been collected for the investigation they were performing and could make little in the way of firm conclusions. The authors also seem to use a dubious surrogate measure of reliability by measuring the number of changes made. A pre-planned case study was performed by Marconi (Draper, Treharne, Boyce and Ormsby, 1996) in the use of the B-method on a parallel project. The study found that errors were detected earlier in the lifecycle and that the project costs were similar to the parallel, real project using their conventional design methods. Another parallel projects case study (Brookes, Fitzgerald, and Larsen, 1996), which found similar results, was performed by British Aerospace.

2.1.3 Statistical Analysis

Statistical analysis techniques assess the likelihood of the recorded sample against a known or assumed population distribution. The more powerful parametric methods assume that the

underlying population is normal. They provide the most definitive results because they use all the available information in the data. If the normality of the parameter's distribution is in doubt then more robust methods should be used. One such class of methods are non-parametric methods that reduce the data to an ordinal scale and make use of ranking properties. Rank statistics obey a normal distribution even when the parameter itself does not, however, because information has been discarded, the results are usually less powerful than parametric methods. A comparatively modern technique is 'bootstrapping' or 'resampling' (Efron and Tibshirani, 1993). This technique uses computer processing to take many samples from the original sample and calculate the statistic of interest for each of these resamples. If the original sample is representative of the overall population, then each resample, and hence each value of the statistic calculated from the resample, is just as valid as if it was sampled from the population. Hence a distribution for the statistic of interest can be generated. Bootstrap techniques do not make assumptions about the distribution of the underlying population distribution, but can be just as powerful as traditional parametric analysis techniques. More details of the statistical techniques used will be presented in Chapter 4.

When performing comparative experiments we are usually interested in detecting a difference in some attribute under two treatments. Following the classical null hypothesis statistical testing process (NHSTP) we would construct a null hypothesis stating that there is no difference and attempt to reject this on the basis of the sample data being unlikely if it were so, leaving an alternative hypothesis that there is a difference. In our experiment in Chapter 4, our substantive hypothesis is that there will be no significant (in the practical sense) difference. Unfortunately not rejecting a null hypothesis is a much weaker result; all we may say is that this sample didn't cause us to reject the null hypothesis. It does not give us any basis for saying that the null hypothesis is likely to be true or any evaluation of its probability. One way round this problem would be to take the approach that a null hypothesis is a hypothesis that we wish to nullify (rather than one of no difference). Then we could formulate the null hypothesis that there is a significant difference and see if we can reject it. However this would require us to arbitrarily define what we mean by a difference (Rozeboom 1960). Note that it would invalidate the NHSTP method if we were to choose this definition in the light of our sample data. Traditionally, when we reject a null hypothesis the meaning of 'different' is not discussed because it 'falls out' of the statistical analysis. A 'difference' is that magnitude such that a sample of differences greater than this magnitude would be unlikely to occur by chance if the 'no difference' hypothesis were true. Hence when we talk about statistically significant differences we are referring to the reliability of the evidence that there is a difference and not to the importance of the magnitude of the difference. Chow (1996) gives a good overview of criticisms of NHSTP (as well as making a case in its favour) in his book 'Statistical

Significance'. Further criticism of the misuse of NHSTP is given by Bakan (1960) and Rozeboom (1960). Many statistical authors (e.g. Wonnacott and Wonnacott 1985) recommend using confidence intervals to explain the results of experiments rather than NHSTP, and we take this approach partly due to our problem with the null hypothesis but also because it is more informative and less reliant on arbitrary choices of criteria.

2.2 Formal Methods

Formal specifications are descriptions of behaviour expressed in a mathematical notation that has a well-defined syntax and semantics. Formal methods are processes of specification, refinement and verification based on formal specifications. We introduce two formal methods, Z and B, that are used in subsequent chapters.

2.2.1 The Z notation

The Z language (Spivey, 1988) is a state based, formal specification language that is based on Zermelo Fränkel axiomatic set theory and first order predicate logic. Schemas are used to structure Z specifications. Schemas associate state variables with predicates based upon them. Schemas can be used within other schemas as state declarations, types, or in predicates. To build a Z specification firstly state variables and invariants that hold on them are defined. Then schemas that define events that alter the state are added. Events are defined in terms of precondition predicates and postcondition predicates. Event schemas can be combined by conjunction and disjunction to compose more complex changes. Once defined, invariants can be relied upon to hold throughout the specification. That is, in event schemas, it is not necessary to define state changes to maintain the invariant, these can be assumed. However, apart from variables controlled by the invariant, it is necessary to fully specify the postcondition over the complete state space referenced in the schema. It is necessary to define what has not changed as well as what has.

Z has a powerful, but rather unapproachable, facility called promotion. Promotion allows hierarchical structuring of a specification. The event schemas for a defined type (i.e. local sub-state space) that are used by a higher-level parent object (by defining instances of the type) can be promoted for use in the parent's operation schemas. Considering the importance of a hierarchical class structuring mechanism in coping with the scale of large systems it is unfortunate that promotion is so difficult to grasp initially and consequently off-putting to students.

Z is popular to the extent that it is probably the most commonly used formal specification language. Craigen, Gerhart & Ralston (1995) put this down to the close interaction between the developers and industrial users and to a substantial pedagogical literature. There are a good number of tools to support the use of Z although many are not industrial strength, supported products and there is little integration of tools.

The following example is a Z specification for a telephone book.

NAME, NUMB

PB

Pbook: NAME ⇸ NUMB
∀ n1,n2 ∈ dom(pbook) | n1 ≠ n2 • pbook(n1) ≠ pbook(n2)

Init

PB
pbook = ∅

lookup

Ξ PB name? :NAME numb! :NUMB
name? ∈ dom(pbook) numb! = pbook(name?)

add

Δ PB name? :NAME numb? :NUMB
name? ∉ dom(pbook) pbook' = pbook ∪ {name? ↦ numb?}

remove

Δ PB name? :NAME
name? ∈ dom(pbook) pbook' = {name?} ⩤ pbook

The schema, PB, defines the state variable, pbook, which models the phonebook and an invariant that ensures that numbers must be unique. (This is not the most succinct form, but we wish to illustrate the methods that would be used in a bigger example). The schema, Init, defines the initial value of pbook. The schema, lookup, returns the number corresponding to a given name (the use of ? and ! in local variable names is a convention to indicate inputs and outputs, respectively, of an operation). The schema includes the state schema PB so that pbook can be accessed. The symbol, Ξ, includes two copies (one copy is decorated, indicating post operation

state) of all the variables in the schema and a predicate to ensure that the post operation value is equal to the pre-operation value. This ensures that pbook is unchanged by lookup. The remaining schemas, add and remove, define events that alter pbook. The symbol, Δ, includes two copies of PB. Again, one is decorated to indicate post operation state, but this time there is no equality predicate. Note that, in the add operation, a precondition to ensure that numb? does not already belong to ran(pbook) is not necessary because the invariant already ensures this.

2.2.2 The B method and notation

The B language (Abrial, 1996) is a state model-based, formal specification notation that has strong structuring mechanisms and good tool support. There are 2 commercial tools for B, Atelier-B (ClearSy) and the B-Toolkit (B-Core, 1996). We have used the B-Toolkit for our translation and animation work, and Atelier-B for performing proofs. B is designed to support formally verified development from specification through to implementation. To do this it provides tool support for generating and proving proof obligations at each stage of refinement. The B-Toolkit also provides animation facilities so that the validity of the specification can be investigated prior to development. To make large-scale development feasible, B provides structuring mechanisms to decompose the specification and its subsequent refinements. These are machines, refinements and implementations. We are mainly concerned with specification and therefore machines. Machines allow an abstract state to be partitioned so that parts of the state can be encapsulated and segregated, thus making them easier to comprehend, reason about and manipulate. One machine may include ('INCLUDES') another machine. If machine A includes machine B, the state of B is visible to A and alterable via B's operations. Another form of machine inclusion is 'EXTENDS'. This is the same as INCLUDES but makes the included machines operations accessible as if they were operations of the including machine. A weaker form of interfacing between machines is provided by 'USES'. The using machine has only read access to the used machines variables and cannot invoke its operations. A machine may be used by any number of other machines but may only be included (or extended) by one other machine.

It is worth noting that, unlike Z, in B the invariant is a verification property which operations are expected to achieve. The invariant is an abstract state specification that is used for checking the correctness of the behavioural specification.

The following example is the same telephone book as above, but this time expressed as a B machine.

```
MACHINE phonebook
SETS        NAME; NUMB
VARIABLES   pbook
```

```
INVARIANT   pbook : (NAME +-> NUMB) &
            !(n1,n2).((n1:ran(pbook) & n2:ran(pbook) &
                        n1/=n2) => (pbook(n1) /= pbook(n2))
            )
INITIALISATION pbook := {}
OPERATIONS
numb <-- lookup(name) =
      PRE   name:dom(pbook)
      THEN  numb:=pbook(name)
      END;
add(name,numb) =
      PRE   name:NAME & numb:NUMB &
            name/:dom(pbook) &
            numb/:ran(pbook)
      THEN  pbook:=pbook\/{name|->numb}
      END;
remove(name) =
      PRE   name:dom(pbook)
      THEN  pbook:={name}<<|pbook
      END
update(name,numb) =
      PRE   name:NAME & numb:NUMB &
            name:dom(pbook)
      THEN  pbook(name) := numb
      END;
END
```

In the B notation, invariants define the type of a variable. In this case, a variable represents the phone book and its type is a partial function from names to numbers. An invariant ensures that numbers in the phonebook are unique. Initially, `pbook` is empty. In the machine's operations, preconditions define the type of any arguments. Additional preconditions may be specified on the arguments or on the state variables. For example, in the `add` operation, `name` must not be a member of the domain of the partial function, `pbook`, and `numb` must not belong to its range. (We cannot rely on the invariant for the latter, as we did in the Z example). Operation postconditions are defined via 'substitutions' that show how the final state of machine variables depends on their initial state and the arguments. (Any state variables not defined in an operation body are not altered by it). Operations may return values. The identifier(s) representing the return value(s) are defined at the beginning of the operation signature (e.g. `numb` in operation `lookup`). Other symbols used in the example are: union `\/`, maplet `|->` and domain subtraction `<<|`.

2.3 Semi-Formal Notations

Semi-formal notations are notations that provide a set of symbols to represent specific roles in the description of a system, but have a loosely defined semantics. The use of a syntactically consistent notation generally brings a more formal feel to descriptions of systems than an

English language description would. This can be misleading as the lack of a precise semantics leaves the description open to different interpretations.

2.3.1 The UML

The Unified Modelling Language (Rumbaugh, Jacobson & Booch, 1998) emerged as a standardisation of the leading object-oriented analysis and design methods that were competing for favour in the late 1980s and early 1990s. This unification was brought about by three of the methods advocates joining forces at a major software tools company, Rational Software. Responsibility for the standardisation was subsequently taken over by an independent consortium, the Object Management Group (OMG). Several software tool manufacturers market tools to support the use of the UML. We use Rational Software's 'Rose' tool.

The UML is a notation for use in modelling object-orientedobject-oriented designs. A unified process, Rational Unified Process (RUP), exists, but is not necessary to use the UML. The UML consists of the following parts:

Use Case diagrams are a means of organising requirements descriptions into event sequence scenarios. A scenario is triggered by an actor (an external object such as a person interacting with the system) and parts of the system's responsive actions are then packaged and represented by named symbols. The meaning of a particular symbol is defined textually, usually in natural language.

Class Diagrams are used to model the static structure of a problem or system. Entity types are represented by classes and the relationships between them are shown as associations and generalisations. Classes represent sets of like instances and are given attributes that represent state variables and values associated with each instance of the class. Classes also have operations that define how an instance's attributes and associations alter in response to events.

Collaboration Diagrams and Sequence Diagrams are equivalent to each other. They both show dynamic behaviour as objects (of the classes introduced in the class diagram) interacting, by passing messages or calling each other's operations, to perform a particular behaviour or task scenario. Sequence diagrams show the interaction as a time ordered sequence of messages passed between objects. Collaboration Diagrams show the same sequence of messages but overlaid on a network of connected objects rather than a time sequence.

State Diagrams and Activity Diagrams – Statechart/Activity models, constructed and viewed via state diagrams and/or activity diagrams, show behaviour in terms of a set of states and transitions between them. Each transition can be annotated with the event that causes it to occur,

any guards, which must be true before it can occur, and actions that are performed when it occurs. Activity diagrams are a development from state diagrams that also allow 'forks' to activate more than one state simultaneously and synchronisations that require more than one state to be active before a transition can occur. (When drawing activity diagrams, states are called activities). Statechart/Activity models can be used at several levels. For example, they can be attached to the logical model, to use cases or to classes

In Chapter 6, we use class diagrams to build the basic structure of a formal model and statecharts to assist in the definition of the class' dynamic behaviour.

2.4 Integrating Formal and Semi-Formal Notations

Semi-Formal Notations such as UML are gaining widespread popularity in industry but lack precision for describing detailed behaviour unambiguously. Conversely, formal notations have not gained widespread use in industry despite their recognised benefits. An integration of semi formal and formal notations may address the deficiencies of the semi formal notations while making the formal notation more approachable. Craigen, Gerhart and Ralston (1995) found that better integration of formal methods with existing software assurance techniques and design processes was commonly seen as a major goal. They concluded, "Successful integration is important to the long term success of formal methods". Fraser, Kumar and Vaishnavi (1994) discuss some of the reasons why this may be true and go on to describe a framework for classifying current formal specification processes according to the degree of transitional semiformal stages. The categories are direct (no transitional stages), sequential transitional (transitional stages developed prior to the formal specification), and parallel successive refinement (formal specification derived in parallel with semiformal specification through iterative process). Paige (1997) analyses the composition of compatible notations and derives a meta-method for formal method (and semi-formal method) integration. Jackson (2000) has developed a formal notation, Alloy and associated tool Alcoa. The Alloy notation has a partial graphical equivalent notation in which state can be expressed. This can then be converted into the textual version of the notation where operations can be added and analyses performed. Without tools to investigate the implications of different structures however, the graphical format is limited to illustration of structure. The work of several research groups that have developed integration between graphical object-oriented notations, including the UML, and formal notations such as B and Z are described in Chapter 8. The precise UML group[1] is a collaborative effort to precisely define UML semantics via formalisation. The object constraint

[1] (http://www.cs.york.ac.uk/puml/maindetails.html).

language, OCL, (Warmer and Kleppe, 1999) is a formal notation that is part of the UML. It can be used to attach formal constraint statements to elements of UML models to constrain their values. For example the behaviour of an operation can be precisely defined by attaching OCL statements for the pre and post conditions of the operation. A more detailed comparison of work combining semi-formal and formal notation will be given in Chapter 8.

2.5 Cognitive Dimensions

In Chapter 5 we are interested in the comparative merits of the formal notation, B, versus the semi-formal UML for specification design. It would be useful to be able to discuss the various attributes of these notations in order to formulate theories and to explain results. Green (1989) presents a framework and vocabulary for discussing cognitive artefacts. Cognitive Dimensions provide a broad-brush qualitative tool for reasoning about the relative merits of information systems with respect to particular types of tasks. The cognitive dimensions framework consists of 14 terms that describe generalised facilities of information systems, notations or artefacts. For example 'viscosity' is the degree of difficulty in making structural changes to descriptions expressed within the system. The 14 dimensions are listed below. For an introductory tutorial see Green and Blackwell (1998).

Abstraction Gradient – How the notation copes with abstractions. Some notations don't allow abstractions, for some they are optional and others are hungry for them. Abstractions are good for clarity but difficult to get right.

Closeness of Mapping – How well constructs map on to problem domain entities.

Consistency – If a notation does something one way in one situation then it should do it similarly for all similar variant situations.

Diffuseness/Terseness – How terse the notation is. Terseness and diffuseness can both cause comprehension problems, a compromise is best.

Error-Proneness – How much the notation leads one to make mistakes or slips.

Hard Mental Operations – Does the notation itself induce 'brain-teasers'. (If it cannot be expressed more clearly in another notation it may be an inherently difficult semantics)

Hidden Dependencies – Links to other information elsewhere that are not visible at the place they affect.

Premature Commitment – How much thought needs to go into future actions when a decision is made

Progressive Evaluation – Whether facilities exist to check what has been achieved so far.

Role-expressiveness – How easy is it to tell what this bit is for.

Secondary Notation – facilities for expressing extra information outside the formal syntax (e.g. indentation, grouping, comments)

Viscosity – How difficult it is to make structural changes to what has been achieved so far.

Visibility – How much of the whole can be viewed and juxtaposed.

A number of types of activity that might be performed on an information system are identified. One of these activities, exploratory design, consists of the identification and evaluation of possible architectures and is applicable for our purposes in Chapter 5. Applicability profiles of the Cognitive Dimensions can be identified for each activity type. For example viscosity is inconsequential for transcription but critical for exploratory design. In chapter 5 we use cognitive dimensions to assess the suitability of the B notation with respect to exploratory design.

Chapter 3

Practitioners Views on the Use of Formal Methods

This chapter reports on a series of structured interviews, which have been conducted with formal methods practitioners. In Chapter 2, the need for empirical assessment, especially in formal methods, was introduced. The types of empirical assessment were described and the contribution each makes to the establishment and investigation of a hypothesis was discussed. This provides a context for the report on the conduct and findings of the series of structured interviews that form our survey. The chapter concludes by describing how subsequent work arose from the results of the interviews, including the formulation of two hypotheses.

The survey covered a broad range of topics associated with the effects that using formal methods might have on a company and its products. The survey was conducted by structured interviews based on a questionnaire (see Appendix A.1).

3.1 Purpose of Survey

The aim of the survey was to explore the experiences of practitioners directly. There are many popular theories about formal methods that have questionable validity and it is often unclear whether they are based on actual experience. Hall (1990) discusses some of these myths, as do Bowen and Hinchy (1995). Therefore it was seen as important to investigate the effects of using formal methods directly with individuals who had first hand experience. Of course, the results still depend on the subjective opinions of these individuals and the environments in which their experiences were obtained. This must be borne in mind when the results are interpreted and the results should be viewed as provisional until further empirical assessment has been carried out to corroborate them.

We wanted to discover the main issues involved in the use of formal methods. In particular, issues surrounding comprehensibility and the difficulty of creating and using formal specifications. It was hoped that significant points would be raised that would warrant further empirical assessment. In this way the survey was seen as the first stage of a 'Multi-method' programme of research as described by Daly (1996). The purpose of this first stage was to raise

interesting and relevant provisional findings for further research rather than firm conclusions, which would be suspect, based on such a small survey.

3.2 Conduct of Survey

The companies were initially contacted by email with a brief outline of our aims and the questions that would be asked. Meetings were set up at the company's premises where the representatives were interviewed. The interviews were structured around a questionnaire but the interviewees were encouraged to digress and elaborate on topics as much as they felt necessary. The questions were used to trigger discussion and as a checklist, but, in an effort to explore the subject widely, the discussions were conducted in an open, free form without constraining the topic to the initial question. The interviewees related answers to their experiences to provide justification and in the process the context of the interviewees' answers and their understanding of key phrases were discovered. This happened mostly as a natural part of the discussions without conscious effort. The final question asked the interviewee if there were any important issues that had not been covered. In most cases the interviewee recapped some of the more important issues at this point but did not raise any new issues. This indicates that the questionnaire covers the main points of interest with respect to formal methods. Each interview lasted approximately 2 hours. The author conducted all the interviews. The interviews were tape-recorded. It was felt that recording the interviews avoided the interviewer from being distracted by note taking. It also meant that the interviewees' opinions could be summarised and distilled with greater consideration and care than would have been possible if taking notes 'on the fly'. The tapes were analysed in detail and comments categorised and matched with like comments from other interviewees. From this process a table (Appendix A.2) of summary notes was built up with rows representing each point made by the interviewees and each column representing the summaries of a particular interviewees responses. The text of this report was written from the summary table. Despite the careful analysis of the actual conversation on the tapes it is still possible that the authors might misinterpret responses or inappropriately emphasise a point. To guard against this the report was circulated to the interviewees for review. A few adjustments arose from this review stage, but on the whole the interviewees agreed that the report was an accurate representation of their views.

All of the interviewees had at least some experience of using formal specifications on full-scale products. Some had also performed refinement, model checking and verification proofs. For various reasons only one company was using formal methods to the same extent as previously but all retained a capability or interest. Market sector varied greatly, including commercial computing systems, safety critical embedded systems and high street consumer products. Table

3.1 lists the companies and Table 3.2 gives an outline of their background and experience.

Company	Identification in this report
(wishes to remain anonymous)	Interviewee A
IBM United Kingdom Laboratories, Hursley Park, Winchester, Hants	IBM
Marconi Electronic Systems. Avionics Systems, Airport Works, Rochester, Kent	Marconi
Philips Research Laboratories Crossoak Lane, Redhill, Surrey	Philips
Praxis Critical Systems, Manvers Street, Bath	Praxis

Table 3.1 - Participating Companies

Company	Market Sector	Notations Used	Extent of Use	Approx. Size of Systems	Current Level of Use
Interviewee A	Contractor with personal experience	Z, VDM(some), CSP (some)	Experience with large and small applications	-	Introducing formal methods into a company
IBM	Commercial computer systems	Z, B	Mainly specification	50 Kloc	Isolated usage - at option of project manager
Marconi	Military Embedded Systems (some safety critical)	B	Full development incl. refinement proofs etc.	3 Kloc	Completed case study - bidding for contracts
Philips	Consumer Products	set theory and first order logic	Mainly specification	10+Kloc	Isolated usage - investigating applicability
Praxis	Safety Critical systems	Z, VDM, CSP (some) CCS (some)	Some full developments, others specification only	10Kloc - 100+Kloc	Continuing full scale use

Table 3.2 - Main characteristics of contributors

At this stage of investigation the wide spread of market sector backgrounds is an advantage to the broad information gathering process. In subsequent stages less variability will be needed as we focus more narrowly on selected issues. The companies are, in most cases, the market leader in their sector and the interviewees are the technical experts within those companies. In several cases the interviewees have published in the area of formal methods. It is reasonable therefore to claim that the interviewees are knowledgeable and experienced in the use of formal methods. It might be argued that the interviewees are all proponents of formal methods and the results might therefore be a biased view. We believe that the commercial pressures upon the interviewees would not allow them to maintain an unrealistic stance. It was apparent however that market sector has a bearing on the stance taken, with the safety critical areas having much

more compelling reasons for supporting the use of formal methods, and the others having a more guarded response.

Each interviewee was asked to define a formal method. Most answers indicated that a mathematical notation or underlying theory was needed (one interviewee required a precise syntax and semantics). Some required there to be methods for manipulation and refinement, others recognised these as possible extensions but did not require them. It was thought that some companies might have a looser definition of formal methods. To test this the interviewees were asked if they would include modelling languages such as UML. All would not, although several interviewees suggested that some parts of UML (e.g. statecharts) are close to being a formal notation. Some added that UML did not contain facilities to express the semantic details of the behaviour of systems.

The formal methods that had been used by the interviewees are as follows: Z and B, which were introduced in Chapter 2. VDM (Jones, 1986) (The Vienna Development Method) is a notation and set of techniques for modelling computing systems, analysing those models and progressing to detailed design and coding. VDM has its origins in the work of the IBM Vienna Laboratory in the mid-1970s. CSP (Hoare, 1985) (Communicating Sequential Processes) is a notation for concurrency based on synchronous message passing and selective communications designed by Hoare in 1978. CCS (Milner, 1985) (Calculus of Communicating Systems) is a mathematical model for describing processes, used in the study of parallelism. It was developed by Milner.

3.3 Results

3.3.1 The Customer's Viewpoint

The companies interviewed had very different market sectors and this led to large variations in answers to questions about customer views on their use of formal methods.

Marconi, being a UK defence contractor, often bids for contracts with Def-Stan 00-55 as a mandatory standard (Ministry of Defence 1997). Hence Marconi's use of formal methods is imposed by its main customer (or at least by the regulatory authorities that its customer has to satisfy). Marconi also supplies outside of the UK, e.g. USA, and for these customers it is expected that persuasion would be needed to convince them to accept formal proof in place of other verification methods such as testing and reviewing.

Note that there is an implication here that formal verification is seen as a partial replacement for other verification methods rather than an additional activity. Formal proof provides an absolute

guarantee of the properties it proves and hence verification of those properties by other means becomes redundant. We have found this to be true from other sources. For example, when software was developed, using the B method, for the Paris underground, unit and integration level testing was not performed. (Boehm, Benoir, Faivre and Meynadier, 1999)

Praxis also supply to the UK MoD and to other authorities that are very safety conscious such as aviation authorities. It also supplies to other markets and finds that some of these customers resist the use of formal methods because of the barrier it creates between supplier and customer. Typically, the customer will need to train some of its employees if it wants to be involved in verification and validation activities during the software development.

The remaining interviewees felt that their customers (which for IBM and Philips were internal) were usually impressed by the use of formal methods, and assumed they would lead to high quality products. Where the formal specifications were used as interfaces to customers, the customer's technical staff (who sometimes needed special training) usually found formality helpful because they knew the precise behaviour of the specified system. It was recognised that the audience may be restricted by formality but this is the case for any technical specification.

Both IBM and Praxis commented that one of the main barriers to the widespread use of formal methods is the general acceptance that software is error prone. One interviewee said "if you want highly reliable software then formal methods are the most cost effective way to produce it, but if the customer will accept unreliable software then it is cheaper not to use formal methods". From the suppliers point of view, any subsequent re-work is either covered in the initial price or is paid for by the customer as a maintenance contract. IBM went on to say that some customers do not want to be tied down to what they require, but would rather have a vague specification of requirements and hope the supplier produces something over and above it, than to be forced to address compromises in order to precisely specify their requirements and then take responsibility for the systems validity.

3.3.2 Impact on Company

Quality Assurance

Opinion on how formal methods affect quality assurance issues was uniform. All (except one company that, independent of the method used, had dispensed with its quality assurance function) agreed that the quality assurance function is not changed. The auditors may need to have some appreciation of the records that they will be examining, but this is true of any new method. They did not feel that quality assurance personnel would need a full understanding of

the formal specification notation. They only need to satisfy themselves that the record has been produced and that the right sort of people have verified and authorised it.

Consultancy and Skills

Several companies had employed external consultants during the initial projects that introduced formal methods to the company. This was seen as necessary. Training was given to all staff involved in formal methods. Generally two weeks of training was found sufficient for staff to assist in formal methods projects. However, it was not thought feasible to train existing staff to a degree that they could successfully use formal methods without expert guidance on hand until they had built up some experience and practice. Not many experienced modellers are required as the majority of the project staff need to be able to comprehend specifications and write detailed sections as directed, but do not need to be able to create the overall structure of the specification.

One interviewee felt that external consultants, who are typically extremely intelligent, would make any project successful, no matter what method they used. This could give a biased view in favour of formal methods. Similarly, companies that use formal methods only recruit personnel who demonstrate the ability to use formal methods, thereby increasing the quality of their staff. Evidence of this was provided by another interviewee who reported that his company tended to recruit from research areas to fill vacancies involving formal specification. This filtering effect inherent in the adoption of formal methods could be seen as a beneficial effect on culture. However, there can be detrimental effects if, having altered the company's methods and culture, none of the permanent staff are sufficiently skilled to take over when consultants leave.

3.3.3 Impact on Product

Reliability

Only IBM and Praxis had any evidence of product improvement. IBM had found (based on informally collected data) a 40% reduction in post-delivery failures compared to their own average product performance. (This data is reported in previous publications by Phillips (1989) Collins, Nicholls and Sorenson (1991) and Houston and King (1991)). Praxis referred to published data, (Pfleeger and Hatton, 1997) which compares a Praxis software product favourably with industry average data. As with most case studies, the cause of this improvement cannot be identified with certainty to the use of formal methods, since other factors such as culture may be atypical, but it does provide a positive empirical indication of the possible benefits of formal methods. Of the other interviewees, Marconi's experience was based on a

study that did not go into service, and Philips and Interviewee A did not have personal knowledge of the relevant product service histories.

There was, however, an implicit assumption from the interviewees that the product would be more reliable. This was indicated by comments such as, "if you want software that works, then the only cost effective way to do it is with formal methods". This implies that formal methods produce a level of reliability that may only be achieved at significantly greater cost using conventional methods. This may be a subjective view but it is the view of those who have used both formal and conventional methods in software development.

Efficiency

Praxis had noted that the code produced from a formal specification was more efficient than conventionally specified software. The precise and accurate nature of the specification makes the coding task straightforward and the coder is less likely to build in redundant code. Note that this observation is supported in the findings of a comparative study by Brookes, Fitzgerald and Larson (1996).

Functionality Growth

Praxis also noted that the effort that is needed in formal specification tends to deter the functionality growth that afflicts many software systems.

Traceability and Maintenance

The interviewees were asked if the structure of the specifications is reflected in the code. Generally, the answer was affirmative and this was thought to be beneficial in aiding traceability between the specification and code. Some noted that this structuring of the code might not be the most efficient implementation but that the traceability benefit outweighed this. Philips questioned whether the specification should influence the structure of the code or not. One view is that the specification should not if it is at the right level of abstraction to be a requirements document. Another is that it would be beneficial if the specification could impose structuring requirements, for example, to improve reuse.

Two interviewees, Praxis and Philips, felt that the formal specification helped a maintainer to understand what changes were needed and therefore to get them right. Marconi felt that the specifications had little impact on maintenance but that the B-Toolkit helped a lot in automatically detecting affected components and re-checking them. IBM said that they do not normally use the documentation for maintenance, although, in one case, when they did and it

was a formal specification, the project leader estimated a 50% reduction in the cost of the maintenance.

The interviewees were asked if they thought that formal specifications help prevent degradation of code structure through maintenance and also whether the specification itself degrades through maintenance. IBM did not use or maintain the specifications after delivery and therefore could not answer. Interviewee A had not been involved in the product maintenance stages. Marconi felt that the B-Toolkit was largely responsible for preventing code degradation since it maintains the traceability from the specification. Philips thought that the formal specification would help prevent code degradation if traceability could be maintained but that this had been a problem (see comments under Lifecycle). Praxis thought that the formal specification prevented code degradation by supporting good practice (i.e. changing the specification first when implementing changes).

3.3.4 Impact on Development

Development Lifecycle

All agreed that there is no change to the sequence of activities performed during the software development lifecycle, but the effort involved in some of the stages is dramatically altered. The specification stages take a lot longer. However everyone agreed that generally the resolution of specification problems discovered during this stage was well worth the effort because these problems would otherwise have arisen later during the development with increased re-work consequences. Similarly, interviewee A believed the primary benefit of formal specifications to be the improved analysis of the problem domain that results from the process of writing them. This leads to a better understanding of the requirements prior to starting a design, which may be another reason for the reduction in problems occurring later in the lifecycle. Verification stages, particularly testing, were much reduced since far fewer errors remain to be discovered. The net effect was that the overall timescales were usually very similar or possibly better for the development that started with a formal specification.

However, Philips found that formal specification did not fit easily with the iterative lifecycle used for some products. Since Philips does not normally have an end-customer performing the requirements specification role, they have to develop the requirements themselves. Also, they typically have very short timescales to develop new products and often refine the requirements as the product is being developed. The time consuming first phase of formally specifying to resolve requirements issues does not fit into this type of lifecycle easily. In fact Philips had examples where the product was finished before they could complete the specification. To

address this, the company are looking at different levels of specification formality appropriate to different product lifecycles.

Formal specification was also found to aid the verification testing process. Marconi, Philips and Praxis all reported that testing was more efficient and more effective when a formal specification was available. This was the primary driving force for improving specification techniques, as far as Philips was concerned. From the formal specification, it is easy to derive test cases and some companies had gone as far as automating this process. Marconi had used B specifications to generate expected results automatically and Philips had generated test cases from statecharts automatically.

3.3.5 Size of system

A guide to the size of the systems developed using formal methods is shown in Table 3.2. The figures should be taken as a rough guide only due to possible variations in the measurement of a line of code and the programming languages used. However, they indicate that formal methods were used on systems typically in the region of 10s of Kloc. The interviewees were asked if large systems were a problem when using formal methods (compared with any other method). Answers varied somewhat but generally, the impression was that size is not a major obstacle any more than other methods. Marconi and Praxis indicated that proving becomes problematic with large systems and that the proof checkers and, to a lesser extent, model checkers may not scale up very well. For formal specification, though, IBM said that large systems are dealt with by breaking the system down into 'encapsulated' sub-components that could be dealt with separately. Marconi, using the B-Toolkit, felt that the system specification was difficult to cope with due to the fact that it could not be subdivided, but that as soon as the design was refined, the system naturally was divided into encapsulated sub-components. It appears that the concept of breaking down the system via encapsulation is crucial in dealing with industrial scale problems.

3.3.6 Comprehensibility

The interviewees did not feel that there were any significant understanding problems with formal notations (although some commented that this may be because they recruit people who will understand them). The notations were not seen as being a problem in this respect. In fact Praxis felt that formal specifications should be easier to understand than code.

Several interviewees said that it is essential to comment Z with English text to explain the structure of the model. This is not so necessary with B as it is more structured. Most companies

impose some styling (e.g. lexical) rules on top of the formal notation in order to improve the consistency of style throughout the organisation, although the general impression was that this was not a major factor in comprehensibility. Interviewee A had used a 'friendly' style of Z (a reduced subset avoiding the less intuitive constructs and annotation in a light style to enhance the friendly feel of the document) and felt that it had been beneficial to understanding for unpractised readers.

Only one specific feature that affects understanding was mentioned. Praxis had found that over-reliance on invariants can be confusing. It is sometimes better to explicitly state things that change during an operation rather than rely on implicit changes as a result of satisfying a state invariant, even if this is, strictly speaking, redundant.

The area that the interviewees did think was difficult was in creating the formal specifications. IBM and Praxis had both employed expert consultants to facilitate this stage. Marconi said that the most highly skilled or experienced people were needed to do the initial or higher level structuring, although others could then cope with adding in the detail. IBM said that the ability to create the right (i.e. useful) model requires the most skill and experience. It is too easy to create a model that is consistent but does not contain the abstractions that are useful in describing the problem.

3.3.7 Tools and Notations

The interviewees were not questioned specifically about tools but during the course of these discussions the B-Toolkit stood out as the only tool that had been used to any extent. IBM had started with Z but switched to B so that the B-Toolkit could be used. Marconi's entire experience was based around the B-Toolkit and they were very pleased with it in most respects. They relied on it heavily and found that it helped in tracing, proving and maintenance work. Praxis said that there are few industrial strength tools but agreed that the B-Toolkit is an exception. A Praxis interviewee thought that B was not as suitable as Z for the system level specification. However, Marconi has used B for all levels of specification.

Philips thought that tool availability has a big impact on the decision to use certain specification techniques. In particular, tool support to maintain traceability between specifications, implementation and test cases is an area of concern.

Interviewee A was in the process of installing the UML as a company wide documentation language. They were anticipating using formal specification in conjunction with the UML. Philips was also adopting the UML in some sectors of the company.

3.4 Conclusions

As this is a first stage, opinion gathering, exercise we are wary of drawing any firm conclusions. The results described above are considered indicators for further investigations. However, we summarise some of the main opinions recorded. Formal methods are worthwhile in terms of improved quality of software with little or no additional lifecycle costs, but only when compared to a rigorous development lifecycle where the cost of software errors is high. If the market does not demand high quality software then it is more difficult to justify their use. The introduction of formal methods affects a company's workforce, processes and culture through effects such as skills filtering and consultancy syndrome. It may also impact on the relationships with a customer through kudos, and communication implications. Overall the effects are usually beneficial but there can be some problems to overcome. There is no real problem with understanding specifications: given suitable training they are no more difficult to understand than programs. The difficult part is creating the specification as appropriate modelling requires practice and skill. Encapsulation is important within the context of large systems. There is a lack of industrial scale tools, the B-Toolkit being the only suitable tool.

Many interesting points have arisen from the structured interviews. We select two hypotheses for further investigation. The first is a comparatively straightforward hypothesis that is suitable for formal experimentation in a laboratory setting. The second is a more complicated issue and will require ingenuity in order to facilitate further empirical investigation.

