SUPPORTING THE LENGTH OF FORMAL DEVELOPMENT: FROM DIAGRAMS TO VDM TO B TO C

Juan Bicarregui
Rutherford Appleton Laboratory, Oxfordshire, UK.
Jeremy Dick
B-Core UK. (Work done whilst at Bull Information Systems, Hemel Hempstead, UK.)
Eoin Woods
LBMS, London. (Work done whilst at Bull Information Systems.)

ABSTRACT

This paper reports on the experiences gained in the MaFMeth project[1] which is undertaking a formal development with tool support for several parts of the life cycle from requirements capture through to C code generation. A number of issues arise from the limitations of different notations and tools and from their combination when applied to the development of software components destined to be integrated into a broader system context.

We describe the problems we encountered, the mistakes we made, and the solutions we adopted, for the benefit of both those applying formal methods to real product development, and for the creators and designers of methods and tools.

Keywords: Formal Methods, VDM, B, Formal Specification, ESSI.

1 INTRODUCTION

The purpose of this paper is to report on our experiences in applying a combination of formal notations, tools and techniques to the development of a system component from specification through to code and test.

We wish to describe the problems we encountered, the mistakes we made, and the solutions we adopted, for the benefit of both those applying formal methods to real product development, and for the creators and designers of methods and tools.

The nature of the experiment was influenced by

- the desire to cover as much of the development life cycle as possible by formal techniques, from diagrams for capturing the initial specification, to code generation, unit test and integration test;
- the nature of the tool support available for each stage of development; and
- the fact that the resulting code has to be closely integrated with other code, which is either supplied by a third party as an existing package, or was developed using more traditional techniques.

[1] The MaFMeth project is an application experiment funded under the EC ESSI programme. It is a collaboration between the Bull development centre (Hemel Hempstead), Bull S.A. (Paris), B-Core Limited (Oxford) and the Rutherford Appleton Laboratory.

"Proceedings, 7th International Conference on: Putting into practice method and tools for information system design", Nantes (France), October '95, IUT de Nantes, H. Habrias (Editor)

The software developed is a component of a distributed transaction processing monitor deployed in financially critical banking environments. This software allows applications running on heterogeneous platforms to coordinate their transactional activities via message passing. The monitor provides message storage and transformation services.

The kinds of problems encountered arise from:

- the combination of different formal techniques;
- limitations of the techniques and tool support chosen; and
- the need to generate integratable code.

This paper gives a description of the experiment and highlights a number of issues concerning the notations, techniques or tools used. Note that, as an experiment in the application of existing techniques and tools, there was no intention (or resources) to attempt to resolve any general problems.

2 THE METHOD

Motivated by the desire to cover as much of the life-cycle as possible, from requirements capture to integration testing of code, and the lack of a single tool providing all these facilities, we found it necessary to select a number of different notations, each with their own forms of tool support. This introduced concerns about the co-existence and interaction of the various formalisms and tools.

The use of C as a target language was imposed by the development organisation. The B-Toolkit was chosen for its support of development from the design phase to C code generation. The decision to employ VDM rather than the Abstract Machine Notation (AMN) of the B method to capture the initial specification was motivated by three reasons:

- previous experience [Bicarregui93] that AMN encourages the specifier to think very much in terms of assignments to state variables, whereas VDM-SL facilitates the capture of the initial specification at a more abstract level.
- the desire to evaluate the diagrammatic approach to formal specification offered by the “VDM through Pictures” tool [Dick91], and the style of specification imposed by this approach.
- the desire to evaluate and take advantage of the ability to generate test cases from the VDM specification using the VDM Analysis Tool [Dick93].

For the purpose of the project, we identified 13 development activities, with varying degrees of tool support. These are depicted in Figure 1.

Measurements relating to these activities are being taken for two purposes:

1. to compare a formal development process with a conventional one;
2. to compare the relative effectiveness of various stages of the formal process.

