Test Case Preparation using a Prototype

H. Treharne[1], J. Draper[2] and S. Schneider[1]

[1] Department of Computer Science, Royal Holloway,
University of London, Egham, Surrey, TW20 0EX, UK.

[2] Mission Avionics Division, GEC-Marconi Avionics Ltd,
Airport Works, Rochester, ME1 2XX, UK.

E-mail: helent@dcs.rhbnc.ac.uk

Fax: +44 (0)1784 439786

Abstract. This paper reports on the preparation of test cases using a prototype within the context of a formal development. It describes an approach to building a prototype using an example. It discusses how a prototype contributes to the testing activity as part of a lifecycle based on the use of formal methods. The results of applying the approach to an embedded avionics case study are also presented.

Keywords: Prototype, B-Method, Formal Software Lifecycle.

1 Introduction

This paper describes a formal development lifecycle and the practical application of the B-Method [1] within that lifecycle to the production of an executable prototype. The prototype is derived from an abstract specification for use in test case preparation. Knowledge of the B-Method is assumed when reading this paper.

The prototyping approach described in this paper forms part of a software development process which addresses the requirements of the UK Defence Standard 00-55 [13]. It addresses the requirement that tests generated from the executable prototype of a software specification are repeated on the final code.

This work was mainly carried out as part of the Measurable Improvement in Specification Techniques (MIST) [5] project and extended during the SPECTRUM project within GEC Marconi Avionics Limited (GMAv). The MIST project was an ESSI Application Experiment which investigated the development of safety-critical software using an integrated approach which combined formal and conventional software engineering techniques for software development. SPECTRUM was an ESPRIT RTD project looking at the feasibility of integrating the formal methods B and VDM and the industrial benefits of such an integration.

The rest of this paper is laid out as follows. Section 2 describes a software development lifecycle which uses the B-Method. Section 3 details the typical testing activity carried out in a software development lifecycle. Section 4 describes the testing process in a formal development lifecycle. Section 5 shows how test cases are built based on the use of a prototype. The final sections contain results of building a prototype for an embedded avionics case study, a discussion, and conclusions.

2 Overview of Formal Development Lifecycle

A formal development lifecycle (shown in figure 1), which would be used to develop critical functions of embedded avionics systems within GMAv, starts with a set of requirements written in an informal but structured notation. The B-Method is used to re-specify these requirements and produce a formal abstract specification written in Abstract Machine Notation (AMN) to provide an operational description of the software. This includes a description of all the inputs and outputs of the system and all the critical functions that will be provided by the formally developed software. Some of the lower level implementation details are not included at this abstract level.