3.4.1 Comprehensibility

One area that was expected to be rich with discussion was that of comprehensibility. It is often said that one of the problems with formal notations is that they are difficult to understand and that highly trained mathematicians are needed to read them. However, the interviewees did not support this view. This is significant because it conflicts with popular opinion: all the experienced interviewees agreed that typical software engineers have no real difficulties with understanding formal notations. As one interviewee put it, formal specifications are no more difficult to understand than code. In Chapter 4, we design and conduct an experiment to test this, by writing a specification using Z and implementing it in a programming language.

3.4.2 Modelling

The interviewees thought that the difficulties with using formal specifications were in finding the useful abstractions from which to create models. This is surprising, because the same engineers are practised at creating models of problems and solutions using less formal notations

as a transitory step in programming. The criteria for selecting a model on which to base a formal specification, may differ from that of less formal design, nevertheless one would expect similar skills to be applicable. One is led to suspect that there may be something lacking in the available notations and methods compared to informal program design methods.

Comparing the available formal specification methods with informal program design methods we find that program design methods concentrate on structure. Their aim is to provide the engineer with mechanisms for visualising the structure of problems from different viewpoints. Engineers are encouraged to explore the relationships between the entities in their models in order to try different abstractions before committing to them. The tools supporting program design methods are designed to enable them to build up an outline model of the problem in their mind. In contrast, if we look at formal methods, they concentrate on detailed behaviour rather than problem structure. This is what formal notations are designed to tackle, accurate precise detail. Tool support for formal methods has concentrated on verification rather than creation. Consequently, tool support for the initial process of exploratory design leading to the creation of a specification may be lacking compared with those available for informal notations. The engineer attempting a formal specification is faced with the need to make difficult and critical choices of model structure but has little support for such work. In chapter 5 we discuss these issues in more detail and compare the process of formal specification with that of program design.

Our hypothesis is that formal specification would be easier if an informal or semi-formal transitory modelling stage were performed, as is done in program design. Fraser, Kumar and Vaishnavi (1994) have described such transitory modelling stages and Bruel and France (1998) have investigated the use of UML as an aid to producing formal specifications. In Chapter 6 we present a formal notation that is based on a combination of UML and B, along with a prototype tool for converting the notation into the equivalent B specification so that verification and animation may be performed using the B-Toolkit. We assess the benefits that this method may bring to formal specification.

3.5 Summary

We have carried out a survey of the opinions of practitioners who use formal methods for software specification and development. The size of the sample is small (5 companies were visited) but covers a range of different market sectors including commercial computing systems, defence and avionics systems and consumer products. The interviewees are experienced experts in the use of formal methods in real systems. The results cover a wide range of issues including the impact on the company, its products and development processes as well as pragmatics such

as scalability, comprehensibility and tools. The survey is the first stage of an empirical assessment of the comprehension and creation of formal specifications. The remainder of this thesis focuses more narrowly on the two hypotheses that we have selected from the survey results:

- Hypothesis 1 - formal specifications are no more difficult to understand than code.
- Hypothesis 2 - a tool supported, graphical modelling notation would be of benefit in the process of writing a formal specification.

Chapter 4

Comprehensibility of Formal Specifications

It is a common perception that one of the problems with formal notations is that they are difficult to understand and that highly trained mathematicians are needed to read them. In Chapter 3 we surveyed the opinions of industrial experts and found that experienced formal methods users thought that typical software engineers have no real difficulties with understanding formal notations. As one interviewee put it, formal specifications are no more difficult to understand than code. This chapter describes the design and conduct of an experiment to test this by comparing subjects' comprehension of a Z specification with its implementation in Java. A close correspondence is maintained between the specification and the implementation, both in functionality and in structure. Subjects were given either the formal specification or the code and their understanding was tested using questionnaires. The results indicate that there is little if any difference in comprehensibility between the two.

4.1 Description of Experiment

The objective of the experiment was to investigate the theory that formal specifications are no more difficult to understand than code. Since comprehensibility is a complex attribute for which we have no absolute measures we need to test this theory by measuring comprehension between two examples that are comparable in some sense. Many attributes could affect this comparison such as size, structure and inherent problem complexity. In order to make the link as tangible as possible we chose to compare a Z specification with its implementation. We do not expect to use this result to conclude whether formal specifications should be used. There are many other factors requiring empirical assessment before a conclusion can be reached. However the comparison with implementation is attractive because the community of potential formal specification users is likely to have extensive experience of code maintenance and hence a 'good feel' for comprehension of code. Having a comparative measure for a specification couched in terms of the comprehensibility of its implementation will transfer this 'good feel' to the realm of formal specification. Therefore the theory can be re-phrased as "a Z specification is (at least) as understandable as its implementation". To investigate this a Z specification of an example

system was constructed. This was then implemented in the Java programming language. Subjects were asked to describe either the functionality represented by the specification or by the code. The mean level of understanding of each group (specification or code) was compared.

4.2 Design of Experiment

The Experiment was a one-way unrelated between-subjects design. This means that the treatments were applied to different sets of subjects and only one set of data (pertaining to one example treatment) was recorded. The Subjects were split into 2 equal sized groups by random distribution of the experimental materials. A two-way experiment (where 2 examples are used so that each subject attempts each of the treatment types) would have provided more statistical power but it was felt that doubling the effort involved would deter many of the volunteers. Another difficulty with 2 way experiments is that a second example is needed which is closely equivalent to the first but is also different enough to avoid significant learning effects. The subjects were given as much time as they required and were asked to record the time they had taken. (There was a 50 minute timetable slot, but all completed within this limit). They were then free to leave the room. It is hoped that this induced the subjects to work as efficiently as possible. The data are analysed below taking into account the time taken by each subject so that the effect of differing work rates can be accounted for.

4.3 Consideration of Influencing Attributes

The preparation of the materials used in the experiment may affect the experiment results. Hence, the author's experience and training is relevant when considering the influencing attributes described below. The author had been trained at postgraduate level in computer science including several courses on programming and programming languages. Postgraduate training included a small amount on formal specification. This was supplemented by a one-week course on formal specification using Z. The author had extensive experience (approx. 20 years) of programming in industry but virtually no experience of formal specification.

Comprehensibility is affected by structure (Finney, Fenton and Fedorec, 1999). The same system could be modelled in Z in many ways. Different specification structures could be adopted without changing the meaning of the model. Similarly the implementation could be structured in many ways and this might affect the comprehensibility of the implementation. To avoid the introduction of un-quantifiable influences on comprehensibility due to differing choices of structure, the specification and code were written with the same structure. There is a close correspondence between the schema and data entities in the Z specification and the

component modules in the Java code. This may mean that to some readers the Z specification, or Java code appears to be unnaturally structured.. Experienced formal methods academics and practitioners have commented that the Z is unusual and appears to be derived from the code. In fact the Z was written first and the Java was written to match its structure. The style of the Z may be influenced by the author's limited experience with writing formal specifications and considerable experience in writing programs. The question pertinent to this experiment is, how does the style of the Z specification affect the experiment results? It is possible that if the Z specification had been written differently understandability would be increased. In this case the experiment results would support the hypothesis even more strongly. On the other hand, if writing the Z specification differently decreases understandability then the experiment has been performed with a better style of Z specification. The effect of structure on the comprehensibility of Z specifications and Java code would be an interesting topic for subsequent work.

Similarly no commenting has been used in the Z specification or in the Java code. This is unnatural in both cases; one would not normally be expected to understand specification or code without a natural language explanation. However, if natural language commentary were provided in the experimental materials, the measure would no longer be of the comprehensibility of the notations. It would be severely and un-quantifiably influenced by the natural language descriptions.

4.4 Subjects

The 36 subjects were 2nd year computer science students who had been taught a course on formal methods and a similar length course on the Java programming language. The subjects were therefore familiar with the notations used, but were not very experienced. The experiment was voluntary, so there may be some self-selection effects, but since the allocation of either the Z specification or Java code was random and unknown to the subjects this should have no bias effect on the experiment.

One threat to validity may be that although the subjects have been taught to equivalent levels in these particular notations, they are likely to be more familiar with reading code in general than reading formal notations. This would bias the results in favour of understanding the Java code. Similarly the subjects' lecturers made several comments to the effect that the subjects did not like using formal methods. There may be a self-fulfilling lack of confidence in the subjects' abilities to read the Z specification leading to another bias towards the Java code.

4.5 Experimental Materials

A short specification was written in Z (Appendix B.1) to describe a road layout with vehicles moving along the roads and across the junctions. The specification was then implemented in Java (Appendix B.2). The Z specification was structured according to an abstract data type paradigm so that it was possible to maintain a close correspondence in terms of structure and allocation of functionality with the Java implementation. The Z specification and Java implementation are shown in the appendices.

4.6 Conduct

The subjects were allocated to one of the descriptions (Z or Java) at random. This was done by randomly distributing a set of envelopes (equal in number to that of the subjects) half containing Z specifications , the other half Java code. In order to ensure that the person marking the answer sheets did not introduce any bias, they were marked blind so that the marker was unaware to which representation (Z or Java) they related.

4.7 Data Collection Procedures

The subjects were given a questionnaire (Appendix B.3) to test their comprehension of the description they had been given. The questions asked were very open. The subjects were asked to describe the real-world objects and behaviour represented by the complete description and then asked what a particular named section of the description represented in real-world terms. The openness of the questions has the disadvantage that it allows a wider scope for interpretation by the subjects of what the required answer is. However, it was found to be impossible to construct more specific questions that would reflect comprehension without strongly suggesting the answer within the question. Additional background questions were asked in case such qualitative information might aid understanding of anomalous results. In the event, it was not necessary to use this additional information. Since the results consisted of an English language description of the system, we were concerned to ensure that the interpretation of the answers did not introduce experimental error. A marking sheet (Appendix B.4) was prepared which listed all the points that a subject might mention in describing the functionality of the system. A subject gained one mark for each point that was mentioned at some point in their answers. The marking sheet thus made the interpretation of answers as objective as possible. A summary of the marks awarded to each subject, along with a summary of their answers to the qualitative questions, is shown in Appendix B.5.

4.8 Analysis of Results

In this section we examine the experimental data set in order to see whether, and to what extent, it supports the hypothesis. An initial examination reveals that the data recorded for the Z specification closely matches that for the Java. In particular the means and medians of the data sets are very similar, however further statistical analysis is necessary in order make quantified statements of probability. First we look at the distribution of the data. This indicates that its adherence to a normal distribution is questionable. We therefore select a bootstrap analysis that is powerful but robust. (That is, it doesn't make any assumptions about the distribution of the data). Using the bootstrap analysis we obtain an outer limit for the difference in comprehensibility at a specific confidence level.

4.8.1 Variables

The independent variable is the notation (Z specification or Java code) used for the description. Two dependent variables are analysed. Firstly, the score which is an integer value ranging from 0 to 22 representing the number of marks gained as a measure of comprehension. Secondly the rate of scoring was found by dividing the score by the time taken. This was used as an alternative measure of comprehension.

4.8.2 Method of Analysis

Since our hypothesis is that there is no significant difference between the comprehensibility of a Z specification and that of its Java implementation, standard null hypothesis testing techniques are not suitable. Instead, we construct confidence intervals to quantify the mean difference for various confidence levels. Initially we constructed confidence intervals using parametric methods, which assume that the population distribution is a normal distribution. Examination of the sample data for score revealed that it is not obviously skewed, and roughly approximates a normal distribution, but this does not guarantee that the population distribution is normal. In fact the data is fundamentally non-normal because it is truncated at 0. We should therefore treat the parametric analysis with some mistrust. For the sample data for rate the distribution appears even less normal. Therefore, we construct confidence intervals based on non-parametric bootstrap methods, which make no assumptions about the underlying population distribution other than the sample data is representative of it.

		Z (marks)	Java (marks)	(Z-J)/J (%)
SCORE	mean	8.28	8.83	-6%
	median	7.50	8.00	-6%
	std.dev	3.37	3.90	-14%
RATE	mean	0.48	0.46	6%
	median	0.37	0.44	-15%
	std.dev	0.32	0.22	44%

Table 4.1 - Summary of Results

4.8.3 Examination of Data

The size of the data samples for the Z specification and the Java program were both 18. Each sample consisted of a score out of a maximum 22 marks and the time taken by the subject in minutes. A measure of the rate of scoring was obtained by dividing the score by the time taken.

An initial look at the medians, means and standard deviations (Table 4.1) of the data indicates that the Z and Java results appear to be very similar in both score, and rate of scoring. The most significant difference between the Z and Java results is in the standard deviation of the rate of scoring, which shows that the rate of scoring varies significantly more between subjects when reading a Z specification than when reading code. This is despite the fact that, when time is not taken into account, score varies less when reading a Z specification than when reading code.

We also examined histograms (using SPSS) showing the actual data and a superimposed normal distribution curve (Figs. 4.3 & 4.4). This showed a fairly good fit but with a slightly high proportion of readings around the mean, indicating a low standard error. For the rate of scoring data the histograms (Figs. 4.5 & 4.6) appear to be skewed towards the lower end indicating that this data is not a very good approximation to a normal distribution.

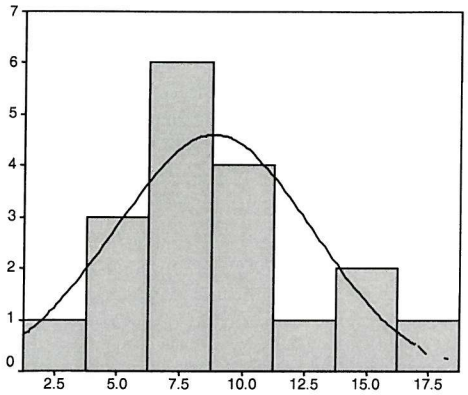


Fig. 4.3 - Histogram of Java scores

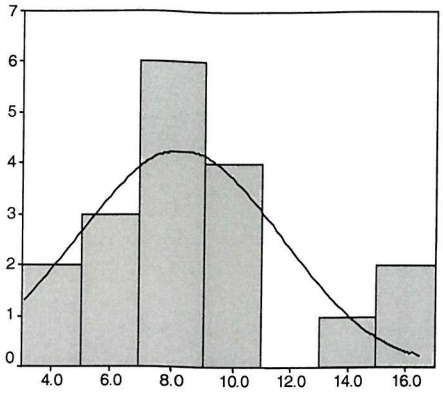


Fig. 4.4 - Histogram of Z scores

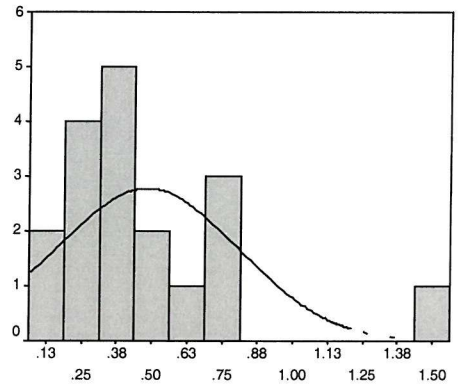


Fig. 4.5 - Histogram of Java rate of scoring

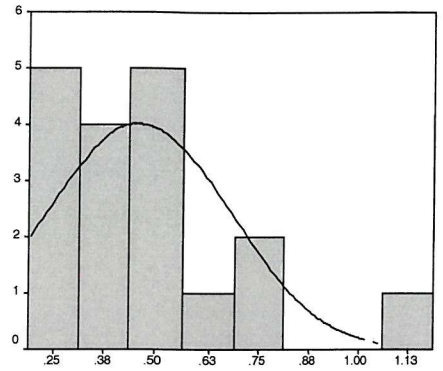


Fig. 4.6 - Histogram of Z rate of scoring

4.8.4 Bootstrap Confidence Intervals

We used the robust bootstrap analysis (Efron and Tibshirani, 1993) to construct confidence intervals. This uses the minimum possible assumption in any analysis based on a sample: that the data sample is representative of the real population. It does not make any assumptions about the nature (e.g. normality) of the real population distribution. Samples of the same size as the original sample are taken repeatedly from the sample data (it is permitted to select the same data point more than once within a sample). The statistic of interest is calculated for each sample and plotted to give a distribution that approximates its distribution in the real population. From this distribution a confidence interval can be deduced for any confidence level. We used MathSoft's S-PLUS 2000 (Professional Release 2) statistics package to perform the bootstrap calculations. Despite the robust nature of the bootstrap analysis, the confidence interval gives a 'better' (i.e. tighter margin at the same confidence level) answer than the traditional parametric confidence interval.

Score. The bootstrap results data output by S-PLUS is shown in Fig. 4.7. The bootstrap calculation for mean(java score)-mean(Z score) gives a difference in means of 2.22 at the 95%

confidence level (25% expressed as a percentage of the mean for the Java sample). Hence we have a 95% confidence that the overall population would have a mean Z score no worse than 75% of the Java score.

```
   *** Bootstrap Results ***
Call:
bootstrap(data = just.the.data,
   statistic = mean(jscore) - mean(zscore),
B = 20000, trace = F, assign.frame1 = F, save.indices = F)
Number of Replications: 20000
Summary Statistics:
      Observed      Bias    Mean     SE
Param   0.5556 -0.006478 0.5491 1.053
Empirical Percentiles:
      2.5%        5%       95%     97.5%
Param -1.5 -1.166667 2.277778 2.611111
BCa Percentiles:
          2.5%        5%       95%     97.5%
Param -1.555556 -1.222222 2.222222 2.611111
```

Fig. 4.7 Bootstrap Analysis Results from SPLUS for Score

The bootstrap density distribution of mean Java score – mean Z score for the 20,000 bootstrap resamples was obtained from Splus (Fig. 4.8).

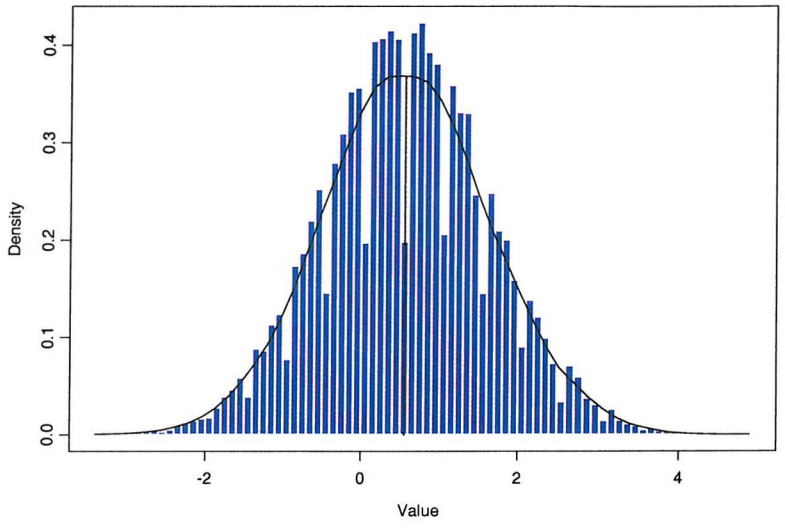


Fig. 4.8 Distribution of means of Java score – Z score for 20,000 resamples

Rate. The bootstrap results data output by S-PLUS is shown in Fig. 4.9. The bootstrap calculation for mean(java rate)-mean(Z rate) gives a difference in means of 0.082 at the 95% confidence level (18% expressed as a percentage of the mean for the Java sample). Hence we have a 95% confidence that the overall population would have a mean Z rate of score no worse than 82% of the Java rate of score.

```
   *** Bootstrap Results ***
Call:
bootstrap(data = just.the.data,
   statistic = mean(jrate) - mean(zrate),
   B = 20000, trace = F, assign.frame1 = F,
   save.indices = F)
Number of Replications: 20000
Summary Statistics:
      Observed       Bias      Mean       SE
Param -0.02692 0.0003885 -0.02653 0.07046
Empirical Percentiles:
            2.5%          5%         95%     97.5%
Param -0.1668326 -0.1430569 0.08754796 0.109355
BCa Percentiles:
            2.5%          5%         95%     97.5%
Param -0.1744356 -0.1489462 0.08211454 0.102708
```

Fig. 4.9 Bootstrap Analysis Results from SPLUS for Rate of Score

The bootstrap density distribution of mean Java rate of score – mean Z rate of score for the 20,000 bootstrap resamples was obtained from Splus (fig. 4.10).

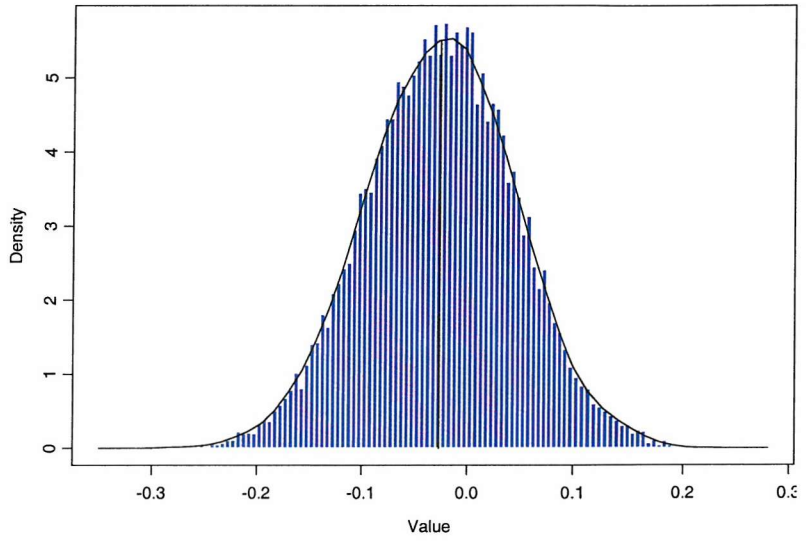


Fig. 4.10 Distribution of means of Java rate – Z rate for 20,000 resamples

In Summary, we have quantified the results in terms of confidence intervals for the usual 95% confidence level and found that we need to allow approximately a 25% margin, for score, and 18% margin for rate of scoring, to achieve this confidence (i.e. Z is within 25% as understandable as Java). Note that this does not mean that the data indicates that there *is* a 25% difference. (In fact, the data indicates that there is very little difference in comprehensibility).

4.8.5 Analysis of Qualitative Data

The questionnaire included some questions to collect some subjective, qualitative data. (See questions 3 to 7 of Appendix B.3). (One subject in the Z group did not complete these

questions). We are careful not to draw firm conclusions from this data due to inevitable variations in interpretation of both the questions and the answers. The following summarises the responses to these questions.

In question 3 the subjects were asked how difficult they thought the specification or program was to understand compared to an English language equivalent. The answers were almost all positive (i.e. harder to understand than English) and there was very little difference between the answers for the Z spec and for the Java program. The means of the answers (interpreting the answers on a scale from -5 to +5) were +2.35 (Z) and +2.39 (Java).

In question 4 the subjects were asked how difficult they found mathematical subjects (i.e. to judge their mathematical abilities compared to their peers). Here there was more of a tendency towards 'easy' indicating that most subjects thought they had an aptitude towards mathematics. This was slightly more so in the Z group than the Java group (-1.65 versus -0.31), which may indicate a mathematical bias in favour of the Z group.

In question 5 the subjects were asked for their mathematical qualifications. All but 5 of the subjects had mathematics A-level. Three of the five without A-level mathematics were in the Java group, 2 in the Z group. This indicates a uniform mathematical ability throughout the two groups.

In question 6 the subjects were asked how much experience they had with the notation or language used in the specification or program. The form of the answers varied slightly, some referring to length of time in months and others referring to course modules or semesters. However, all the answers apart from two in the Java group indicate that they only have experience of the notation/language from a course module in the previous year. Two answers from the Java group indicated a frequent use of Java leading to more of a familiarity.

In question 7, subjects were asked for any other comments. Many left this blank but of those that offered comments seven (all from the Z group) said that Z or formal specification is difficult or more difficult than code, whereas only 3 (from the Java group) said that Java or code is hard to understand. In fact 4 (again from the Java group) said that programs are easy to understand. Hence there appears to be a tendency to believe that formal specifications are more difficult to understand than code. This has not been borne out by the results of this experiment but may be a bias towards understanding the Java.

4.9 Threats to Validity

The degree of credibility of any study depends on its validity. We have already discussed 'Conclusion Validity', the validity of the statistical analysis. In this section we consider other threats to the validity of the experiment and its conclusions (Basili, Shull and Lanubile, 1999).

4.9.1 Internal Validity

Internal Validity defines the degree of confidence in a cause-effect relationship. Thus under this heading we must consider whether the subjects understanding of the specification and program could have been influenced by any factors other than the independent variable. There are 2 categories of factors that could be a threat here. The first category is attributes of the subject that might influence their understanding, such as ability or degree of training in relevant subjects. This was minimised by selecting the subjects from the same cohort of a course. There will still be differences in background and ability but the random allocation to groups should distribute such factors between the 2 groups. As with any sample method there is, however, always the chance that an unfortunate allocation has occurred. The second category is attributes of the materials other than the notational difference such as style. As discussed above, the structure, style, naming and font of the two descriptions were made consistent to eliminate these factors. A further threat to the internal validity was discovered after the experiment had been performed. The Java program had been tested in order to verify its correctness but the Z specification was only verified by inspection. Three errors were left undiscovered in the Z specification when it was used for the experiment. The errors are as follows:

1. The blank predicate part of the schema *VehicleType* should either contain *true*, or be omitted

2. The identifier *Destination already occupied*, used in the definition of *Report*, should contain underscores instead of spaces,

3. The schema *pickRoad* is incorrectly used as a function in the schemas $moveNewRoad_0$ and *destinationAlreadyOccupied.*

The first two errors are minor and unlikely to cause any misunderstanding or confusion to a reader. For these errors it is reasonable to assume that the subjects were able to easily identify the correction to the syntax if they noticed the error. The third error is much more significant since a correction is not easily identified even if the intended meaning is recognised. If the errors made it more difficult for the subjects to understand the Z specification the support for our hypothesis is strengthened. However, since the subjects did not comment on the errors, and

there does not appear to be a correlation between the errors and an area that was misunderstood, we assume that the subjects correctly deduced the intended meaning of the schemas. The limited experience of the subjects may have led to them assuming that there was no error, even if they did not recognise the syntax, and correctly guessing the meaning. The corrected version of the Z specification is shown in Appendix B.6. This version is written in an ASCII form of the Z notation, ZSL, and has been checked using the ZTC type checker (Jia, 1998).

4.9.2 External Validity

External Validity defines the extent to which the conclusions from the experimental context can be generalised to the context specified in the research hypotheses. Having established the experimental hypothesis we must consider how well it supports the substantive hypothesis. There are several threats to the inductive process needed to assess the substantive hypothesis. Firstly, the notations used in the example are particular whereas the substantive hypothesis is general in terms of notations. However, both Z and Java are typical and representative of the majority of other notations. We feel that practitioners will accept that if the hypothesis is true for these notations then it is, to some extent, generally true. There may be notations that deviate one way or the other. For example, Java is an object-oriented language and procedural languages may be easier to understand (although, in the experiment, we have not used many object-oriented concepts, such as inheritance, that are likely to affect understanding). However, similar experiments using alternative notations would clarify the generality in this respect.

Secondly, we must consider whether using students as subjects poses a threat to the validity of the experiment. The subjects were students who had undertaken an equivalent level of training in both notations. Lecturers reported that the students generally expressed a dislike of the formal notations. This is probably representative of the general population of practitioners in industry. We accept that students have less experience to rely on than practitioners. The extra experience of practitioners is likely to aid understanding of the program rather than the formal specification, but if our results reflect the situation without this bias in experience we view this as a desirable attribute. That is our results reflect the situation in the absence of a strong experiential bias as might be found in industry and therefore reflect the situation once an equivalent experience of formal specification has been obtained.

Thirdly, we should consider how the small size of the example problem affects the validity of the generality. This is a cause for concern, because the example problem is tiny compared with a real problem. Unfortunately it is impractical to use representative problems in this kind of experiment. We accept that scalability is an issue that could have a significant effect on the

results. The experimental results therefore reflect the situation in the absence of scalability issues, which require further investigation.

4.9.3 Construct Validity

Construct Validity defines the extent to which the variables successfully measure the theoretical constructs in the hypotheses. The theoretical construct in the hypothesis is comprehensibility. Under construct validity we must therefore consider whether the dependent variable and its measure are valid measures of comprehensibility. The measure consists of 2 stages: an analogy between comprehensibility and being able to describe the functionality of the represented system; and the validity of the scoring system used to measure the described functionality.

A threat to the first stage is that the subject may not have given a description that portrays their understanding. It seems reasonable to assume that the ability to describe something is proportional to the subject's understanding of it. This assumption is widespread in education via examination methods. The subject's written communication skills will affect their description as well as other factors such as their perception of what is relevant to the answer. However, these influences will not affect the validity of the results unless they affect one group significantly more than the other. We do not foresee any factors that could be influenced by the independent variable and hence might affect one group more than the other. (It may be that it is more difficult to describe the functionality of a program than a specification because of the difference in abstract level. However we consider this to be an essential part of what we are measuring rather than a source of bias. By 'comprehensibility' we mean ability to understand the functionality). The random assignment of subjects should therefore eliminate the effect of the ability-based factors, but as with any sample method there is always the chance that an unfortunate allocation (such as a disproportionately high number of more able subjects in one of the groups) has occurred.

The threat to the second stage is the method of scoring the written descriptions. The descriptions were marked according to a list of points (objects, properties or behaviour) and given one mark for each point mentioned. The answers were marked without knowledge of which group they belonged to so that no prejudice of the marker was introduced. Some points were easier to obtain than others and this means that the measure is non-linear affecting the scale validity. However, we feel that this will not be a significant problem as those who obtained harder marks generally obtained the easier marks. We considered weighting the points with differing amounts of marks but this would be a subjective judgement and in most cases it is not obvious what the weighting should be.

4.10 Possible Areas for Replication

Confidence in experimental results and further knowledge of influencing factors is gained by replication of experiments. Basili, Shull and Lanubile (1999) discuss a framework for organising related sets of experiments with the aim of building up a complete picture of the results over a wide range of contexts. (The term 'replication' is generally taken to include variations in the experimental work as well as strict replications). An experiment (or other empirical assessment) using practitioners with varying degrees of experience would be useful to establish that the results may be generalised to industrial situations. The area of scalability and an evaluation of its importance to formal specification compared with program design would illustrate its effects on comprehensibility. Further work on the effects of different styles and structures on comprehensibility would also be an interesting and valuable area to explore. Existing work in this area includes that of Finney, Fenton and Fedorec (1999), who conducted an experiment that concluded that the degree of schema structuring in a Z specification affects its comprehensibility, schemas of approximately 20 lines being optimal. Vinter (1998) conducted experiments that showed that subjects are likely to misinterpret certain forms of logical statements including disjunction, conjunction and quantification in the same way that people commonly misinterpret equivalent natural language descriptions. This implies that some forms will be more susceptible to misinterpretation than others, depending on context.

4.11 Summary

We set out with the intention of testing the substantive hypothesis that formal specifications are no more difficult to understand than code. Our experimental evidence strongly supports a hypothesis that subjects such as the ones we used could understand the Z version of the example approximately as well as the Java version of the same example. The data recorded for the Z specification closely matches that for the Java. The means for both score and rate of scoring were very close. The variance for score was also closely matched but there does appear to be a slightly higher variance in the times taken for the Z specification. This may be due to a wider variation in mathematical background, familiarity and confidence.

At the usual 95% confidence level we needed to allow a 25% margin for score and 18% margin for rate of scoring (i.e. Z is within 25% as understandable as Java).

We have chosen to adhere to the commonly used arbitrary confidence level of 95%. To give a guide to how the quantitative margin of the results would be improved by a looser choice of confidence level, we calculated alternative margins for the bootstrap result at the 80% and 75% levels. The corresponding results for scores were Z is within 18% and 14% as understandable as

Java respectively. The corresponding results for rate of scoring were Z is within 7% and 4% as understandable as Java respectively.

In the previous section we discussed various threats to the validity of the results and in particular, threats to the generalisation of the experiment needed to support the substantive hypothesis. There are some areas that would benefit from further investigation, however, subject to these reservations, we conclude that formal specifications are no more difficult to understand than code. Consequently, industry should expect similar levels of effort in reading and understanding formal specifications as they already experience in reading and understanding programs provided they allocate similar resources to the task.

The threats to validity illustrate the difficulties involved in performing empirical assessments involving human performance. In particular the consideration of construct validity illustrates some of the difficulties of finding suitable and valid measures of complex attributes associated with human behaviour such as comprehension.

Chapter 5

Why Writing Formal Specifications is Difficult

Perhaps the most powerful method we use for solving new problems is our ability to recognise similarities with, and differences from, our past experiences. We have the ability to recall situations, actions that were taken and resultant outcomes from our ever-increasing memory of past experiences. We are able to recognise similar instances and from this generalise to find desirable actions for classes of scenarios. Furthermore, we are able to recognise differences so that we can adapt these general strategies to new experiences.

Within computer science, as in other disciplines, such techniques are so basic and commonplace that they are used as a routine technique. For example new computer based solutions are invariably developed based on a collection of techniques learnt from previous projects. Experienced software engineers debug software by matching faulty behaviour with that of the past to lead them to probable causes. Working by similarity has been used in a more explicit manner by Brereton, Budgen and Hamilton (1998) when discussing the maintenance problems of hypertext.

In Chapter 3 we found that formal methods practitioners generally agree that writing formal specifications is difficult. In this chapter we make some suggestions as to why this might be so. First we outline a general definition of 'specifications' that is widely applicable to items at any stage in the programming process. Then we discuss the process of creating a formal specification and why it is difficult. We make some comparisons with writing procedural programs. Finally we use a cognitive dimensions analysis to assess B with respect to exploratory design. During this analysis we consider the design process and tools for formal specification in comparison with that of computer programming. From this comparison we identify one of the main differences between the two processes as the lack of equivalent design visualisation tools for formal specification.

5.1 Models, Specifications and Implementations

A specification is a description. This is a very broad and flexible definition and therefore encompasses many things. One kind of specification is a requirements specification where we describe things we desire to be true of a system. Another is a functional specification where we describe the actual behaviour of a system. It would be difficult to combine these views because we would need to maintain the distinction between things that are reported as fact and things that are stated as desired. Parnas (1997) defines specification to mean requirements descriptions, excluding 'actual' descriptions. Parnas warns that, unless explicitly stated, many descriptions could be interpreted as either requirements specifications or actual descriptions leading to confusion over an important distinction.

Different specifications, therefore, describe different viewpoints. Even within one viewpoint, specifications are rarely complete. A specification usually concentrates on one aspect such as functionality, or materials, or performance. We use many varied notations for specification because different notations allow us to express different views or aspects most effectively. Nuseibeh and Finkelstein (1992) recognise the importance of different viewpoints in their framework for the development of heterogeneous, composite systems.

One technique for describing things that is often used in specifications is modelling. A model is an object that resembles a 'target' object in some ways. A model is a way of describing the target object, so a model is a form of specification. According to FOLDOC, the free on-line dictionary of computing[2], a model is "A description of observed behaviour, simplified by ignoring certain details. Models allow complex systems to be understood and their behaviour predicted within the scope of the model, but may give incorrect descriptions and predictions for situations outside the realm of their intended use".

A model boat resembles the target object in shape and colour; perhaps also, to some extent, in its functionality if it floats, but in many other ways, such as size and materials, it does not. The 'reader' needs to understand the scope of the model in order to interpret it correctly. That is, the reader needs to know which attributes of the model are intended to describe the target and which are not. In the model boat example the reader is left to make their own judgement (based on common knowledge of the generic class of the object) on which attributes are similar in a real boat and which are not.

[2] http://foldoc.doc.ic.ac.uk/foldoc/index.html

Another example of a model is a Z specification. Here the representative attributes are the abstract mathematical state and behaviour information. The model may completely specify this attribute of the target, but it leaves many implementation options unspecified. The reader distinguishes the representative attribute as a convention of the notation. That is, the reader knows that with Z specifications, attributes such as the notation and the choice of mathematical structures is not representative of the target.

Wills and D'Souza (1997) are careful to point out that the attributes in their types (part of the Catalysis method which utilises the UML as notation) do not imply that the things represented by the types have any features with these names. "The only requirement is that the operations.... exhibit the behaviour implied by the model". Clearly, they do not feel that this is obvious enough from the modelling notation to go unmentioned.