To meet the first of these objectives, the results of a number of development projects, all producing sub-products with similar characteristics, will be compared. One of the development projects will be the application experiment, developing its product using the formal development methodology. The rest of the development projects will act as the control for the experiment and so will use the existing software

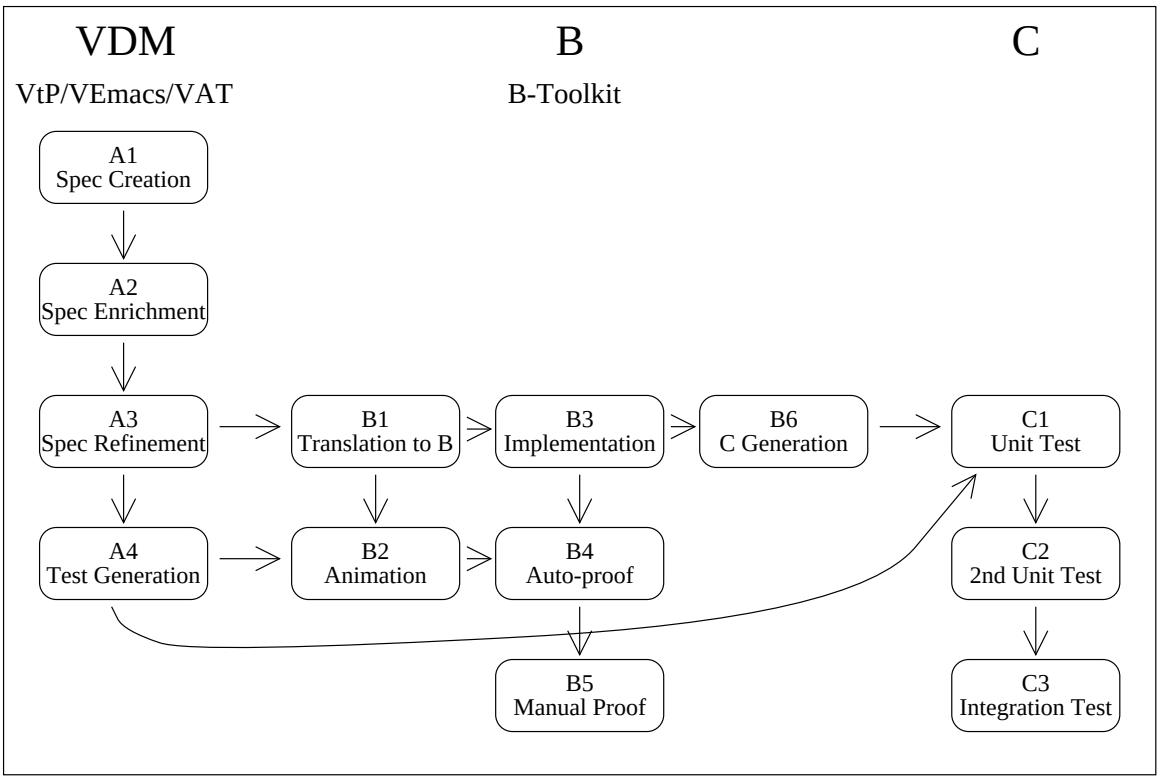


Figure 1: Development activities identified in the MaFMeth project

development process without any changes. The measurements made will be via the existing metrics programme and will consist of at least the following standard metrics being collected for each of the development projects under study:

- number of faults per thousand lines of code found during unit test,
- number of faults per thousand lines of code found during validation test,
- number of faults per thousand lines of code found during customer use,
- person months of effort per thousand lines of code produced.

To meet the second objective, counts of faults discovered will be kept during each of the stages named above. Since some of the activities in the formal process may take place in parallel, some faults may be discovered at more than one stage. For this reason, each fault discovered will be given a unique identity, and for each fault we will record the stage it was first discovered, and the stage it was introduced. From these figures, we will be able to estimate the effectiveness of each stage of the process in terms of the actual numbers of faults introduced and detected.