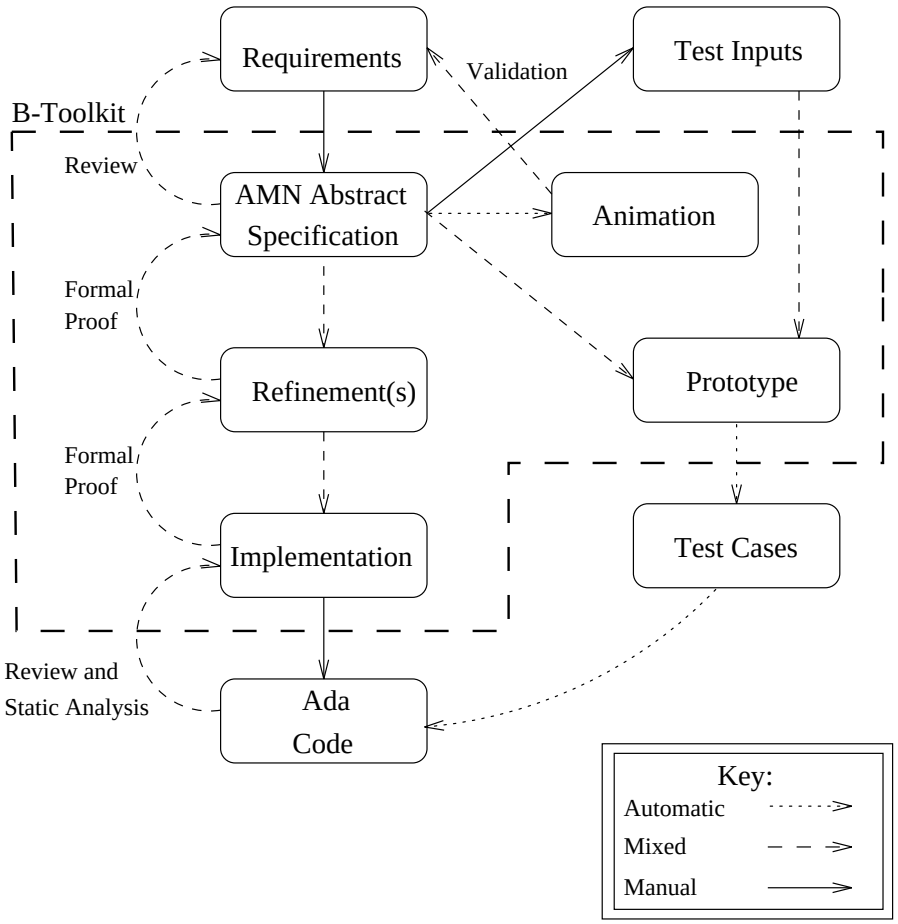

Fig. 1. Formal Development Lifecycle

The abstract specification is animated, formally proved consistent and, typically, reviewed by an independent verification and validation team. The abstract specification contains enough detail to continue development along two independent parallel paths. The abstract specification may contain some limited non-determinism. The issues arising from prototyping non-deterministic specifications are discussed in section 7.

Both parallel development paths produce executable code. The main development is the formal refinement process which leads to an AMN implementation that is currently hand translated into

Ada code. Ada is the preferred programming language for safety-critical software.

In the refinement there are a number of intermediate levels where algorithms and data types are expanded by adding more detail. This design process is discussed in [7] and [8]. The Ada code is verified by review and tested using test cases that are generated using the prototype, as described in section 4.

The secondary development path involves the production of a prototype and the automatic generation of C code which can be executed to produce test cases which are applied to the Ada code. The efficiency of the C code of the executable prototype is not a concern because it is not used in the final system.

3 Testing in a Conventional Software Development Lifecycle

In a software development lifecycle the testing process can be broken into 4 stages; Test Planning, Test Case Preparation, Test Performance and Test Output Review.

Test planning is the process of identifying and documenting the required tests for a software system. In software testing there are three levels; Software Unit Testing, Software Integration Testing and Hardware/Software Integration Testing.

The levels of testing are illustrated in figure 2, which shows the scope of each level in terms of the code objects being tested and their relationship with the design or requirement target.

Software Unit Testing (SUT) tests each individual software module against the low level requirements for that module specified in the software detailed design.

Software Integration Testing (SIT) test groups of integrated modules which implement a specific function described in the top level design. The main aim is to demonstrate that the modules work together correctly with respect to data flow and end-to-end correctness.

Hardware/Software Integration testing (HSIT) tests the fully integrated software within the target environment. It tests the software against the external interfaces and functionality described in the software requirements. It may also test individual interface modules as a addition to the basic unit tests.

The test environment for each level of testing (host or target) is also documented during test planning. In addition to deciding what functionality will be tested, test coverage measurements are defined that must be achieved during testing of each level. This will provide some assurance of the quality of the software in certification and provide evidence that the software has been exercised thoroughly.

These measurements fall into two groups based on functional (black box) and structural (white box) testing [12]. For example, during Software Unit Testing the minimum functional measurement would be that each function or procedure is tested at least once. More stringent requirements would be that each parameter of a procedure or function is is tested with a range of values. For numeric values there would need to be a test case for the minimum, maximum, intermediate and out of bound values. Measurements based on structural testing would consider which statements have been exercised and which branches have been traversed. In addition to traversing each branch of a conditional statement it may also be necessary to achieve a decision coverage measurement which ensures that every boolean value is combined in all the different combinations at least once.

During Software Integration Testing it is expected that the functional test coverage measurement will ensure that combined functions and procedures are being called rather than individual functions and procedures. Furthermore the structural measurement will ensure that all the lower level procedures and functions have been called.

