A Semi-Formal Approach to Introducing Formal Methods

A. J. Cowling

Department of Computer Science, Sheffield University,
Regent Court, 211 Portobello Street, Sheffield, S1 4DP, UK.
Phone: +44 114 222 1823; Fax: +44 114 222 1810.
A.Cowling @ dcs.shef.ac.uk

Abstract. This paper describes the development of an approach to introducing students to formal methods that emphasises the links between the main formal activity, of constructing a specification for a system, and the other activities that comprise the process of developing a software system. In particular, this approach emphasises two such links. One is the way in which a specification provides information about a system which is additional to that normally contained in the requirements document for it, and which is needed during the design activity. The other is the way in which a specification provides a basis for deriving the test cases for an implementation of a system. After reviewing the relevant pedagogical principles, the paper describes how this approach had been developed, starting from a conventional approach to teaching formal methods, and then progressing through the creation of a process for constructing specifications based on diagrammatic models, firstly using Z and subsequently using extreme X-machines. It then describes the new approach, and illustrates it with a simple example. This shows how the approach is semi-formal, in that it relies on diagrams to represent the structure of the specification, and then on formal notations to express the detail that can not be represented in the diagrams.

1 Introduction

One of the most challenging aspects of teaching formal methods in undergraduate computing programmes is that of finding the best approach to introducing the subject. This involves a wide range of issues, some of which are primarily pedagogical, while others are concerned with the details of particular formal methods and how they are used in practice. Furthermore, some of these issues potentially conflict, so that the set of criteria for identifying a best approach has to include that the approach must resolve these conflicts satisfactorily.

Given this framework, the purpose of this paper is to analyse how one particular computer science department has worked towards a solution to this problem of how best to introduce formal methods within its undergraduate programmes. The search for this solution has evolved over a number of years, and so part of the paper is concerned with describing this evolution. Along the way this has

involved challenging some assumptions that were common because they were apparently obvious, not least assumptions about what actually constitutes a formal method. Consequently the solution that has now been reached might well be regarded as controversial, and so the other part of the paper is concerned not just with describing this solution, but also with justifying it.

The paper therefore has to alternate between discussing the underlying pedagogical issues and describing the approaches to introducing formal methods that have resulted from them, and so the rest of it is structured as follows. Firstly, Section 2 introduces the main pedagogical issues that have determined the overall approach to introducing formal methods, and in particular that have determined the context within which this introduction occurs. This context is described in Section 3, and then Sections 4 and 5 describe the two main approaches that had been tried in the course of this evolution, and discuss why they had only partially succeeded in achieving the pedagogical goals that had been set. From this Section 6 analyses the goals for the synthesis of these approaches that has been created, and Section 7 actually describes this new approach. Section 8 then discusses how the details of this approach relate to the principles developed in Section 6, and finally Section 9 summarises the work and draws conclusions from it.

2 Introducing Formal Methods

There are a number of well-established pedagogical principles that relate to the teaching of formal methods, and in particular to how this topic should first be introduced to students. The most fundamental of these principles is that one must be clear about the reasons for teaching this topic, and these reasons must appear both relevant and convincing to students. Here, the principle of relevance means that, while those teaching the formal methods may regard as important abstract reasons such as supporting the concepts of "computational thinking" [26], these are unlikely to appear important enough to the students to actually help in motivating their study.

The principle that the reasons for studying formal methods need to appear convincing also means that these reasons need to arise from links that are made between formal methods and the rest of the computing curriculum [25], and in particular from links with the other activities involved in developing software [9, 18]. In doing this, though, it is important not to make exaggerated claims for the role of formal methods [16]. Of course, there are well-known examples of systems where the development has benefitted from the use of formal methods, such as IBM's CICS system [10] and the control system for the Paris Metro line 14 (Météor) [2], and others can be found in [11]. On the other hand, as Hoare has acknowledged [12], there are probably far more examples of systems that have been developed successfully without using formal methods at all. Students may well be aware of this, and if so then trying to suggest that formal methods are essential is more likely to create suspicion of these methods than motivate them to study the topic.