3 THE APPLICATION

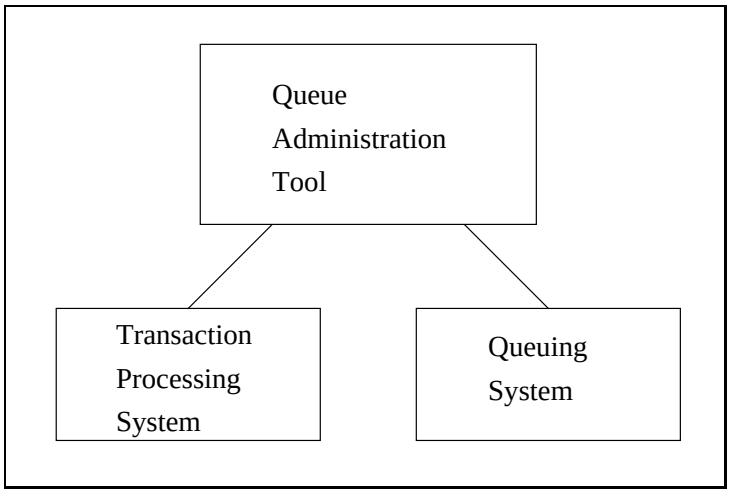
The MaFMeth project addresses the development of one component of a product designed to enhance the functionality of an existing distributed transaction processing system.

Three additional services are offered by this enhancement:

- a dynamically configurable message routing service
- a message transformation service
- a message queuing service.

The last of these contains the component that is being developed in MaFMeth: a Queue Administration Tool (QAT), which monitors the states and contents of queues in the system, raises alarms under certain conditions, allows messages to be examined and modified, and certain actions to be taken on queues.

To obtain the information it requires, QAT must interface directly with the existing queuing and transaction processing systems. This fact has a significant impact on the style of the formal approach taken, as will be discussed later.



4 MODELLING THE CONTEXT

To determine the boundary of the system to be developed, it is necessary to specify some of its context as well. In the case of QAT, we had to give specifications of the queuing service and parts of the transaction monitor, as well as the new alarm service.

Whilst developing the system model, we used three specifications:

Abstract specification: a specification of the external interfaces of the whole system including the new functionality introduced by QAT. This took no account of the fact that some of the functionality was already provided by existing components, such as the queuing system.

First decomposition: here QAT, the alarm system, the queuing system and part of the transaction monitor appear as separate components. This defined the scope and interfaces, internal and external, of the proposed development.

Second decomposition: this takes into account the limited nature of the existing implementation of the queuing system, by introducing additional operations.

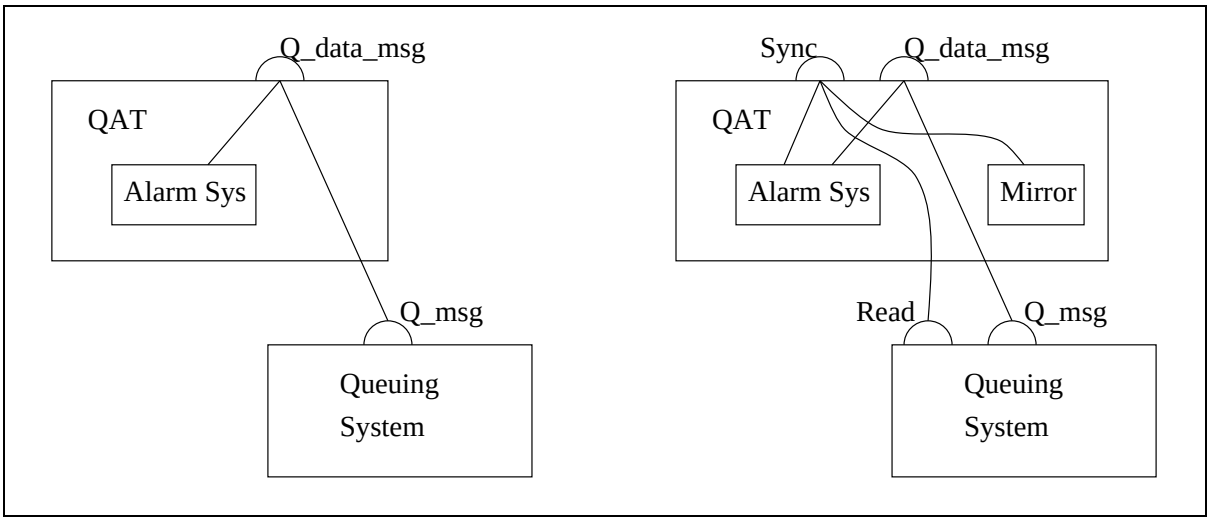
ISSUE 1:

> Although a considerable degree of implementation detail was available for the existing components from the outset, we still had to give an abstract model of the entire system functionality to provide a context for the expression of the global requirements, independently of the prescribed system architecture.
>
> In general, this kind of concern will arise whenever integration with existing system components is required.