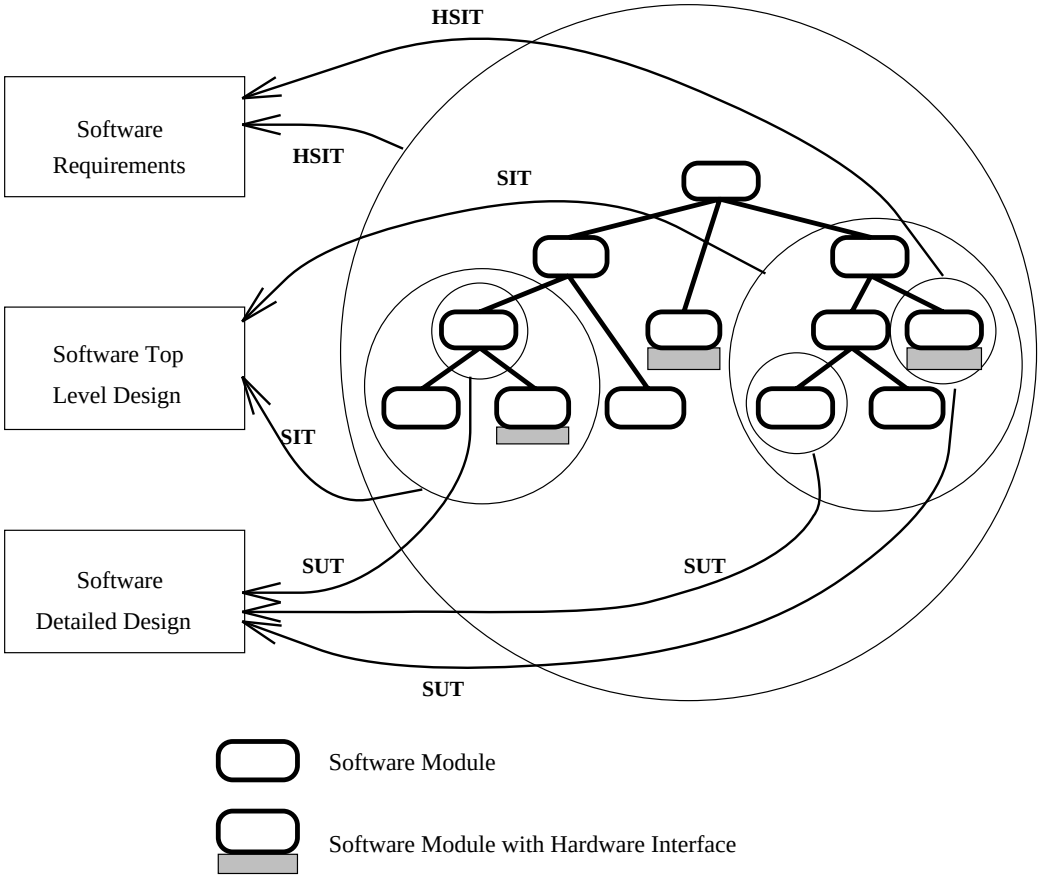

Fig. 2. Levels of Software Testing

Test case preparation is the next stage in the testing process. In this stage, test case inputs are selected and expected outputs (or a range of acceptable outputs)are identified. The test cases are validated to ensure that the functional coverage metrics have been met.

The test performance stage is the process of applying the test inputs to the software and recording the generated outputs. Tools support the test performance process by automatically applying the test and recording the results, usually by generating a test harness. Tools can also generate stubs for lower level modules to allow higher level modules to be tested before they are integrated.

The final stage in the testing process is the test output review. This checks that the generated outputs are equivalent to the expected outputs and that the structural coverage metrics have been satisfied. Tools can help the output review process by automatically comparing the actual and expected output and by calculating the coverage metrics.

4 Testing in a Formal Development Lifecycle

If the software has been developed using the B-Method then it is possible to use the formality of the top level specification and the automatic features of the B-Toolkit to assist in the test case preparation of the testing process. A prototype is built, as shown in section 5, and is used in the

following three phases [14].

Firstly, the abstract specification is analysed in order to identify all the behaviours to be tested. At present it is done manually but tools are being developed which will automate the process for B and Z [9]. Once the behaviours have been identified, input values are chosen which will exercise these behaviours. Boundary value analysis and equivalence partitioning are used to choose these inputs. The inputs for the different behaviours are combined using a cross product to give a large set of test inputs. In order for the executable prototype to run with the test inputs they are translated into a suitable format. The prototype has a simple menu driven interface, offering the system operations. The inputs are extracted from the input list and written into a text file with the appropriate operation calls inserted where required. This formatting is currently performed with ML [16] but could be done with any suitable tool.

The executable prototype is run with the test inputs and the resulting outputs are stored in a file. The combination of test inputs and expected outputs gives a large suite of test cases.