Hence, a particular principle that arises from this is that formal methods need to be introduced early in the curriculum [20], as otherwise students will certainly gain experience of developing software without needing to make any use of them, and so will become much more sceptical of any claims as to why they might be important in practice. Furthermore, this principle can not be side-stepped by focussing on the development of practical skills, such as reading or writing formal specifications, since this does nothing to help explain why such skills might be important enough to justify the effort needed to acquire them. Hence, the motivation for studying formal methods when they are first introduced must come from them being linked positively with the introductions to the other parts of the process of creating software. Moreover, these links must not simply involve formal specifications being created for systems, but the specifications must then be used in other activities within this process, in order to avoid the situation of the specifications simply becoming write-only documents.

In practice, since a formal specification for a system can not be created until after the requirements for it have been analysed, these links will mainly arise through such a formal specification being used in some way in at least one of the activities that come later in the lifecycle. Typically this means design, coding or testing, and out of of these three the two that are most suitable for linking directly to formal methods are design and testing, since the activity of coding a system is usually driven primarily by whatever forms of design documentation have been created for that system.

When students are first being introduced to the design activity, the main concern will be to teach them some form of design method, which will enable them to produce a workable architecture and detailed design for a system from the requirements given for it. Such design methods usually start from the results of the requirements analysis, but a strong arguments for using formal methods is that an ordinary requirements document will not usually define the requirements precisely enough to form a sound basis for a design method. To reinforce this it is important to limit the scope of the requirements analysis process when it is introduced, so that the subsequent development of a formal specification will actually add further information to the results of this analysis. Then the design method will have to use the formal specification of the system as its basis, rather than just starting from the basic requirements document.

The links between formal methods and testing are well-established [24, 8], and can be made easily when testing is first introduced, because of the fundamental role that testing methods play in it. These are conventionally divided into ones that start from a specification of the system to be tested and ones that start from its implementation, and so if any specification-based methods are taught then it is natural that they will take the formal specification for the system as their starting point.

The final principle identified here is that there are three key aspects of formal methods that must be taught when they are being introduced. Two of these are commonly recognised, and consist of declarative knowledge about formal methods, concerned respectively with the notations and the underlying computational

models that particular methods use. The third aspect is often ignored, however, and is the procedural knowledge about a formal method: that is, knowledge of how to carry out the task of creating a specification. Because most methods are sufficiently flexible that the specification for a particular system could be structured in a number of different ways, textbooks do not usually prescribe any one structure or process. While this is appropriate once students have gained experience of using a method, in the early stages of learning it they certainly do need clear guidance as to how best structure both the specification itself and the task of creating it, which is why it is important to teach this procedural aspect as well.

3 The Context for Introducing Formal Methods

The structuring of a curriculum based on these principles has involved two stages, or levels of detail. The first stage has been concerned with developing the overall structure of an introduction to software development, while the second stage (described in the next Section) has been concerned with the level of detail of the individual activities that make up software development, and in particular those that involve formal methods. Here, the term software development is being used with a specific meaning, namely the process of creating feasible solutions to sets of functional requirements for software systems [3, 6]. This process is therefore a very restricted form of software engineering, in that it is purely concerned with the structural aspects of software systems and how they are created. The significance of this restriction is that students need to have mastered these basic skills before they can start to consider the qualitative and quantitative aspects of both products and processes with which software engineering is concerned.

The particular curriculum structure that has been created for introducing software development partly reflects the constraints of typical UK three-year honours degree programmes, in that it runs in the first year of the three, and so assumes no prior knowledge of the subject. It has two components, each worth 10 ECTS credits, and each consists of a pair of modules (one in each semester), where the material progresses seamlessly from the first module to the second. Indeed, the only reason for regarding each component as two separate modules is that the university's regulations require each semester's work to be assessed separately, but this is changing, and so eventually each component may become a single year-long module.

One of these components is a fairly conventional programming course, and this is not discussed further here. The other component is primarily structured as a software development project that students carry out in teams, and which is supported by a programme of lectures and other classes that provide the students with the knowledge needed for each part of the project. This project follows a strict waterfall lifecycle, with stages for requirements analysis, formal specification, design, construction, and finally systems evaluation. It is known as the *crossover project*, because one of the features of its structure is that at the end of each stage each team hands over the work that it has done (and the doc-

umentation from the previous stages) to another team, which then continues the work on the next stage. Finally, each team receives back for the final evaluation stage the project for which they originally wrote the requirements, so that they can see how it has evolved as it has been through three other sets of hands for the intermediate stages. Further details of this project structure can be found in [4], and the only feature of it that needs to be noted here is that this structure has the benefit of requiring the student teams not only to create documentation, but also to read the documentation created in the previous stages. This emphasises to the students the importance of creating material that is well-structured, and hence readable, in each of the development activities.