An additional advantage of the existence of an overall abstract specification was that test cases could be generated from this VDM model to form part of the eventual integration test suite. It is interesting to note that test case generation from our specification of individual existing components in the first and second decompositions can be used to test the accuracy of the model with respect to existing implementations.

The development path from the abstract specification defining the external interfaces to the first decomposition in which the boundaries of the component of interest are defined, is a form of refinement. In other words, the functionality of the initial specification should be preserved by the restructured one. In our case, the external interface was also modified at this stage, and hence no formal refinement could be demonstrated. In more ideal circumstances, time would have permitted us to recover the formal relationship between the models by the definition of an extra enclosing layer of specification.

The second decomposition was necessary due to inadequacies in the implementation of the queuing system. In general, the prior existence of certain specified components constrains the designer's approach, in that the nature of these components is pre-determined. In the case in hand, the interface of the underlying queuing system does not signal the occurrence of significant events, such as the arrival of a message on a queue. And yet such events were modelled in the abstract specification of the system as a whole. The occurrence of these events can only be discovered by polling the state of the queuing system, and by comparing the current state with the last state. Differences in state indicate that events have occurred. This led to the extension of the QAT system state to include a mirror copy of the state of the queuing system, and the addition of a polling operation, SYNC-ALARM-AND-QUEUE-SYSTEMS, which updates this mirror state to reflect the real state of the queuing system. The structure is depicted in the following diagram



Demonstrating that the second decomposition is a refinement of the first would be difficult. In the first decomposition, the operation QUEUE-DATA-MESSAGE affects the state of both the alarm component

and the queuing system. In the second decomposition, it effects only the queuing system; it is a subsequent occurrence of SYNC-ALARM-AND-QUEUE-SYSTEMS that detects the arrival of the new data message (along with other new messages), updates the mirror state, and brings about the required changes in the state of the alarm component. The functionality of a single operation in the first decomposition is preserved over sequences of operations in the second decomposition.

ISSUE 2:

A considerable degree of flexibility is desirable when attempting to prove refinements with the goal of using pre-implemented components. Interfaces between modules may have to be revised to take account of inadequacies in existing modules. A formalism is required for situations where operations are to be implemented as combinations and sequences of others.

5 VDM THROUGH PICTURES: DIAGRAMS TO VDM

VDM through Pictures (VtP) allows an initial VDM-SL specification to be created through the use of two kinds of diagrams:

Type Structure Diagrams (TSD): an almost complete visual syntax for VDM type definitions, are used to define the state of a system.

Operation State Diagrams (OSD): allow the implicit definition of operations in a very particular style.

The semantics of these diagrams is defined in terms of the VDM that they represent. They are not intended to correspond to the semantics of entity relationship or state transition diagrams.

TSDs are neutral in terms of the style imposed on specification. OSDs, however, impose a style of specification in which operations are defined in implicit pre- and postcondition style, and in which the pre- and postconditions are conjunctions of auxiliary boolean functions. The diagrams allow the operation signature to be given, and the makeup of their pre- and postconditions to be specified. The auxiliary functions are merely identified, their detailed definition being of no relevance to the diagram; they are later fully defined by enriching the VDM-SL text generated from the diagram.

VDM-SL can be generated from diagrams, and vice versa, allowing a flexible working environment for initial specification capture. VtP works in conjunction with a VDM-SL multi-font editor and type checker, but no support for proof obligation generation or reasoning, such as that given by Mural [Jones91, Bicarregui91], is provided.

6 HAND TRANSLATION: VDM TO B

Although it can be argued that VDM and B encourage different levels of abstraction [Bicarregui93], it is also possible to use them at the same level making translation merely a matter of transliteration. However, our initial attempt at translation from VDM-SL to AMN was not as successful as we had hoped. We introduced a large number of errors into the specification at this stage which, although they were quickly caught, we believe could have been avoided. With hindsight, we realise that we were mixing the change of notation with some other concerns; namely, a change of data model, a change from implicit to explicit operation definitions, and the introduction of more structure into the specification.