We can even consider a natural language description to be a model. The representative attribute is the semantics within the text. This is a very flexible form of model, which is why natural language description is so popular and widespread.

Given this loose definition, all specifications are models; they are synonymous, interchangeable terms. FOLDOC describes a specification as "A document describing how some system should work". According to FOLDOC therefore the main difference between models and specifications is that models describe observed behaviour while specifications describe required behaviour. This distinction might be pertinent when we are *modelling* the observable behaviour of a system (which may or may not already exist) as a systems analysis stage prior to *specifying* the requirements for the implementation of a component of the system.

Despite this possible distinction, for our purposes, a model of an observable system is a specification of its behaviour and the specification of a component is a model of its responses to events. The term, model, highlights the concept of representative attributes while the term, specification, highlights the descriptive role, but they are names for the same thing and both may refer to required or actual behaviour of an observable system or component thereof.

We tend to treat computer programs as the target object of many of our specifications, but programs are not the final product. They are a description, in a notation (the programming language), of the operations that a machine will carry out. The program is a specification for the behaviour of the concrete machine, the computer. However we could view the computer as an imaginary object. If we know what the imaginary computer is like, we can deduce a behaviour that is represented in the program. In this sense the program is a model of a possible behaviour of the computer and is very similar to behaviour specifications written in formal notations such as B or Z. If we use a program to specify from this viewpoint (i.e. the actions of a computer) the

scope of the model is well defined but if we shift our viewpoint to required functionality, it becomes more difficult to distinguish required functionality from implementation decisions. However, similar problems arise in formal specification. There are many ways to write a Z specification (e.g. choice of schemas, choice of data structures) all of which are modelling details lying outside the attribute representation scope of the specification. We could agree similar conventions for distinguishing the scope when we use a program as a specification of required functionality. So perhaps, computer programs can be viewed as specifications in several senses, of the computers behaviour, but also of the required functionality.

Often the target of a specification is not a physical object but an abstract property or behaviour, which might be attached to a physical object. The level of abstraction away from concrete details can be varied providing a means for coping with scale. A highly abstract specification can first be produced to specify abstract properties of behaviour, which will be made more visible by not being obscured in detail. Further functional detail can be added in stages of refinement. Initially these refinements may add purely functional detail and remain requirements specification. Later refinements may introduce implementation decisions. Generally, as we move from requirements specification to implementation specification we also tend to move from declarative to imperative styles.

In some cases, a physical object is within the scope of the specification. In these cases we could consider the object to be the ultimate specification of itself. It is clearly accurate and complete but certain properties are not readily visible and may be difficult to measure. A design specification might be required in order to perform maintenance for example. So there are desirable properties of specifications other than accuracy and completeness. We generate specifications (models) because, even though they may be lacking in accuracy and completeness, they give us different views of the target object. In fact, in order to achieve this, to accentuate a particular view, we often deliberately suppress the accuracy or completeness of a specification so that it doesn't obscure the desired view. So the target object is a specification of itself but is not necessarily the ideal one, there are different ideal specifications for different roles.

By specification we mean any form of description of an object including the object itself. We appreciate that specifications can differ in form, accuracy and scope and different forms will be more suitable for different purposes, even if they lack accuracy and completeness. In order to avoid any confusion with preconceived ideas of specifications we use the term 'representation' to mean a specification in this extended sense. We take model to be an alternative word for specification (and hence representation) that has a different emphasis but refers to the same concepts.

5.2 Writing Formal Specifications

The process of computer programming can be viewed as a sequence of two or more representations, starting with an undocumented knowledge of a need for a computer to perform a task and ending with a program that enables a computer to perform a task that to some degree satisfies the initial representation. Hence, programming can be viewed as the generation of an alternative representation (the program) to an initial representation (the requirement). As described above, these generations may involve many steps of decreasing abstraction. The B method embodies this process via its concept of progressive verified refinements from formal specification through to program code. In fact the B method relies on making many small refinement and decomposition steps starting from a very abstract initial specification. Each refinement or decomposition introduces more specification details until a complete specification is achieved. Thereafter, further refinements and decompositions make implementation decisions until an implementation is reached. We are concerned with the difficulty of creating the first formal representations that make up a complete formal specification.

In their paper, 'Strategies for Incorporating Formal Specifications', Fraser, Kumar and Vaishnavi (1994) perform a morphological analysis to derive a framework for classifying strategies for using formal methods. Their classification is very simple, whether or not a semiformal, intermediate representation is used and whether or not computer assistance is used to generate the formal specification. One of the main reasons for analysing these strategies, they say, is because formal notations do not encourage exploration of the problem structure and this is detrimental to the resulting specification. They conclude that direct specification from an informal description into a formal notation without computer assistance is only practical for small well-structured or prototypical problems, and that iterative transitional (i.e. using a semiformal intermediate representation) strategies are needed for elicitation, problem structuring and validation of real-life problems. Further, to cope with the labour intensive generation of formal specifications, computer assistance provides most promise in addressing the problem of scale. Craigen, Gerhart, and Ralston (1995) carried out a survey of industrial applications of formal methods. After analysing the use of formal methods in a dozen industrial applications they observed that: "Industry will not abandon its practices, but it is willing to augment and enhance its practices." One of their recommendations was that research should concentrate on integrating formal techniques with software engineering practices, both in the area of assurance and in design methods.

Our survey of opinions of formal methods experts has led us to similar conclusions. When questioned about difficulties in understanding formal notations, these practitioners said that there were no fundamental difficulties; software engineers find that formal notations are no

more difficult to understand than code. Despite this, highly academic and talented consultants were generally employed to write the specifications. It was reported that the processes of creating a formal specification are extremely difficult and requires great skill.

The task of creating a model-based formal specification often starts from an informal, poorly structured and incomplete description of the problem. The next step is to choose and create abstractions that will be useful in the following step. (Here we use the term 'abstraction' to mean a grouping of elements that is to be treated as a single entity. Note, however, that often we need to choose abstractions before deciding the details of the elements they represent). The following step is to specify the detailed rules that govern the state, structure and behaviour of a model that represents a well-structured, complete and consistent specification. However, choosing appropriate abstractions is notoriously difficult and it seems that current formal notations are not conducive to exploring alternative abstractions before detailed behaviour is added. Green and Blackwell (1996) point out the "ironies of abstractions": that the difficulties involved in finding appropriate abstractions are similar to the difficulties they remove. Formal specification notations such as B and Z are 'abstraction hungry'. That is, they require the user to choose abstractions before they can be used. (Green and Blackwell describe abstraction hungry systems as those that "can only be used by deploying user-defined abstractions). The primitives in the Z notation are such that very little can be said without choosing variables that represent relationships between elements of state, operations that collate sets of elemental actions and groupings of these variables and operations to form further abstractions. Furthermore, in order to specify behaviour succinctly, a coherent collusion of abstractions must be built. This requires look-ahead, we need to predict what abstractions will be useful and what their interdependencies are. A collection of abstractions provides an ontology and, hence, the choice of abstractions changes the basis of reasoning. Therefore changing abstractions later will be difficult because the behaviour will need to be re-specified within the context of a different ontology.

The following example illustrates how the notation affects the choice of abstractions, how it determines the ontology and how it affects the visualisation and expression of certain relationships. The example models the movement of traffic on a road system using the Z notation.

[VEHICLES]

Road

traffic: seq VEHICLES dest: P Road

traffic is an abstraction that groups a sequence of vehicles. dest is an abstraction that gives a particular significance to a set of Roads. Road is an abstraction that captures and collates significant attributes of a road. Roads have destinations and associated traffic.

Unfortunately, this is not a valid Z specification because schemas cannot be self-referencing. The chosen abstractions are not suitable for expressing the relationship between a Road and its destination Road(s). We need higher-level abstractions to do this:

layout: Road → P Road

layout is an abstraction that captures the connectivity of the roads in a system. (dest has been removed from Road). Note that we prefer the total function from roads to (possibly empty) sets of roads, rather than a mapping from roads to roads. This is partly because it seems a more natural representation of the real world abstraction and also because we use this form in the U2B translator described in chapter 5. We can now add an event of a vehicle moving from one road to another. go is an abstraction that represents an event and comprises a precondition and some state changes defined by a postcondition.

go

from?,from?' : Road
to?, to?' : Road

---

to? ∈ layout(from?)

from?'.traffic = tail from?.traffic
to?'.traffic = to?.traffic⌢⟨head from?.traffic⟩

Alternatively, since we have had to remove dest from Road, maybe it would be better to elevate the abstraction traffic to the level of the road system, Roadsys:

[VEHICLES,ROADS]

Roadsys

layout: ROADS → P ROADS
traffic: ROADS → seq VEHICLES

go

ΔRoadsys
from?, to?: ROADS

---

to? ∈ layout(from?)

layout'=layout
traffic'(from?) = tail (traffic(from?))
traffic'(to?) = traffic(to?)⌢⟨head(traffic(from?))⟩

In this example, Road is a very simple object and its representation (in the first alternative) as an abstract data type schema is not worthwhile. However, generalising to more complex objects, which might have other attributes, initially it is not clear whether the encapsulation of traffic

within the abstract data type Road is better or worse than modelling traffic at the higher, Roadsys level. It is not until we start using these abstractions that we start to find out the effect of such decisions. The system has a different ontology; traffic has a different meaning since it now refers to all the queues of vehicles in the system, rather than just that on a specific road. ROADS is a basic type, whereas before, Road was a complex structure with attributes.

Moving from one road to another is constrained so that vehicles don't collide at junctions. We need some concept of a road being enabled, which is dependant on other roads not being enabled. We could add this to Roadsys thus.

Roadsys

enabled: ℙ ROADS depends: ROADS → ℙ ROADS layout: ROADS → ℙ ROADS traffic: ROADS → seq VEHICLES
∀ rr:Roads| rr ∈ enabled • depends(rr) ∩ enabled = ∅

The invariant ensures that the road cannot be enabled when a road it depends on is already enabled

However, the abstractions do not provide the concept of a junction within the ontology. If we need to introduce concepts related to a junction (perhaps closing a junction for maintenance of the traffic lights) it is difficult to envisage the effect from the depends abstraction.

As with any complex construction, formal specification involves the construction of multiple layers, as a description is structured into a hierarchy. This entails ordering abstractions, a difficult cognitive task. One way to find abstractions is to generalise instances, but this leads to a set of abstractions with low coherence (they may be good abstractions but they don't fit together well), another look ahead failure.

We might look to similar tasks with which we can draw parallels. Programming is a task that is very similar in nature to writing a formal specification. The Programming language is a formal notation. Programming is a similar task in terms of the level of detail and precision required in the process. In the early days of computers, a handful of enthusiast and specialist programmers hand wrote code, but only for simple well-structured problems. As the problems have grown in size and complexity, programming has become a widespread profession practised by well-trained but average graduates; it is no longer the province of specialist academics. Now, through visual interfaces, it is beginning to become available on a widespread level to the general public. Winograd (1995) describes these typical stages that new technologies go through; 'technology-driven' when the technology is used by enthusiasts, 'productivity-driven' when it is used by professionals and 'appeal-driven' when it is used by consumers. In order to achieve these

conflicting developments programmers have added more and more intermediate transitional stages into the design process. First Assemblers, then higher-level languages, then architectural design stages. Languages have become more natural for expressing the problem solution and program design paradigms have been developed to encourage better structuring of programs.

Formal specifications have not had the opportunity to develop in this way. Formal specification has lagged behind programming and only become of serious, widespread interest when the problems we want to solve with them are complex. While problems were simple, formal specification was not necessary. Formal specification has been used initially for safety critical systems and these have been kept simple for safety as well as practical reasons, but this has led to the view that formal specifications are not viable for other domains. Formal specification has suffered from a motivational lag. If the motivation to use them had been there in the early days of programming, methods to enable their effective use would have developed in pace with the scale of problems being solved.

Formal specification also suffers from its verification role. Structuring mechanisms for design purposes are often antagonistic to decomposition for proof purposes. For example Object Z usually has to be 'flattened' for manipulation. B contains significant restrictions to enable proof composition. For example, only one machine is allowed write access to the data of a shared machine. (Buchi and Back (1999) have suggested an amendment to B to allow write-shared machines). It is important to consider the purpose of a specification before selecting a notation (Hall, 1999). Design structuring mechanisms are important for an industrial scale task because they allow the problem to be decomposed into manageable parts and allocated to different teams or individuals. The structuring mechanism must allow the problem to be decomposed into natural coherent parts and must allow their interfaces and relationships to be understandable and manageable. We take the view that the first stage of transferring formal methods to industry is formal specification, and it is important not to significantly degrade design structuring for mathematical manipulation. A translation to a more suitable form for verification may be a later, possibly automated, stage. However, for pragmatic reasons, the techniques and tools we present in the next chapter restrict structuring such that both purposes are served.

Stepwise Refinement (Wirth, 1971) is an established technique for decomposing large systems into manageable sub-parts by hierarchical stages. The technique works well in developing a formal specification because a more detailed specification can be formally proven to be a refinement of a more abstract one. The decomposition at each stage is dependent on structuring mechanisms, which may be restricted as discussed above, but the introduction of detail in stages is, itself, beneficial. However, contractual requirements may dictate that complete and detailed requirements are expressed for customer agreement, and hence the contractual specification may

include several stages of refinement. Refinement may also be used to add implementation details. Refinements for implementation purposes would need to be kept separate from Refinements that are part of specification.

Formal methods can be used to verify the implementation against the specification and to prove properties of the specification such as its internal consistency. Usually, formal methods cannot be used to fully validate the specification. This is because validation involves the examination of the specified system to determine whether it is useful. The user's requirements for the system are usually informal and only partially recorded. Validation is when the user assesses whether the system will be useful in practice. This assessment can involve undeclared background knowledge, such as working practices, culture etc. Both Hayes and Jones (1989) and Fuchs (1992) agree that formal specifications improve validation at the specification stage. This is an important benefit because, otherwise, most validation is done on the implemented system, where changes are much more costly. Since validation inherently involves users who normally have no training in formal specifications, a barrier to validation is communicating the meaning of the specification. One method of overcoming this barrier is to translate the specification into a form that can be executed so that users can test the specification in specific scenarios. Hayes and Jones, argue that many of the techniques used to make a specification clear (such as inverses, negation and quantifiers) and non-determinism, which has an important role in avoiding over-constraining the implementation, are so hard to implement that doing so compromises other roles of the specification. Note that Hayes and Jones distinguish prototyping from specification validation. Prototyping is a method of discovering undeclared requirements for input into the specification, making validation more successful but not replacing it. Fuchs refutes the arguments of Hayes and Jones by demonstrating the translation of the same examples used by Hayes and Jones, into a declarative logic language. He succeeds in providing an executable version of each example that is similarly structured to the specification, at the same level of abstraction and does not introduce additional algorithmic details. For some examples limits have to be introduced where otherwise the computation would be infinite. Gravell and Henderson (1996) discuss, amongst others, Hayes and Jones and Fuchs work and conclude that although clarity, expressiveness and abstraction level must be given priority to enable inspection and review, executable translations of specifications are often achievable and provide a cost effective means of detecting some kinds of errors. The B-Toolkit includes an animation facility that is useful for validating B specifications by execution. However we have found that some specification constructs, such as set construction are not successfully handled. Leuschel and Butler (2002) have proposed and implemented an alternative animation and model-checking facility for B that is based on automatic translation into Prolog.

Since Formal specification is a similar kind of task to programming, it is reasonable to assume that similar stages will be necessary to create formal specifications for real-life problems using average engineering skills. Formal mathematical notations based on set theory have the advantage that properties can be expressed extremely simply and succinctly compared to a programming language. Even so, methods for organising these expressions and composing them into a meaningful and manageable specification are crucial. Already attempts have been made to develop more useable formal notations. The Z notation has a simple but effective composition mechanism in its notion of schema. However, schemas do not provide full encapsulation. A collection of schemas is necessary to cover state, initialisation and operations of a subcomponent. Also, promotion and binding mechanisms used for composing schema into higher levels, although mathematically simply and powerful, are not intuitive from the system designers perspective. Students often find these concepts difficult to grasp. Object-Z and B add more sophisticated building mechanisms that improve encapsulation, albeit with disadvantages discussed above. As notations develop, some researchers are beginning to investigate the need for transitional hierarchical design stages, as noted by Fraser, Kumar and Vaishnavi (1994). Other references to such examples include Facon, Laleau & Nguyen (1996), Bruel and France (1998) and Meyer & Souquières (1999). Here, most attempts actually adapt the program design methods directly. In Chapter 6 we discuss a translation that we have developed using the UML as a transitional stage with computer assistance to generate B specifications.

While we have been arguing that there are similarities between formal specification and programming we recognise that there are significant and fundamental differences. Often, when writing a specification our aim is to describe requirements or observable behaviour rather than specify an implementation. That is we are describing what happens rather than how it should be achieved. This implies different aims, levels of abstraction and techniques. A common difference is that most formal specification notations are declarative whereas procedural programming notations are imperative. Declarative notations are good for specification because they enforce a description of what happens to state when an event occurs without allowing a description of how it is achieved. However the removal of the facility to decompose behaviour into sequential stages is a descriptive limitation that is unfamiliar to programmers.

In comparing formal specification and programming we are considering imperative, procedural programming languages because they are usually used for implementing systems. Declarative languages such as the logic language, Prolog, (Sterling and Shapiro 1986) have more similarities with formal specifications.

The following summarises the main differences between set-based formal specification and procedural programming.

Purpose - The aim of formal specification is usually to describe something whereas the aim of programming is to implement something. This can lead to different aims and priorities.

Process - Program design has received a lot of attention over years of development. Tools and techniques have been developed to a greater extent than those for formal specification.

Abstractness - Programs are fixed at the concrete implementation level by the machine they are instructing, whereas formal specifications can be pitched at any desired level of abstraction.

Declarative - Formal specifications are usually declarative whereas procedural programs are imperative. Programmers are used to decomposing problems into a sequence of steps rather than a conjunction of truths.

Animation - While animation of formal specifications is possible, current tools to support this are not entirely satisfactory and hence animation is not widely used. This makes validation difficult. In contrast, programs are executable by purpose.

Mathematical - Formal specifications are mathematically manipulable enabling reasoning and formal verification to be carried out.

5.3 Cognitive Dimensions of B

In this section we perform a cognitive dimensions analysis of the B notation with respect to exploratory design. Exploratory design is the process that is undertaken to create a formal specification. The 14 dimensions that were introduced in chapter 2 are ordered according to our rough subjective ordering of their importance in exploratory design. We assess the B notation as an example of a formal notation and attempt under each verdict to generalise to indicate whether the dimension contributes to making the process of formal specification difficult. We also consider how each dimension affects program design and how program design tools are used to alleviate the problems. We selected the B notation because the analysis will be useful in supporting chapter 6. We view the B notation as being one of the more practical formal notations because it has good structuring and encapsulation mechanisms and good tool support.

5.3.1 Abstraction

FOLDOC defines Abstraction as "Generalisation; ignoring or hiding details to capture some kind of commonality between different instances". An abstraction gives a new meaning or role to an object or group of objects and allows the group to be referred to by a new name. Formal notations are very abstraction hungry. This means that they require you to invent abstractions at

an early stage. In B, abstractions are created by naming sets, defining types, variables, definitions and abstract machines. You cannot say anything at all in B without choosing abstractions. This is to be expected because B is a modelling language and is intended to be used to describe things by assigning roles to the mathematical constructs of set theory. For Exploratory design, abstraction hunger is a double-edged sword. On the one hand abstractions enable you to create a higher-level problem specific language; they determine the ontology of the problem domain. Once the abstractions have been made the problem can be expressed very clearly and important properties will be made visible. On the other hand choosing appropriate abstractions that will fit together in a coherent way is extremely difficult. Abstraction hunger is a property of any general purpose modelling language and B is not particularly beneficial or deficient in this respect compared to similar notations, however, we identify abstraction hunger as one of the main, inherent, difficulties in formal specification. Programming languages involve similar levels of abstraction hunger. In most large-scale program design, some form of design support is used. This normally includes a guideline or method for choosing abstractions and a drawing format for representing their relationships. Often several drawing formats are involved, giving different viewpoints of the relationships between abstractions (e.g. data dependencies, invocation sequences, functional hierarchy). We will refer to this support as 'Program design tools', although in some cases the tool consists of nothing more than an instruction on how to employ the method. For example, in the 1980's the UK Ministry of Defence required suppliers of real-time computer systems to document their software designs using the standard, JSP188 (Ministry of Defence, 1980). No, particular drawing tool was mandated, but the standard defined a framework for decomposing the software first into 'facilities', then 'tasks' and finally into 'modules'. It also defined the types of diagrams that would provide a visualisation of the relationships amongst these components (functional decomposition, component decomposition, data flow, and control flow diagrams). The 'MASCOT' method for software design (Simpson, 1986) was developed to comply with JSP188. Later in the 1980s structured design methods such as that proposed by Ward and Mellor (1985) and 'Jackson Systems Development' (JSD) proposed by Jackson (1983) were widely advocated. Software packages supporting these methods with drawing tools that encompassed and enforced their rules were available. During the 1990's, object-oriented programming became popular and introduced more kinds of relationships (and consequently views). Tool support became more necessary and tool vendors were more successful than the structured design ones. Three main variants of the object-oriented methods emerged and were competing for popularity (and tool sales). Eventually, a need for unification of the object-oriented method variants was recognised. This resulted in the 'Unified Modelling Language' (UML), introduced in chapter 2. Unification benefited both the tool vendors and the companies

developing software. Each tool vendor had a larger customer base and the software design companies no longer had to risk committing to a particular vendors method.

Program design tools assist in making abstractions by providing a visualisation of them to assist the designer in assessing them. In particular this visualisation assists in assessing the coherence and coupling of the abstractions by making clear their interactions. Formal methods tools enable properties of a completed specification to be analysed and verified. In, comparison, program tools provide very little real assistance in analysing the completed model. However it is their support for the subjective assessment of the emerging model in the early stages of its creation that makes them attractive for this exploratory design stage.

5.3.2 Premature Commitment

Premature commitment is when decisions must be made (and committed to) without fully knowing how those decisions will affect later work. The very nature of exploratory design implies a lack of knowledge about how the later features of the design will turn out. The less commitments need to be made the better. With respect to writing formal specifications this dimension goes hand in hand with abstraction hunger. The main premature commitment that needs to be made is to the abstractions used in the specification and we have already noted that formal specification requires these at an early stage. We see the premature commitment to abstractions as the main difficulty in writing formal specifications. Again, programming involves similar levels of commitment to abstractions such as data structures and modularisation before detailed coding, but program design tools have been developed which allow the designer to visualise and explore different structures before making that commitment. This process allows the designer to make better predictions about which structures are likely to be more successful when the detailed code is added.

5.3.3 Viscosity

Viscosity is the amount of effort needed to make significant (i.e. structural) changes to a completed or partially completed description. This is very important in exploratory design because the nature of exploration makes it virtually certain that a significant amount of re-arrangement will be needed as the true nature of the specification and the best way to express it unfold. Formal specifications are highly viscous. The detailed mathematical notation requires a significant investment and any structural re-arrangement is likely to require extensive and careful revisions. However even more significant than this is the effort to revise the abstractions that the notation was so hungry for and made us prematurely commit too. These abstractions provide the very ontology of the specification and making revisions will entail revising both

structural elements and mathematical details. This is the third side of a vicious triangle of dimensions: Abstraction hunger, premature commitment and viscosity. Together they account for the main difficulties with writing formal specifications. Yet again programs are similarly viscous. To alter data structures or modularisation in any significant way usually involves significant effort in recoding. Program design tools have been developed which reduce viscosity by allowing the designer to change the structure with graphical drawing tools. The aim is to obtain a successful architecture before committing to code, but if automatic code generators are used the viscosity reduction is extended. Small alterations to a program can often be accommodated fairly easily without substantial changes to the architecture, but as the number of alterations increases the suitability of the architecture gradually decreases until a 're-factoring' is needed in order to create a new architecture that better supports the changed functionality. At this point the viscosity of program architecture is a substantial overhead to the required change. If a program design tool with automatic code generation is used, effort is saved because the infrastructure code associated with the structure is automatically produced. This is a desirable route that we would like to adopt in formal specification. A second less desirable outcome often occurs when such tools are not used. The structure is not changed because of the viscosity. Instead re-factoring is avoided and the detail code is made to work within the unsuitable structure.

5.3.4 Progressive Evaluation

During exploratory design it is important for the designer to be able to check and review work performed so far at regular intervals. This is part of the feedback required for exploration. The B method provides two mechanisms that can be used for progressive evaluation. Abstract machines provide an encapsulation mechanism, allowing component parts to be independently analysed, animated and proved within the B-Toolkit. This method imposes a certain ordering on the evaluation since lower level components need to be checked prior to use in the evaluation of higher-level components. The order may be counter to the natural order of exploration. It may be beneficial to design a specification at an abstract level, evaluate this, and then add further detail in a series of levels. This concept of refinement is central to the B method, where an initial abstract specification is written and verified before further detail is added in the form of a refinement that is verified to comply with the more abstract version. The specification can be built up in levels until the specification becomes the implementation. This is especially obvious in the B method but similar concepts apply in other formal notations (less so the component encapsulation). We conclude that progressive evaluation is catered for quite well in formal specification methods, however we also note that verification proofs are recognised to be difficult. Since this is the primary means of evaluation, progressive evaluation may still be a

barrier to formal specification. Furthermore, most of the verification is aimed at verifying the internal consistency of the specification; there is still the question of whether the specification is the right one (validation). Animators test the specification from this point of view. Unfortunately, writing successful animation tools is difficult. The animator in the B-Toolkit becomes unusable if some kinds of set constructions are used in a specification. In program design, the situation is very similar although the methods of verification (such as reviewing) are usually not rigorous. Modularisation is achieved according to the design paradigm and program design tools are used to achieve layered design stages. Testing by dynamic execution of the code (the equivalent of animation) is used for both verification and validation. Although testing is rarely exhaustive, if performed incrementally as the program is developed, it normally provides good feedback.

5.3.5 Closeness of Mapping

A close mapping between the elements of the notation and entities in the problem domain makes exploratory design much easier because less effort is expended describing the problem domain entities allowing more effort to go into describing their behaviour. The B notation is a general modelling notation and therefore its elements are more abstract than the problem domain. However, if abstractions have been chosen appropriately, a new set of elements is created that have a close mapping with the problem domain. Therefore, considering that we require a general-purpose (rather than domain-specific) notation, we do not view B as being deficient with respect to closeness of mapping. However, from the point of view of discovering why formal specification is difficult, the lack of closeness of mapping in our notations causes difficulties unless we make abstractions, and, as noted above, finding appropriate abstractions is difficult. The situation in program design is, once again, very similar. In some areas domain specific languages have been developed (e.g. control algorithm languages used in avionics control systems software) which improve closeness of mapping to such an extent that programming becomes relatively easy and error free. Where more flexible, general-purpose languages are needed a compromise solution is achieved by using the design paradigm of the language to create an ontology from abstractions for each sub-domain. For example, in an object-oriented paradigm, classes are used to model entities in the problem domain and methods represent the behaviour of those entities in response to events. The class's methods are equivalent to the constructs in a domain specific language.

5.3.6 Hard Mental Operations

Hard mental operations affect the designer's ability to express semantics in a specification. In general, formal notations (mathematics) have a relatively high incidence of expressions that most people find difficult to cope with and this dimension is probably responsible for the majority of the prejudices against them. However, in most cases, with practice, these expressions become more accessible, indicating that the dimension is a less significant obstacle to creation for experienced users. The B notation, although a declarative formal notation based on set theory and similar to Z, is expressed in a form that resembles program statements and organisation. The post condition is expressed as a set of changes to the state variables accessible to the operation. Like a program, but unlike Z, any variables not mentioned are assumed to be unchanged and assignment is used to express the changes. There is still no sequential composition (at the specification level) but this is made explicit and more accessible by a 'simultaneous' operator instead of relying on conjunction as Z does. We conclude that this dimension may be a moderate obstacle to formal specification but B mitigates this by using a form that helps the designer envisage what is being expressed. We think this mitigation will be especially important for novices, although there is a danger that they will misinterpret the notation as imperative.

5.3.7 Visibility and Juxtaposability

Visibility is the ability to view component parts of a description easily. Juxtaposability is the ability to view several components side by side. For example, juxtaposability is important when two components are being compared or when information is needed about a component when another is being developed or altered. The B-Toolkit allows several abstract machines to be displayed on screen in separate windows so that this can be achieved. Initially it might be assumed that this is sufficient for a textual notation and that this dimension does not cause a problem in writing formal specifications. However, Craigen, Gerhart and Ralston (1995) reported that one of the tools that the commercial sector (as opposed to the regulatory-governed sector, who are more interested in formal verification) desired most was specification navigation tools such as browsers and cross-referencing tools. Visibility issues should not be underestimated and in program design perhaps one of the biggest driving factors for using graphical design tools is the visibility they provide through multiple views of the emerging design. We therefore conclude that formal specification suffers quite badly with respect to visibility through a lack of such tools compared with program design and integrated development environments (IDEs).

5.3.8 Hidden Dependencies

A hidden dependency is a dependent relationship between two components where the dependency is not clearly visible. Hidden dependencies tend to fall into two categories. In one-way dependencies, the relationship is only visible from one of the end components of the relationship. In local dependencies an overall relationship can only be deduced by traversing many local relationships. Hidden dependencies affect exploratory design because they increase viscosity (i.e. they are difficult to find when a major change is needed). For example, in Z, invariants expressed in the state part are assumed to hold in operations. This means that some state changes that take place when an operation occurs might not be stated explicitly in the operation schema. Some users avoid this hidden dependency by stating the operation post conditions even if they are redundant (see Chapter 3). B does not suffer from this hidden dependency. In B, the invariant is a property that must be proven to hold throughout all operation events, it is not assumed to hold and 'supplement' the operation semantics. Hence operations must explicitly state all changes to state variables including those that are necessary to maintain the invariant. While the negative effects of hidden dependencies have been identified and addressed within the programming community, most general purpose programming languages and practices still allow the programmer to create hidden dependencies via global data accesses. Hence depending on discipline, conventions and culture within the organisation, programs may be worse than formal specifications with respect to this dimension. We conclude that formal notations, and B especially, score highly with respect to this dimension.

5.3.9 Error-Proneness

Error-Proneness is the tendency to make minor slips (rather than errors of design judgement) in the notation. This would hamper exploratory design. There may be a tendency, especially in novices, to make errors in the mathematical expressions. However, we do not view this as a major contributor to problems with formal specification once some practice has been gained. It would seem no worse, and perhaps easier, than writing programs. We note that another dimension, progressive evaluation, is important in mitigating the effect of Error-Proneness. The ability to detect and correct errors at a unit level is of great benefit in developing the overall specification. This has similarities in programming where modules are individually compiled and tested so that the problem of error detection and correction is manageable. The importance of tools for progressive evaluation (and hence in mitigating error-proneness) has been discussed.

5.3.10 Consistency

A notation is consistent if similar semantics are expressed in similar forms. For example, if the syntax for expressing conjunction were different in an invariant and an operation, this would be inconsistent. Consistency is beneficial for exploratory design because it reduces the number of syntactic rules that need to be remembered when writing a specification. We know of no inconsistencies in B. Typically formal notations, by their mathematical nature are highly consistent. We conclude that this dimension is not a reason for the difficulty in writing formal specifications.

5.3.11 Diffuseness/Terseness

Diffuseness is the verbosity of a notation. Terseness is the opposite of diffuseness. Verbose notations tend to slow thinking performance. Terseness is beneficial for exploratory design because it reduces the time taken to express properties in the notation but can also increase error-proneness. The negative effect of terseness on comprehension is not likely to be apparent during the design stage since the design team will recall what they have expressed. B, like most formal mathematical languages based on set theory, tends to be terse. We conclude that this dimension is not a reason for the difficulty in writing formal specifications.

5.3.12 Role-Expressiveness

Role-expressiveness is the degree to which it is obvious what each component of the specification or program is for. This is more relevant to comprehension than design. The designer will generally appreciate the role of each element, being the one who selected it. We conclude that this dimension has very little bearing on design. However, we note that the genericness of the constructs and notation in B detracts from its role-expressiveness. It is not apparent what role each machine, operation or data structure plays in the specification without deducing the behaviour of each component. This can be overcome if secondary notation such as comments and well-chosen names are provided.

5.3.13 Secondary Notation

Secondary notation (i.e. information conveyed outside of the formal syntax of the notation) can convey extra information, such as the grouping and role of related statements. Secondary information can be of two types. It may be 'redundant' if it is already present in the formal syntax (e.g. indentation of code) or may be additional information provided by an 'escape from formalism' (e.g. commenting). It is common practice to add secondary information to programs

in the form of comments and indentation. Similarly, it is seen as an essential part of writing formal specifications in Z, to intersperse each schema with a paragraph of natural language to describe its role and explain how the mathematics models the real problem. The natural language description is so integral to a Z specification that the resulting document can be seen as a description in two complimentary notations, rather than a formal specification with supporting comments. This resembles the literate programming ideas of Knuth (1984).

Similarly to role-expressiveness and for the same reasons, secondary notations are more important for comprehension than for design. We conclude that this dimension has only a minor bearing on design, but note that B has facilities for adding secondary information. For example, comments can be embedded, B statements can be indented or grouped and capitalisation conventions may be employed to aid comprehension.

5.4 Summary

In summary we see the main problems in writing a formal specification as being the requirement to commit to abstractions at an early stage and the difficulty of subsequently altering these abstractions. Abstractions are needed to achieve a suitable closeness of mapping. The B notation is typical in this respect. Progressive evaluation is difficult in formal specification even though it is generally catered for. Improved animators would address this. Visibility is not adequately addressed. Formal specification notations often involve hard mental operations, although B is better than many in this respect. Formal notations tend to tackle hidden dependencies, error-proneness and consistency fairly well, so that these dimensions are not problematic and their terseness is, if anything a benefit during design. Role-expressiveness and secondary notation are of little relevance during design.

Considering that program design suffers from similar problems leads us to the hypothesis that the solutions adopted from program design would similarly benefit formal specification. A graphical design, transitional, stage would provide better visibility of abstractions and how they interact to compose the whole and this would be of value when assessing abstractions thereby alleviating premature commitment. The tool would also lower viscosity by automatically providing the infrastructure of a formal notation version. Fig. 5.1 represents the relationships between the main problematic dimensions for formal specification and illustrates where a graphical design tool would alleviate these problems.