4 Formal Introduction to Formal Methods

Within this framework, the first approach that was taken to introducing formal methods was a relatively conventional one, using Z [21] as the specification language and the CaDiZ tool [23] to process the specifications. This approach was not very successful, largely because the quality of the specifications that the students produced was so variable: a few teams created very good ones, while others struggled to produce anything of any significance, and most were somewhere in between. Consequently, it was a matter of luck as to whether a team working on a later stage in the project saw a specification that actually contained much useful information, and many did not.

Analysing this, it became clear that a large part of the problem was that the procedural aspects of creating formal specifications in Z were not being given nearly enough attention. To rectify this, concepts were borrowed from the research field of methods integration, such as the results in [15], in order to define a process for creating Z specifications. This process then formed the core of the second approach, which was described in detail in [5]. It can be summarised here as having two stages: firstly to create the data model for the specification and secondly to use this in creating the processing model. Thus, the first stage starts from a UML class diagram for the system being specified, which the students will have produced as part of the requirements analysis stage. This will mainly focus on the business classes for the system, so that in content it is effectively almost an entity-relationship diagram, and this is then translated into Z in the following steps.

1. Any basic types needed for the entity attributes are specified.
2. Each entity (ie business class) is specified, in terms of a schema that defines the attributes, using either the built-in types or those specified in step 1.
3. For each class the collection of objects of that class is specified, in terms of a schema that defines the set of objects of this class. This schema also defines the property of whichever attribute (or attributes) form the key for this class, namely that two objects must be equal if they have the same key, which is expressed in Z in a standard way.
4. For each association a schema is defined to represent it, as either a function or a relation, and to specify its multiplicities. A complication here is that,

because values in Z do not have an identity, and so do not behave quite like objects, this function or relation needs to be between keys rather than between objects. Consequently, some additional schemas also need to be defined for each class: one for a function that constructs the set of key values from a set of objects, and the other for a function that extracts an object from a set given its key. These, along with the representations of the multiplicities, are also expressed in Z in standard ways. Then the function or relation that represents an association between classes can be specified as a composition of these with the function or relation that represents the association between the keys, again in standard ways.

5. A final schema is constructed, to collect together the schemas for the collections of classes and for the associations between the classes, in order to specify the complete data model.

In these steps the various standard ways of expressing features involve particular constructions in Z that are taught as part of the process, and further such standard constructions are introduced in its second stage. This second stage, of creating the processing model, starts from a UML use case diagram for the system being specified, and it also uses as input a prototype GUI for the system, where again the students will have produced both of these as part of the requirements analysis stage. Each use case, together with the accompanying interface screens of the protoype GUI, is then specified as an operation in Z in the following steps.

1. The inputs and outputs for the operation are identified from the GUI.
2. The effect of the operation on the data model is identified, as some combination of adding or removing objects from a collection, changing the attributes of an object, or adding or removing instances of associations. As part of this the preconditions for the operation also need to be identified.
3. These are used to create a schema to specify the normal course of the operation. Here each of the different possible effects that an operation can have is specified in Z in a standard way, and similarly there are standard ways of specifying basic preconditions such as objects belonging to the appropriate collection, or having keys that are different from any existing object.
4. For each precondition an associated error message is specified, and an error schema is created to specify that this error message should be returned if the precondition is not specified.
5. Very basic schema calculus is used to create a robust specification of the overall behaviour of the operation, as the disjunction of the normal and error schemas for it.

The main effect observed from the introduction of this approach was that the quality of the specifications that the students produced became much more uniform, and also generally improved. Thus, using this approach nearly all students managed to construct reasonably realistic data models, except that quite a common error was to use functions to represent associations in a way that did not actually match the multiplicities in the UML class diagrams. Also, nearly all

students managed to specify at least some aspects of some operations, although the weaker ones often just produced specifications of the simpler operations, such as those that altered a particular attribute of some object. Furthermore, this improvement also meant that the students were actually making some use of the formal specifications during the activities of designing the systems and defining the sets of test cases for them.