We believe a better separation of concerns could have been achieved by performing these three kinds of refinement within one or other of the two notations.

The change of data model came about because composite types are not directly supported in AMN, rather they can be modeled by giving explicit constructor and selector functions. The mismatch could have been resolved in a number of ways, for example, the use of composite types could have been avoided altogether, or they could have been refined away before translation was undertaken.

The move from implicit to explicit operations, could also have been separated from the translation. The refinement could have been undertaken in VDM by using explicit operations, albeit with implicit constructs in the defining expressions. Alternatively, definitions could have been used in the AMN to mimic the implicit operations.

The introduction of more structure into the specification arose from the use of the extensive machine structuring facilities available in AMN. However, the B-style of incremental presentation of the overall specification did not sit well with the monolithic VDM style. A safer way to perform the translation might be to use the same structuring mechanism on both sides of the translation. The AMN structuring facilities could then be reintroduced in the B development as desired.

ISSUE 3:

In a tool supported development, the unsupported stages are likely to be the most error prone. It is clearly advantageous, therefore, to minimise the complexity of the unsupported stages.

ISSUE 4:

Even if the models are sufficiently close for the syntactic translation between notations to be straightforward, there may be pitfalls arising from the different semantic bases of the notations.

For instance, VDM-SL has a denotational semantics [ISO93] whereas B semantics is based on weakest precondition [Abrial93].

Looseness in VDM expressions is interpreted as underspecification, whereas VDM operations may be truly non-deterministic [Larsen94]. In the semantics of B, loose constructs are only available in substitutions so the expression language would appear to be fully deterministic.

To give us greater confidence in the uniformity of semantics across the various tools used for supporting VDM-SL and B, we chose to interpret all loose constructs as underspecification This approach is aided by the style imposed by VtP, where all predicate expressions are "named" by their definition as auxiliary functions.

It is worth noting also that the ability to type-check the VDM before translating it into AMN gives greater type security than is offered by the Abstract Machine Notation alone.

7 ANIMATION IN B

Animation was used as a validation technique to reveal faults as early as possible in the life cycle. There are two kinds of animation supported by the B-Toolkit: the direct “execution” of specifications using values supplied by the user, and the actual execution of implementing code using an automatically generated test harness.

Execution of specifications occurs once static checking has been successful. (The specification is "analysed" in B-Toolkit terminology.) This allows test cases to be run against the specification to reveal faults before refinement begins. Two sorts of faults could be discovered at this stage: verification faults, relating to the consistency of the formal model with itself which could also be found by discharging proof obligations; and validation faults, which reveal a mismatch between the specification and the informal

requirements, the kind of fault that would not be found by discharging proof obligations alone. The latter, in particular, makes animation a valuable tool to use to ensure a common understanding of the operation of the system, albeit at a rather abstract level.

The experience of this development was that most of the operations in the QAT specification could be animated as they were written and this allowed valuable early testing to be performed. However, three kinds of problem were encountered during animation and test execution of the implementation, which had an influence on the way the specification was written:

Over-complex constructs in substitutions: some operations could not be animated as they were written simply because the set of rewrite rules in the supplied animation theory was not rich enough to simplify the instantiated substitutions. For example, there was no rewrite rule for domain deletion. This problem could have been overcome either by adding additional rewrite rules to the animator's theories or by writing the substitution in a different way so that the existing rewrite rules could manipulate it. For pragmatic reasons, during this project, the latter option was taken. For example, specific domain anti-restriction substitutions were rewritten using the set subtraction operation which had a corresponding rewrite rule. A more satisfactory long-term solution would be to develop (and prove) general rewrite rules for the complex constructs.

Interface for animation: within the B animator, there is no way to manipulate the system state except via the specification's operations. This was occasionally a problem when the person animating the specification wished to initialise the state in a manner which could not be done easily by the use of the operations. The solution chosen was to add temporary operations for the animation step which were removed before completing the specification. While slightly irksome, this process solved the problem well enough for this situation.