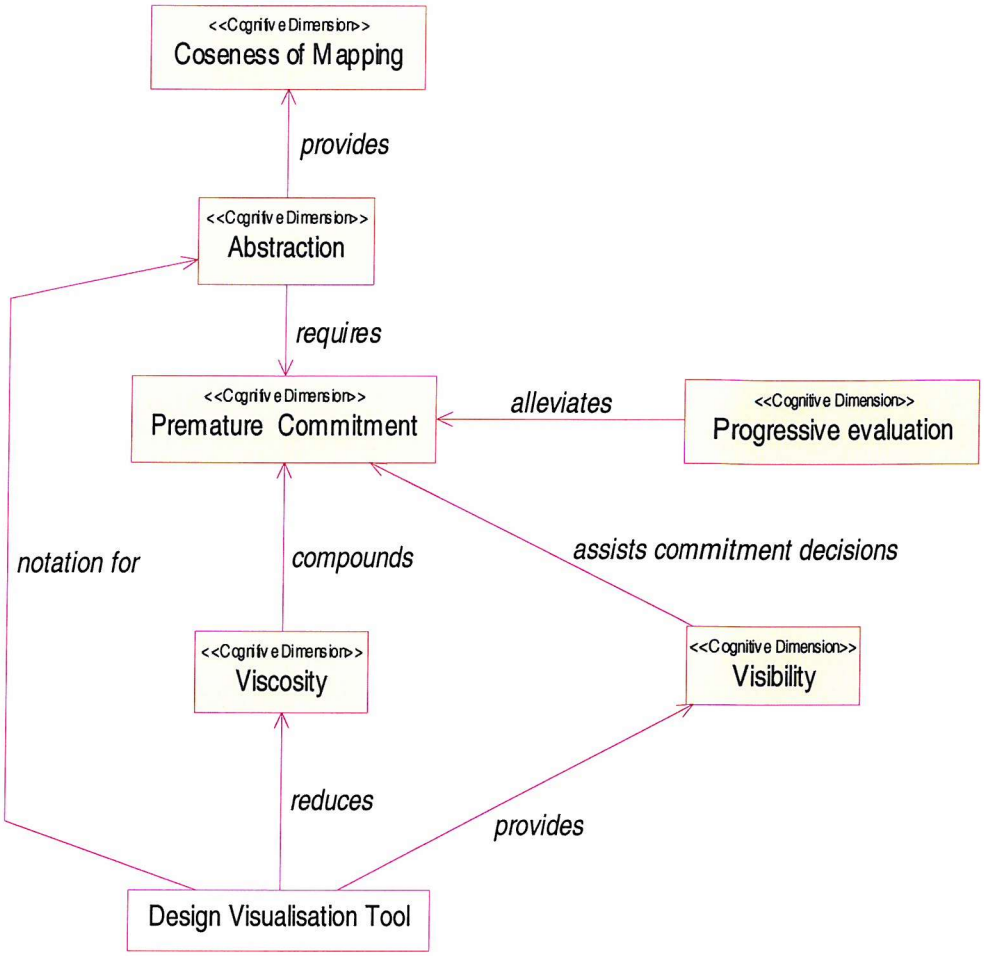


Fig.5.1 relationships between cognitive dimensions that affect formal specification and how a design visualisation tool would alleviate these problems

Chapter 6

B-UML and U2B: Adapting the UML for Formal Specification

In Chapter 5 we discussed specification and why we think formal specification is so difficult. We analysed the process of writing a formal specification using the B notation as an example and contrasted it with the process of writing a computer program, which is, itself, a kind of formal specification. We established that the difficulties are very similar in computer programming but that design tools such as the UML alleviate these difficulties. We believe that graphical modelling tools similar to those used for program design would aid the process of formal specification.

With this in mind we have used diagrammatic notations of the UML for formal specification. To support this we have developed a prototype tool to convert adapted forms of UML class diagrams and statecharts into specifications in the B language. The aim is to use some of the features of UML diagrams to make the process of writing formal specifications easier, or at least more approachable to average programmers. We view this work as a feasibility investigation rather than a final method or product. The translation relies on precise expression of additional behavioural constraints in the specification of class diagram components and in statecharts attached to the classes. These constraints are described in an adapted form of the B 'abstract machine notation'. The type of class diagrams that can be converted is restricted in order to comply with constraints of the B-method without making the B unnatural. The resulting UML model is a precise formal specification but in a form which is more friendly to average programmers, especially if they use the same UML notation for their program design work. The diagrammatic notation and tool support brings its benefits to the modelling process for formal specification. The translation to textual B specification does not add anything to the specification; it merely provides an alternative mathematical, textual form. In this textual form, however, the benefits of the B method are obtained. The translation also demonstrates the validity of the graphical forms and defines their semantics. We envisage benefits to B users (especially novices) from being able to develop models in the UML diagrammatic form and we see this as a possible way to overcome some of the psychological barriers that programmers have against formal specification.

6.1 Benefits of a Diagrammatic Form for Specification

The majority of students on computer science courses express an aversion to formal specification whereas they are quite comfortable using graphical program design notations such as the UML[3]. We believe that this is largely an unwarranted fear and that formal specification, given the same level of tool and language support should be no more difficult than programming. Advantages of graphical design aids are more to do with the creation of models than with conveying information. Graphical descriptions can be misleading to read, they often convey different meanings to different readers and require experience to interpret secondary features (Petre, 1995) but to the writer they provide a quick way to express their ideas and to assist in visualizing prototype models that must otherwise be built entirely within the mind. Textual representations, although often more accurate in conveying precise meanings, are much more cumbersome for creating some aspects of these models. Graphical representations are good for helping to visualize structures, composition and the relationships between elements. Modelling large systems usually requires initially a structural design, which is then populated with more precise semantic detail. It is this first modelling stage that benefits from program design tools such as UML. Class diagrams allow the types of objects in the problem domain and the relationships between them to be modelled, visualized, prototyped and altered quickly. Attempts to add the semantic detail to these models may result in deficiencies in the model being discovered and lead to refinements to the model. These changes can be made quickly because the model is highly visible and easily alterable with the aid of the graphical design tools. Readability and ambiguity is not an issue because it is the creators that are using the tools for modelling. These features have made graphical design techniques such as UML popular for developing programs. We contend that the process of writing formal specifications is in many ways similar to programming and involves similar difficulties in abstraction, look-ahead and viscosity. Therefore tools that programmers have evolved for writing programs, or ones very similar to them, should bring similar benefits when writing formal specifications. In particular the UML and associated tools attack viscosity in order to alleviate the difficulty of choosing and committing to appropriate abstractions.

---

[3] This view was based on the comments of several lecturers. In order to test it we asked computer science students at The University of Southampton whether they liked using formal methods such as Z and B, and whether they liked using graphical design notations such as UML. Of the 118 students that responded, 67% preferred using graphical design notations and 15%, formal methods. The data from the poll and further results are shown in Appendix C.

6.2 Benefits of Translating UML to B

As will be seen, the translatable UML model with formal annotations is just as precise and complete as the equivalent B specification. This is demonstrated by the fact that it can be translated to B automatically. However, there are still benefits to translating into a B specification:

- The textual B specification is a complete mathematical description that may be more readable to experienced formal methods users.
- The B specification can be manipulated mathematically, enabling reasoning and proof to be performed.
- Tools are available for type analysis, proof assistance and animation.
- The translation demonstrates the semantics of the UML version.

A B specification can be animated with the B-Toolkit to explore the dynamic behaviour of the modelled system. In UML terms this means that operations of an object can be invoked and the B animator will check preconditions, and invariants and display the new state of the system in terms of the object's attributes and relationships with other objects. Animation is useful, especially to novices, because it provides feedback and debugging of the specification. It is also useful for validation, i.e. demonstrating to users that the specification describes a system which will be useful.

A class' dynamic behaviour can be proven to conform to its invariants. In UML terms this means that the proof tools will provide assistance in proving that no sequence of invocations of an object's operations can produce a resultant state (in terms of the class' attributes and associations with other objects) that disobeys the invariant. A safety or business critical property of the system could be specified and verified in this way.

UML models prepared for translation to B contain invariant and method specifications written in B notation. The annotated UML diagram is given a precise semantics by the B generated by the translator.

6.3 The U2B Translator

The U2B translator converts Rational Rose[4] UML Class diagrams (Rational 2000A), including attached statecharts, into the B notation. U2B is a script file that runs within Rational Rose and converts the currently open model to B. It is written in the Rational Rose scripting language, which is an extended version of the Summit BasicScript language (Rational 2000B, Rational 2000C). U2B is configured as a menu option in Rose. U2B uses the object-oriented libraries of the Rose Extensibility Interface to extract information about the classes in the logical diagram of the currently open model. The object model representation of the UML diagram means that information is easily retrieved and the program structure can be based around the logical information in the class rather than a particular textual format. U2B uses Microsoft Word[5] to generate the B Machine files. The current version of U2B is a prototype for exploring the translation rules and the efficacy of the concept. The translator could be improved in efficiency and robustness as outlined in Chapter 9.

6.4 Structure and Static Properties

The translation of Classes, attributes and operations is derived from proposals for converting OMT to B (Meyer & Souquières 1999). However, since our aims are primarily to assist in the creation of a B specification rather than to generate a formal equivalent of a UML specification, our translation simplifies that proposed by Meyer and Souquières. This is achieved by restricting the translation to a suitable subset of UML models.

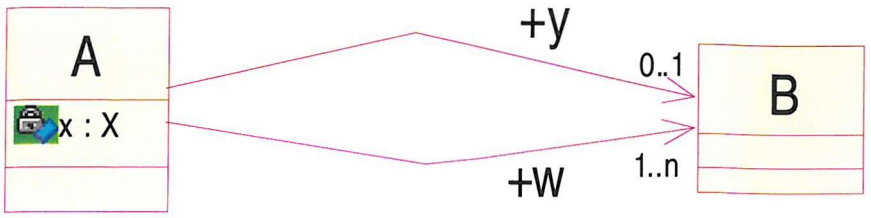
A separate machine is created for each class and this contains a set of all possible instances of the class and a variable that represents the subset of current instances of the class. Attributes and (unidirectional) associations are translated into variables whose type is defined as a function from the current instances to the attribute type (as defined in the Class diagram) or associated class.

For example consider the following class diagram with classes `A` and `B`, where `A` has an attribute `x` and there is a unidirectional association from `A` to `B` with role `y` and 0..1 multiplicity at the target end. A second association, `w`, has a 1..n multiplicity:

---

[4]Rational Rose is a trademark of the Rational Software Corporation

[5]Microsoft Word97 is a trademark of the Microsoft Corporation



This will result in the following machine representing all instances of A:

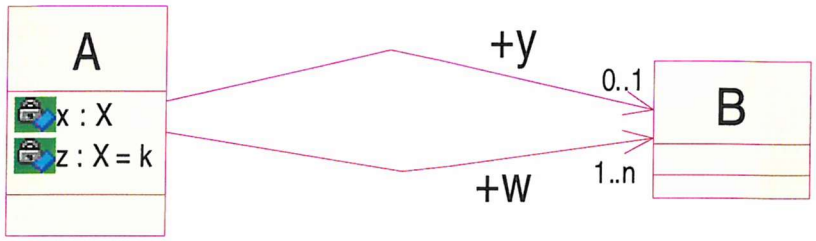
```
MACHINEA
EXTENDS
     B
SETS
     ASET
VARIABLES
     Ainstances,
     x,
     w,
     y
INVARIANT
     Ainstances <: ASET &
     x : Ainstances --> X &
     w : Ainstances --> POW1(Binstances) &
     y : Ainstances +-> Binstances
INITIALISATION
     Ainstances := {} ||
     x := {} ||
     w := {} ||
     y := {}
```

Note that the multiplicity of the association w is handled as a function from instances of class A to sets of instances of class B using the POW (powerset) operator. Multiplicities of associations are discussed in more detail later. The machine is initialised with no instances and hence all attribute and association functions are empty. A separate machine will be generated for class B.

In the example above, as well as in the examples that follow, we use the usual B conventions for capitalisation of names. That is, type sets, including given or enumerated sets, are named in upper case and variables are named in lower case. Hence attributes and association roles are named in lower case. Class names are given in upper case since they are used to generate the name for the given set of possible instances of the class. This results in the variable representing the set of possible instances being part upper and part lower case, however this reflects its main role as a type specifier.

6.4.1 Instance Creation

A create operation is automatically provided for each class machine so that new instances can be created. This picks any instance that isn't already in use, adds it to the current instances set, and adds a maplet to each of the attribute and association relations mapping the new instance to the appropriate initial value. Note that, according to our definition (via translation) of class diagrams, association means that the source class is able to invoke the methods of the target class. The example below is similar to the first example but class `A` has an additional attribute, `z`, that has an initial value, `k`.




```
Return  <-- Acreate =
  PRE
    Ainstances  /= ASET
  THEN
    ANY new
    WHERE
      new : ASET - Ainstances
    THEN
      Ainstances := Ainstances \/ {new } ||
      ANY xx WHERE xx:X THEN
        x(new):=xx END ||
      z(new):=k ||
      ANY xx WHERE xx:POW1(Binstances) THEN
        w(new):=xx END ||
      ANY xx WHERE xx:Binstances THEN
        y(new):=xx END ||
      Return := new
    END
  END
```

Attribute `x` has no initial value specified and is therefore initialised non-deterministically to any value of the type `X`. Attribute `z` is initialised to the specified initial value, `k`. Association `w` must be initialised to a non empty set because its multiplicity may be greater than one but is definitely greater than zero. (Currently, we have no means of specifying initial values for associations). It is initialised non-deterministically to any non-empty subset of instances of `B`. The association, `y`, is initialised non-deterministically to any instance of `B`. (Since its multiplicity is 0 or 1 it could have been left undefined. This is discussed further below).

6.4.2 Association Multiplicities

In UML, multiplicity ranges constrain associations. The multiplicities are equivalent to the usual mathematical categorisations of functions: partial, total, injective, surjective and their combinations. Note that the multiplicity at the target end of the association (class B in the example above) specifies the number of instances of B that instances of the source end (class A) can map to and vice versa. This can be confusing when thinking in terms of functions because the constraint is at the opposite end of the association to the set it is constraining. The multiplicity of an association determines its modelling as shown in Table 6.1. We use functions to sets of the target class instances (e.g. POW(B)) to avoid non-functions. Note that n is assumed unless otherwise specified in the UML class diagram.

Multiplicity also affects the initialisation of an association that is performed when new instances of the source class are created. Currently this has not been adequately addressed in the U2B translation. For example, in the first case in Table 6.1 (0..n→0..1), the translator selects any existing instance of class B. This is unnecessarily restrictive since creating a new instance of B or leaving the association undefined are equally viable options. In the case (1..1→1..1) the translator's action is invalid since the only allowable initialisation is to create a new instance of B to map the association to. The (automatically generated) create operation supplies a new instance as an output of the operation but this can only be assigned to a local variable or output variable. Assignment of an operation output to a global variable would require the use of sequential composition, which is not allowed in specifications in B. An alternative 'create' operation that accepts a parameter identifying the new instance to be created is required. Similarly, for the case 0..n → 0..n, because the multiplicity at the target class may be greater than 1, it should be possible to initialise the association to a set consisting of any combination of existing and newly created instances of B. In the last case in Table 6.1 (1..1→1..n) the translator's action is, again, invalid since the only valid action is to create a (non-empty) set of new instances of B. To create and assign a set of new instances, an alternative create operation is needed that accepts as a parameter the set of new instances.

Association Representations in B for Different Multiplicities		
Ai and Bi are the current instances sets of class A and B respectively (i.e. Ainstances and Binstances) and f is a function representing the association (i.e. the role name of the association with respect to the source class, A). `disjoint(f)` *is defined in B as:* `!(a1,a2).( a1:dom(f) & a2:dom(f) & a1/=a2 => f(a1)/\f(a2)={} )`		
UML association multiplicity	**Informal description of B representation**	**B invariant**
0..n → 0..1	partial function to Bi	`Ai +-> Bi`
0..n → 1..1	total function to Bi	`Ai --> Bi`
0..n → 0..n	total function to subsets of Bi	`Ai --> POW(Bi)`
0..n → 1..n	total function to non-empty subsets of Bi	`Ai --> POW1(Bi)`
0..1 → 0..1	partial injection to Bi	`Ai >+> Bi`
0..1 → 1..1	total injection to Bi	`Ai >-> Bi`
0..1 → 0..n	total function to subsets of Bi which don't intersect	`Ai --> POW(Bi) & disjoint(f)`
0..1 → 1..n	total function to non-empty subsets of Bi which don't intersect	`Ai --> POW1(Bi) & disjoint(f)`
1..n → 0..1	partial surjection to Bi	`Ai +->> Bi`
1..n → 1..1	total surjection to Bi	`Ai -->> Bi`
1..n → 0..n	total function to subsets of Bi which cover Bi	`Ai --> POW(Bi) & union(ran(f))= Bi`
1..n → 1..n	total function to non-empty subsets of Bi which cover Bi	`Ai --> POW1(Bi) & union(ran(f))= Bi`
1..1 → 0..1	partial bijection to Bi	`Ai >+>> Bi`
1..1 → 1..1	total bijection to Bi	`Ai >->> Bi`
1..1 → 0..n	total function to subsets of Bi which cover Bi without intersecting	`Ai --> POW(Bi) & union(ran(f))= Bi & disjoint(f)`
1..1 → 1..n	total function to non-empty subsets of Bi which cover Bi without intersecting	`Ai --> POW1(Bi) & union(ran(f))= Bi & disjoint(f)`

Table 6.1 – How associations are represented in B for each possible multiplicity constraint

In Fig. 6.1 a mapping represents an association between the classes `A` and `B` with multiplicity 0..n → 0..1. The representation in B is a partial function. It is not a total function because the element `a4` doesn't map to anything in `B` (as indicated by the 0 at the right hand end of 0..n → 0..1). It is not injective because `b2` is mapped to by both `a2` and `a3` (as indicated by the n at the left hand end of 0..n → 0..1). It is not surjective because `b3` is not mapped to by anything in `A` (as indicated by the 0 at the left hand end of 0..n → 0..1)

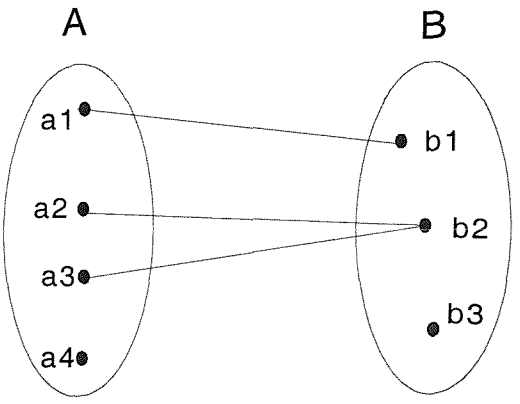
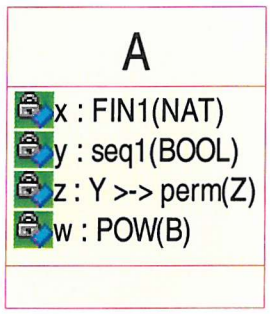


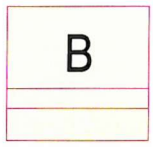
Fig 6.1 Mapping representing a 0..n → 0..1 association

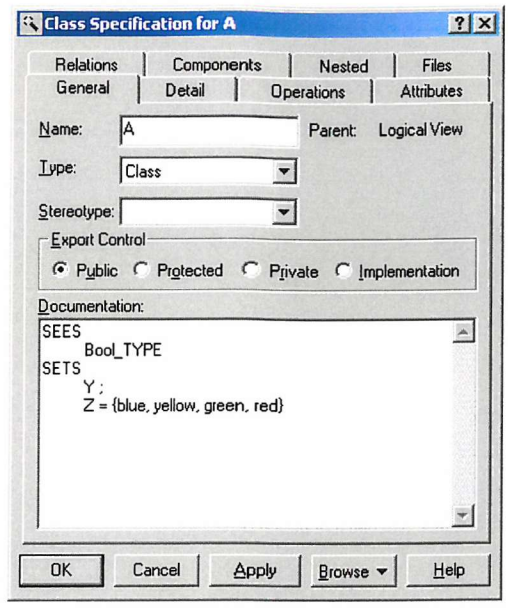
6.4.3 Attribute Types

Attribute types may be any valid B expression that defines a set. This includes predefined types (such as NAT, NAT1, BOOL and STRING) functions, sequences, powersets, instances of another class (referenced by the class name), and enumerated or deferred sets defined in the class specification documentation window. (If translating to B-Core B, the appropriate B library machines must be referenced via a SEES clause in the class's specification documentation window). If the type involves another class (and there is no unidirectional path of associations to that class) the machine for that class will be referenced in a USES clause so that its current instances set can be read. If there is a path of unidirectional associations to the class it will be extended (EXTENDS) by this machine in order to represent the association and this will provide access to the instances set. (Note that only unidirectional associations are interpreted as associations. Unspecified or bi-directional associations are ignored and can therefore be used to indicate type dependencies diagrammatically if required). Any references to the class in type definitions of variables or operation arguments will be changed to the current instances set for that class.

For example, the following shows a class that has an attribute `x` of type, non-empty finite subset of natural numbers. It has an attribute `y` that is of type, non-empty sequence of booleans. The library machine `Bool_TYPE` has been referenced via a SEES clause in the class's documentation window (this would not be necessary for Atelier-B). It has an attribute `z` that has type, total injection from `Y` to permutations of `Z`. A 'SETS' clause has been added to the class' documentation window that defines `Y` as a deferred set and `Z` as an enumerated set.







Note that 'Export Control' settings in the class specification are not used in the U2B translation. The corresponding B machine for class A is shown below.

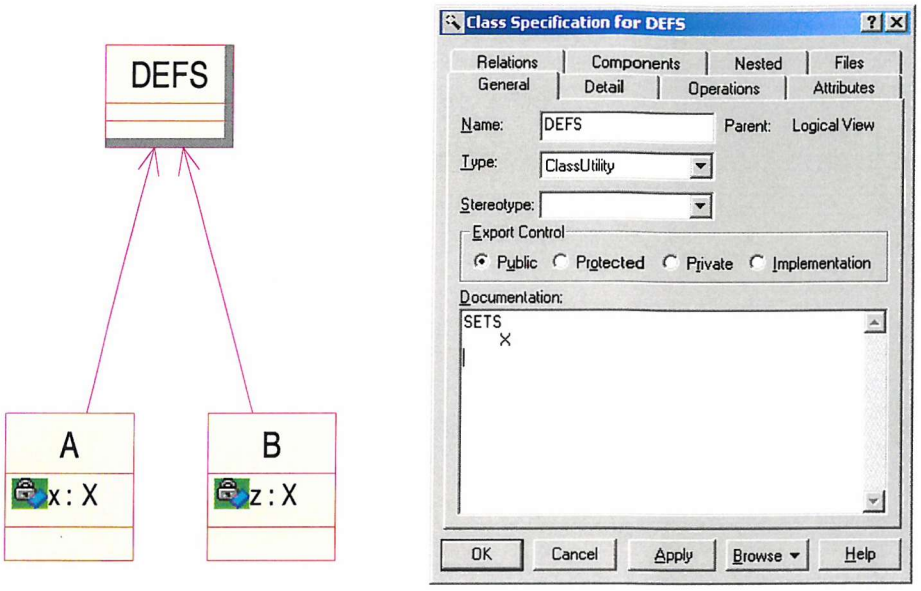
```
MACHINEA
SEES
     Bool_TYPE
USES
     B
SETS
     ASET ;
     Y ;
     Z = {blue, yellow, green, red}

VARIABLES
     Ainstances,
     x,
     y,
     z,
     w
INVARIANT
     Ainstances <: ASET &
     x : Ainstances --> FIN1(NAT) &
     y : Ainstances --> seq1(BOOL) &
     z : Ainstances --> Y >-> perm(Z) &
     w : Ainstances --> POW(Binstances)
```

6.4.4 Global Definitions

It is often useful to define types as enumerated or deferred sets for use in many machines. We use 'class utilities' for this. In UML, a class utility is a class that doesn't have any instances, only static (class-wide) operations and attributes. The U2B translator creates a machine for each class utility and copies any text in the specification documentation window of its class specification into the machine. Hence definitions, sets and constants can be described in B clauses in the

documentation window. Any machines that reference things defined in this way must have an association to the class utility. (This association will not be interpreted as an association to an ordinary class). In the following example a class utility, DEFS, is used to define a set X that is used as a type by 2 other classes.



The corresponding machine for class utility DEFS is:

```
MACHINEGLOBALS
SETS
        X
END
```

The machines for classes A and B will reference DEFS via a 'SEES clause:

```
SEES
    DEFS
```

6.4.5 Local Definitions

As we have seen in a previous example, such sets can also be defined locally to a class in the class' specification documentation window. In fact, any valid B clause can be added in this window. For example, we use this method to specify invariants for the class. Each clause must be headed by its B clause name in capitals and starting at the beginning of a line, the text that follows that clause, up until the next clause title (if any) will be added to the appropriate clause in the machine. Any text before the first clause is treated as comment and added as such at the top of the machine

6.4.6 Singular Classes

Often, a B machine models a single generic instance of an entity, rather than an explicit set of instances (in the same way that a class in UML leaves instance referencing implicit). The resulting specification is simpler and clearer for not modelling instances. If the class multiplicity (cardinality) is set to 1..1 in the UML class specification, the U2B translator creates a machine with no instance modelling. Note that this can only be done at the top level of an association hierarchy since at lower levels the instance set is used for referencing by the higher level. Below is shown the machine representing class A from the first example above if the class' multiplicity is set to 1..1. Note that there is no modelling of instances; the types of attributes are simpler because it is no longer necessary to map from instances to the attribute type. There is no instance create operation, attributes are initialised in the machine initialisation clause.

```
MACHINEA
EXTENDS
     B
VARIABLES
     x,
     w,
     y
INVARIANT
     x : X &
     w : POW1(Binstances) &
     y : Binstances
INITIALISATION
     X :: X ||
     w :: POW1(Binstances) ||
     y := {}
END
```

6.4.7 Restrictions

The B method imposes some restrictions on the way machines can be composed. These restrictions ensure compositionality of proof. Their impact is that no write sharing is allowed at machine level (i.e. a machine may only be included or extended by one other machine). Also, the inclusion mechanism of B is hierarchical. Hence, if M1 includes M2, then M2 cannot, directly or transitively, include M1. We reflect these restrictions in the UML form of the specification, which must therefore be tree like in terms of unidirectionally related classes. Non-navigable (and bi-directional) associations are ignored but may be used to illustrate the use of another class as a type (i.e. read access only). However, multiple, parallel associations between the same pair of classes are permitted.

Although we would like to adhere to the UML class diagram rules as much as possible, since our aim is to make B specification more approachable rather than to formalise the UML we are

relatively happy to impose restrictions on the diagrams that can be drawn. That is, we only define translations for a subset of UML class diagrams. Other authors (Facon, Laleau & Nguyen, 1996, Meyer & Souquières, 1999, Meyer & Santen, 2000, Nagui-Raiss, 1994, Shore, 1996) have suggested ways of dealing with the translation of more general forms of class diagrams. However, the structures of B machines that result from these more general translations can be cumbersome. If the specification were written directly in B, it would be highly unlikely that the resulting B would have this form. Since we also desire a usable B specification we prefer to restrict the types of diagrams that can be drawn.

6.5 Dynamic Behaviour

The dynamic behaviour modelled on a class diagram that is converted to B by U2B is embodied in the behaviour specification of class operations and invariants. UML does not impose any particular notation for these definitions; they could be described in natural language or using UML's Object Constraint Language (OCL). However since we wish to end up with a B specification it makes sense to use bits of B notation to specify these constraints. The constraints are specified in a notation that is close to B notation but needs to observe a few conventions in order for it to become valid B within the context of the machine produced by U2B. When writing these bits of B the writer shouldn't need to consider how the translation would represent the features (associations, attributes and operations) of the classes. Also we felt we should follow the object-oriented conventions of implicit self-referencing and the use of the dot notation for explicit instance references. This is illustrated in examples below.

6.5.1 Invariant

Unfortunately there is no dedicated text box for a class invariant in Rational Rose. One suggestion is to put invariant constraints in a note attached to the class (Warmer & Kleppe, 1999), but notes are treated as an annotation on a particular view (diagram) in Rational Rose and not part of the model. This makes them difficult to access from the translation program and unreliable should we extend the conversion to look at other views. Therefore we include the invariants as a clause in the documentation text box of the class' specification window. Invariants are generally of two kinds, instance invariants (describing properties that hold between the attributes and relationships within a single instance) and class invariants (describing properties that hold between different instances). For instance invariants, in keeping with the implicit self-reference style of UML, we chose to allow the explicit reference to 'this instance' to be omitted. U2B will add the universal quantification over all instances of the class automatically. For class invariants, the quantification over instances is an integral part of the

property and must be given explicitly. Hence, U2B will not need to add quantification and instance references.

For example, if `bx: NAT` is an attribute of class `B` then the following invariant could be defined in the documentation box for class `B`:

```
bx < 100 &
!(b1,b2).((b1:B & b2:B & b1/=b2)=> (b1.bx/=b2.bx)
```

This would be translated to:

```
!(thisB).(thisB:Binstances =>
   bx(thisB) < 100 &
   !(b1,b2).((b1:Binstances & b2:Binstances & b1/=b2)
=> (bx(b1)/=bx(b2))
)
```

The translation has added a universal quantification, `!(thisB)`, over all instances of `B` and this is used in the first part of the invariant. It is not used in the second part where the invariant already references instances of class `B`. (Note that currently the translator adds one universal quantification for the entire invariant whether or not it is needed).

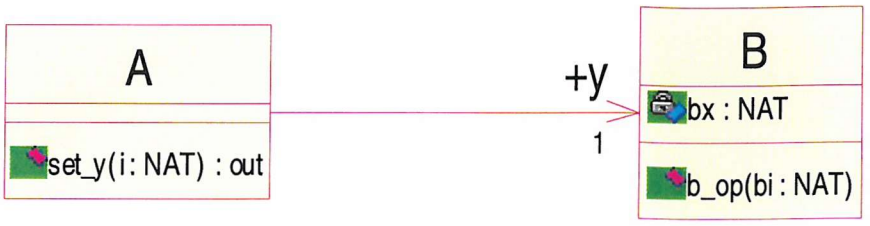
6.5.2 Operation Semantics

Operation preconditions are specified in a textual format attached to the operation within the class. Details of operation behaviour are specified either in a textual format attached to the operation, or in a statechart attached to the class. Operation behaviour may be specified completely by textual annotation, completely by statechart transitions, or by a combination of both composed as simultaneous specification.

Operation textual behaviour specification - In Rational Rose, 'Specifications' are provided for operations (as well as many other elements) and these provide text boxes dedicated to writing preconditions and semantics for the operation. (A postcondition text box is also provided. Initially we used this for the operation body. Reviewers found this strange because operation bodies in B do not look like postconditions predicates. In fact they are mathematically equivalent, but since our motivation is to achieve a more user-friendly and intuitive form of formal specification, we decided to use the semantics box because it suits the pseudo-operational style of B).

Operations need to know which instance of the class they are to work on. This is implicit in the class diagram. The translation adds a parameter `thisCLASS` of type `CLASSinstances` to each

operation. This is used as the instance parameter in each reference to an attribute or association of the class.



In the above example, `set_y` might have the following precondition:

```
i > y.bx
```

and semantics

```
y.b_op(i) ||
IF y.bx <100
THEN
    out := FALSE
ELSE
    out := TRUE
END
```

which would be translated to

```
i > bx(y(thisA))
```

and

```
b_op(y(thisA)) ||
IF bx(y(thisA)) <100
THEN
    out := FALSE
ELSE
    out := TRUE
END
```

Operation Return Type - UML operation signatures contain a provision for specifying the type for a value returned by the operation. Since B infers this from the body of the operation we use it instead to name the identifiers that represent operation return values. The string entered in the return type field for the operation will be used as the operation return signature in the B machine representing the class. For example, the `set_y` operation in the above example has its return field set to `out`. The operation signature for `set_y` in the B machine `A` will be:

```
out <-- set_y (thisA,i) =
```

Statechart Behavioural Specification - For classes that have a strong concept of state change, a statechart representation of behaviour is appropriate. In UML a statechart can be attached to a class to describe its behaviour. The underlying model representing the statechart is constructed and viewed via a set of one or more state diagrams. A statechart consists of a set of states and a set of transitions that represent the changes between states that are allowed. If a statechart is attached to a class the U2B translator combines the behaviour it describes with any operation semantics described in the operation specification semantics windows. Hence operation behaviour can be defined either in the operation semantics window or in a statechart for the class or in a combination of both.

The name of the statechart model is used to define a state variable. (Note that this is not the name of a state diagram, several diagrams could be used to draw the statechart of a class). The collection of states in the statechart is used to define an enumerated set that is used in the type invariant of the state variable. The state variable is equivalent to an attribute of the class and may be referenced elsewhere in the class and by other classes. State chart transitions define which operation call causes the state variable to change from the source state to the target state, i.e., an operation is only allowed when the state variable equals a state from which there is a transition associated with that operation. To associate a transition with an operation, the transition's name must be given the same name as the operation. Additional guard conditions can be attached to a transition to further constrain when it can take place. All transitions cause the implicit action of changing the state variable from the source state to the target state. (The source and target state may be the same). Additional actions (defined in B) can also be attached to transitions. The translator finds all transitions associated with an operation and compiles a SELECT substitution of the following form:

```
SELECT statevar=sourcestate1 & transition1_guards
THEN statevar:=targetstate1 || transition1_actions
WHEN statevar=sourcestate2 & transition2_guards
THEN statevar:=targetstate2 || transition2_actions
etc
END ||
```

This is composed with the operation precondition and body (if any) from the textual specification in the operation's precondition and semantics windows:

Let `Popw` be the precondition in the operation precondition window, `Sosw` be the operation body from the operation semantics window and `Gstc` the SELECT substitution for this

operation composed from the statechart. Then the translator will produce the following operation:

```
PRE
   Popw
THEN
   Gstc ||
   Sosw
END
```

This can be represented as: `Popw | (Gstc || Sosw)`

Hence the pre condition, `Popw`, has precedence and, if false, the operation will abort. If an event B style systems simulation (Abrial 2000) is desired, the specifier should take care not to define preconditions that conflict with the transition guards. (For example, if an event only occurs if an attribute, bx, is positive, and this is modelled by a guarded transition; adding the precondition bx>0 would change the meaning of the model to represent a system where the event can occur at any time but aborts if bx is not greater then 0).

Note that it would be entirely valid (although somewhat obtuse) to write a precondition within the operation semantics window: `Sosw = Posw | S1osw`. However, preconditions take precedence in simultaneous substitutions, so

`(Gstc || (Posw | S1osw)) = Posw | (Gstc || S1osw)`

Hence, writing the precondition in the operation semantics window is equivalent to writing it in the precondition window. It has the same precedence and possible conflicts with the operation guards derived from the statechart. We feel that writing the precondition in the operation semantics window should be discouraged because the precedence may not be obvious to readers of the specification.

If the precondition (`Popw` $\wedge$ `Posw`) is true, then the guard from `Gstc` takes precedence over the simultaneous substitution, `Sosw`. This means that the textual operation body from the operation semantics window, although defined separately from the statechart and not associated with any particular state transition, is only enabled when at least one of the state transitions is enabled. That is, if

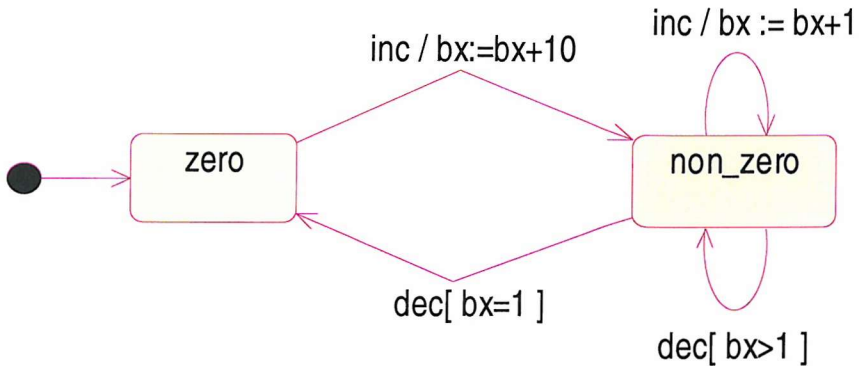
`Gstc = (G1 ⇒ T1) □ .... □ (Gn ⇒ Tn)`

then,

`(Gstc || Sosw) = (G1 ⇒ (T1 || Sosw)) □ ... □ (Gn ⇒ (Tn || Sosw))`

where □ represents choice.

Actions should be specified on state transitions when the action is specific to that state transition. Where the action is the same for all that operation's state transitions, it may be specified in the operation semantics window in order to avoid repetition. The following example illustrates how a statechart can be used to guard operations and define their actions. It also shows how common actions can be defined in the operation semantics window and how a precondition could upset the constraints imposed by the statechart.



The statechart has 2 states, `zero` and `non_zero`. The implicit state variable, `b_state` (the name of the statechart) is treated like an attribute of type `B_STATE = {zero,non_zero}`. An invariant, `(b_state=zero) <=> (bx=0)`, defines the correspondence between the value of the attribute `bx` and the state `zero`. The invariant would be written in the class specification window. When an instance is created its `b_state` is initialised to `zero` because there is a transition from an 'initial' state to `zero`.