While this approach was therefore producing some successes, there was also evidence that it was not being completely successful, for three reasons. Firstly the students were finding the CaDiZ tool increasingly unsatisfactory. It is actually a collection of tools, that are driven from a specialised unix shell, and so it uses a command line interface rather than a GUI, which meant that it increasingly looked quite unlike any other tools that the students were using. In particular, once Eclipse became the standard tool for most software development, the fact that CaDiZ had to be used separately from it, and in this quite different fashion, made it extremely unpopular.

Secondly, although the specifications that the students were producing were more complete than they had been in the first approach, mostly they were still omitting some important information. In particular, many students' specifications did not get as far as defining how a system's operations should change the associations between its data objects, and yet this is probably the most important piece of information that formal specifications should be adding to what is contained in requirements documents. Thus, while other teams in the project were referring to the specifications, they were not gaining as much from them as had been intended when the approach was designed.

Thirdly, although the students were consulting the specifications when using the test method to construct their sets of system-level test cases, they were finding difficulties in relating the formal specification to the expected behaviour of the system. This is because the expected behaviour was being described partly by the prototype GUI illustrated in the requirements document, which presented each operation as a sequence of screens, where the progression from one to the next was determined partly by which buttons a user might press, and partly by what other inputs they might supply, and particularly by whether those inputs were valid. In the Z specification, however, these possible sequences were simply flattened into a single step consisting of a disjunction of alternatives.

When creating their specifications the students could work out fairly easily how this flattening should occur. Similarly, in the early stages of applying the test method, which was a simplified version of the category-partition method [17] where the categories and partitions were determined from the Z specification, they could relate the flat structure of the specification to the similar structure of the categories and partitions. In converting these to test cases, however, they were then having to reverse this flattening process, in order to get back to the sequences of inputs that were required for each test case, and they found this more difficult.

Given these problems, various alternative approaches were considered, although not actually explored to the extent of trying to use them with the stu-

dents. For instance, the possibility of switching from Z to Object-Z [7, 19] was investigated. This clearly would require simpler structures for some aspects of the specifications, in that associations between objects could be modelled directly. On the other hand, to model the operations realistically, which was important for the links with testing, the specifications would still need to represent keys, and so would still need many of the standard constructions from the existing approach. Hence, such a change would not actually have resulted in much simplification of the specifications, and so it also seemed unlikely to address the other shortcomings of this fully formal approach.

5 A Diagrammatic Introduction

Apart from the issue of the tools, these main shortcomings were the two mismatches between the Z models and the underlying computational models used in the other stages of the software development. One of these mismatches was that the system models needed to represent the sequencing of inputs and outputs, whereas the Z models ignored this completely. The other was that the system models ideally needed to take a top-down view of both processing and data as being equally important. This did not match the bottom-up nature of a Z specification, which requires a complete data model to be built before any of the processing model can be specified, effectively de-emphasising the processing model.

There did not seem to be any easy way of adapting the Z approach to cure these mismatches, and so the radical decision was taken to abandon Z and switch to a specification method that would match more closely the characteristics of the systems models. In their emphasis on the sequencing of inputs and outputs these models were essentially viewing the systems as collections of abstract machines, and so the specification method that was adopted used machine models: to be precise, eXtreme X-Machines (XXMs). The XXM model was described originally by Thomson and Holcombe [22], and is based on the conventional X-machine model, as described for instance in [13]. Here, the name eXtreme refers to the way in which this model reflects the typical structures used in eXtreme Programming [1] in the story cards that document the requirements for operations. Thus, the states in the XXM model for an operation correspond directly to the points where the system is waiting for input from the user. The transitions then correspond to the processing involved in accepting some input and generating the response to the user that indicates what input is expected next. Formally, the distinctive feature of the XXM is that a transition may have a guard associated with it that is defined over the memory of the machine, with the requirement that for every guarded transition out of a state there must also be other transitions with complementary guards, so that each state of the machine is complete in respect of the possible values of the guards. In practice, constructing an XXM model for an operation therefore begins with the construction of the state transition diagram (STD) for the machine, and this is why an approach to formal specification that uses XXM models can be described as a diagrammatic one, although of course

the STDs are simply a notation for representing a very precise mathematical formalism.