Limitations in interface generation: before executing an implemented machine in the B-Toolkit, an interface, or test harness, has to be generated. The generator as it stands is limited in the types of argument with which it can cope. The effect of this is that one is tempted to build an implementation according to the limitations of the interface generator. The resulting interface to the C code was not, in our case, satisfactory for interfacing with other components of the system.

ISSUE 5:

For maximum benefit, animation should be possible without any supplementary design input on the part of the user, so encouraging its use whenever a specification is created or changed. The desire to animate should have a minimum influence on the design choices made.

Despite these problems, which were simply shortcomings of the support tools and in no way foundational, animation, immediately following the hand translation into B from VDM-SL, was a valuable means to validate the translation process. It revealed 5 faults, 4 of which related to errors in the translation from VDM, but one of which reflected a fault already present in the VDM specification (where the current alarm configuration could be deleted leaving an inconsistent state). This last fault could in principle have been found earlier by discharging the VDM "feasibility" proof obligation, however, this had not been attempted. (The fault was also discovered independently during the proof of the B Specification discussed in Section 10.)

8 REFINEMENT IN B

Once the specification had been expressed satisfactorily in B and validated by animation (as far as was possible and considered desirable) the project activity was directed towards producing a working implementation of the software system. The B Method and notation allow a number of steps (refinements)

to be taken between the specification and a final design (implementation) step. Each of the refinement steps results in a B "refinement" and the final design step results in a B "implementation". The implementation can then be used to generate compilable source code using the B-Toolkit's code generation technology. The key difference between the intermediate B refinements and the final B implementation is that while non-determinism is allowed in the definition of the operations of a refinment, this must be resolved in an implementation, hence allowing code generation to take place more directly.

In the case of this project, it was decided (considering the relatively simple nature of the system being developed) to perform a single implementation step directly from the specification and not to produce any intermediate refinements.

When performing the design process with the B-Toolkit, it is important to understand the library technology (including the Base Generator) which is available to support the development. Apt use of this technology can significantly reduce the development time and also reduce the likelihood of errors due to the large amount of "free" implementation that can be gained. Briefly, there is a library of reusable machines supplied with the B-Toolkit (the "B-Library"). It is mainly concerned with data storage and includes some abstract types (such as boolean and scalar) as well as sets, sequences, arrays, variables and functions. Within the library there are pre-implemented specifications of the machines which manipulate this data.

The Base Generator (GenBase) in B-Toolkit allows the user to describe complex structured system states in terms of library specifications. It creates the appropriate B machines and implementations for the system state described, resulting in an implemented B machine with operations to manipulate the elements of the desired state in a standard manner. No more design input is required from the developer.

In this project, consideration of these aspects of B development led to a three stage design process:

Abstract Data Type Implementation: the state of the desired implementation was designed in terms of a number of abstract data types. These data types were described using the GenBase script language and then GenBase was used to generate their implementation to provide a basis for implementation of the system.

State Implementation: the outputs of the GenBase generator were used as the basis of an initial implementation of the system state. The B-Library was used to implement any of the elements of the system state which could not be conveniently represented using GenBase as well as those elements that were felt to be too simple to justify the overheads of using the generator. This step resulted in a completely implemented system state from which (in principle) code could be generated.

Operation Implementation: an implementation of each operation in the specification was defined using the deterministic constructs of the AMN notation. These operation implementations manipulated the concrete system state (via the operations defined on the machines from which it was constructed) to produce the system behaviour defined by their B specification.

Inevitably the implementation of the operations revealed limitations in the implementation of the system state and so the above three steps were iterated until a satisfactory system was constructed from which source code was generated and compiled.

Although the design process for the system was relatively straightforward, three significant problems were encountered, namely:

Method Definition: for an approach to software development to be considered to be a method, four aspects must be present; an underlying model of the development process, a set of well defined notations, a clear definition of a series of steps to follow from start to end and finally, a set of

guidelines for the application of the approach. The B Method has a set of well related notations and there is a concept of an underlying model of the development process. However, the lack of a clear method definition and associated guidelines did cause confusion at the outset.

Interface Refinement: the B Method does not directly support refinement of the system interface. So the "abstract" specification interface must be the required implementation interface. The full implications of this were not completely understood early enough in the development process for this project and rework of the specification was required to allow a satisfactory interface to be constructed.