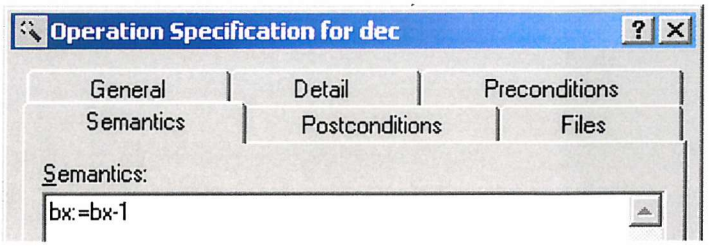
```
MACHINEB
SETS
     BSET;
     B_STATE={zero,non_zero}
VARIABLES
     Binstances,
     b_state,
     bx
INVARIANT
     Binstances  <: BSET &
     b_state : Binstances --> B_STATE &
     bx : Binstances --> NAT &
     !(thisB).(thisB:Binstances =>
      (b_state(thisB)=zero) <=> (bx(thisB)=0)
      )
INITIALISATION
     Binstances := {} ||
     b_state := {} ||
     bx := {}
```

```
OPERATIONS
Return  <-- Bcreate =
   PRE
      Binstances  /= BSET
   THEN
      ANY new
      WHERE
         new : BSET - Binstances
      THEN
         Binstances := Binstances \/ {new } ||
         b_state(new):=zero ||
         ANY xx WHERE xx:NAT THEN
            bx(new):=xx END ||
         Return := new
      END
   END
;
```

Operation `inc` can occur in either state. Its action is different depending on the starting state and so actions have been defined on the transitions and are combined with the state change action.

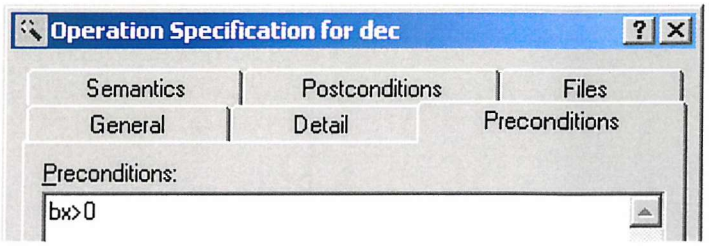
```
inc (thisB) =
   PRE
      thisB : Binstances
   THEN
      SELECT b_state(thisB)=zero
      THEN   b_state(thisB):=non_zero ||
             bx(thisB):=bx(thisB)+10
      WHEN   b_state(thisB)=non_zero
      THEN   bx(thisB) := bx(thisB)+1
      END
   END
```

Operation `dec` has two guarded alternatives when in state `non_zero` but does not occur while in state `zero`. Since the action is the same for both transitions it has been defined in the operation's semantics window.



```
dec (thisB) =
   PRE
      thisB  :  Binstances
   THEN
      SELECT b_state(thisB)=non_zero &
             bx(thisB)=1
      THEN   b_state(thisB):=zero
      WHEN   b_state(thisB)=non_zero &
             bx(thisB)>1
      THEN   skip
      END ||
      bx(thisB):=bx(thisB)-1
   END
END
```

If we had put a precondition in the operation specification precondition window (or even in the operation semantics window), the guard would no longer function since the precondition would fail resulting in an abort when bx=0.




```
dec (thisB) =
   PRE
      thisB  :  Binstances &
      bx(thisB)>0
   THEN
      SELECT b_state(thisB)=non_zero &
             bx(thisB)=1
      THEN   b_state(thisB):=zero
      WHEN   b_state(thisB)=non_zero &
             bx(thisB)>1
      THEN   skip
      END ||
      bx(thisB):=bx(thisB)-1
   END
```

This could be avoided by repeating the precondition and decrement substitution in the action field of each dec transition on the statechart in which case the guard would take precedence.

6.6 Summary

In this chapter we have described a method for attaching formal constraints to class diagrams drawn in the Rational Rose UML tool. The class diagram becomes a graphical formal specification notation, B-UML, which we hope will bring benefits to the process of creating a

formal specification. We define a translation to the B notation, which ensures a precise definition of the semantics of B-UML. The translation also provides a pure textual equivalent in a recognised formal notation that has good tool support.

Chapter 7

Examples of B-UML and U2B in Use

In this chapter we present three examples of B-UML models and show how they translate into equivalent B specifications. The first example, a raffle game, demonstrates the features of the class diagram translation. The second example, a railway station, introduces the use of statecharts to specify operation behaviour within a class. The third example, part of a teletext page selection system, is based on a real industrial project. It is a simplified version of a model initially developed jointly with M. Satpathy at Reading University. This example illustrates some techniques for coping with more complicated statecharts. Although the teletext example is suitable for the purposes of illustrating the translation techniques, it is apparent that a statechart description is not the most suitable means to describe the problem. This demonstrates the importance of having the textual form in the operation semantics windows. The example also illustrates some limitations of the current translation methods.

7.1 Raffle Game

This example describes a raffle game system. Newly created games must be initialised by setting their set of prizes before tickets can be sold. When a ticket is sold a record of the player that bought it is kept. A draw of the winning tickets can be attempted at any time but is only achieved when enough tickets have been sold to win all the prizes. A ticket can be checked to see if it is a winning ticket. A prize can be claimed by submitting a winning ticket and identifying the player that bought it correctly.

Fig. 7.1 shows a class `GAME` that has typed and initialised attributes, parameterised operations (some with return values), three association relationships with a class `TICKET` and an aggregate relationship with another class, `PRIZE`. The class also uses another class, `PLAYER`, as a type. The associations have role names `Prizes`, `Tickets`, `Winners` and `Claimed`, which are used to refer to the instances of the associated class involved in the association. The class `GAME` has an operation `setprizes` that allows the associated prizes to be defined for a particular instance of `GAME`. When this has been done, operation `buy` allows players to buy tickets for a game by incrementing attribute `Sales` and non-deterministically selecting an unused instance of class `TICKET`, calling its `sell` operation (which sets its `Owner` and `Sold` attributes) and adding it to

the association `Tickets`. Once a minimum number of tickets have been sold for a game, operation `draw` allows the winning tickets for that game to be selected, one for each prize, and added to the association `Winners`. Players can check to see whether their ticket belongs to the association `Winners`. If it does, they can use operation `claim` to obtain one of the prizes, which is selected non-deterministically. The ticket, for which a prize has been claimed, is added to the association `Claimed`. Attribute `Owner`, of class `TICKET`, records which player bought the ticket so that this can be checked when a prize is claimed.

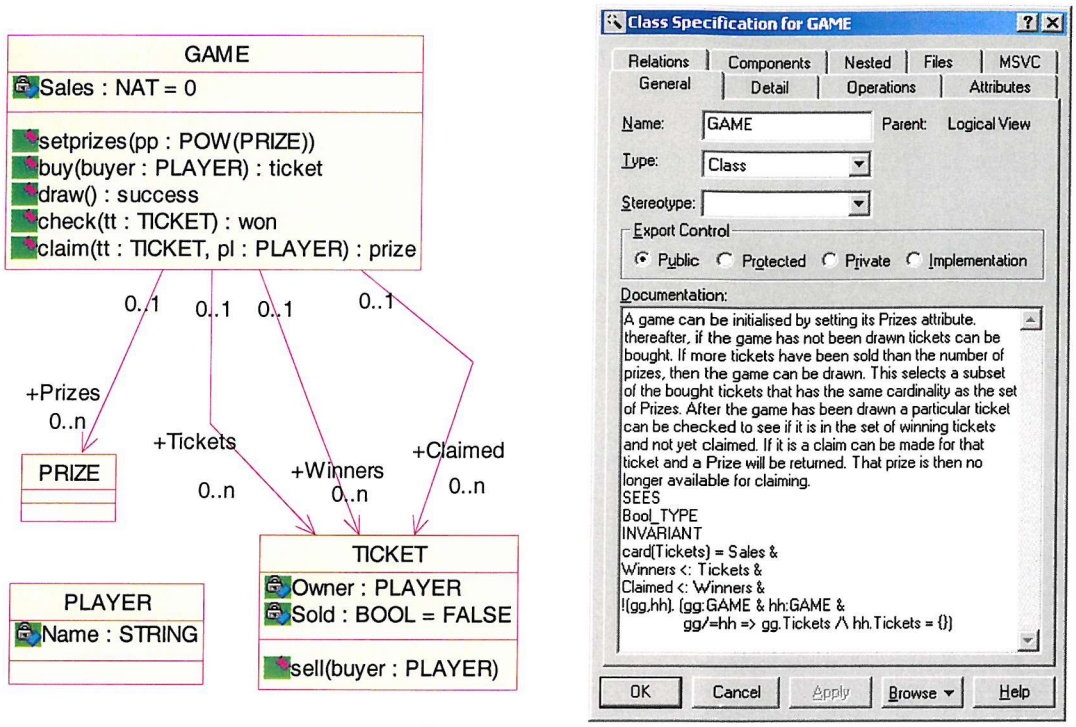


Fig. 7.1 Class Diagram and Class Specification for Game

Alongside the class diagram is shown the Rational Rose specification for the class `GAME`. Following the natural language description in the 'Documentation' box some class invariants are given. These express the requirements that the number of sales is equal to the number of tickets sold, winning tickets are a subset of the tickets sold and tickets for which a prize has been claimed are a subset of the winning tickets. These invariants describe relationships between the attributes and associations within a single instance of the class `GAME`. The last invariant ensures that a ticket cannot be sold for two different games and describes a relationship between instances of the class. This last invariant was entered before the translator supported multiplicities on associations. It is now redundant since the association multiplicity 0..1 at the source end expresses this constraint and U2B automatically generates the equivalent predicate `disjoint(Tickets)`. Note that the attribute `Sales` is also redundant and could be removed. Apart from requiring extra operations to maintain it, redundant information requires invariants

to ensure it is kept consistent and these will generate additional proof obligations. (Both have been left in the example purely for illustrative reasons.) The Atelier-B proof tools were used (by a colleague) to prove that these invariants were preserved by the operations of the example. The proofs uncovered a mistake in the original version of the buy operation that allowed a ticket that already belonged to another game to be resold. In the buy operation described below, the precondition and selection predicate of the ANY substitution contained `tt:TICKET-Tickets` (i.e. tt is a ticket that doesn't already belong to this game) instead of `tt:TICKET-UNION(gg).(gg:GAME|gg.Tickets)` (i.e. tt is a ticket that doesn't already belong to any game).

Each operation of the class also has a Rose Specification window with appropriate tabs for the definition of the operation. The operation preconditions and body shown in Fig. 7.2 are taken from the precondition and semantics tabs of the specification for the `buy` operation in class `GAME`. The ANY construct is a statement of the B language that selects a value for a variable (here `tt`) satisfying some condition. In this case the condition is `tt:TICKET-UNION(gg).(gg:GAME|gg.Tickets)`, i.e. select an unused ticket. The second part of this expression is a generalised union of the association `Tickets` over all instances of the parent class, `GAME`. This is expressed as the union of `gg.Tickets` for all `gg:GAME`. Also, note the call to operation `sell` of the `Tickets` class. The operation is called for the instance `tt` of `TICKET`.

```
precondition
Prizes /= {} &
Winners = {} &
TICKET-UNION(gg).(gg:GAME|gg.Tickets) /= {}

semantics
ANY tt WHERE tt: TICKET - UNION(gg).(gg:GAME|gg.Tickets)
THEN
        Tickets := Tickets \/ {tt} ||
        tt.sell(buyer) ||
        Sales := Sales +1 ||
        ticket := tt
END
```

Fig. 7.2 Precondition and Semantics for operation *buy* of class *GAME*

Below is shown the automatically produced B machine for the class `GAME`:

```
MACHINEGAME
/*" A game is initialised by setting its Prizes attribute. "*/
.....

SEES
     Bool_TYPE
EXTENDS
     PRIZE,
     TICKET
USES
     PLAYER
SETS
     GAMESET
VARIABLES
     GAMEinstances,
     Sales,
     Prizes,
     Tickets,
     Winners,
     Claimed
```

Machines of associated classes are extended so that their operations are accessible to higher level classes. Classes used as types only need USES access.

Current class instances is a variable which is a subset of the possible instances, a given set, GAMESET.

Variables model the attributes and associations of the class.

The types of variables used to model attributes and associations are defined in the invariant as functions from the current instances to the attribute/association type. Association multiplicities affect these functions and impose constraints on their ranges. In this case the functions map to subsets of the target class that don't intersect

```
INVARIANT
     GAMEinstances  <: GAMESET &
     Sales : GAMEinstances --> NAT &
     Prizes : GAMEinstances --> POW(PRIZEinstances) &
     disjoint(Prizes)
     Tickets : GAMEinstances --> POW(TICKETinstances) &
     disjoint(Tickets)
     Winners : GAMEinstances --> POW(TICKETinstances) &
     disjoint(Winners)
     Claimed : GAMEinstances --> POW(TICKETinstances) &
     disjoint(Claimed)
     !(thisGame).(thisGame:GAMEinstances =>
      card(Tickets(thisGame)) = Sales(thisGame) &
     Winners(thisGame) <: Tickets(thisGame) &
     Claimed(thisGame) <: Winners(thisGame) &
     !(gg,hh). (gg:GAMEinstances & hh:GAMEinstances &
                    gg/=hh => Tickets(gg) /\ Tickets(hh) = {}) )
```

Invariants from the class documentation window are copied into the machine invariant and have universal quantification over all current class instances added. Dot notation of explicit instance references has been converted to parameterisation.

```
INITIALISATION
     GAMEinstances := {} ||
     Sales := {} ||
     Prizes := {} ||
     Tickets := {} ||
     Winners := {} ||
     Claimed := {}
OPERATIONS
Return  <-- GAMEcreate =
   PRE
     GAMEinstances  /= GAMESET
   THEN
     ANY new
     WHERE
       new : GAMESET - GAMEinstances
     THEN
       GAMEinstances := GAMEinstances \/ {new } ||
       Sales(new):=0 ||
       ANY xx WHERE xx:POW(PRIZEinstances-union(ran(Prizes)))
       THEN Prizes(new):=xx END ||
       ANY xx WHERE xx:POW(TICKETinstances-union(ran(Tickets)))
       THEN Tickets(new):=xx END ||
       ANY xx WHERE xx:POW(TICKETinstances-union(ran(Winners)))
       THEN Winners(new):=xx END ||
       ANY xx WHERE xx:POW(TICKETinstances-union(ran(Claimed)))
       THEN Claimed(new):=xx END ||
       Return := new
     END
   END
;
/*" Initialise the Prizes attribute with a set of Prizes "*/
 setprizes (thisGame,pp) =
   PRE
     thisGame  :  GAMEinstances &
     pp:POW(PRIZEinstances) &
     Prizes(thisGame) = {}
   THEN
     Prizes(thisGame) := pp
   END
;
/*" If the game has had its Prizes set and has not been drawn then "*/
/*" a ticket is sold to the buyer and added to Tickets and returned"*/
ticket <--  buy (thisGame,buyer) =
   PRE
     thisGame  :  GAMEinstances &
     buyer:PLAYERinstances &
     Prizes(thisGame) /= {} &
     Winners(thisGame) = {} &
     TICKETinstances-UNION(gg).(gg:GAMEinstances|Tickets(gg)) /= {}
   THEN
     ANY tt WHERE tt: TICKETinstances -
                      UNION(gg).(gg:GAMEinstances|Tickets(gg))
     THEN
             Tickets(thisGame) := Tickets(thisGame) \/ {tt} ||
             sell(tt,buyer) ||
             Sales(thisGame) := Sales(thisGame) +1 ||
             ticket := tt
     END
   END
;
```

All machine variables are initialised to empty sets. An instance creation operation is automatically provided. This initialises the attribute values for the new instance according to the initialisation values specified for the class or non-deterministically where no initialisation value is given.. The new instance is returned.

```
/*" If the game has been set up and not been drawn already and "*/
/*" enough tickets have been sold to provide a winner for each "*/
/*" prize then the game is drawn by selecting a subset of the "*/
/*" tickets sold as winners of the prizes and true is returned. "*/
/*" If  the game has been set up and not been drawn already but "*/
/*" not enough tickets have been sold, false is returned "*/
success <--  draw (thisGame) =
   PRE
      thisGame  :  GAMEinstances &
      Prizes(thisGame) /= {} &
      Winners(thisGame) = {}
   THEN
      IF  card (Prizes(thisGame)) < card (Tickets(thisGame))
      THEN
              ANY ww WHERE
                      ww : POW (Tickets(thisGame)) &
                      card (ww) = card (Prizes(thisGame))
              THEN
                      Winners(thisGame) := ww
              END ||
              success := TRUE
      ELSE
              success := FALSE
      END
   END
;
/*" If tt is in the set of winners but not in the set of claimed "*/
/*"  true is returned, otherwise false is returned "*/
won <--  check (thisGame,tt) =
   PRE
      thisGame  :  GAMEinstances &
      tt:TICKETinstances
   THEN
      IF tt : Winners(thisGame) - Claimed(thisGame)
      THEN
              won := TRUE
      ELSE
              won := FALSE
      END
   END
;
/*" If tt is in Winners but not in Claimed and pl is the owner of "*/
/*" tt one of the prizes in Prizes is returned and is removed from "*/
/*"  Prizes and the ticket is added to claimed "*/
prize <--  claim (thisGame,tt,pl) =
   PRE
      thisGame  :  GAMEinstances &
      tt:TICKETinstances &
      pl:PLAYERinstances &
      tt : Winners(thisGame) - Claimed(thisGame) & Owner(tt) = pl
   THEN
      ANY pp WHERE pp :Prizes(thisGame)
      THEN
              Claimed(thisGame) := Claimed(thisGame) \/ {tt} ||
              Prizes(thisGame) := Prizes(thisGame) - {pp} ||
              prize := pp
      END
   END
END
```

This example demonstrates how effective the semi-diagrammatic method is for creating formal specifications. In producing the specification, we found the representation of its main elements (such as GAME and TICKET) and the organisation of attributes, associations and operations, helpful in visualising and deriving the model. Much of the infrastructure of the B machines was generated automatically, which left us free to concentrate on adding the operation semantics and invariants. The separation of the parts of textual specification by 'hanging' them onto diagrammatic entities seemed to help psychologically in making them seem easier to consider. The resulting specification closely resembles the familiar UML class diagram making it approachable and comprehendible to software engineers. Using the textual B version of the model enabled us to detect a mistake in it.

7.2 Railway Station

This example is a model of a railway station. It is an extension of the example in Lano (1996). A station has a number of platforms and extra platforms can be added. Arriving trains are allocated to an available platform if one exists or are queued until a platform becomes available or an error occurs. In the latter case a queued train moves to one of the platforms whether or not it is available and hence a crash may occur. Platforms may be opened and closed. A platform is available when it is open and no train is occupying it. A crash occurs if a train arrives at a closed or occupied platform. If a crash occurs at a platform it may be cleared and made available by opening it. If a multiple crash occurs (i.e. more than 1 train occupies the platform) opening the platform will leave it closed and a subsequent opening is required to make it available.

The class diagram in Fig. 7.3 consists of a class STATION that has a typed and initialised attribute, parameterised operations (one with a return value), and an association with another

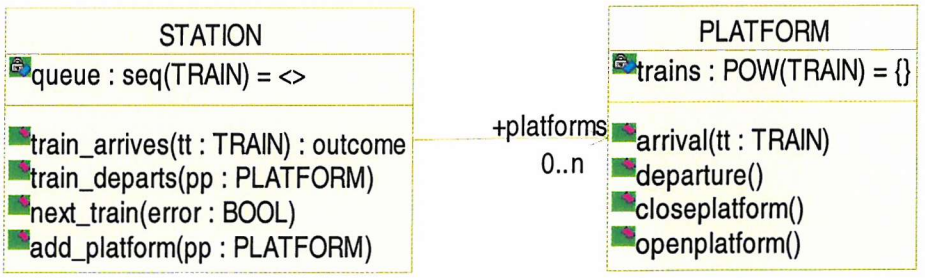


Fig. 7.3 Class Diagram for Example Station

class PLATFORM. The association has a role name platforms, which is used to refer to instances of the associated class.

In Fig. 7.4 the Rational Rose specification window for the class `PLATFORM` is shown. Following the natural language description in the documentation box some class invariants are given. The first part of the invariant contains three instance invariants that implicitly apply to all platform instances. The final part of the invariant is a class invariant that is explicitly quantified for all pairs of platforms. Note that some of these invariants refer to a state variable, `platfom_state`, and its possible values that have been obtained from a statechart attached to the class.

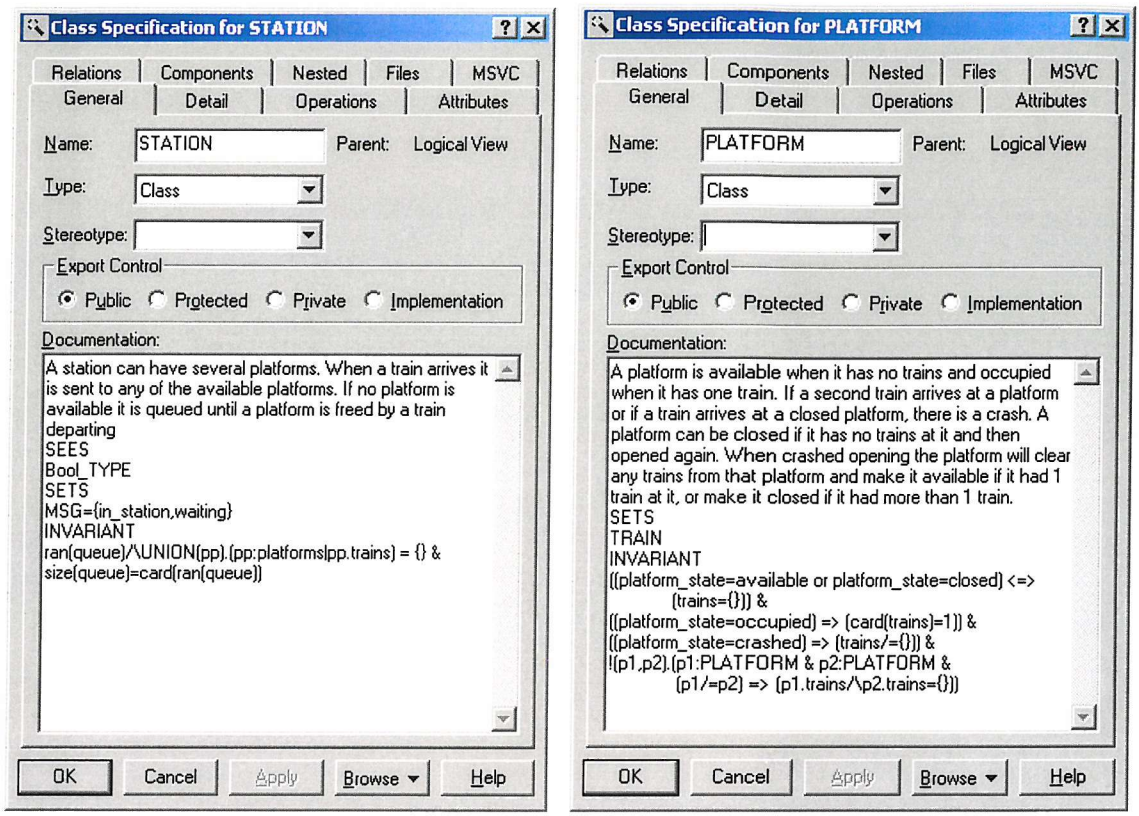


Fig. 7.4 Class specification windows for the classes STATION and PLATFORM

The multiplicity of the STATION class has been set to 1 by setting the multiplicity field in the detail tab of the class's specification box (not shown). This will prevent the U2B translation from modelling instances of the class.

Each operation of the class also has a Rose Specification window with appropriate tabs for the definition of the operation. The operation precondition and body, shown in Fig. 7.5, are taken from the precondition and semantics tabs of the specification for the `train_arrives` operation in class `STATION`. The precondition states that `tt` must not belong to the range of queue and it must not belong to the union of the set `trains` for all platforms associated with this station. That is, the arriving train must not be waiting to get into the station or at a platform already. If an empty platform exists at this station, the operation sends the train to any such empty platform and returns the outcome `in_station`. Note that the arrival at a platform is handled by calling

the `arrival` operation of class `PLATFORM`, specifying the selected platform, `pp`, using the dot prefix notation. If no platform is available the train is appended to the queue and an outcome, `waiting`, is returned.

```
train_arrives precondition

tt/: ran(queue) &
tt/: UNION(pp).(pp:platforms|pp.trains)

train_arrives semantics

IF #(qq).(qq:platforms & qq.platform_state=available)
THEN
    ANY pp WHERE
         pp:platforms &
         pp.platform_state=available
    THEN
         pp.arrival(tt) ||
         outcome:=in_station
    END
ELSE
    queue:=queue^[tt] ||
    outcome:=waiting
END
```

Fig. 7.5. Precondition and semantics for operation train_arrives of class Station

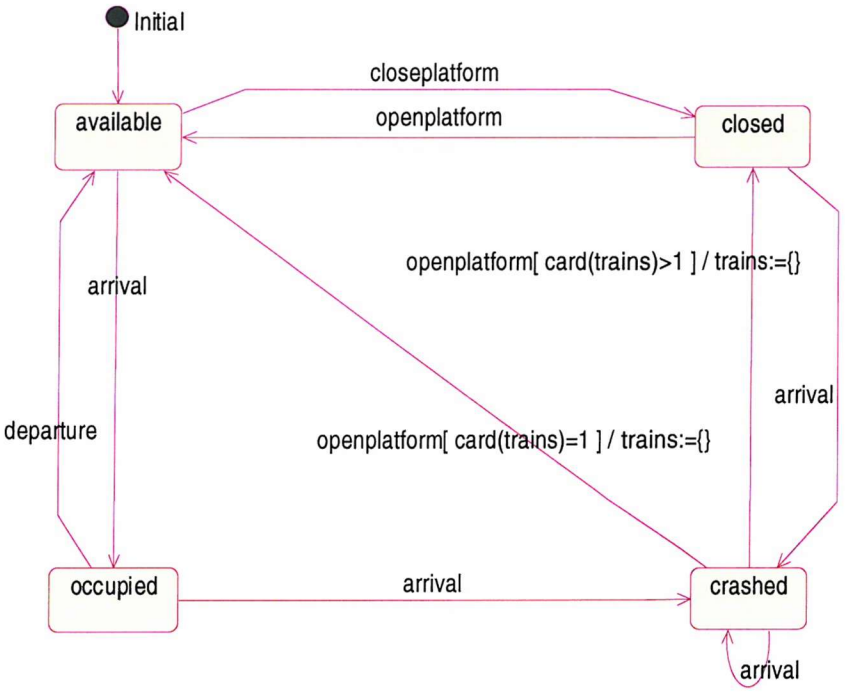


Fig. 7.6. State chart attached to class platform

Fig. 7.6 shows the statechart attached to class PLATFORM. The statechart describes the states that a platform can be in and which transitions between states are possible. Each transition corresponds to an operation of the machine and has its event named after an operation. For example, when a platform is in the state available, two operations are allowed: closeplatform and arrival. Execution of the arrival operation in this state changes the control state to occupied, while executing the closeplatform operation changes the control state to closed. The transitions associated with the operation openplatform have additional guards which determine which of the transitions will be taken when openplatform occurs from the state crashed. These transitions also take a different action from the openplatform transition that occurs from the state closed.

Below is shown the B machine for the class PLATFORM.

```
MACHINEPLATFORM
/*" A platform is... etc. "*/

SETS
     PLATFORMSET;
     PLATFORM_STATE={available,closed,occupied,crashed} ;
     TRAIN
```

PLATFORMSET is the set of all possible instances of PLATFORM. PLATFORM_STATE has been generated from the states on the attached statechart. TRAIN has been generated from the SETS machine clause in the class specification documentation window.

```
VARIABLES
  PLATFORMinstances,
  platform_state,
  trains
INVARIANT
  PLATFORMinstances  <: PLATFORMSET &
  platform_state : PLATFORMinstances --> PLATFORM_STATE &
  trains : PLATFORMinstances --> POW(TRAIN) &
```

PLATFORMinstances is a variable subset of PLATFORMSET, representing the current instances of PLATFORM. A variable, platform_state represents the state that each PLATFORMinstance is in. A variable, trains, represents the subset of TRAIN belonging to each instance of PLATFORM.

```
!(thisPlatform).(thisPlatform:PLATFORMinstances =>
 ((platform_state(thisPlatform)=available or
   platform_state(thisPlatform)=closed) <=>
              (trains(thisPlatform)={})) &
 ((platform_state(thisPlatform)=occupied) <=>
              (card(trains(thisPlatform))=1)) &
 ((platform_state(thisPlatform)=crashed) <=>
              (trains(thisPlatform)/={})) &
 !(p1,p2).(p1:PLATFORMinstances & p2:PLATFORMinstances &
              (p1/=p2) => (trains(p1)/\trains(p2)={}))
)
```

Further invariants reflect the invariants specified in the specification documentation text box for class PLATFORM. Universal quantification over PLATFORMinstances has been added and the dot notation of explicit instance references has been converted into parameters.

```
INITIALISATION
     PLATFORMinstances := {} ||
     platform_state := {} || trains := {}
OPERATIONS
 Return  <-- PLATFORMcreate =
  PRE  PLATFORMinstances  /= PLATFORMSET
  THEN
     ANY new
     WHERE
        new : PLATFORMSET - PLATFORMinstances
     THEN
        PLATFORMinstances := PLATFORMinstances \/ {new} ||
        platform_state(new):=available ||
        trains(new):={} ||
        Return := new
     END
  END ;
```

Initially PLATFORMinstances is empty, and hence all variables are empty sets. A create operation is provided which non-deterministically picks any unused instance from PLATFORMSET and initialises state and attribute variables to the initial values given in the statechart and UML class specification respectively

```
arrival (thisPlatform,tt) =
  PRE
    thisPlatform  :  PLATFORMinstances &
    tt:TRAIN &
    tt/:UNION(pp).(pp:PLATFORMinstances|trains(pp))
  THEN
    SELECT platform_state(thisPlatform)=available
    THEN   platform_state(thisPlatform):=occupied
    WHEN   platform_state(thisPlatform)=closed
    THEN   platform_state(thisPlatform):=crashed
    WHEN   platform_state(thisPlatform)=occupied
    THEN   platform_state(thisPlatform):=crashed
    WHEN   platform_state(thisPlatform)=crashed
    THEN   skip
    END ||
    trains(thisPlatform):=trains(thisPlatform) \/ {tt}
  END ;

 openplatform (thisPlatform) =
  PRE
    thisPlatform  :  PLATFORMinstances
  THEN
    SELECT platform_state(thisPlatform)=closed
    THEN   platform_state(thisPlatform):=available
    WHEN   platform_state(thisPlatform)=crashed &
           card(trains(thisPlatform))=1
    THEN   platform_state(thisPlatform):=available ||
           trains(thisPlatform):={}
    WHEN   platform_state(thisPlatform)=crashed &
           card(trains(thisPlatform))>1
    THEN   platform_state(thisPlatform):=closed ||
           trains(thisPlatform):={}
    END
  END
END
```

Operations are defined for each operation of the class. (Only two operations are shown). A parameter, `thisPlatform`, has been added to define the instance that the operation is to operate on; this is implicit in the UML class diagram version. The type of this and any other parameters are defined as operation preconditions. Other preconditions are derived from the operation preconditions specification window of the class diagram. The operation body is derived from the operation semantics specification window of the class diagram (see Fig. 7.5) and from the statechart. The body of operation `arrival` consists of a 'SELECT' guard, which defines the state transitions that take place when this operation (event) occurs, and, in parallel, the action specified in the semantics window, which occurs for each state transition. In operation `openplatform` additional conditions determine the final state when the initial state is `crashed` leading to two different SELECT branches for the `crashed` state.

The B machine for the class `STATION` does not model instances (because the class multiplicity has been set to one) and therefore variables representing attributes and associations are typed

directly rather than as functions. This machine EXTENDS the PLATFORM machine so that it can call operations of PLATFORM if required.

```
MACHINESTATION
/*" A station can have several platforms...etc. "*/

SEES
     Bool_TYPE
EXTENDS
     PLATFORM
SETS
     MSG={in_station,waiting}
VARIABLES
     queue,
     platforms
INVARIANT
     queue : seq(TRAIN) &
     platforms : POW(PLATFORMinstances) &
     ran(queue)/\UNION(pp).(pp:platforms|trains(pp)) = {} &
     size(queue)=card(ran(queue))
INITIALISATION
     queue:=<> ||
     platforms := {}
```

A variable, platforms, which is a subset of PLATFORMinstances, is used to model the association with class PLATFORM. No create operation is generated because instances are not modelled. Instead, the variables are initialised in the INITIALISATION clause to the values specified in the class diagram (in this case both are initialised to empty). The precondition and semantics for the train_arrives operation of STATION shown in Fig. 7.5 is as follows:

```
outcome <--  train_arrives (tt) =
  PRE
    tt:TRAIN &
    tt/: ran(queue) &
    tt/: UNION(pp).(pp:platform|trains(pp))
  THEN
    IF #(qq).(qq:platforms &
           platform_state(qq)=available)
    THEN
        ANY pp WHERE
             pp:platforms &
             platform_state(pp)=available
        THEN
             arrival(pp,tt)  ||  outcome:=in_station
        END
    ELSE
        queue:=queue^[tt]  ||  outcome:=waiting
    END
  END
;
```

This operation makes various references to components of the PLATFORM class, including reading the state variable platform_state and calling its arrival operation. The object-oriented dot notation of Fig. 7.5 has been changed to standard B notation.

This example demonstrates how statecharts can be used to specify the behaviour of a class in terms of the state changes that occur when its operations are invoked. The example shows how common information specified in the operation semantics is composed with the statechart defined operation actions when the model is translated to B. One of the benefits of using statecharts in this way is that an overall view of the behaviour of a class through its combined operations is presented

7.3 Teletext

The following example is a simplified version of a teletext page selection system. Pages are selected in a two tier hierarchical pair of columns where the selected group determines the column of pages available for selection. Selection of items in each column is made by *left*, *right*, *up* and *down* arrow keys. An *ok* key confirms the selected page. The example illustrates how statecharts as well as semantics windows can be used flexibly with suitable machine definitions to define class behaviour. The example also uncovers and illustrates some limitations with the current translation, which will be the subject of future work.

The system was modelled as two classes (Fig. 7.7), OVERVIEWTABLE (of which there is only one instance) and COLUMN of which there are two distinct instances each being associated with the overview table in a different role. The COLUMN class models the scrolling behaviour of a column so that all pages are accessible even if the display is too small to show the complete column. (Note that in the following, to avoid confusion with up and down arrow keys and cursor movements, when we refer to scrolling movements we refer to the movement of the displayed portion rather than the column movement behind the display. Hence scroll down means that the column moves up relative to the display). The class has operations, Up and Dn, to respond to up and down commands. It also has a Reset operation to re-initialise the column (for example the page column is reset every time a new group is selected in the group column). The class also keeps track of the index of the item currently selected in that column. Note that the column does not contain the actual sequence of pages of a column; these belong to the other class. The OVERVIEWTABLE class has two attributes, GPS and G2P, that contain the current list of groups and a mapping that gives the list of pages corresponding to each group. The type PAGE is declared as a deferred set in the class specification. The attributes are initialised via the operation Init. The operation OK, corresponding to the OK key pressed event, returns the

currently selected page. The operation `Display` returns the information necessary to produce a display corresponding to the current selection state. That is, the current list of groups and the one that is currently selected, and the current list of pages and the one that is currently selected. The remaining operations define responses to the cursor movement (arrow) keys and are defined by a statechart described below.

The remaining symbol, `SQUASH`, is a parameterised class utility used to define a function constant needed for manipulating sequences.