In teaching this approach, the process that was presented to the students for developing the processing model had two main stages: firstly the creation of the STDs for the operations, and secondly the specification of the functions needed for the XXMs. In the first stage, the STD to model an operation is created by identifying the states of the XXM with the screens in the prototype GUI. Then, the transitions are identified initially by analysing the possible inputs to each screen, to determine which screen is to be displayed next. Where this analysis indicates that the choice of the next screen depends on the data stored in the system, then a set of transitions is required, each with an appropriate guard. Finally, the transitions are labelled to correspond to the inputs that cause them, such as the input of some particular piece of data, or some button being clicked. These transition labels then identify the functions in the XXM model, and subsequently these will then correspond to the units of code that will need to be defined in the design stage of developing the system, to actually handle the various possible inputs.

The second stage in the process, specifying the functions for the XXM models, does then involve the construction of a data model. Given the the UML class diagrams created in the requirements analysis stage of the software development process were effectively being viewed as entity-relationship diagrams, the approach that was selected was to use the conventional relational model, with each entity being represented in terms of a relational database table, and the associations being represented by foreign keys in these tables.

This approach was partly succesful, but not completely so. Where it succeeded is that the students found the STD parts of the XXM models easy to understand and construct. They also found it easy to see how these models related to the tasks that had to be carried out in the design stage of developing systems, so that in their projects they were actually using the specifications during the design stages. Also, the testing method that was being taught was modified to embed the category-partition concepts into the framework of the standard X-machine testing method [14], with the result that the students were also using the specification in the system evaluation stage, too. Thus, this approach was even more successful than the previous ones in linking the formal specification stage with the other stages in the software development process.

Where this approach was not so successful, though, was in enabling the students to create specifications that defined precisely the behaviour of each operation. In particular, it did not result in any real improvements in the extent to which students' specifications defined how operations should affect associations between objects, in terms of either creating or deleting instances of associations. In part this, as with previous approaches, was a consequence of time constraints, in that the students would naturally create the easy bits of their specifications first, so that if they ran out of time it would be the hard bits that were not done. It has then been a common feature of all of the various approaches that

thinking about how operations affect associations has apparently been seen by the students as one of the hard bits.

More fundamentally, though, it appeared that in this particular approach this effect was also a result of the decision to model associations in terms of foreign keys. This kind of model is more concrete than the use of functions or relations in Z, and as a result it does not represent so naturally properties of associations such as multiplicities. Since the need for a system to alter instances of associations usually derives directly from its business rules, and the multiplicities play an important part in encoding these, this meant that this more concrete representation was actually obscuring one of the key features that determine how the system should behave.

6 The Principles of a Semi-Formal Approach

In the light of this evaluation of the diagrammatic approach, it was clear that it needed to be revised. In planning this revision, a number of curriculum goals were identified as a basis for guiding the development of a new approach. The most important of these was that the use of machine models should be retained, as should the associated top-down structure of the diagrammatic approach. Thus, students would still begin their specifications by constructing XXM models, because this then imposed a clear structure on the specifications.

Where the new approach should differ was in the treatment of the data model, which should use a structure that was still formal, but that would reflect more closely the key features of the behaviour of a system. Specifically, the data model needed to make a proper distinction between the individual objects that represent entities and the collections of these objects that the system then manipulates. Also, while this model should be a formal one, the formalisms used should be kept as simple as possible, in that they should not repeat information that could be expressed clearly in the diagrams, or using (possibly stylised) natural language. Finally, the formalisms used should, as far as possible, be applied uniformly, so that if a particular part of the model required some formal construction to be used in one situation, then where possible the same construction should be used in all situations.

Three specific applications of this were then identified: associations, multiplicities, and the preconditions of operations. For associations, this aim of uniformity meant that they should be represented as far as possible in terms of relations, but recognising where appropriate that functions are a special case of relations. This approach would then mean that multiplicities could be represented uniformly, in terms of the sizes of the sets formed by the relational image operator. Finally, the application to the preconditions of operations reflected the need to balance this general goal of uniformity with the one of simplifying the formalisms used. While there are some obvious cases of preconditions that could be expressed informally in a clear way, such as whether an object exists that has a given key, or whether two objects are associated, there are plenty of other realistic preconditions where precise statements would have to be formal ones.

Hence, the goal of uniformity meant that in principle all preconditions should be expressed formally, although in practice a number of standard constructions would be introduced for the more common situations, and these would be used as vehicles to help the students understand the underlying formalisms.