ISSUE 6:
A means to support the 'realisation' of the types of operation parameters and result during development is required. Whilst not a formally refinements, such design decisions are commonplace during the removal of abstraction from an overall design.

Implementation Interface: some difficulty was encountered in producing an acceptable API (Application Program Interface) for the product using only the C code translator in the B-Toolkit. Those adopting a similar approach should investigate the concrete interface that will be produced for a particular abstract interface to ensure that it is acceptable for use in their situation.

ISSUE 7:
Code generators may not be flexible enough to produce the style of interface required for components that are to be integrated into existing environments. A hand-coded interface may be necessary to encapsulate the generated code.

9 CODE GENERATION: B TO C

The process of generating C language source code from the B implementation is totally automatic, requiring no human intervention. Our experience was that in general it was very easy to generate an initial C implementation from the toolkit. The toolkit is simply instructed to generate source code for one or more B implementations and the code is created in a number of files. Provided that this source code is exactly what is required, no further action is required (other than to compile and run it). However there are a number of potential problems associated with this part of the lifecycle:

Packaging: no control is available over how the generated code is packaged. If it is not in the appropriate form for delivery (for example to unit test teams) then this complex and error prone task must be performed manually.

Coding Standards: there is no ability within the tool to influence the style or specific form of the generated source code. This means that the generated source code will not conform to any particular coding standards which may be in force within the organisation. This may not be acceptable within the software process in use and so manual (or possibly automated) reformatting and presentation of the code may be required. For example, no comments are inserted which implies that if they are required, this will be a major task whenever code is generated.

Trace and Debug Code: most software products use standard code for tracing and debugging the product in the field. This code must always be inserted at predefined points within the product. With automated code generation there is no means of performing such a task and so again this must be achieved by other means.

Use of the Generated Code: the C code generated is not particularly robust and would not generally be considered to form an acceptable API for general use. Before releasing the product for general use, it may well be necessary to write a "wrapper" API which presents a more robust and conventional interface for conventional applications.

The above problems with code generation while irritating would not stop the product being used in a commercial environment. Most of them could be solved with special purpose post processing tools which could be written on a site by site basis. However, this is an aspect of code production is generally ignored by automatic code generation tools.

ISSUE 8:

To be useful for coding in an engineering environment, where procedures and standards exist in a wider context, and where components are to be integrated with others not created in the same way, the flexibility of code generation is a vital issue.

It could be argued that, if something like the B-Toolkit is used to create code, then maintenance of that code should take place at the AMN implementation level using the toolkit and code generator, removing the need to examine the C code at all.

In reality, conformity of coding standards across a complex system is an important factor for ease of maintenance of the whole. A balance is needed between reuse and prescription.

10 THE USE OF PROOF

As stated earlier, no proof was attempted at the VDM stage, however, some significant effort was devoted to the use of the support for proof provided by the B-Toolkit.

The proof activities performed were:

- automatic proof of proof obligations,
- inspection of the remaining proof obligations,
- interactive proof of proof obligations including the development of new “theories” of the application domain and new primitive rules for the basic language.

in practice the autoproving stage was used as a quick (in human time if not machine time) filtering of the proof obligations in order to focus attention on the remaining ones.

The inspection of the remaining proof obligations was a fruitful exercise even before any attempt to proof them was made. Many of the proof obligations were considered likely to be true and so attention was focused on those which remained in doubt. Although not rewarding in terms of the number of proof obligations discharged, this approach did lead directly to the uncovering of faults.

Typically the rectification of these faults required the strengthening of a precondition or an invariant, however one more significant error was uncovered which required the rewriting of the specification to distinguish the dual purpose of an operation by breaking it down into two parts. In the proof of one machine 4 out of 19 unproved proof obligations disclosed faults, 13 of the remaining 15 were proved by addition of assumptions, 2 were left unproven. In one exercise, 7 faults were found in 3 days work and a good level of confidence was achieved in the parts of the system considered. Although admittedly many of these faults were also found (more cheaply) in animation, some had not been detected there.