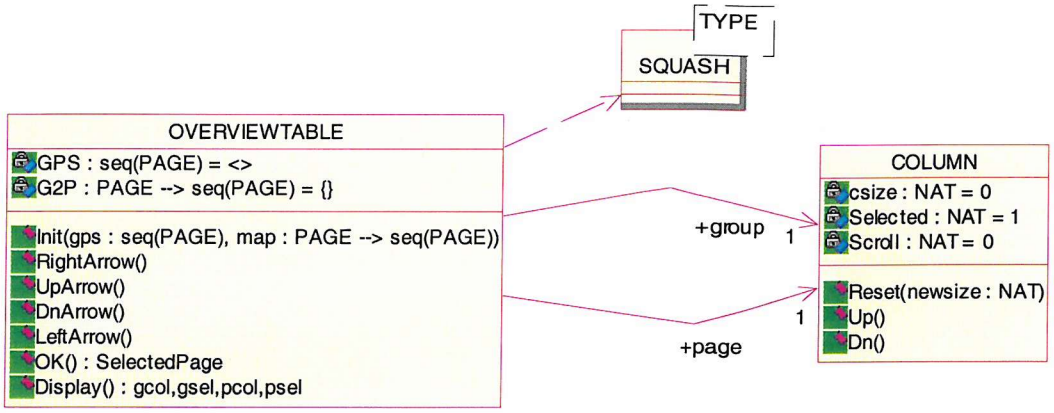


Fig. 7.7 Class Diagram for Example Teletext

The specification for class `COLUMN` (Fig. 7.8) contains some definitions that are used in the statechart describing its behaviour. The definitions aid readability as well as making the amount of text on the diagram more readable and mitigating repetition of expressions on different transitions. We found the use of definitions in this way essential to make state diagrams more manageable.

(Note that the translation of dot notations to parameterisation is currently not very robust. For example the dots in number ranges can be mistaken for explicit instance references preventing the addition of a 'thisClass' type instance reference. To avoid this we have put brackets around the upper bound of the number range.)

The definitions introduce the concept of cursor position on the display. This is an essential concept in the display of a column. However, `Scroll` and `Selected` (item in list) are even more fundamental concepts within the aims of the system, and, as shown in Fig 7.9, cursor position can be calculated from them. Since redundant information necessitates invariants to ensure consistency, which generate additional proof obligations, it is undesirable to introduce `Cursor` as an attribute. Instead we use a definition, which allows us to write '`Cursor`' instead of '`Selected-Scroll`' to aid readability. Two types of response to a vertical movement are defined, a cursor movement when the cursor changes position within the display area to select a

new item and a scrolling movement where the information on the display moves up or down and the cursor position remains constant and thereby selects a new item. The final definition, a boolean expression `NrBottom`, illustrates the use of definitions to aid readability in transition guards.

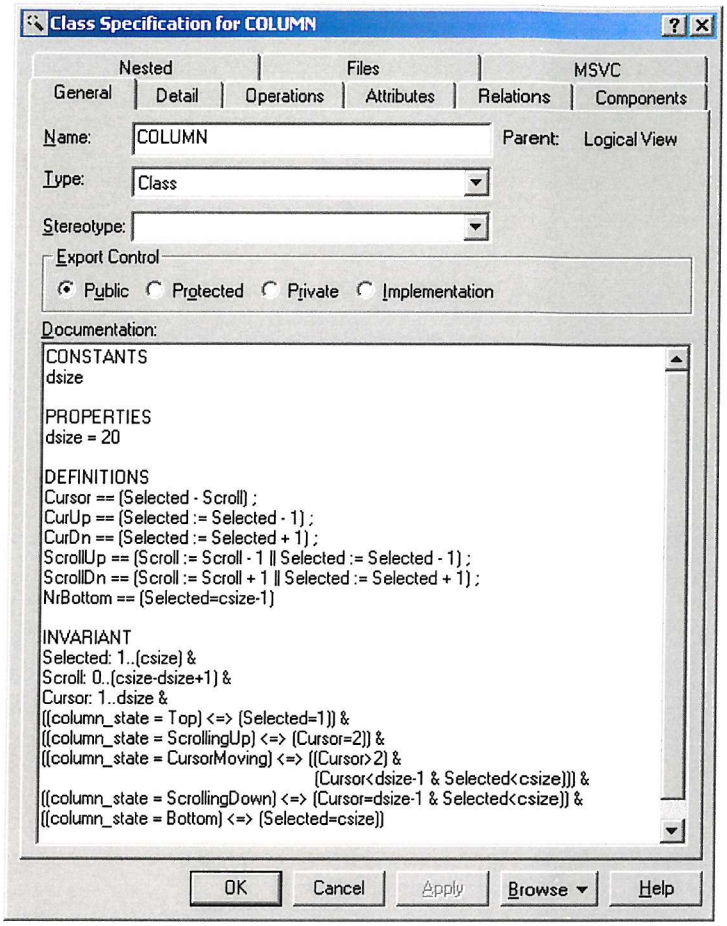


Fig. 7.8 Class Specification for the COLUMN class

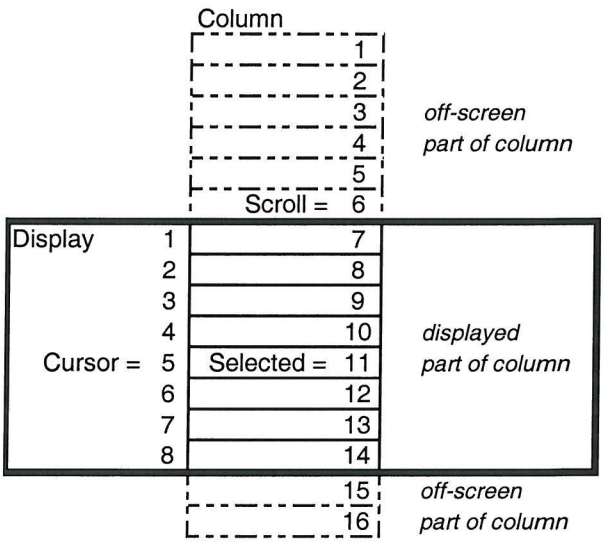


Fig 7.9 Relationship between *Selected*, *Scroll* and *Cursor*

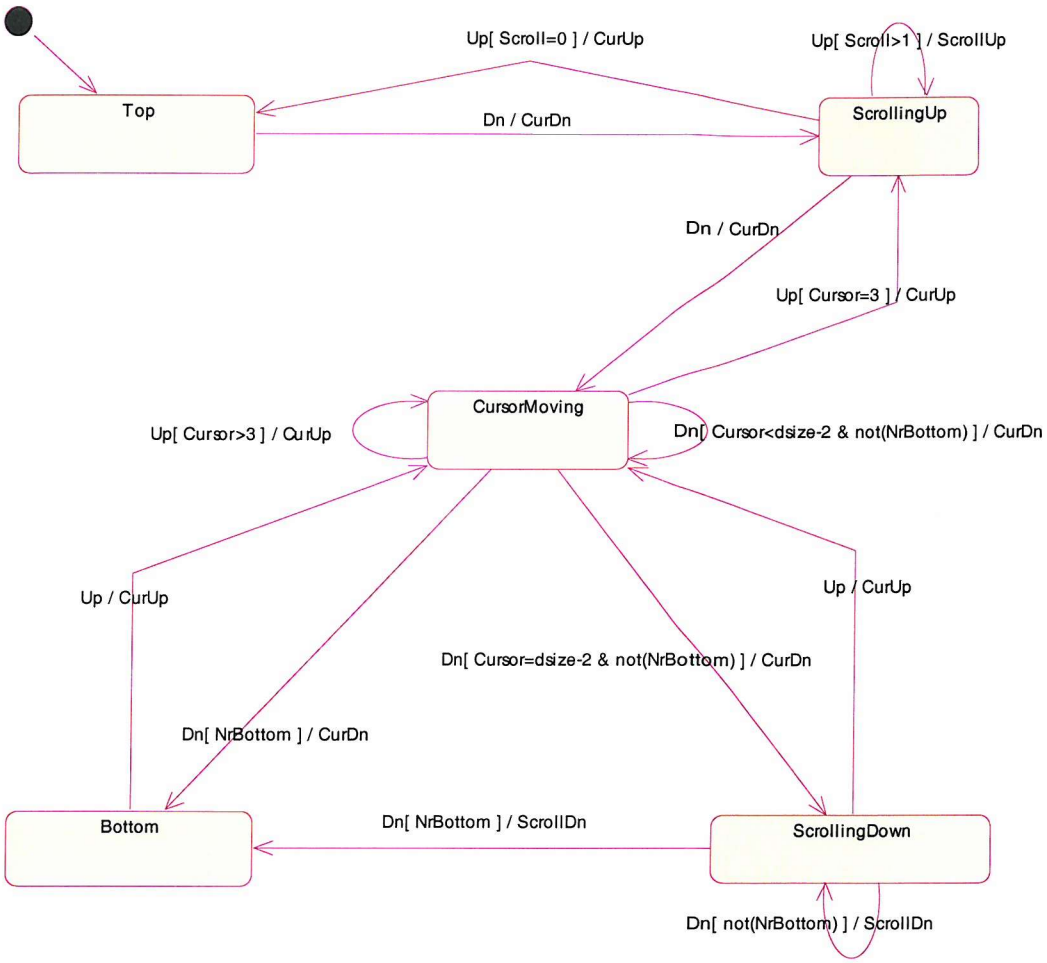


Fig. 7.10 State chart model of semantics of *Up* and *Dn* operations of class *COLUMN*

The statechart (Fig. 7.10) describes the behaviour of the operations `Up` and `Dn`. Initially the `column_state` is `Top`. An invariant requires the equivalence of this state to the condition where the `Selected` item is the top of the column. From this state the operation `Up` is not available. (For a less abstract model we might wish to allow the Up event to occur and specify that nothing happens). A `Dn` event from `Top` will move the cursor down one place (i.e. increase `Selected` item by one while leaving `Scroll` at zero) and change the state to `ScrollingUp`. Since `Scroll` is zero a subsequent `Up` event would return the state straight back to `Top`. In general, while `Scroll` is greater then zero, `Up` events in the `ScrollingUp` state result in the amount of `Scroll` and the `Selected` item both decreasing by one (i.e. `Cursor` remains at position 2 on the display). A `Dn` event from `ScrollingUp` moves the cursor down one place and changes the state to `CursorMoving`. In this state further `Dn` events will keep moving the cursor down until it is two places from the bottom of the display. At this point, if the selected item is not the one before last in the column the cursor is moved down one position and the state

is changed to ScrollingDown. From ScrollingDown further Dn events will cause Scroll and Selected to both be increased by one until Selected is one before the last item in the column when, in addition, the state will change to Bottom. From the state Bottom, further Dn events do not occur but an Up event will move the cursor up one position and change the state straight to CursorMoving (we circumnavigate ScrollingDown because we have already scrolled past the bottom of the column and ScrollingDown changes to Bottom via another ScollDn action). CursorMoving can return straight to Bottom, with an increase cursor action, if the cursor is two places from the bottom of the display and Selected is one before the last item in the column.

In some problems a clear concept of state is involved with a few discrete states that segregate the system behaviour cleanly into different conditions. In these cases a statechart is clearly an appropriate means of description. In other cases this is not the case and a textual form of description is clearer. The example here tends toward the latter. The example statechart provides a visualisation of some of the conditions that the column can exist in, but the behaviour in response to an event is often the same for several states. The statechart requires a substantial investment of effort in order to understand it and to glean significant information from it. The equivalent textual specifications for the two operations are shown below (Fig. 7.11) and can be described as follows. Dn: Downward movements can occur while *Selected* has not reached the last item in the column. The cursor is increased to the next item on the display unless it is one before the bottom of the display in which case a scroll down is made instead. Up: Upward movements can occur while *Selected* has not reached the first item in the column. The cursor is decreased to the previous item on the display unless it is one below the top of the display and the display has been scrolled (*Scroll*>0) in which case a scroll up is made instead.

```
Dn
SELECT Selected<csize THEN
    IF Cursor = dsize-1 THEN
        ScrollDn
    ELSE
        CurDn
    END
END

Up
SELECT Selected>1 THEN
    IF Cursor = 2 & Scroll>0 THEN
        ScrollUp
    ELSE
        CurUp
    END
END
```

Fig. 7.11 Equivalent textual semantics definitions for operations *Up* and *Dn*

The specification for the class OVERVIEWTABLE (Fig. 7.12) contains two definitions used in the statechart describing the class's behaviour. These definitions conditionally select the next, or previous (respectively), group and reset the column of pages accordingly.

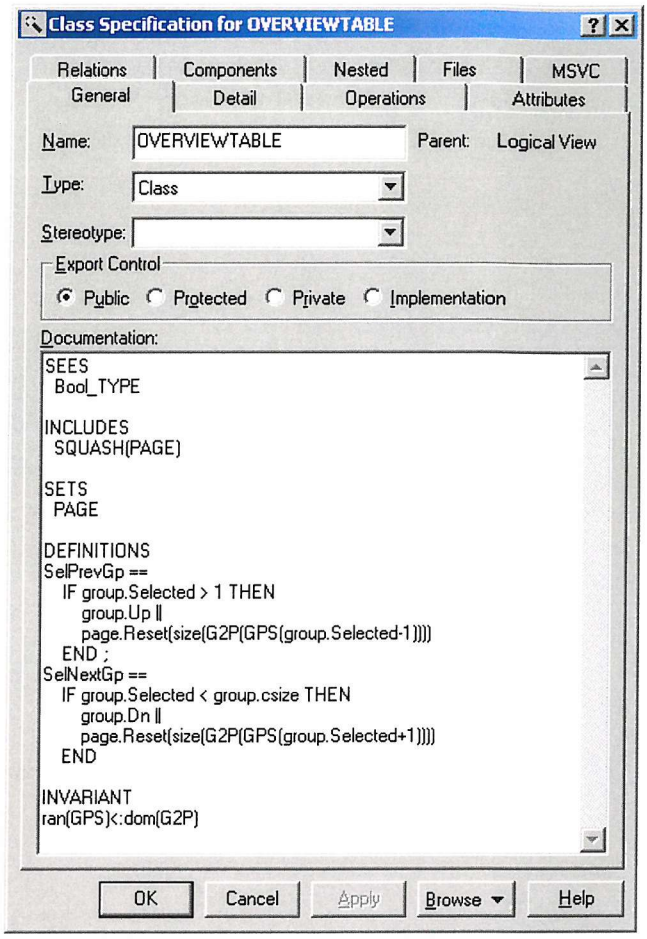


Fig. 7.12 Class Specification for *OVERVIEWTABLE*

The definitions illustrate a problem with the relationship structure between classes. When the group is changed via the operation call `group.Up` (i.e. the `Up` operation, of class `COLUMN`, with instance parameter `group`) the pages column has to change simultaneously via the operation call `page.Reset` (i.e. the `Reset` operation of class `COLUMN` with instance parameter `page`). Since, to ensure consistency, B does not allow the simultaneous invocation of two operations in the same machine, the definitions are illegal. The problem is inherent in any association between two classes. Systems can only be modelled when each event alters the state of at most one instance of each associated class. The class relationship structure shown in the class diagram is a special case where instances are known via different association roles. A solution to the problem in this case would be to model the two instances as separate classes. Future work on inheritance and on parameterisation of classes would mitigate the consequent repetition. In the more general case where there is no suitable role distinction between the instances being

simultaneously altered, a possible solution might be to allow the illegal form in the UML operation semantics (where an instance based reference is beneficial) but detect it during translation and convert it to an allowable form (such as a single operation that accepts two instance references).

The invariant ensures that the group's pages that are in the `GPS` attribute are contained in the domain of the groups to pages mapping `G2P`.

The statechart (Fig. 7.13) describes the behaviour of the class in response to the four arrow keys. Two states are used in the statechart which correspond to which column is the focus for up and down arrows. Initially it is the `Group` state. `LeftArrow` and `RightArrow` events switch between the `Group` and `Page` states. `LeftArrow` events only occur while in the `Page` state and `RightArrow` events only occur while in the `Group` state. While in the `Group` state, `UpArrow` and `DownArrow` events result in the actions defined by `SelPrevGp` and `SelNextGp` respectively. While in the `Page` state they invoke the `Up` and `Dn` operations of the `COLUMN` class upon the instance, `page`. Note that the guards in the `Up` and `Dn` operations of the `COLUMN` class mean that `UpArrow` and `DnArrow` events only occur when a new selection can be made. It is not necessary to re-specify these guards. Currently the state transition to final state (event, `OK`) has no meaning and is ignored. (We anticipate that its meaning should be that the state model will not respond to further events except perhaps an initialisation event to return it to its initial state).

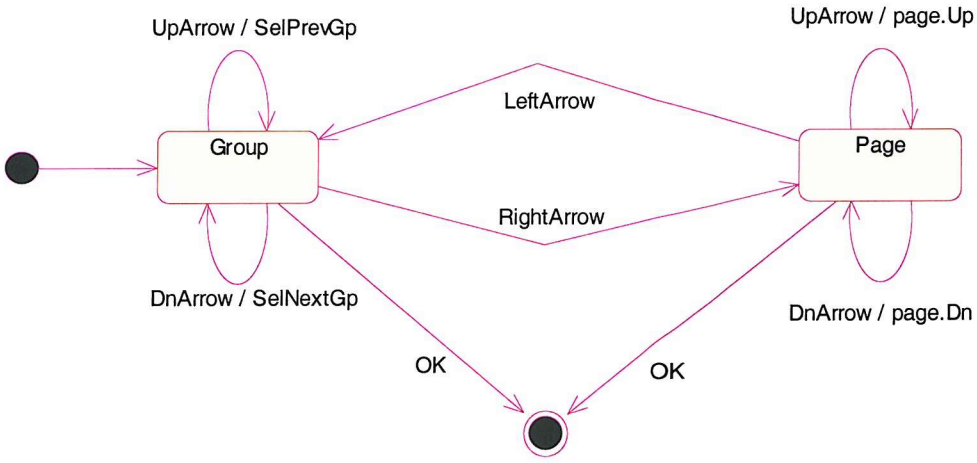


Fig. 7.13 State chart model for the *OVERVIEWTABLE* class

Finally, the operation semantics (from the operation specifications) for the `OK` and `Display` operations are shown below (Fig. 7.14 and Fig. 7.15).

```
OK
SelectedPage:= GPS(page.Selected)
```

Fig. 7.14 Semantics for operation OK of *OVERVIEWTABLE* class

```
Display
gcol:= squash(group.Scroll..(group.Scroll+dsize) <| GPS) ||
gsel:= group.Selected ||
pcol:= squash(page.Scroll..(page.Scroll+dsize) <|
G2P(GPS(group.Selected))) ||
psel:= page.Selected
```

Fig. 7.15 Semantics for operation *Display* of *OVERVIEWTABLE* class

The operation `Display` uses a function, squash, which converts a function whose domain is a set of integers into a sequence by replacing the smallest integer in the domain with one, the second smallest with two, and so on. This function is not available in B. We defined it as a constant in a separate parameterised machine represented by a parameterised class utility (Fig. 7.7). The parameter defines the type for the range of the sequence and enables us to define the squash function generically, rather than specifically for the type, `PAGE`, that we currently require it for. Parameterisation is currently not supported by the U2B translator. For now we manually add `(TYPE)` to the machine header of `SQUASH.mch`. The content of `SQUASH.mch` is copied from the class utility specification window shown in Fig. 7.16.

This example explores the practicalities of using statecharts to model the behaviour of classes and how this information is composed with textually specified operation semantics. We have found that displaying guard and action information in a statechart can become unwieldy but this can be solved by using declarations in the class specification. We have found that statecharts are not always the most appropriate specification medium. In some cases the textual operation specifications are clearer and more succinct and in many cases a combination of the two forms will be most appropriate. Currently we have assumed an event-based approach that is more appropriate for abstract models of observed systems rather than specifications of implementations.

The example shows that class relationships where more than one instance of an associated class is modified simultaneously cannot be translated to valid B by the current translator. Future work will include developing the translation rules to solve this problem.

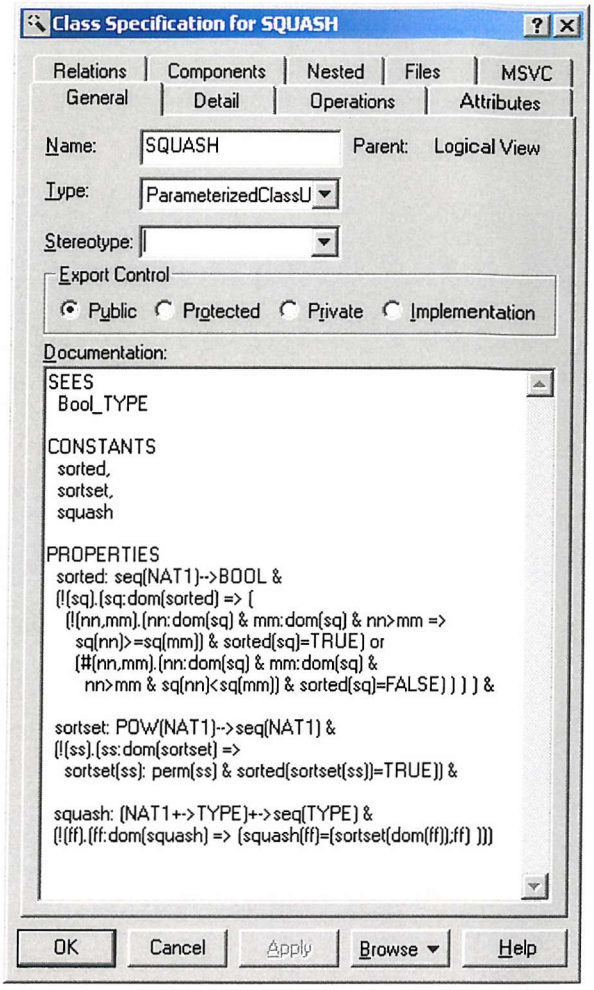


Fig. 7.16 Class Specification for *SQUASH*

7.4 Summary

In this chapter we have presented example specifications written in B-UML that illustrate its use. We have illustrated how a formal specification model can be built up within the UML class diagram and statechart notations using the specification windows that Rational Rose provides for textual annotation. The examples have also demonstrated the importance of choosing appropriate notations for different problems and hence the significance of B-UML's flexible combination of statechart and operation semantics for specifying the behaviour of classes. The examples have also raised some limitations of the current translator that will be addressed in future work.

Chapter 8

Related Work

In this chapter we summarise related work that is similar to, or relevant to, the U2B translation. In summarising each work we point out similarities and differences from U2B. Table 8.1 Lists work we consider to be relevant.

	Reference	From	To	Tool
		UML's official formal notation		
OCL	**Warmer and Kleppe**	Constraints in UML	N/A	n/a
		Tool supported translations of UML		
RoZ	**Dupuy and du Bousquet**	UML CD	Z	y
IFAD	**IFAD**	UML CD	VDM++	y
		Translations to B		
	Nagui-Raiss	ERD	B	n
CEDRIC-HE	**Facon, Laleau and Nguyen**	UML CD	B	n
	Facon, Laleau and Mammar	UML CD, SD and ID	B	p
LORIA DEDALE	**Meyer and Souquières**	UML CD, SD	B	n
	Meyer and Santen	Class hierarchies	B	p
	DEDALE: Ledang and Souquières	UML CD and ID	B	p
iSTATE	**Sekerinski and Zurob**	SD	B	p
		Translations to other notations		
	Kim and Carrington	UML CD	Obj Z	n
	France Bruel Larrondo-Petrie and Shroff	UML CD	Z	n
	DeLoach, Smith and Hartrum	CD and SD	O-Slang	n
	Börger, Cavarra and Riccobene	SD	ASM	n
	Bolton and Davies	UML AD	Z & CSP	n
SOFL	**Liu & Sun**	Integrated SM-OO-FM	-	n/a

(CD=Class Diagram, SD=Statechart Diagram, ID=Interaction Diagram, AD=Activity Diagram, ERD=Entity Relationship Diagram, p = proposed)

Table 8.1 Related Work

8.1 OCL

The diagrammatic notations of the UML are not sufficient to express all the information needed in a model. Typically, annotations of constraints, invariants and operation semantics are needed to complete the information in a specification. The UML therefore contains within its definition, a precise textual notation in which these annotations can be expressed. This notation is called

the Object Constraint Language (OCL) (Warmer & Kleppe 1998). OCL is a formal declarative notation but uses few mathematical symbols. It was conceived with the aim that it should be precise but approachable to engineers and programmers without experience in formal notations or extensive mathematical training. It also follows an object-oriented style dot notation for accessing attributes, associations and operations of an instance. While OCL may have achieved its aim of being approachable to typical programmers, a number of problems have been raised by Vaziri and Jackson (1999) including:

a) OCL has an implementation style in that it uses operations in constraints. Operations can be undefined (e.g. if an infinite loop is caused) leading to constraints being undefined.

b) OCL expressions are overly verbose due to frequent use of coercions (oclIsKindOf). Classes are not treated as sets of objects and hence set operators cannot be used.

c) OCL constraints can be difficult to read due to stacking (via navigation) of quantifiers and collection operators, but not of logical operators.

d) OCL is not a stand-alone language. The notation is intended to apply constraints to objects described in the other notations of the UML. Therefore it relies on the diagrammatic specifications of entities to which its constraints can be applied.

We see d) as being the most significant problem with OCL since it is difficult to reason about properties when a complete textual description is not available. Even if a complete mathematical specification could be obtained, no logic system or tools are available to enable mathematical manipulation. Although our main aim is to make formal specifications easier to write we do not wish to sacrifice one of the important benefits of formal specification in order to achieve it. A UML model with OCL constraints is not a complete formal specification and so does not meet our aims even though, at first sight, it appears to provide a similar type of modelling notation.

8.2 RoZ

RoZ (Dupuy and du-Bousquet, 2000) translates Rational Rose class diagrams to Z specifications. Constraints representing invariants on the attributes and associations of a class can be expressed in Z Latex, in the class diagram specification documentation windows. This is similar to our approach except that, in RoZ, constraints for a class may be in the specification windows for the attributes or associations to which they relate. Upon translation to Z, these constraints are collated as the predicate for a schema describing the attributes of an instance of

the class. Only class invariants (i.e. constraints between different instances of a class) are written in the class specification window. These are used as the predicate in a schema describing the set of instances of the class (the attribute schema being used as the element type for the set of instances). A minor inconsistency is that a constraint between two attributes of a single instance has to be placed in one of the attribute specifications (so that it is translated into the attribute's schema). This is because any predicate in the class' specification will be translated into the instance's schema. Type definitions for attributes have to be added in a separate text file. Our approach was to define them in the specification of the class that used them or, if used by several classes, in a class utility.

Basic operations to modify each attribute and to add and remove an instance of the class are generated automatically. The behaviour of the generated operations is defined, using Z Latex (Spivey 1990), in the post conditions tab of their specification window. (Currently we do not generate basic operations automatically, however this could easily be added. We generate operation signatures automatically where they are not present in a class but appear in a statechart attached to that class).

The class operations are translated into Z schemas using the postcondition predicates. Attribute modifying operations, which are recognised by a keyword 'intension operation' embedded in their semantics field, are translated to a schema that changes the attribute schema. These operations are promoted via a general-purpose promotion schema ('promotion operation'). Operations that change the set of instances of a class ('extension operation') are translated into schemas that alter the instance's schema of the class. Non-basic operations can be added to the class manually and these can have the above types as well as 'composed operation' for an operation that is composed from other operations.

Note that for abstract classes (an abstract class in UML is one that doesn't have any instances), the instances schema and operation promotions are not generated. This is similar to our singular machine (representing a class that has multiplicity 1), which does not model instances explicitly. However we have taken the approach that implicitly one instance exists, currently we do not support abstract classes.

Finally, associations are represented by schemas that define the relationship between the classes at their ends. The two roles are modelled by functions between the class type (attribute) schemas with finite powerset used to represent multiplicities. Predicates reflect any constraints attached to the association and a predicate defines the inverse relationship between the 2 roles. (It is not clear to us how mutable associations could be accessed from operations).

The translation also handles inheritance, which we have not tackled as yet.

Precondition validation theorems can be automatically generated for proof with the Z-EVES theorem prover. This is not needed for our translation to B because the B-Toolkit and Atelier-B contain facilities to generate proof obligations.

The RoZ tool is similar to our approach in its implementation techniques, such as the use of Rational Rose scripting language, embedding of invariant text and operation semantics in specification windows. RoZ has some features, such as basic operation generation, that we have not tackled but lacks the ability to use statecharts to define class semantics that U2B has. RoZ does not constrain models to hierarchical structures in the way that U2B does. RoZ appears to treat associations as a higher level (above the class layer) rather than a navigable (by operations) link to an associated class' attributes and operations.

8.3 IFAD Rose-VDM++ Link

IFAD's VDM++ (IFAD 2000a) is an object-oriented extension of VDM. A tool is provided which performs syntax and type checking and code generation. IFAD provide an extension to the tool (IFAD 2000b) that enables conversion to and from Rational Rose class diagrams. The tool is also capable of merging specifications that exist in both formats. Conversion is straightforward because the formal notation (VDM++) is object-oriented and therefore, most UML concepts have a corresponding feature in VDM++. In some cases, however, features of VDM++ are not represented directly in UML and a stereotype is used to make distinctions. For example, class values of VDM++ are represented by a UML attribute with stereotype <*value*>. The stereotype distinguishes it from an instance variable. Stereotypes are also used to explicitly distinguish between operations and functions. Since UML does not provide a way to define a result identifier, the special identifier RESULT is used in pre and post conditions

However, only class diagrams are converted and operation semantics are not handled in the UML representation. Hence, the tool allows the designer to develop a model of a system in terms of the classes, associations and operations but doesn't allow the operation behaviour to be specified within the visual representation. It is necessary to convert to the VDM++ form in order to add behaviour. Similarly, invariants are not represented in the UML form. A link to the VDM++ file representation of each class is embedded as an 'external file' for each class. This ensures that the behavioural information added to the VDM++ version remains associated with its UML class. The tool provides reverse engineering facilities to update the UML model for alterations.

The method provides most of what is needed to provide a visual formal specification tool, but does not treat the UML version as the primary specification medium. UML is seen as an

ancillary form and hence the relatively simple step of making it a complete specification has not been taken.

8.4 Other Work on Translating to B

Several groups have proposed translations from object-oriented notations to B. As well as those discussed below, see earlier work by Nagui-Raiss (1994), Shore (1996) and Lano (1996). The suggestions for modelling the static class data structure and relationships are similar to each other and are the basis for our own approach. Our approach differs from these because our aim is to provide a graphical notation for expressing B specifications rather than a formal representation of a UML model. The main difficulties in mapping from classes into machines are in representing mutable associations and operation behaviour. This is because of the restrictions that B imposes in order to ensure compositionality of proof. Whereas most groups attempt to accommodate all valid class structures as far as possible, we allow only those UML models that have natural B representations. Hence we impose restrictions on our UML models to only allow strictly hierarchical structures with uni-directional navigable associations.

We look at the major groups that have contributed in the past and are continuing ongoing research in the area. The methods differ for modelling the dynamic behaviour represented in UML operations. At the time of writing, none of these groups had a translation tool available for evaluation, although all have proposed them or claim to be in the process of developing them.

8.4.1 Work at CEDRIC-IIE Laboratory

Researchers at the CEDRIC-IIE Laboratory have developed schemes for translating UML class diagrams and dynamic behavioural diagrams into B specifications. Facon, Laleau & Nguyen (1996) provide a comprehensive mapping of static class diagram features into B and structure this into machines. Later work at CEDRIC has concentrated on Information systems and database applications (Facon, Laleau, & Mammar, 1999) that are data-centric and generally involve simple basic operations. These types of systems involve a high degree of data relationships modelling and our approach of restricting the use of UML would probably be intolerable. Conversely they require only simple operations and so our use of operation behaviour modelling techniques would be largely redundant. The approach taken at CEDRIC has been to automatically define basic operations of a class according to class properties such as mutability and multiplicity. Class statecharts are then used to define how external (to the class) events invoke the basic operations of the class according to state and guard conditions. Collaboration diagrams define which class events occur in response to each external (to the system) transaction.

External, use case, transactions with the system are described with functional sequence diagrams (i.e. a sequence diagram involving users and the system). Each step on a functional sequence diagram is a transaction message that is further described by a simplified collaboration diagram. The collaboration diagram identifies a system level operation and its implementation in terms of events at the class statecharts level. (Note that the sequence diagram itself is not represented in B since the aim of the translation is to check the consistency of data modifications rather than to model functional scenarios).

Thus, the hierarchy of system behaviour is represented in layers made up of different UML modelling notations (collaboration, state and class) rather than by imposing hierarchy in the class structure as we do. Functionality is still largely encapsulated within the class behaviour, but the statechart describes an additional layer of class behaviour that is not represented by operations shown in the class diagram. A third layer describes functionality that involves more than one class. The CEDRIC approach is more suited to data intensive systems that fit a collaborative class oriented description whereas our approach is more suited to process intensive systems where the emphasis is on process/data encapsulation.

8.4.2 Work at LORIA – Universite Nancy

Meyer & Souquières (1999) proposed a method for transforming OMT diagrams (on which UML class diagrams are based) including operations and dynamic behaviour expressed in statecharts. Similarly to the approach of CEDRIC-IIE, classes have very basic and simple operations and the class' statechart provides additional functionality by defining the events and state transitions under which these basic operations are used. Unlike CEDRIC-IIE's approach, the statechart layer is represented as operations within the class machine. To avoid calling operations within the same machine, basic operations are translated to definitions (B's equivalent of macros) using a DEFINITIONS clause rather than as B operations. (For different reasons, we have also used this technique to define actions that are repeated in several places on a statechart). The resulting structure of B machines consists of a top-level system machine, a machine for each class (including subclasses and aggregate components) and a machine for each unfixed (or attributed) association. (Associations that have no attributes and are fixed for at least one class are handled within the class for which they are unfixed). The disadvantage of this is that functionality that might be naturally associated within a class is elevated to the top-level machine in order to obtain write access over association links. This is probably more significant in process control applications, where operation behaviour is more complex, than in information systems where the accent is on data maintenance.

Meyer & Santen (2000) go on to describe how Atelier-B can be used to verify behavioural conformance of inheritance (generalisation/specialisation) relationships in a UML class diagram by using the translation proposed by Meyer and Souquières. Currently we are concentrating on issues involved in writing specifications, however, we recognise verification as an important benefit of writing formal specifications. In the translations used for presenting this work, non-basic operations are specified, not in the UML, but by post-translation additions to the B machine. It would be a simple step to attach the operation bodies to the UML classes as we have done but the example chosen illustrates that combinations of basic operations defined in a statechart are not always suitable.

Further work by Ledang and Souquières (2001) considers techniques for arranging non-basic operations into separate machines to comply with the operation calling restrictions of B. The calling sequence defined in a collaboration diagram (which must not contain any cyclic calling dependencies) is analysed and allocated into layers so that,

a) there is no calling-called dependency amongst operations in the same layer;

b) basic operations (which do not call any other operation) are in the bottom layer;

c) system operations (which do not have a calling operation) are in the top layer;

d) operations above the bottom layer only call operations of the next lower layer.

A structure of B machines is constructed with one machine for each layer except at the bottom layer where there is one machine for each class. However, an operation at one level may call several operations at the next lower layer. Since this is not allowed in machines and machine inclusion, implementations and imports are used to define the operations instead.

8.4.3 Sekerinski and Zurob - Statecharts to B

Sekerinski (1998) describes how reactive systems can be designed graphically using statecharts (Harel, 1987) and how these designs can be converted to B for analysis and refinement to code. A full treatment of statecharts is given, including hierarchies, concurrency and various equivalents to shortcuts used in statecharts. An example of a conversion to B is then given.

The treatment differs from ours in that statecharts, although similar to UML state machines, are treated as an independent form of design notation rather than as a subnotation to class diagrams. On the other hand, hierarchical statecharts (i.e. states may have substates) and concurrency (i.e. states may have groups of substates which may progress independently and concurrently) are included. These are areas that we would like to tackle as future work and note that Rational

Rose state machines are able to express both features. In addition, communication between concurrent sub-parts is available via internal events that are generated as part of the action of one transition and are referenced as the event triggering another. This is translated to a B definition in order to ensure the event is not available externally (this also avoids the fact that operations cannot be called from within the same machine). Externally available events are modelled as operations, as we have done. However, the approach is to model the implementation of a reactive component with operations representing called procedures, rather than an action system approach in an event B style with operations representing actions. Therefore operations are treated as procedures with conditional substitutions rather than guarded actions.