7 The Semi-Formal Introduction

The approach that has therefore been developed is as follows, where the description of it will be illustrated by the example of a very basic library system. This library system simply handles books, individual copies of books (where there may be multiple copies of some books), and users, who may borrow copies of books. Details of the information that needs to be stored for each of these kinds of entity will be discussed as the example is developed. The library imposes limits on the number of copies that any user may have on loan at any given time, and these limits will vary from user to user, but further details of how they might vary will be ignored. The library system provides the usual range of operations for managing the data about books, copies and users, and the usual operations of checking out copies to a user who is borrowing them, and checking in copies that a user is returning. Other typical functions of such a library, such as allowing users to reserve books that are currently out on loan, will be ignored.

Like the previous approaches this one has two main parts, one for the construction of the processing model and one for the construction of the data model, but these are now split into three stages. Firstly the basic processing model is constructed in a top-down fashion, using XXM models, and identifying where guards are needed, but without defining what the conditions for them are. Secondly the data model is constructed, and then thirdly the remaining features of the processing model are specified in terms of it, namely the guards and the actual changes that XXM transitions must make to it. For instance, the operation to check a book out of the library would be modelled in the first stage by the XXM shown in Fig. 1.

The second stage then builds up a data model following a simplified version of the process described in Section 4, but without expressing all of the details from the UML class diagram in formal notation. Specifically, the constraints that are assumed to apply to this introduction to formal methods, and consequently the ways in which the various details do need to be defined, are as follows.

1. Attribute types are named in the class diagram, and it is assumed that the sets of values that these refer to are understood (ie in Z they would be either be basic types or built in types).
2. Business classes (such as *Copy* in this example) are named in the class diagram, and are understood to be tuples of their attributes. It is assumed that situations where there might be invariants between the attributes will be sufficiently uncommon that the need to specify them can be ignored. However, since UML class diagrams do not identify keys, the key for each class must be specified using natural language, for instance as "class *Copy* has field *copyId* as key".

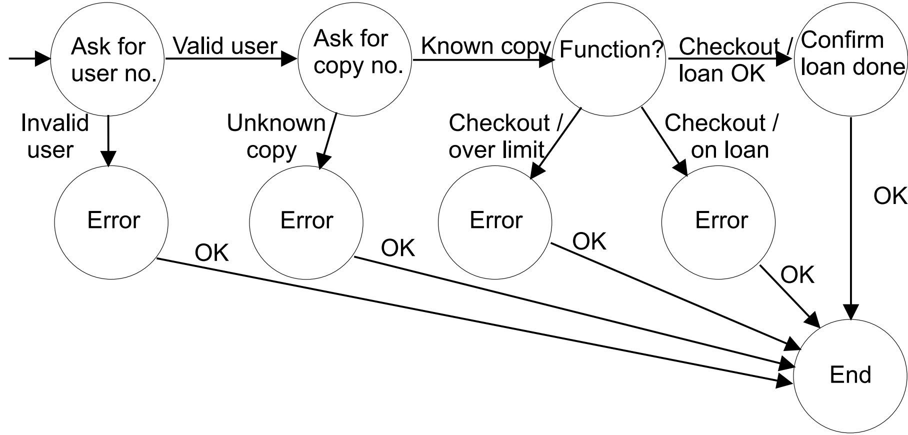

Fig. 1. The XXM for the checkout operation

3. For each class, a variable corresponding to the collection of objects of that class must be specified, using similar syntax to a Z declaration (ie using its powerset notation), eg as $clist : \mathbb{P}\ Copy$.
4. Similarly, for each association a corresponding variable must be specified, as either a function or a relation, again using similar syntax to a Z declaration, eg as $borrowedBy : Copy \rightarrow User$.
5. Also, for each association the multiplicities must be specified, in terms of the relationships between its domain and range and the corresponding collection sets. Here it is assumed that there are just two possibilities that need to be considered for the lower bound of a multiplicity, namely that an association is either optional or compulsory (ie lower bounds 0 or 1 respectively), and two possibilities for the upper bound (ie some value, or unspecified). For each of these there is a standard form of constraint that expresses it, so that for *borrowedBy*, which is optional for both entities, the constraints for the lower bounds would be $\mathsf{dom}\ borrowedBy \subseteq clist$ and $\mathsf{ran}\ borrowedBy \subseteq ulist$, while the constraint on the number of copies that a user can borrow would be $\forall\ u : User \mid u \in ulist \bullet \#(borrowedBy \sim (\!| u |\!)) \leq u.limit$.
6. It is assumed that the possibility of constraints between different associations can be ignored, although in principle they could be specified once the relevant variables have been declared.