Some of the rules added to the user theories were rather unwieldy. When defining rules with the proof of given formula in mind, there is a distinct temptation to add rules concerning the particular constants of the specification rather than asserting generalities. Such rules cannot be validated without a detailed knowledge of the specification and, as they are effectively axioms of the proof system, could easily introduce an inconsistency into the proof theory reflecting a fault in the specification. In this project, all the rules added were defined purely in terms of metavariables (“jokers” in B terminology), however, despite this, the goal directed nature of the proof task did lead to the use of some rather problem specific axioms.

One fault worth noting arose in the definition of the rules themselves. A simple typographical error involving the capitalisation of a variable name was made in a proof rule which made the rule unsound. This rule could potentially have been used to prove a whole raft of untrue obligations and thus verify an erroneous specification. It is salutary lesson in the pitfalls of adding unproven rules to an automated proof theory.

On the other hand the benefits of automation cannot be overstated, without it, the shear number of trivial proof obligations would have made the task of even an initial inspection infeasible and would thus have obscured some obvious falsities. A complementary use of automatic proof would also have been useful. Many of the obligations which reflected specification faults were patently false. An initial pass with an automated tool trying to prove these obligation false is likely to have detected a high proportion of them.

ISSUE 9:

> The purpose of verification is to find faults in a specification. Thus effort should be devoted to establishing the falsity of proof obligations rather than their veracity. It is pointless expending effort in proving the 99% of obligations which are true if the status of those which are false is not uncovered.

11 CONCLUSIONS

In this paper, we have described our experiences concerning the the tools and techniques employed in this project. We have highlighted a number of general issues which we feel should be addressed in order to make their use more successful. A quantitative analysis of nature of the errors specific to this development, and a breakdown of the stages at which they were introduced and detected is in preparation.

The experiment has demonstrated the feasibility of tool-supported formal development but initial results highlight the problems of any remaining unsupported tasks. The largest number of the faults were introduced by the hand translation between notations. Naturally, it would have been better to have used a single notation throughout, perhaps through the use of diagrammatic tools for B, but in a short timescale project such as this, we were constrained to use tools that were available. However, the errors that were introduced by the hand translation were, in any case, soon revealed in animation and in proof.

The experience of this project has been a positive one, and the development process undertaken successful. Furthermore we believe that, with hindsight, we could refine the process and circumvent many of the problems described here in subsequent developments. For example, one lesson learned was that it is easier to do several small steps in refinement than one large one, and in particular that any unsupported steps should be made as trivial as possible.

We have reported on the experience of the use of formal methods in an industrial project. The problems we encountered were, on the whole, relatively minor and no doubt will be addressed by future generations of support tools. However, existing tools already make the formal approach possible and the time is ripe to gather further evidence on benefits and difficulties of using these techniques.

REFERENCES

[Abrial93] J.R. Abrial. The B-Book. Cambridge University Press, 1995. To appear.

[Bicarregui91] J.C. Bicarregui and B. Ritchie. Reasoning about VDM developments using the VDM support tool in Mural. In Proceedings of VDM '91, Formal Software Development Methods, Lecture Notes in Computer Science, Springer-Verlag, 1991.

[Bicarregui93] J.C. Bicarregui and B. Ritchie. Invariants, frames and postconditions: a comparison of the VDM and B notations. In Proceedings of Formal Methods Europe '93, Lecture Notes in Computer Science, Springer-Verlag, 1993.

[Dick91] Jeremy Dick and Jerome Loubersac. A Visual Approach to VDM: Entity-Structure Diagrams. Technical Report DE/DRPA/91001, Bull, 68, Route de Versailles, 78430 Louveciennes (France), January 1991.

[Dick93] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J.C.P. Woodcock and P.G. Larsen, editors, FME'93: Industrial-Strength Formal Methods, pages 268–284, Formal Methods Europe, Springer-Verlag, April 1993. Lecture Notes in Computer Science 670.

[Jones91] C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. Mural: A Formal Development Support System. Springer-Verlag, 1991.

[Larsen94] Peter Gorm Larsen. Evaluation of underdetermined explicit definitions. In FME'94: Industrial Benefit of Formal Methods, Lecture Notes in Computer Science, Springer-Verlag, October 1994.

[ISO93] VDM-SL. VDM Specification Language Proto-Standard. ISO/IEC, JTC1/SC22 N1346, BSI IST/5/50, April 1993.