Sekerinski and Zurob (2001) go on to describe a meta-model of statecharts via a class diagram with semantics formally defined in a B like notation. A normalisation of statecharts is formally described (to add arrows that may be left out as shortcuts). This is the first stage in translation to B. A flawed condition, when states are unreachable, is formally described. This condition is translated to B but warnings are given. Finally, illegal statechart conditions (such as transitions between two concurrent groups of sub-states), which prevent translation to B, are described.

8.5 Translations to Other Formal Notations

We have reported on OCL, two well developed tools that transform UML class diagrams into formal notations (not B) and several groups that have proposed translations to B from various combinations of the UML notations, class diagrams, statecharts and interaction diagrams. Others have proposed translations of UML notations into formal notations other than B.

France, Bruel, Larondo-Petrie and Shroff (1997) propose a formalisation of UML class diagrams in Z. This work focuses on formalising the UML, rather than using UML to assist in formal specification, but in the process translation rules are developed and illustrated by example. The use of Z, and hence the freedom from the proof composition restrictions of B, enables more complex class diagram structures to be catered for. This is developed in France (1999) where the equivalent Z specification is used to analyse the semantics of class diagram structures. Again the focus is on defining a precise semantics for the UML. For example outline proofs are given for various inferences that can be made about incomplete class diagrams involving generalisation relationships.

Kim and Carrington (2000) give a formal definition of UML class diagrams using Object Z. They also provide a formalised meta-model of Object Z and hence a formal mapping from class

diagrams to Object Z. The object-oriented facilities of ObjectZ make the translation more natural and simpler than that for Z.

Börger, Cavarra and Riccobene (2000 and 2001) have used Abstract State Machines (ASMs) (Gurevich, 1995) to rigorously define UML statecharts and activity diagrams. This work is strong in its treatment of the UML's integration of statecharts with the object model, including the concepts of events, actions and activities. This will be relevant to future work enhancing the U2B translation tool to cover these features of UML statecharts and/or activity diagrams.

Bolton and Davies (2000) present a formal semantics for UML activity diagrams using a combination of Z and CSP. They use Z to model the static objects within an (possibly hierarchical) activity diagram and parallel CSP processes, one per state, to model its behaviour. The synchronisation of the CSP processes, upon their respective alphabets, models the sequence of events expressed by the transitions between states. Since a class diagram can also be translated into Z, it could be verified for consistency with a requirements specification expressed as an activity diagram by translating the activity diagram into Z/CSP.

DeLoach, Smith and Hartrum (2001) define a translation from UML class diagrams and statecharts into O-SLANG (DeLoach and Hartrum, 2000). O-SLANG is an object-oriented extension of Slang; a theory based algebraic specification language. The translations were verified by defining a formal semantics for the UML notations and mapping both O-SLANG and UML to those semantics. A prototype system has been developed to demonstrate automation of the transformation. The motivation is similar to ours, to facilitate the creation of formal specifications via semi-formal diagrammatic stages. O-SLANG is not as widely used as B but benefits from being specifically designed to describe object-oriented models making translation from UML more natural and complete.

We have reported mostly on translations from existing object-oriented diagrammatic notations to existing or new formal notations. For some researchers the primary aim is to formalise the existing object-oriented notation so that the descriptions that use it are precisely understood. For others the aim is to gain the benefits of an approachable diagrammatic modelling notation while creating a specification in their favoured formal notation. Most researchers opt for the pragmatic route and use an existing object-oriented diagrammatic notation. However, an alternative approach is to propose a new integrated diagrammatic and formal notation, which can then be designed with integration in mind. Liu and Sun (1995) have taken this approach with their SOFL (Structured Object-Oriented Formal Language). Data Flow diagrams are used to decompose the functional requirements into 'condition processes' in a stepwise hierarchical manner. Each condition process on a data flow diagram receives and creates data items. Its

process is described formally via pre and post conditions and it can either be further decomposed via another data flow diagram or can be declared to have an implementation module. Implementation modules are described in an executable programming language that may be structured using procedures and classes. The use of diagrammatic forms is limited to the hierarchical refinement data flow diagrams. Although the diagrams are highly integrated with the formal specification this appears to demote their significance to an outline structure viewer with most reliance placed on the textual form. Object orientation is used only at the (textual) programming level and not represented diagrammatically.

Chapter 9

Conclusions

This chapter summarises the research and how it meets the aims introduced in Chapter 1. Future directions are proposed for development of the B-UML notation and the U2B translator that will facilitate further exploration of the adaptation of UML notations for creating B specifications. Further evaluations of the use of B-UML in realistic situations are proposed.

9.1 Meeting the Research Aims

The overall aim of the research was to explore the barriers to using formal specification techniques. This has been achieved through the following steps.

The first stage of research was an exploratory survey of formal methods practitioners in order to identify some of the main barriers to use. The survey was limited to a small set of experienced users from a range of market sectors. The purpose of the survey was to identify the most relevant issues for further investigation. The survey achieved a broad exploration of the use of formal specifications in industry and identified several possible issues related to barriers to their use. The survey's strength was that it derived empirical evidence from some of the market leading organisations using formal techniques for commercial products. Despite a varied range of market sectors, there was a reasonable degree of convergence in the interviewees' responses. The survey was presented at the Empirical Assessment of Software Engineering (EASE2000) conference (Snook and Harrison, 2000) and published in the journal, Information and Software Technology (Snook and Harrison, 2001a). Glass reported the publication in his newsletter 'The Software Practitioner' (Glass 2001). Glass recognises the contribution of such surveys, saying: "Formal methods have been lauded by academics and ignored by practitioners for over 30 years. Both camps are locked into their positions; almost no one on either side does the deeply-needed evaluative research which could determine which camp is closer to the truth." In our reply, also published in the same newsletter (Glass 2001) we suggested that: "Perhaps academia is not prioritising the problems it researches to the greatest effect. Targeting the pragmatic problems that practitioners initially face would lead to increased interest and funding from industry, and a more widespread take up of formal specification would later lead to faster development of subsequent research areas in formal methods. Our research" (in combining UML with B) "has

attempted to find ways of making formal specification easier or at least more accessible to novices".

From the survey findings, two issues were selected for further investigation. The first was comprehensibility, which was thought not to pose a significant problem for suitably trained software engineers. The second was the difficulty in writing formal specifications, which was thought to be problematic.

The second stage of research was a further investigation into whether the comprehension of a formal specification could be a barrier to their use. Comprehension depends on the skills and training of the reader and so could be a barrier in several different situations such as customer approval of the specification, quality assurance processes as well as the software design and maintenance processes. In this stage of the research we explored comprehensibility of formal specifications by suitably trained software personnel and hence focused on the last of these situations. We devised an experiment that tested the hypothesis that formal specifications are no more difficult to understand than code. The experiment compared subjects understanding of a Z specification with that of its implementation in Java. The experiment was presented at the Empirical Assessment of Software Engineering (EASE2001) conference (Snook and Harrison, 2001b). Subject to the threats to validity discussed in Chapter 4, we found that comprehension is not a barrier for software personnel .

The remainder of the research focused on the second issue selected from the survey. This was that formal specifications are difficult to write. This was recognised as a significant barrier by those interviewed. We looked at the similarities and differences between formal specification and program design and applied the cognitive dimensions framework in order to assess a formal specification notation with respect to exploratory design. We reasoned that the processes involved in formal specification are similar in many respects to that of program design. Both involve the selection of suitable abstractions in an exploratory design phase. We concluded that one of the main differences is that, for program design, tools and notations have been developed to assist in the difficult process of choosing a coherent set of useful abstractions. Experienced formal specifiers may have developed sufficient experience and expertise to be able to form these abstractions mentally, but novices find the task insurmountably difficult. This leads to a strong deterrent to their increased uptake. In order to test this theory, we adapted two notations from the UML (class diagrams and statecharts) so that they could be used to write semi-diagrammatic formal specifications using one of the leading UML design tools, Rational Rose. We call the adapted notation B-UML. Using the extensibility facilities within Rose, we provided a translation facility, U2B, so that the verification benefits of an existing, tool supported, formal notation (B) could be used to verify the B-UML specifications. The

translation also clarified the semantics of B-UML. The notation, B-UML, and its translator, U2B, are not fully developed methods or tools. Rather, they are prototypes used to test the feasibility of using such techniques and whether they are beneficial. We developed several examples that illustrated the approach. One example is a simplified version of an industrial application. B-UML and U2B were presented at the UML 2000 workshop 'Dynamic Behaviour in UML Models: Semantic Questions' (Snook and Butler, 2000) and at the 13th Annual Workshop of the Psychology of Programming Interest Group (PPIG2001) (Snook and Butler, 2001). We felt that specifications were easier to write using B-UML than they would have been in B. However, we recognise that this is a subjective opinion and further evaluation of the technique is required before firm conclusions can be made. This is discussed below. The examples also uncovered limitations in the current method and threw up possible routes for extending and enhancing B-UML and U2B. Subject to further evaluation and development we believe that the research carried out so far supports the hypothesis that modelling notations and tools similar to those used in program design would benefit the difficult task of writing formal specifications.

9.2 Lessons Learned Using B-UML

The use of the UML provides a visual modelling interface that assists in developing a structure for the specification. This is likely to be most significant for programmers who are familiar with using the UML for software design and unpractised at using formal notations. The automatic generation of B machines from the diagrammatic components of a UML model and the isolation of formal annotations for class invariants and operation semantics makes the formal specification more manageable. It may be more difficult to gain a complete view of the specification from the UML model but this is available via the translation to B. State charts can be used successfully to model the behaviour of classes and this information can be combined with textually specified operation semantics. We have found that displaying guard and action information on a statechart can become unwieldy but this can be solved by using B definitions in the class specification. We have found that statecharts are not always the most appropriate specification medium. In some cases the textual operation specifications are clearer and more succinct and in many cases a combination of the two forms will be most appropriate.

In order to achieve compositionality of proof, B contains restrictions on how machines can access the operations of other machines and on simultaneous changes to machine variables. The restrictions are as follows:

1. A machine cannot have more than one other machine that makes calls to its operations. This restriction disallows data sharing involving multiple write access.

2. Operations cannot call other operations within the same machine

3. Each operation may make, at most, one call to the operations of each other machine.

4. Each variable of a machine can be altered by at most, one of the simultaneous substitutions of an operation

Note, however, that a machine can promote an operation of a machine it includes. Promotion is equivalent to defining an operation of the promoting machine that invokes the operation of the included machine.

The first compositionality restriction of B means that the natural mapping of class operations into machine operations (where the machine represents the class) does not permit associations to be altered by both the associated classes. In addition, non-hierarchical class relationship structures, which imply that a class is alterable by more than one other class, are not permitted. Since we were primarily concerned with enhancing the process of creating B specifications, restricting the use of UML class diagrams to match these B restrictions was acceptable. We therefore restricted our models to hierarchical class structures using uni-directional associations. Since these restrictions are equivalent to the restriction in B, we do not expect them to be any more problematic than they are in writing B specifications. Our experiences so far have not revealed any difficulties arising out of these restrictions.

The second compositionality restriction is not restrictive since it can always be avoided by repeating the substitutions of the 'called' operation within the 'calling' operation in place of the call. The disadvantages of repeating blocks of substitutions can be avoided by using B definitions (a DEFINITIONS clause in the class specification window).

The third compositionality restriction is restrictive. Operation semantics where more than one instance of an associated class is modified simultaneously cannot be translated to valid B by the current version of the translator. This restriction is imposed partly by the object-based nature of the modelling. In a normal B specification, the called operation could be designed to modify multiple instances.

Similarly, the fourth compositionality restriction is restrictive if more than one instance of the machine requires modification of the same attribute. Again it is the imposition of an object-based notational style that leads to the problem. In normal B the function representing the mapping from instances to attribute values could be altered using set operators so that all instances were altered within the same substitution.

Future work will include developing the translation rules to solve these problems.

9.3 Further Work

In this section we outline further work that could be done. The section is split into the following subsections: further evaluation of the effectiveness of B-UML in making formal specification easier; further development of B-UML and U2B to solve current deficiencies and extend the notation and techniques. Further work to improve confidence and generalizability of the experimental conclusions concerning comprehensibility of formal specifications was suggested in Chapter 4.

9.3.1 Evaluation of B-UML and U2B

Further evaluation should be carried out to assess the prime motivation for devising a semi-graphical formal notation, that it will make formal specifications easier to create. We envisage two possible forms of evaluation. Firstly, a formal experiment which would involve two groups of subjects writing a formal specification of the same example but one using B-UML and the other using B. Possible dependant variables which would indicate a difference in suitability of the formal notations and supporting tools might be: correctness of the specification, usefulness of the specification for various tasks and time taken to produce the specification. The subjective qualitative opinions of the subjects would also be of interest.

The second form of evaluation we envisage is a case study using the techniques. Ideally, this would involve an independent organisation using B-UML and U2B to write formal specifications that are required for real applications. Some form of comparison with writing formal specifications without the technique is desirable. It is unlikely that any organisation will be able to replicate the case study, but it may be possible to develop part of the project with, and part without, the techniques. The case study should involve some experienced and some novice personnel to evaluate whether the techniques benefit one group more than the other. The evaluation would rely mostly on qualitative feedback augmented by measurements comparing the treated and untreated parts of the project.

Some progress towards such an evaluation (mostly on the use of statecharts) has been made by Åbo Akademi as part of a more general case study (Matisse, 2001). The initial feedback is favourable but the work is at too early a stage to report in this thesis.

9.3.2 Development of B-UML and U2B

The translator could be improved in several areas. Firstly, we intend to improve the way it works. Currently it is a prototype that works by building files representing B machines using the

text replacement facilities of Microsoft Word. The files are added to, and edited, as the program progresses through the UML model. Although this method was quick to implement and achieved the aim of providing a prototype for feasibility testing, the translation is not very robust. For example, when new examples of operation interactions, association navigations and attribute accesses are attempted, there is a high risk that the text replacement commands will not have anticipated the new formats. Before embarking on further enhancements, we intend to re-write the translation so that it builds an internal representation of the B machines before generating the text files. This will also allow us to dispense with any reliance on Word, which will improve the performance of the translation.

Once the method of translation has been strengthened, we intend to enhance it in various ways to extend the facilities for modelling, provide additional checking of the model before and during translation and to facilitate other model based activities such as refinement.

The UML provides options for different types of association relationships. These imply differences in the creation and destruction of instances of the associated class. For example, composition implies that instances should be created and destroyed with the parent class instance. Currently the U2B translator does not do this automatically.

In Chapter 6 we discussed the implications of different association multiplicities for initialisation of newly created instances. In some cases the current translator is unnecessarily restrictive and in a few cases multiplicities are not supported because no valid initialisation is possible with the current options available in the translator. We envisage the addition of (or possibly the selection of appropriate) creation operations to enable new instances, and new sets of instances, to be specified by, and hence used by, a higher-level class. For example the create operation of a higher-level class, `A`, might use the new create operation of class `B` to initialise an association with multiplicities 1..1→1..1 as follows:

```
PRE
  Binstances  /= BSET
THEN
  ANY newB
  WHERE
    newB : BBBSET - BBBinstances
  THEN
    Bcreate (newB) ||
    Aassoc(newA) := newB
  END
END
```

The same class `A` would use the createSet operation of class `B` to initialise an association with multiplicities 0..n→0..n as follows:

```
ANY oldBs,newBs
WHERE
   oldBs:POW(Binstances) &
   newBs:POW(BSET-Binstances)
THEN
   BcreateSet(newBs) ||
   Aassoc(newA):=oldBs \/ newBs
END
```

Other features of UML class diagrams, such as generalisation, class parameterisation and abstract classes have not been considered at present. It may be that these facilities are useful in translation to corresponding B facilities.

Enhancements to tackle the problems described in the previous section concerning simultaneous changes to instances of an associated class and simultaneous alteration of the same attribute for multiple instances of the class are envisaged. It may be possible to allow apparently illegal B forms in the B-UML specification that are converted into a legal B form at translation.

Enhancements to the use of state machines might include the use of hierarchically structured state machines, the use of activity chart constructs such as parallel state transition paths, entry and exit actions, event actions. Currently we assume an event style model of a system. It may be useful to allow the choice between this and a non-event style model.

Other notations within the UML have not been considered. Component diagrams may be useful as a higher-level structuring mechanism as used by Åbo Akademi (Matisse, 2001). It may be useful to use interaction diagrams, as do Ledang and Souquières (2001) for preliminary definition of inter-class operation calling structures.

Enhancements to the facilities provided by the translator may be useful. Currently the translator performs no checking of the model prior to translation. The B-Toolkit or Atelier-B is relied upon to check that the model represents valid B. The translator could provide, at least basic syntax checking. For example, that the class association restrictions discussed earlier in this chapter have been obeyed in the B-UML.

Many of these improvements are currently being worked on as part of the EU projects, MATISSE (Snook and Waldén, 2002) and PUSSEE (PUSSEE, 2002). Current work within the PUSSEE project has concentrated on extending U2B to support the B refinement method (rather than a single layer of specification). The translation described in this thesis concentrates on the abstract machine specification level and de-composition of a large machine into smaller machines based on the UML classes and their relationships. However, the primary decomposition mechanism in B is not the inclusion of other machines but decomposition by

refinement. An abstract machine specification is refined until an implementation specification is reached. The implementation imports other abstract machines that encapsulate its variables. A hierarchy of modules is constructed, each component consisting of a refinement chain, from abstract machine to implementation. Our current work extends U2B so that refinement and implementation can be modelled in UML using realisation relationships between classes. Furthermore, the UML model can be organised hierarchically using packages so that the B project decomposition technique can be employed within a B-UML model. With this extended version of the U2B translator we will investigate the use of B-UML on the industrial case studies provided by the project partners and contrast it with conventional B project developments.

9.4 Conclusion

In this thesis we have considered formal methods, a software engineering technique that has become very popular as an area for academic research but has only been adopted 'sporadically' within industry. It is clear from current literature that there are benefits to the quality of the software produced using this technique, but there are also barriers that prevent widespread use. Using empirical methods we have investigated what industry believes are the barriers to the use of formal methods. We have investigated further, comprehension, an area that might have been seen as a barrier, and decided (in agreement with those in industry) that it is not. We have then investigated, by constructing an example, a possible technique to assist in the construction of formal specifications, which was seen as a barrier by those in industry. We have found this to be of benefit when writing formal specifications. Through our close work with industrial partners we plan to continue to explore the barriers to formal specification and to investigate ways that they can be overcome.

References

Abrial, J.R. (1996) *The B Book - Assigning programs to meanings.* Cambridge University Press.

Abrial, J.R. (2000) Guidelines to formal system studies. Draft Version 2. *http://www.matisse.dera.gov.uk/formal_guidelines_v2.pdf*

Austin, S and Parkin, G. (1993) *Formal methods: a survey,* National Physical Laboratory.

Bakan, D. (1960). The test of significance in psychological research. *Psychological Bulletin, Vol.66, No.6,* pp.423-437.

Basili, R.V., Shull, F. and Lanubile, F. (1999) Building knowledge through families of experiments. *IEEE Transactions on Software Engineering, Vol.25, No.4,* pp.456-473.

B-Core (1996) *B-Toolkit User's Manual*, Release 3.2. B-Core(UK) Ltd, Oxford (UK).

Behm, P., Benoit, P., Faivre, A. and Meynadier, J-M. (1999) Météor:A successful application of B in a large project. In J. Wing, J. Woodcock, J.Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems, FM'99, Vol. I*, Lecture Notes in Computer Science, Vol.1708, Springer-Verlag, pp.369-387.

Bolton, C., Davies, J. (2000) Activity graphs and processes. In W.Grieskamp, T.Santen and B.Stoddart, editors, *Integrated Formal Methods, Second International Conference, IFM 2000.* Lecture Notes in Computer Science, Vol.1945, Springer-Verlag, pp.77-96.

Börger, E., Cavarra, A. and Riccobene, E. (2000) An ASM semantics for UML activity diagrams. In T.Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000,* Lecture Notes in Computer Science, Vol.1816, Springer-Verlag, pp.293-308.

Börger, E., Cavarra, A. and Riccobene, E. (2001) Modeling the dynamics of UML state machines. *Abstract State Machines Theory and Applications. International Workshop, ASM 2000,* Lecture Notes in Computer Science, Vol.1912, Springer-Verlag, pp.223-41.

Bowen, J.P. and Hinchey, M.G. (1995) Seven more myths of formal methods. *IEEE Software, Vol.12, No.4.*

Brereton, O.P., Budgen, D. and Hamilton, G. (1998) Hypertext the next maintenance mountain. *IEEE Computer, Vol.31, No.12,* pp.49-55.

Brooks, R.E. (1980) Studying programmer behaviour experimentally: the problems of proper methodology. *Communications of the ACM, Vol.23, No.4,* pp.207-213.

Brookes, T.M., Fitzgerald, J.S., and Larsen, P.G. (1996) Formal and informal specifications of a secure component: final results in a comparative study. In M-C. Gaudel and J. Woodcock, editors, *Industrial benefit and advances in formal methods, FME'96,* Lecture Notes in Computer Science, Vol.1051, Springer-Verlag, pp. 214-227.

Bruel, J. and France, R. (1998) Transforming UML models to formal specifications. *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98). Workshop on Formalising UML. Why? How?*

Buchi, M. & Back, R. (1999) Compositional symmetric sharing in B. In J. Wing, J. Woodcock, J.Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems, FM'99, Vol. I,* Lecture Notes in Computer Science, Vol.1708, Springer-Verlag, pp.431-451.

Chow, S. (1996) *Statistical significance.* SagePublications.

ClearSy System Engineering, Aix-en-Provence (F). *AtelierB User Manual V3.6.*

Collins, B.P., Nichols, J.E. and Sorenson, I.H. (1991) Introducing Formal Methods: the CICS Experience with Z. *Mathematical Structures for Software Engineering,* Clarendon Press Oxford, pp.153-164.

Courtney, R.E. and Gustafson, D.A. (1993) Shotgun correlations in software measures. *Software Engineering Journal, Vol.8, No.1,* pp.5-13.

Craigen, D., Gerhart, S. and Ralston, E. (1995) Formal methods reality check: industrial usage. *IEEE Transactions on Software Engineering, Vo.l21, No.2,* pp.90-98.

Curtis, B. (1980) Measurement and experimentation in software engineering. *Proceedings of the IEEE, Vol. 68, No.9,* pp.1144-1157.

Daly, J. (1996) *Replication and a multi-method approach to empirical software engineering research.* PhD Thesis, University of Strathclyde.

DeLoach, S., Hartrum, T. (2000) A theory-based representation for object-oriented domain models. *IEEE Transactions on Software Engineering, Vol.26, No.6,* pp.500-517.

DeLoach, S., Smith, J. and Hartrum, T. (2001) Translating graphically-based object-oriented specifications to formal specifications. *To be published - http://www1.coe.neu.edu/~jsmith/Publications/publications.html*

Draper, J., Treharne, H., Boyce, T., Ormsby, B. (1996) Evaluating the B-method on an avionics example. *Data Systems in Aerospace (DASIA) Conference,* pp.89-97 (European Space Agency Publication Division WPP-116).

Dupuy, S. and du Bousquet, L. (2000) A multi formalism approach for the validation of UML models. *Formal Aspects of Computing, Vol.12, No.4,* pp.228-230.

Efron, B. and Tibshirani, R.J. (1993). *An introduction to the bootstrap.* Chapman & Hall.

Facon, P., Laleau, R., and Nguyen, H.P. (1996) Mapping object diagrams into B specifications. In *Methods Integration Workshop, Electronic Workshops in Computing (eWiC),* Springer-Verlag.

Facon, P., Laleau, R., Nguyen, H.P. and Mammar, A. (1999) Combining UML with the B formal method for the specification of database applications. *Research report, CEDRIC Laboratory,* Paris.

Fenton, N. (1993) How effective are software engineering methods. *Journal of Systems and Software, Vol.22,* pp.141-146.

Fenton, N. (1996) *Software metrics: a rigorous approach.* 2nd Edition, International Thomson Computer Press.

Finney, K., Fenton, N. and Fedorec, A. (1999) The effects of structure on the comprehensibility of formal specifications. *IEE Proceedings of Software, Vol.146, No.4,* pp.193-202.

Finney, K., Rennolls, K. and Fedorec, A. (1998) Measuring the comprehensibility of Z specifications. *The Journal of Systems and Software, Vol.42,* pp.3-15.

France, R.B. (1999) A problem oriented analysis of basic UML static requirements modeling concepts. In proceedings of *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99).* pp.57-69.

France, R.B., Bruel, J., Larrondo-Petrie, M. and Shroff, M. (1997) Exploring the semantics of UML type structures with Z. In H.Bowman and J.Derrick, editors, *Formal Methods for Open Object-based Distributed Systems. Vol.2* IFIP TC6 WG6.1 Chapman & Hall, pp. 247-57.

Fraser, M.D., Kumar, K. and Vaishnavi, V.K. (1994) Strategies for incorporating formal specifications in software development. *Communications of the ACM, Vol.37, No.10,* pp.74-86.

Fuchs, N.E. (1992) Specifications are (preferably) executable. *Software Engineering Journal, Vol.7, No.5* pp.165-179.

Glass, R. (1994) The software research crisis. *IEEE Software, Vol.11, No.6,* pp.42-47.

Glass, R. (2001) Formal methods: a largely positive study, but with some interesting questions. In *The Software Practitioner, Vol.11, No.5* September 2001.

Gravel, A. and Henderson, P. (1996) Executing formal specifications need not be harmful. *IEE Software Engineering Journal, Vol.11, No.2,* pp.104-110.

Green, T.R.G. (1989) Cognitive dimensions of notations. In A.Sutcliffe and L. Macaulay, editors, *People and Computers, Vol.5,* Cambridge University Press.

Green, T.R.G. and Blackwell, A.F. (1996). Ironies of abstraction. In Proceedings *3rd International Conference on Thinking.* British Psychological Society.

Green, T.R.G. and Blackwell, A. (1998) Cognitive dimensions of information artefacts: a tutorial *http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf*

Gurevich, Y. (1995) Evolving algebras 1993: lipari guide. In E.Börger, editor, *Specification and Validation methods,* pp.9-36. Oxford University Press.

Hall, A. (1990) Seven myths of formal methods. *IEEE Software Vol.9,* pp.11-19.

Hall, A. (1999) What does industry need from formal specification techniques. *2nd IEEE Workshop on Industrial Strength Formal Specification Techniques.* IEEE Computing Society, pp.2-7.

Harel, D (1987) Statecharts: a visual formalism for complex systems. *Science of Computer Programming, Vol.8, No.3,* pp.231-274.

Hayes, I.J. and Jones, C.B. (1989) Specifications are not (necessarily) executable. *Software Engineering Journal, Vol.4, No.6,* pp.320-338.

Hoare, C.A.R. (1985) *Communicating sequential processes.* Prentice-Hall.

Houston, I. and King, S. (1991) CICS Project Report: Experiences and results from the use of Z in IBM. In Proceedings of *The 4th International Symposium of VDM Europe. Vol.1,* Springer-Verlag, pp.588-596.

IFAD (2000a) VDMTools: The IFAD VDM++ language *http://www.ifad.dk*

IFAD (2000b) VDMTools: The Rose-VDM++ link *http://www.ifad.dk*

Jackson, D. (2000) *Alloy: A lightweight object modelling notation.* Technical Report 797, MIT Lab for Computer Science.

Jackson, M.A. (1983) *System Development,* Prentice Hall.

Jia, X. (1998) *ZTC: A Type Checker for Z Notation User's Guide, Version 2.03,* Division of Software Engineering, School of Computer Science, Telecommunication and Information Systems, DePaul University.

Jones, C.B. (1986) *Systematic software development using VDM.* Prentice-Hall.

Kim, S. and Carrington, D. (2000) A formal mapping between UML models and Object-Z specifications. In J.Bowen, S.Dunne, A.Galloway, S.King, editors, *ZB 2000: Formal Specification and Development in Z and B. First International Conference of B and Z Users,* Lecture Notes in Computer Science, Vol.1878, Springer-Verlag, pp.2-21.

Kitchenham, B.A. (1996) Evaluating software engineering methods and tool – part 1 & 2. *ACM SIGSOFT Software Engineering Notes, Vol. 21, No.1 and No.2.*

Kitchenham, B.A., Linkman, S.G. and Law, D.T. (1994) Critical review of quantitative assessment. *Software Engineering Journal, Vol.9, No.3,* pp.43-53.

Kitchenham, B.A., Pfleeger, S.L. and Fenton, N. (1995) Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering, Vol. 21, No.12,* pp.929-944.

Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El-Emam, K. and Rosenborg, J. (2001) *Preliminary guidelines for empirical research in software engineering.* Technical Report, NRC/ERB-1082, January 2001, NRC 44158.

Knuth, D. (1984) Literate Programming, *The Computer Journal, Vol.27, No.2,* pp.97-111.

Lano, K. (1996) *The B language and method: a guide to practical formal development,* FACIT Springer-Verlag.

Ledang, H. and Souquières, J. (2001) Integrating UML and B specification techniques. In proceedings of *Informatik2001 Workshop on Integrating Diagrammatic and Formal SpecificationTechniques.*

Liu, S. & Sun, Y. (1995) Structured methodology + object-oriented methodology + formal methods: methodology of SOFL . In Proceedings of *First IEEE International Conference on Engineering of Complex Computer Systems. ICECCS'95.*

Matisse (2001) *Methodology of integration of formal methods within the healthcare case study.* Deliverable D7 within the Matisse project, IST-1999-11435, October 2001.

Meyer, E. & Souquières, J. (1999) A systematic approach to transform OMT diagrams to a B specification. In J. Wing, J. Woodcock, J.Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems, FM'99, Vol. I*, Lecture Notes in Computer Science, Vol.1708, Springer-Verlag, pp.875-895.

Meyer, E. & Santen, T. (2000) Behavioural conformance verification in an integrated approach using UML and B. In W.Grieskamp, T.Santen, B.Stoddart, editors, *IFM'2000 : 2nd International Conference on Integrated Formal Methods*, Lecture Notes in Computer Science, Vol.1945, Springer-Verlag, pp.358-379.

Milner, R. (1985) Control flow and data flow: concepts of distributed programming. In proceedings of *NATO Advanced Study Institute International Summer School.* Springer-Verlag, pp.205-228.

Ministry of Defence (1980) *Joint Services Publication 188: Requirements for the documentation of software in military operational real-time computer systems, Ed.3.*

Ministry of Defence (1997) *Def Stan 00-55: Requirements for safety related software in defence equipment, Issue 2.* http://www.dstan.mod.uk/data/00/055/02000200.pdf

Nagui-Raiss, N. (1994) A formal software specification tool using the entity-relationship model. In P.Loucopoulos, editor, *13th International Conference on the Entity-Relationship Approach*, Lecture Notes in Computer Science, Vol.881, Springer-Verlag, pp.315-332.

Nuseibeh, B. and Finkelstein, A. (1992) Viewpoints: a vehicle for method and tool integration. In G.Forte, N.H.Madhavji and H.A.Mueller, editors, *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*, CASE 92, IEEE Computer Press, pp.50-60.

Paige, R. (1997) *Formal method integration via heterogeneous notations.* Thesis for Doctor of Philosophy, University of Toronto.

Pannell, D. and Pannell, P. (1999) Introduction to social surveying: pitfalls, potential problems and preferred practices. SEA Working Paper 99/04, *http://www.general.uwa.edu.au/u/dpannell/seameth3.htm*

Parnas, D.L. (1997) Precise description and specification of software. In V.Stavridou, ed. *Mathematics of Dependable Systems II*, Clarendon Press, pp.1-14.

Petre, M. (1995) Why looking isn't always seeing. *Communications of the ACM, Vol.38, No.6,* pp.33-44.

Pfleeger, S.L. and Hatton, L. (1997) Investigating the influence of formal methods. *IEEE Computer, Vol.30, No.2,* pp.33-43.

Pfleeger, S.L., Jeffery, R., Curtis, B. and Kitchenham, B. (1997) Status report on software measurement. *IEEE Software, Vol.14, No.2,* pp.33-43.

Phillips, M. (1989) CICS/ESA 3.1 Experiences. In proceedings of *The 4th Annual Z User Meeting.* Springer-Verlag Workshops in Computing, pp.179-185.

PUSSEE (2002) *Definitions of Requirements for Integration of UML and B.* Deliverable D4.1.1 within the PUSSEE project, IST-2000-30103.

Rational (2000A) *Using rose – Rational Rose 2000e.* Rational Software Corporation. Part Number 800-023321-000.

Rational (2000B) *Rose extensibility user's guide – Rational Rose 2000e.* Rational Software Corporation. Part Number 800-023328-000.

Rational (2000C) *Rose extensibility reference – Rational Rose 2000e.* Rational Software Corporation. Part Number 800-023329-000.

Rozeboom, W. (1960) The fallacy of the null-hypothesis significance test. *Psychological Bulletin, Vol.57, No.5,* pp.416-428.

Rumbaugh, J., Jacobson, I. & Booch, G. (1998) *The Unified Modelling Language Reference Manual.* Addison-Wesley.

Satpathy, M., Harrison, R., Snook, C. and Butler, M. (2001) A comparative study of formal and informal specifications through an industrial case study.In proceedings of *IEEE/ IFIP Workshop on Formal Specification of Computer Based Systems (FSCBS'01).*

Scanlon, D. (1989) Structured flowcharts outperform pseudocode: an experimental comparison. *IEEE Software, Vol.6, No.5,* pp28-36.

Sekerinski, E. (1998) Graphical design of reactive systems. In D.Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method,* Lecture Notes in Computer Science 1393, Springer-Verlag, pp.182-197

Sekerinski, E. and Zurob, R. (2001) iState: a state chart translator. In M.Gogolla, C.Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools,* Lecture Notes in Computer Science, Vol.2185, Springer-Verlag, pp.376-390.

Shore, R. (1996) An object-oriented approach to B. In H.Habrias, editor, *Putting into Practice Methods and Tools for Information System Design - 1st Conference on the B method.*

Simpson, H.R. (1986) The MASCOT Method. *Software Engineering Journal, Vol.1, Iss.3,* pp.103-120.

Snook, C. and Butler, M. (2000) Verifying dynamic properties of UML models by translation to the B language and toolkit. In G.Reggio, A.Knapp, B.Rumpe, B.Selic, R.Wieringa, editors, *Dynamic Behaviour in UML Models: Semantic Questions,* UML 2000 workshop proceedings, pp.99-105.

Snook, C. and Butler, M. (2001) Using a graphical design tool for formal specification. In G.Kadoda, editor, *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group,* (PPIG'2001), pp. 311-321.

Snook, C. and Harrison, R. (2000) Practitioners' views on the use of formal methods: an industrial survey by structured interview. In *EASE2000, Papers from the Conference on Empirical Assessment in Software Engineering.*

Snook, C. and Harrison, R. (2001a) Practitioners' views on the use of formal methods: an industrial survey by structured interview. *Information and Software Technology, Vol.43, Iss.4,* pp.275-283.

Snook, C. and Harrison, R. (2001b) Experimental comparison of the comprehensibility of a Z specification and its implementation. In *Proceedings of The Conference on Empirical Assessment in Software Engineering, EASE2001.*

Snook, C. and Walden, M. (2002) Use of U2B for Specifying B Action Systems, In proceedings of *RCS'2002 International Workshop on Refinement of Critical Systems: Methods, Tools and Experience.*

Spivey, J.M. (1988) *Understanding Z.* Cambridge University Press.

Spivey, J.M. (1990) *A guide to the zed style option* (available via anonymous FTP at ftp.comlab.ox.ac.uk.).

Sterling, L. and Shapiro, E. (1986) *The Art of Prolog, Advanced Programming Techniques,* MIT Press.

Tichy, W. (1998) Should computer scientists experiment more. IEEE Computer, Vol.31, No.5, pp.32-40.

Vaziri, M. & Jackson, D. (1999) *Some shortcomings of OCL, the object constraint language of UML.* Response to Object Management Group's Request for Information on UML 2.0.

Vinter, R.J. (1998) *Evaluating formal specifications: a cognitive approach.* PhD Thesis University of Hertfordshire.