This set of variable declarations then defines the names that can be used for the persistent data in the third stage, where the detail of the operations is specified. This detail has four components, as follows.

1. Conditions are specified that will then be used in the guards for those transitions that need them. They define any input data that might be used in them as parameters, so that a condition *CopyExists* might be specified as $CopyExists\ (cno : \mathbb{N}) = \exists\ c : Copy \bullet c \in clist \land c.copyId = cno$.

2. For each transition that accepts some input, variables are declared to specify the names and types of the input data items. For instance, both transitions out of the state *Ask for copy no.* are defined to accept input that can be declared as $cn? : \mathbb{N}$, and this variable can then be used in specifying the guards for the transitions. The Z naming conventions of ? for input and ! for output are used for convenience, but are not essential to the approach.
3. For each transition that requires a guard, it is specified in terms of the conditions and input variables. For instance, the guard for the transition *Known copy* would be $CopyExists\ (cn?)$, and the guard for the transition *Unknown copy* would be $\neg CopyExists\ (cn?)$.
4. For each transition that changes the persistent data, the changes are specified in terms of the data model and input variables. For instance, the transition *Checkout/loan OK* uses the input variables $cn?$ and $un?$ to make one change to the data, in that it causes a new maplet $c \mapsto u$ to be added to the function *borrowedBy*, where $c.copyId = cn? \wedge u.userId = un?$.

A specification in this style then provides enough information that one could analyse each operation, both to show that it maintains the invariants for the data model and to extract the new postconditions that it establishes, but these analyses are not required as part of the approach. Rather, the emphasis is on the links with the design and testing stages, where the specification does define the changes to the data model that are needed in the design stage, and provides a good basis for using a form of the X-machine testing method.

8 Evaluation

At this stage this new approach has been designed and is being used, but students have not yet followed it through to completion, and so proper evaluation will have to wait until they have. There will then be two kinds of criteria for evaluating it, one kind relating to its success within the context of the project where it is being used, and the other kind relating to its success beyond this project.

The first kind of evaluation will be possible once one class has completed a run of the crossover project, and as with the previous approaches that have been described here the criteria for it will be concerned with the completeness and accuracy of the specifications that the students have produced, and with the consequences of this for how much use is made of them in the subsequent design and testing stages of the project. The second kind of evaluation will be concerned with how successful the approach has been in convincing students that some use of formal methods can be of benefit in the process of developing software, even if (or perhaps especially if) that use is only informal in nature. To evaluate these criteria it will be necessary to study how the students do undertake this process in later projects, which will be an important study to perform, but the methods that might be used for it must be regarded as beyond the scope of this paper.

9 Summary and Conclusions

What the paper has done is to show how a new approach to introducing formal methods has been developed, which is semi-formal in that it relies on diagrams to express the structure of the formal specification for a system, with formal notations being used just to express those features that can not easily be represented within the diagrams. Thus, the approach enables the specification to be constructed in a top-down fashion as far as possible, rather than the bottom-up process that must be followed for a fully formal specification. As such it also avoids a lot of the more complex notation that would be needed if an equivalent fully formal specification were to be constructed explicitly, so that this approach effectively abstracts away from some aspects of the formal notations. This approach has been aimed at providing strong support for creating effective linkages between the activities of specifying a system formally, designing it and testing an implementation of it, and the evaluation of the approach that is being conducted will focus on these issues.

References

1. Beck, K. and Andres, C.: *eXtreme Programming Explained: Embrace Change.* Addison Wesley, San Francisco (2005).
2. Behm, P., Benoit, P., Faivre, A. and Meynadier, J.-M.: Météor: A Successful Application of B in a Large Project. *FM'99: Proc. World Congress on Formal Methods in the Development of Computing Systems, Volume I*, Lecture Notes in Computer Science **1708** (1999) 369–387.
3. Cowling, A. J.: Modelling: A Neglected Feature in the Software Engineering Curriculum. *Proc. 16th Conference on Software Engineering Education and Training*, IEEE Computer Society Press, Los Alamitos (2003) 206–215.
4. Cowling, A. J.: The Crossover Project as an Introduction to Software Engineering. *Proc. 17th Conference on Software Engineering Education and Training*, IEEE Computer Society Press, Los Alamitos (2004) 12–17.
5. Cowling, A. J.: Translating Diagrams: A New Approach to Introducing Formal Methods. *Proc. 18th Conference on Software Engineering Education and Training*, IEEE Computer Society Press, Los Alamitos (2005) 121–128.
6. Cowling, A. J.: Stages in Teaching Software Design. *Proc. 20th Conference on Software Engineering Education and Training*, IEEE Computer Society Press, Los Alamitos (2007) 141–148.
7. Duke, R and Rose, G.: *Formal object-oriented specification using Object-Z.* Macmillan, Basingstoke (2000).
8. Duke, R, Miller, T. and Strooper, P.: Integrating Formal Specification and Software Verification and Validation. *Teaching Formal Methods 2004: Proc. CoLogNET/FME Symposium*, Lecture Notes in Computer Science **3294**. Springer Verlag, London (2004) 124–139.
9. Habrias, H. and Faucou, S.: Some reflections on the teaching of formal methods. *Proc. Workshop on Teaching Formal Methods: Practice and Experience*, 2003. `http://cms.brookes.ac.uk/tfm2003/`.
10. Hayes, I. J. (editor): *Specification Case Studies, 2nd edition.* Prentice Hall, Englewood Cliffs, New Jersey (1993).

11. Hinchey, M. G. and Bowen, J. P.: *Industrial-Strength Formal Methods in Practice.* Springer Verlag, London (1999).
12. Hoare, C. A. R.: How Did Software Get So Reliable Without Proof? *Proc. Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, Lecture Notes in Computer Science **1051**. Springer Verlag, London (1996) 1–17.
13. Holcombe, M. and Ipate, F.: *Correct Systems: Building a Business Process Solution.* Springer Verlag Series on Applied Computing, London (1998).
14. Ipate, F. and Holcombe, M.: An integration testing method which is proved to find all faults. *Intern. J. of Computer Mathematics*, **63**.3 (1997) 159–178.
15. Mander, K. C. and Pollack, F. A. C.: Rigorous Specification Using Structured Systems Analysis and Z. *Information and Software Technology*, **37**.5 (September 1995) 285–291.
16. Mandrioli, D.: Advertising Formal Methods and Organizing their Teaching: Yes, but .... *Teaching Formal Methods 2004: Proc. CoLogNET/FME Symposium*, Lecture Notes in Computer Science **3294**. Springer Verlag, London (2004) 214–224.
17. Ostrand, T. J. and Balcer, M. J.: The Category-Partition Method for Specifying and Generating Functional Tests. *Comm. ACM* **31**.6 (June 1988) 676–686.
18. Robinson, K.: Embedding Formal Development in Software Engineering. *Teaching Formal Methods 2004: Proc. CoLogNET/FME Symposium*, Lecture Notes in Computer Science **3294**. Springer Verlag, London (2004) 203–213.
19. Smith, G.: *The Object-Z specification language.* Kluwer Academic, Boston (2000).
20. Sobel, A. E. K., Saiedian, H., Stavely, A. and Henderson, P.: Teaching Formal Methods Early in the Software Engineering Curriculum. *Proc. 13th Conference on Software Engineering Education and Training*, IEEE Computer Society Press, Los Alamitos (2000) 55–56.
21. Spivey, J. M.: *The Z Notation: A Reference Manual.* Prentice-Hall, Englewood Cliffs, New Jersey (1989).
22. Thomson, C. and Holcombe, M.: Applying XP Ideas Formally: The Story Card and Extreme X-Machines. *Proc. 1st South-East European Workshop on Formal Methods*, South-East European Research Centre, Thessaloniki (2003) 57–71.
23. Toyn, I. and McDermid, J. A.: CADiZ: An architecture for Z tools and its implementation. *Software: Practice and Experience* **25**.3 (March 1995) 305–330.
24. Utting, M. and Reeves, S.: Teaching formal methods lite via testing. *Software Testing, Verification and Reliability* **11**.3 (September 2001) 181–195.
25. Wing, J.: Computational Thinking. *Comm. ACM* **49**.3 (March 2006) 33–35.
26. Wing, J.: Invited Talk: Weaving Formal Methods into the Undergraduate Computer Science Curriculum (Extended Abstract). *AMAST 2000: Proc. 8th Int. Conf. on Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science **1816**. Springer Verlag, London (2000) 2–7.