Ward, P.T. and Mellor, S.J. (1985) *Structured Development for Real-Time Systems.* Prentice-Hall/Yourdon Press, 1985. Three volumes.

Warmer, J. and Kleppe, A. (1999) *The Object Constraint Language: precise modeling with UML.* Addison-Wesley.

Winograd (1995) Environments for designing. *Communications of the ACM, Vol.38, No.6,* pp.65-74.

Wills, A.C., D'Souza, D. (1997) Rigorous component-based development. *http://www.trireme.u-net.com/catalysis/*

Wirth, N. (1971) Program development by stepwise refinement. *Communications of the ACM, Vol.14, No.4,* pp.221-227.

Wonnacott, R. and Wonnacott, T. (1985) *Introductory statistics.* Wiley.

Wynekoop, J. and Russo, N. (1997) Studying system development methodologies: an examination of research methods. *Information Systems Journal, Vol.7,* pp.47-65.

Zelkowitz, M. and Wallace, D. (1998) Experimental models for validating technology. *IEEE Computer, Vol.31, No.5,* pp.23-31.

Appendix A Survey Materials

A.1 Questionnaire

1) How would you define a formal method?
2) What experience has the company had with formal methods?
3) Which formal methods have you used most?
4) How big are the systems that you use formal methods on?
4a) Does the size of the system affect the practicality of using formal methods?
5) How do formal methods affect the software life cycle?
6) How do formal methods affect software quality assurance activities? (records, audits, certification etc)
7) What are the benefits that you have found?
7a) Are they measurable in terms of quality of software products?
7b) Are they measurable in terms of software process improvements?
7c) Has any quantitative data been collected that demonstrate the benefits?
8) What problems have been encountered?
8a) How do the problems affect the quality of software products?
8b) How were they overcome?
9) Have any understanding difficulties or benefits been found?
9a) If so, has this affected correctness and verification of resulting code?
9b) Is there any pattern to the misunderstanding? i.e. particular constructs or styles
10) Do you use any style rules or codes of practice when writing formal specifications
10a) How do they affect understanding (if at all)?
11) Have you found that the structure of the specification model influences the implementation?
11a) Is this good or bad?
12) How have formal specifications affected maintenance issues?
12a) Do the formal specifications help determine the correct code change?
12b) Are the formal specifications difficult to update? Are they kept up to date?
12c) Do the formal specifications prevent (or worsen) degradation of the code structure through maintenance?
12d) Does the structure of the specifications themselves deteriorate through maintenance?
13) How do customers view the use of formal methods?
13a) Are formal documents used as an interface to customers?
13b) If so, how does this affect understanding of the system by the customer
13c) Has it affected system validation and acceptance stages?
14) Is there anything we haven't covered that you would like to talk about?

A.2 Results Summary Matrix

Question	Interviewee A	IBM	Marconi	Philips	Praxis
How would you define a formal method?	Unambiguous mathematical notation in which you can express system behaviour and structure. Has a precise syntax and semantics. Doesn't count things like UML as formal enough	Mathematical and can prove correctness Distinguish between Spec notations and methods (inc. proof) (notations such as UML can't express the important relationships)	Method with underlying mathematical theory, well defined syntax and semantics and rules for manipulation and verification.	Notation with precise syntax & well defined semantics, with a method for refinement into an implementation and support for requirements elicitation. state charts are sufficiently formal although don't support proofs. {O-O notations s.a. UML also discussed - comments relating to these are shown in braces}	Mathematically based notation, specification optionally with verification proofs (UML some aspects formal but not very expressive)
What experience has the company had with formal methods?	One large project which was specified in Z. Several small projects (electronics components) were specified formally and subcontracted	Since 1983 but not so widespread use now	Full size, parallel case study (18 months duration)	COLD developed by Philips and used on 1 or 2 products (but didn't do formal refinement). Lighter weight version of COLD used more extensively for specification	Lots, specialise in this area
Which formal methods have you used most?	Z, CSP, VDM	Z and B toolkit	B Toolkit	COLD (notation in genre of Z, VDM etc) {UML has been used on a few products esp. use-case for requirements}	Z, VDM very extensively CSP, CCS a bit
How big are the systems that you use formal methods on?	1 large safety critical project (150 s/w engineers) several small projects	50 KLOC	900 LOC (ADA) with full formal spec and verification, 2000 LOC with partial formal specification to establish doesn't affect critical parts.	from 10Kloc upwards. Biggest products have been medical systems (70 people for 2 years) but consumer products are becoming quite large in terms of software systems	Biggest 200Kloc, often 100Kloc, 10s Kloc Def. Stan. 00-55
Does the size of the system affect the practicality of using formal methods?	No more than any other factor/method	No, as long as you break it down into encapsulated components Encapsulation is important and formal spec defines the interfaces between components	Yes, but also affected by other factors such as degree of formality reqd. (proofs etc), complexity. Also encapsulation helps break problem down as get into design stages – size problem mainly at reqmts. spec stage.	Not per se but, rapid growth in size of software systems has meant that recruit programmers to cope and haven't been able to formal specify these systems quickly enough	Mostly the methods scale up but model checkers not very well and proof checkers even worse
How do formal methods affect the software life cycle?	N/A – In large project the formal spec wasn't used for subsequent development, in small projects the development was subcontracted	Don't change its structure but major shift of effort to up front, specification stage away from debugging and testing.	Same stages but shift effort to specification stage, reduce rework later, assists testing and validation later	Increase effort to get spec right, but timescales are short so development has to proceed in parallel and product may complete before spec is finished. There is a difficulty with precise specification since lifecycle is often iterative, developing the requirements as the design evolves. As formal specification aims to sort out requirements issues prior to starting design it doesn't fit with iterative development. {O-O modelling enables the quick animation-	Front loading – spec takes longer to get right but in the process resolve many problems and spec makes subsequent stages much easier (esp. coding and testing) Same stages throughout except conventional lifecycle often omits the system specification and works from the requirement spec instead

Question	Interviewee A	IBM	Marconi	Philips	Praxis
				refine-code generation iteration methods}	
How do formal methods affect software quality assurance activities? (records, audits, certification etc)	N/A – as 5 above	Not applicable, don't do any	Unchanged really. Auditors need to have an outline understanding of the expected outputs of processes in order to verify that the processes are being performed. Procedures have been written to cover formal methods.	beginning to put in place templates and checklists to standardise formats to improve exchange between divisions and this will go into procedures. Quality is the main driving force behind pushing for more formal {mostly in the looser sense} methods.	Doesn't change Q.A. role but does make quality control checks more effective. Improves traceability which helps Q.A.
What are the benefits that you have found?	Makes you think through and understand the problem domain. A tool for thinking Discover spec problems early Image – a very good, clever organisation Expert effect – you need clever people to use them, clever people make quality software with any method.	Allows you to see the users view of the system. Uncovers specification issues early rather than discovering them late in the day Significant improvement in failure rates	Get spec. problems resolved and discover errors early so much less rework. Test cases can be automatically generated which enables efficient and effective validation testing	Discover problems early {e.g. when developing state charts} Clear about the requirements and whether they are complete and consistent	Cheaper (if you want a system that works) High defect removal (still get bugs but easier to detect with a formal spec) Coding is much more straightforward (know exactly what's needed) As a consequence performance is improved Complexity deters functionality/code from growing unnecessarily
Are they measurable in terms of quality of software products?	N/A – not measured	40% reduction in post delivery failures (this is based on fault report data for the CICS system)	N/A product was not put into service but expected to be more reliable	N/A	Yes, Product is more reliable
Are they measurable in terms of software process improvements ?	N/A – not measured	Reduces costs of later development and testing activities due to less problems	Requirements validation process much improved. Testing process much improved by auto generation of test cases and expected results	Benefits in providing something to test against. Knowing what is being developed Maintenance helped	Yes, reviewing and testing is more effective
Has any quantitative data been collected that demonstrates the benefits?	N/A – No (data has been collected at ACSL but they haven't started using fm yet)	Some informally collected data for 7a above	(see paper)	No but plan to collect data	Yes but no baseline for comparison (but see Pfleeger & Hatton) One example where data convinced customer that a formally re-engineered version of the existing system would be worthwhile
What problems have been encountered?	Attitude – reluctance of ordinary engineers to get involved. Complexity – hard work to become fluent	Keeping an expertise base together Ability to create good models with useful abstractions (difficult to teach) Customers don't want to be tied down early Management perceive it as a big risk	Structuring and resourcing the requirements specification. Size of proof at first stages of refinement due to lack of encapsulation at this stage. Tools are not formally developed and validated so any use in verification compromises validity	Resourcing to use effectively Extended timescale of spec means it is of limited benefit Doesn't fit with incremental development lifecycle	Customer resistance, acceptance of incorrect software (see 13) Immature tools not integrated into the rest of the development lifecycle e.g. can't write in standard word processors, being unable to use normal development tools is a big turn off Proof tools not industrial strength (except maybe B)
How do the problems affect the quality of software	They don't get used	Reduced use of formal methods (loss of benefits)	N/A not covered in interview	N/A	Lack of use of formal methods – don't achieve benefits Misunderstanding can still exist between

Question	Interviewee A	IBM	Marconi	Philips	Praxis
products?					formal spec and requirements (i.e. is spec valid)
How were they overcome?	They weren't really but good teaching and team building is seen as a solution.	They weren't Oxford Uni. provided a lot of consultancy on modelling	Specification structured by most highly skilled people, others filled in detail. Take small refinement steps initially.	Plan to standardise on UML which doesn't suffer these problems and then use formal methods for (encapsulated) subcomponents within the UML model for the critical parts	N/A
Have any understanding difficulties or benefits been found?	Yes, See 8	Need English comments to explain Z (not so much in B as this is better structured)	Harder to get top level requirements reviewed but this was solved by using the animation facility.	No, but tend to recruit from research for these roles so limited resources so tending to move away from formal {O-O not so much a problem but UML use-cases and requirement elicitation needs (soft) skills that hard developers don't find easy so now recruiting specialist requirements people}	(see 13 for customer understanding) Not too much of a problem, employ good calibre staff but don't think you need special people to understand formal specs, easier than understanding code, just need practice.
If so has this affected correctness and verification of resulting code?	N/A – no examples of coding from spec	No	No	No because experts used	No significant effect
Is there any pattern to this misunderstanding? i.e. particular constructs or styles	Some constructs are difficult e.g. functions that return functions	N/A	No	No	No, not really
Do you use any style rules or codes of practice when writing formal specifications?	Use simpler forms even if more verbose – reduced subset Friendly style with explanation Teach via metaphors to help visualisation	Had codes of practice for embedding comments in Z, now policy is for literate programming where specification (formal and informal) and design and code are all kept together.	Naming conventions, capitalisation, naming correspondence through refinement levels for traceability	Yes had documented styles and codes of practice {UML – trying to do the same – templates and checklists}	Yes, lexical constraints on spelling etc, use of delta and sigma, common format
How do they affect understanding (if at all)?	Easy to understand and also less off putting	See 9	N/A (not asked but assume aid readability)	{Needed to migrate high end products to other divisions as they become older}	Makes 2 peoples specs look similar so know where to expect things
Have you found that the structure of the specification model influences the implementation?	No examples of writing code from specs but would expect the structure to follow through into the code	Yes, most people structure their design similar to their specification (but, the formality assists in maintaining a purer external requirements view, i.e. the spec contains less design decisions)	To some extent but refinement stages used to re-structure for design purposes. State structure unchanged. Code structure reflects B notation	This is a question under consideration at PRL – PK thinks it won't if it is at the right abstraction to be a requirements document but they are looking at whether it can be done purposefully	Yes, to a fairly large degree but some features such as atomicity, concurrency, timing are design stages that affect this correspondence
Is this good or bad?	Trade off – helps traceability but may not be efficient code	Good, it aids traceability (although a purist might argue that the design will lead to less efficient code the pragmatic view is more important)	Helped in maintaining code to B correspondence	It may be good to express requirements so that they influence the structure of the implementation to make it have reusable components On the other hand it may confuse the requirements	Good helps traceability

Question	Interviewee A	IBM	Marconi	Philips	Praxis
				role of the spec to have this in it.	
How have formal specifications affected maintenance issues?	N/A – no maintenance experience	(answered below)	The formal specs themselves had no effect on maintenance. B tool helped a lot in automatically detected everything that relied on a changed component and assisting in re-checking these,	Improve understanding of code being changed making it easier to get the change right	Spec helps with determining correct code change and the effects on the rest of the system
Do the formal specifications help determine the correct code change?	N/A	In one example the formal spec was used to good effect and gave an estimated 50% reduction in maintenance cost. However this is an isolated case and usually the specification is not used or maintained after development.	No.	Yes, provided traceability is ok from spec to code (but this is often not the case due to timescales during development)	Yes
Are the formal specifications difficult to update? Are they kept up to date?	N/A	Not usually kept up to date.	Were updated, and this is not difficult due to the help from the B toolkit	Specs are fairly easy to update but it is only done if traceable to code	Yes, they are kept up to date. This is partly culture but also formal specs are worth keeping up to date because they are so useful compared to natural language specs. Usually easier to update formal specs because you can work out what needs changing better.
Do the formal specifications prevent (or worsen) degradation of the code structure through maintenance?	N/A	Don't affect it either way, usually don't go back to the spec anyway	Prevent degradation because the B tool allows you to maintain the design structure easily and the code is kept in-line with this structure.	if traceable prevent code degradation since the change is made with the spec structure in mind.	Help prevent degradation indirectly by supporting a good process, tend to do things in the right order starting with the spec and this helps keep good code structure
Does the structure of the specifications themselves deteriorate through maintenance?	N/A	Not much practical experience but perception that the formality will increase the tendency to avoid restructuring leading to degradation.	N/A No experience of post-delivery maintenance. (During early spec development the structure was maintained but may be different later on)	Have found that natural language specs deteriorate quickly with changes whereas Formal Specs do not.	Some degradation.
How do customers view the use of formal methods?	Impressed, usually have someone who is keen to learn the methods to again personal position	General warm feeling that company is doing something good to look after quality	MoD Customers mandate the use of formal methods and want to sort out any spec. misunderstandings early and tie down the requirements. US customers may need persuasion to accept proof instead of testing etc. (MoD vice-versa)	(usually project manager or marketing act as customer proxy) Tend to want good methods to achieve quality in general so supportive	Depends on customer, some (e.g. MoD) mandate due to regulatory pressure Some resist, may not fit in with practices, cost of training etc. General acceptance that software rarely works – this takes away the incentive to use formal methods since it is cheaper to produce a system that doesn't work than to make one that does using formal methods
Are formal documents	Were used to place subcontracts for	Internal customers only (encapsulated	Yes	Yes project manager needs to approve	Sometimes

Question	Interviewee A	IBM	Marconi	Philips	Praxis
used as an interface to customers?	small projects	subsections called domains are developed separately so Formal specs will be used to define the interfaces for the domain and this will be used by the other domain groups)		requirements spec	
If so, how does this affect understanding of the system by the customer?	Interviewee was the customer and wrote the spec, subcontractors were able to understand the friendly, reduced subset Z	Beneficial (but note that these 'customers' are other IBM software developers)	Customers have people that understand the methods, also animation of spec is used to illustrate its meaning. If not mandatory there may be some understanding problems but not insurmountable.	The project manager has problems understanding formal specs. so working towards levels of abstraction and to make readable so that they can understand while still being formal as possible	Technical engineering staff understand better and can answer questions about the behaviour of their future system, but the audience is restricted
Has it affected system validation and acceptance stages?	Has helped in working out test cases. No auto test output generation so far but agreed could be done	No, except that there are less problems at this stage.	Helps derive test cases and also contributes to acceptance evidence (for the right customers)	Yes, one of the main driving forces is to improve the final testing by having a clear spec of what it should do.	Beneficial because traceability of tests to spec is clearer, customer can see that the system does what was specified
Is there anything we haven't covered that you would like to talk about?	Would like to see improved learning techniques. Would Not want notation to be made easier if this contaminated the mathematical purity of the notation	There is no impetus to using formal methods because customers accept the current level of quality without FM and supplier can cover corrective work in price.	Thinking about formal methods for non-functional requirements such as parallelism, timing and also for hardware (def stan 00-54)	Have strong commercial pressures for high quality but also timescale pressures. So looking at UML for re-use component based approach and will formalise the critical bits	No
		Lack of specification precision leads to late changes – people are aware that late changes happen and therefore avoid precision – vicious circle!!	Full formal development with proofs, same as for conventionally developed high integrity software but think variance of estimates may be higher at moment due to limited experience, limited skills base etc. Considering using formal specification with B tool but less proving as this might be more efficient than conventional development.		Problem with B – starts late in the lifecycle, need a z spec to start off then translate to B
		Don't use code gen. facility of b toolkit due to code inefficiency	Did not use B-Tool code gen because – didn't trust it (not formally developed) and didn't want C (unsafe).		The really rich languages like Z are only semi-decidable
		timescales are similar to conventional methods	Structure of requirements spec, proofs. (refinement is ok, much like design)		Agreed, domain specific languages probably the way forward
			B tool – relied on heavily and found it helped a lot in tracing, proving and maintenance rework. Some holes – need to add rules for proving but proofs rely on the validity of these rules, no ADA translator		

Appendix B Experiment Materials

B.1 Z specification

State

RoadType

length : $\mathbb{N}$
length > 0

PositionType

road : RoadType space : $\mathbb{N}$
space $\in$ 1.. road.length

VehicleType

pos : PositionType

RoadsysType

roads : $\mathbb{P}$ RoadType goesto : RoadType $\leftrightarrow$ RoadType
dom goesto = roads ran goesto $\subseteq$ roads

Traffic

roadsys : RoadsysType vehicles : $\mathbb{P}$ VehicleType
$\forall$ v : vehicles $\bullet$ v.pos.road $\in$ roadsys.roads $\forall$ v, w : vehicles | v $\neq$ w $\bullet$ v.pos.road = w.pos.road $\Rightarrow$ v.pos.space $\neq$ w.pos.space

Initialisation

TrafficInit

Traffic′ vehiclesinit? : $\mathbb{P}$ VehicleType roadsysinit_roads? : $\mathbb{P}$RoadType roadsysinit_goesto? : RoadType $\leftrightarrow$ RoadType
vehicles′ = vehiclesinit? roadsys′.roads = roadsysinit_roads? roadsys′.goesto = roadsysinit_goesto?

Operations

Report ::= Okay | Destination already occupied

Success

error! : Report
error! = Okay

moveSameRoad$_0$

Δ Traffic
pos? : PositionType
pos?.road ∈ roadsys.roads
pos?.space < pos?.road.length
∃ v : vehicles • v.pos = pos?
¬∃ w : vehicles • w.pos.road = pos?.road ∧ w.pos.space = pos?.space+1
vehicles′ = vehicles ∪ {v :VehicleType | v.pos.road=pos?.road ∧ v.pos.space=pos?.space+1} \ {v : vehicles | v.pos = pos?}
roadsys′ = roadsys

pickRoad

roadset? : ℙ RoadType
road! : RoadType
roadset? ≠ ∅
road! ∈ roadset?

moveNewRoad$_0$

Δ Traffic
pos? : PositionType
pos?.road ∈ roadsys.roads
pos?.space = pos?.road.length
∃ v : vehicles • v.pos = pos?
¬∃ w : vehicles • w.pos.road = pickRoad roadsys.goesto pos?.road ∧ w.pos.space = 1
vehicles′ = vehicles ∪ {v : VehicleType | v.pos.road = pickRoad roadsys.goesto pos?.road ∧ v.pos.space = 1} \ {v : vehicles | v.pos = pos?}
roadsys′ = roadsys

destinationAlreadyOccupied

Ξ Traffic pos?: PositionType error!: Report
pos?.road ∈ roadsys.roads ∃ v: vehicles • v.pos = pos? ((pos?.space < pos?.road.length ∧ ∃ w: vehicles • w.pos.road = pos?.road ∧ w.pos.space = pos?.space+1) ∨ (pos?.space = pos?.road.length ∧ ∃ w: vehicles • w.pos.road = pickRoad roadsys.goesto pos?.road ∧ w.pos.space = 1)) error!=Destination already occupied

moveVehicle ≙
((moveSameRoad$_0$ ∨ moveNewRoad$_0$) ∧ Success)
∨ destinationAlreadyOccupied

B.2 Java Program

```
-----------------------------------------------------------------------------------------------------
import java.lang.Exception;
class InvariantException extends Exception {

public     InvariantException (String msg) {super(msg);}
}
-----------------------------------------------------------------------------------------------------
class RoadType {
   int   roadlength;

public RoadType(int inp_length) throws InvariantException {
   if (inp_length < 1) {
      InvariantException e = new InvariantException
         ("Invariant: road length must be >= 1");
      throw e; }
   roadlength=inp_length; }
}
-----------------------------------------------------------------------------------------------------
class PositionType {
   RoadType   road;
   int      space;

public  PositionType (RoadType inp_road, int inp_space) throws InvariantException {
   if (inp_space < 1 || inp_space > inp_road.roadlength) {
      InvariantException e = new InvariantException
         ("Invariant: position must be within road");
      throw e; }
   road = inp_road;
   space = inp_space; }

public boolean sameas (PositionType inp_pos) {
   boolean same = false;
   if (inp_pos.road==road && inp_pos.space==space) same =true;
```

```
    return same; }
}
-----------------------------------------------------------------------------------------------------
class VehicleType {
   PositionType pos;

public VehicleType(PositionType inp_pos) {pos = inp_pos;}

public void   moveto(PositionType inp_pos) {pos = inp_pos;}
}
-----------------------------------------------------------------------------------------------------
class RoadsysType  {
   RoadType[]      roads;
   RoadType[][]   goesto;

public RoadsysType(RoadType[] init_roads,RoadType[][] init_goesto) throws InvariantException {
   roads = init_roads;
   if (init_goesto.length < roads.length) {
         InvariantException e = new InvariantException
            ("Invariant: all roads must go somewhere");
         throw e; }
   for (int i=0; i<roads.length; i++) {
      if (init_goesto[i].length == 0) {
         InvariantException e =  new InvariantException
            ("Invariant: all roads must go somewhere");
         throw e; }
      for (int j=0; j<init_goesto[i].length; j++) {
         if (!isaroad(init_goesto[i][j])) {
            InvariantException e =  new InvariantException
               ("Invariant: invalid goesto road");
            throw e; }    } }
   goesto = init_goesto; }

public boolean isaroad(RoadType inp_road) {
   boolean r=false;
   for (int j=0; j<roads.length; j++)
      if (roads[j] == inp_road) r=true;
   return r; }

public RoadType[] allgoesto(RoadType inp_road) {
   int i=0;
   while (roads[i] != inp_road) i++;
   return goesto[i];}
}
-----------------------------------------------------------------------------------------------------
import java.util.Random;
class Pick {
   static Random r=new Random();

static public RoadType pickroad (RoadType[] array) {
   int n=Math.abs(r.nextInt() % array.length);
   return array[n];}
}
-----------------------------------------------------------------------------------------------------
class Traffic {
   RoadsysType   roadsys;
   VehicleType[]   vehicles;

public Traffic(RoadType[] init_roads,RoadType[][] init_goesto,VehicleType[] init_vehicles) throws InvariantException {
   roadsys = new RoadsysType(init_roads,init_goesto);
   for (int i=0; i<init_vehicles.length; i++) {
```

```
        if (!roadsys.isaroad(init_vehicles[i].pos.road)) {
            InvariantException e = new InvariantException
                ("Invariant: Vehicle not in valid road");
            throw e; }
        for (int j=0; j<init_vehicles.length; j++) {
            if (init_vehicles[i].pos.sameas(init_vehicles[j].pos) && i!=j) {
            InvariantException e = new InvariantException
                ("Invariant: 2 vehicles at same position");
            throw e; }    }}
    vehicles = init_vehicles; }

 public void moveVehicle(PositionType inp_pos) throws Exception {
    PositionType destination;
    if (inp_pos.space < inp_pos.road.roadlength) {
        destination = new PositionType(inp_pos.road,inp_pos.space+1);}
    else {
        RoadType exit=Pick.pickroad(roadsys.allgoesto(inp_pos.road));
        destination = new PositionType(exit,1);    }
    if (isVehicleAt(destination)) {
        InvariantException e = new InvariantException
            ("Invariant:: Destination already occupied");
        throw e; }
    getVehicleAt(inp_pos).moveto(destination);    }

 public boolean isVehicleAt(PositionType inp_pos) {
    boolean found = false;
    if (vehicles != null) {
        for (int i=0; i<vehicles.length; i++) {
            if (vehicles[i].pos.sameas(inp_pos)) found=true;}}
    return found;  }

 public VehicleType getVehicleAt(PositionType inp_pos) {
    int i=0;
    while (!vehicles[i].pos.sameas(inp_pos)) i++;
    return vehicles[i];    }

}-----------------------------------------------------------------------------------------------------
```

B.3 Questionnaire

Your email address:

Please record the time taken for each of the first 2 questions including all of the time you spend reading the specification/program).

Q1. Describe the physical objects represented in the system and their behaviour (i.e. the functionality of the specification/program) Time taken for Q1.: mins

Q2. 'PickRoad' represents an indeterministic or random choice. In real-world, functional, terms what is it used for? Time taken for Q2.: mins

Q3. How difficult did you find the specification/program to understand compared to how you think you would have found an English language equivalent?

Easy O O O O O o O O O O O Hard (replace an O with an X)

Q4. How difficult do you find mathematical subjects? (i.e. what is your subjective judgement of your own mathematical abilities compared to your peers)

Easy O O O O O o O O O O O Hard (replace an O with an X)

Q5. What training/qualifications do you have in mathematical (and related) subjects (e.g. GCSE, A-level Maths/physics etc)?

Q6. How much experience have you had with the notation/language used in the specification/program?

Q7. Any other comments? (or things that might have affected your answers)

B.4 Marking sheet

Q1. Roads	
Roads are directional	
Roads have a length	
… which is greater than zero	
Roads are modelled as a sequence of discrete positions	
The end of each road is connected…	
….to one or ….	
….more other roads	
Vehicles	
Vehicles exist on a particular road	
…at a particular position on that road	
2 vehicles cannot occupy the same position	
Vehicles can move along roads…	
…one position forward at a time	
…but only if the destination position is unoccupied	
A vehicle at the end of a road…	
…can move to another road…	
…that is connected to its road…	
…in fact any of the connected roads…	
…the choice is random/undefined	
…but only if the destination is unoccupied	
Q2. it represents the vehicles/drivers choice of …	
… which new road to enter.	
Total	
Total time taken: (Q1+Q2 = +)	
marks per minute	

B.5 Summary of Results

	Time (mins)	Q1+2	Q3	Q4	Q5	Q6 (answers to Q6 and Q7 are summarised)	Q7
Z							
1	20	10	2	2	A	<1yr	
2	12	8	3	-2	B	1course	dreading z would have preferred J
3	20	7	-1	-2	A	module	
4	9	7	5	-1	C	little	formal spec more difficult than code
5	10	15	1	-2	A	fair amount in course	
6	29	8	5	-4	A	module	
7	25	9	2	0	A	semester	
8	27	13	1	2	A	semester	
9	13	10	3	0	A	course	
10	25	9	0	-3	A	course	code easier then spec
11	13	4	-	-	-	-	-
12	20	15	3	-3	A	module	z awkward / symbols
13	23	6	2	-3	A	2modules	
14	35	5	2	-5	A	some last term	difficulty ops
15	33	6	3	-2	A	little	difficult due lack of knowledge of z
16	13	3	5	-2	A	12x45min lectures	spec totally confusing
17	18	7	1	-1	A	1 module	
18	18	7	3	-2	A	2 modules	
Java							
19	22	8	0	-4	A	1yr	
20	22	9	2	3	A	6months	
21	19	4	2	-1	A	6months	
22	15	7	3	-3.5	A	6months	names made it easy
23	14	10	2	-1	A	6months	progs easier than z/more natural to write
24	15	3	2	1	A	some	
25	14	8	2	-3	A	module	
26	15	16	3	-2	A	10months	traffic implies functionality
27	21	10	2	0	A	12months in depth	
28	22	12	3	-3	A	module	lack of comments
29	35	7	4	2	A	3 to 4 months	lack of comments
30	12	6	3	3	B	1st yr	Java confusing
31	25	7	3	-3	A	6 months	
32	25	9	1	1	A	module	
33	15	4	3	1	B	not much	v difficult to understand this prog
34	25	8	3	5	B	semester	not strong at coding - Java diffciult
35	23	17	3	-4	A	lots of Java	not too hard to understand
36	27	14	2	3	A	fair amount - module	weird coding style

key for Q5: A=Alevel maths, B=GCSE maths C=maths as part of french baccalaureat

B.6 Corrected Z Specification in ZSL

```
specification

--- RoadType -----------------
| length: N
|-------------------------
| length > 0
--------------------------

--- PositionType -----------------
| road: RoadType;
| space: N
|-------------------------
| space in 1..road.length
--------------------------

--- VehicleType -----------------
| pos: PositionType
|-------------------------
| true
--------------------------

--- RoadsysType -----------------
| roads: P RoadType;
| goesto: RoadType <-> RoadType
|-------------------------
| dom goesto = roads;
| ran goesto subseteq roads
--------------------------

--- Traffic -----------------
| roadsys: RoadsysType;
| vehicles: P VehicleType
|-------------------------
| forall v:vehicles @ v.pos.road in roadsys.roads;
| forall v,w:vehicles | v/=w @
|        v.pos.road=w.pos.road => v.pos.space/=w.pos.space
--------------------------

--- TrafficInit -----------------
| Traffic';
| vehiclesinit?: P VehicleType;
| roadsysinit_roads?: P RoadType;
| roadsysinit_goesto?: RoadType <-> RoadType
|-------------------------
| vehicles' = vehiclesinit?;
| roadsys'.roads = roadsysinit_roads?;
| roadsys'.goesto = roadsysinit_goesto?
--------------------------
```

```
Report ::= Okay | Destination_already_occupied

--- Success ---------------------
| error!: Report
|-------------------------
| error! = Okay
-------------------------

--- moveSameRoad0 --------------------
| Delta Traffic;
| pos?: PositionType
|-------------------------
| pos?.road in roadsys.roads;
| pos?.space < pos?.road.length;
| exists v: vehicles @ v.pos = pos?;
| not(exists w:vehicles @
|     w.pos.road=pos?.road and w.pos.space=pos?.space+1);
|
| vehicles'= vehicles ||
|     {v:VehicleType | v.pos.road=pos?.road and
|                     v.pos.space=pos?.space+1} \
|     {v:vehicles | v.pos=pos?};
| roadsys'=roadsys
-------------------------

--- pickRoad --------------------
| roadset?: P RoadType;
| road!: RoadType
|-------------------------
| roadset?/={};
|
| road! in roadset?
-------------------------


--- moveNewRoad0 --------------------
| Delta Traffic;
| pos?: PositionType
|-------------------------
| pos?.road in roadsys.roads;
| pos?.space = pos?.road.length;
| exists v:vehicles @ v.pos = pos?;
|
| let roadset?=={rr:RoadType | (rr,pos?.road) in roadsys.goesto} @
|   exists road!:RoadType | pickRoad @
|     not(exists w:vehicles @ w.pos.road=road! and w.pos.space=1)=>
|
|     vehicles' = vehicles ||
|       {v:VehicleType | v.pos.road=road! and v.pos.space=1} \
|       {v:vehicles | v.pos=pos?};
| roadsys'=roadsys
-------------------------
```

```
--- destinationAlreadyOccupied ---------------------
| Xi Traffic;
| pos?: PositionType;
| error!: Report
|-------------------------
| pos?.road in roadsys.roads;
| exists v:vehicles @ v.pos = pos?;
| (pos?.space < pos?.road.length and
|   (exists w:vehicles @ w.pos.road=pos?.road and
|                        w.pos.space=pos?.space+1))
| or
| (pos?.space = pos?.road.length and
|   (let roadset?=={rr:RoadType | (rr,pos?.road) in roadsys.goesto} @
|     not(exists road!:RoadType | pickRoad @
|       not(exists w:vehicles @ w.pos.road=road! and w.pos.space=1)
|        )
|   )
| );
| error!=Destination_already_occupied
--------------------------

moveVehicle =^= ((moveSameRoad0 or moveNewRoad0) and Success)
                 or destinationAlreadyOccupied

end specification
```

Appendix C Results of Student Poll

C.1 Poll Results

Results - all years	**ZB**	**UML**	**pref**
total responses	118		
useable responses	116	116	115
strong dislike	47%	12%	
dislike	29%	21%	
neutral	12%	28%	
like	8%	35%	
strong like	3%	3%	
prefer UML			67%
equal			18%
prefer ZB			15%

Results - first years	**ZB**	**UML**	**pref**
total responses	33		
useable responses	33	33	33
strong dislike	30%	12%	
dislike	21%	21%	
neutral	24%	36%	
like	15%	30%	
strong like	9%	0%	
prefer UML			52%
equal			18%
prefer ZB			30%

Results - second years	**ZB**	**UML**	**pref**
total responses	50		
useable responses	49	49	48
strong dislike	57%	6%	
dislike	29%	29%	
neutral	6%	29%	
like	6%	33%	
strong like	2%	4%	
prefer UML			77%
equal			17%
prefer ZB			6%

Results - Third years	**ZB**	**UML**	**pref**
total responses	35		
useable responses	34	34	34
strong dislike	50%	21%	
dislike	38%	9%	
neutral	9%	21%	
like	3%	44%	
strong like	0%	6%	
prefer UML			68%
equal			21%
prefer ZB			12%

C.2 Poll Data

-2=strongly dislike, -1=dislike, 0=neutral, 1=like, 2=strongly like, dk=don't know

Cohort	ZB	UML	pref
first year	-2	0	2
first year	-2	1	3
first year	-2	1	3
first year	1	-1	-2
first year	-2	1	3
first year	-2	-1	1
first year	1	0	-1
first year	-1	0	1
first year	1	1	0
first year	-2	-1	1
first year	-2	-2	0
first year	-1	-1	0
first year	-1	-2	-1
first year	2	-2	-4
first year	1	0	-1
second year	-2	0	2
second year	-2	1	3
second year	0	1	1
second year	-2	dk	
second year	-2	0	2
second year	-1	1	2
second year	1	-1	-2
second year	-2	0	2
second year	-2	-1	1
second year	-2	-1	1
second year	-1	2	3
second year	-2	0	2
second year	-2	-1	1
second year	-1	1	2
second year	-2	-2	0
second year	-1	-1	0
second year	-1	0	1
second year	-2	0	2
second year	-1	-1	0
second year	dk	1	
second year	-2	-2	0
second year	-2	0	2
second year	-1	0	1
second year	-2	1	3
second year	1	1	0
second year	-2	-1	1
second year	-2	-1	1
second year	-1	0	1
second year	-2	0	2
second year	-2	0	2
second year	-1	0	1
second year	-1	1	2
second year	-2	2	4
second year	0	-1	-1
second year	-1	-1	0

Cohort	ZB	UML	pref
second year	-2	1	3
second year	-1	1	2
second year	-2	1	3
second year	-2	0	2
second year	-1	1	2
second year	1	1	0
second year	0	1	1
second year	2	-1	-3
second year	-2	1	3
second year	-2	-1	1
second year	-2	-1	1
second year	-1	1	2
second year	-2	-1	1
second year	-2	0	2
second year	-2	-2	0
third year	-1	1	2
third year	-1	1	2
third year	-2	1	3
third year	-1	-2	-1
third year	-2	-2	0
third year	0	0	0
third year	-2	-2	0
third year	-2	-2	0
third year	-2	-2	0
third year	-2	1	3
third year	-2	1	3
third year	-2	0	2
third year	0	1	1
third year	-2	0	2
third year	-1	0	1
third year	-2	1	3
third year	-2	1	3
third year	-1	1	2
third year	-1	1	2
third year	-1	1	2
third year	-1	1	2
third year	-1	2	3
third year	-2	0	2
third year	1	1	0
third year	-2	2	4
third year	-2	0	2
third year	-2	1	3
third year	-2	1	3
third year	-1	0	1
third year	-1	-2	-1
third year	0	-1	-1
third year	-1	-1	0
third year	dk	dk	
third year	-1	-2	-1
third year	-2	-1	1