The second phase is the refinement of the test cases into a format that is suitable for use on the Ada code. All the values used in the test cases are abstract representations which must be converted into concrete form to match the interface to the final Ada code using ML. Most of the conversions are simple mappings but some will require complex calculations. For example, an input might be modelled as an enumerated set at the abstract level and refined into a sequence of booleans representing hardware registers at the Ada code level. The result of test case refinement is a test description file which describes the concrete test cases in a format that is suitable for use by the test case application tools.

The last phase is the execution of the Ada code with the test cases. This phase is supported by test application tools, such as AdaTest and TestMate. These tools generate test harnesses and test stubs, as well as applying the test, recording and comparing the actual and expected results. Any test failures are reviewed to see whether they were caused by an error in the main development path or an error in the test case generation. Coverage analysis is also carried out to confirm that all the functionality of the Ada code has been exercised.

The test cases generated using the prototype contribute to the functional testing of all the levels of testing described in section 3. Test cases which exercise the abstract inputs and outputs of the top level functions of the specification will form part of Hardware/Software Integration Testing. The tests of the top level functions which call lower level functions contribute to Software Integration Testing. Testing of the lower level functions will be used in Software Unit Testing.

The test cases are derived from examining the structure of the formal specification, therefore the functional coverage measurement of the unit tests and the structural coverage of the software integration tests are expected to be high. It is also anticipated that, since there are a large number of tests produced, the structural coverage of unit testing will also be high. However, not all the requirements of the system are necessarily embodied in the executable prototype because it is derived from an abstract specification. The abstract specification contains only what is required to describe the safety functions and properties of the software in the system. This obviously affects the coverage metrics that can be achieved during testing. It may be necessary to consider an incremental development of the prototype to cover the new functionality introduced during the refinement process. Alternatively, additional tests can be added manually.

5 Example Prototype

This section uses a small example to illustrate the process of building a prototype using Version 4 Beta Release of the BToolkit [2]. The input to the process is an AMN abstract specification which is a formal description of the functionality of the system. A prototype **MACHINE** and **IMPLE-**

MENTATION are manually written based on the abstract specification, as shown in figure 3. This effectively provides a test harness for the specification. The automatic coding features of the Base Generator are used to generate all the **MACHINE**s which support the prototype **IMPLEMENTATION** [3]. The **IMPLEMENTATION** together with its **IMPORT**ed **MACHINES** form one development layer and capture all the functionality of the system. Once the **IMPLEMENTATION** is written, the Interface generator is used to provide a menu-driven interface which allows testing of the **OPERATIONS** in the prototype. This example only discusses those parts of the process which requires human intervention.

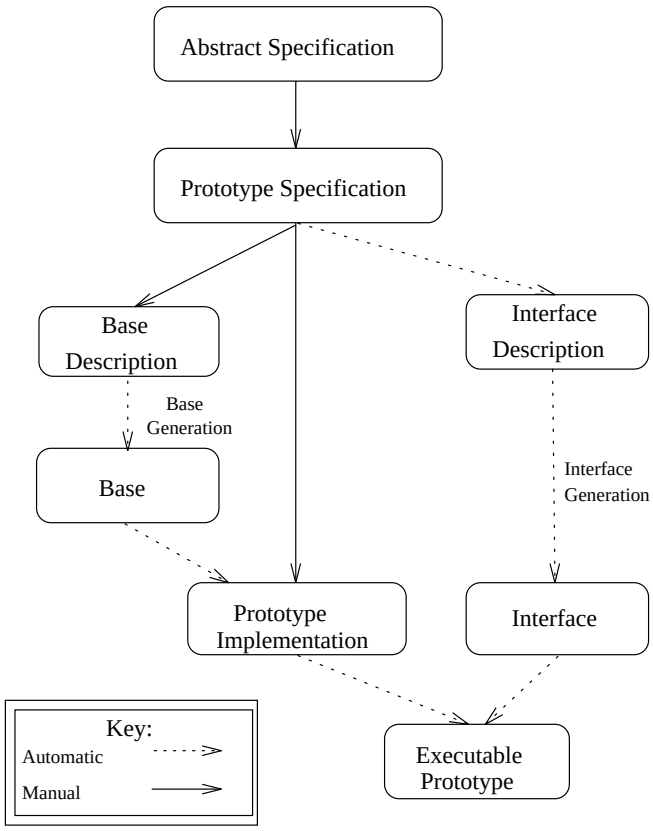

Fig. 3. Prototyping Lifecycle

The example used is a simple, embedded system. The software receives input commands from one hardware interface, validates them and stores the input commands which are used by the main functions of the software. This is typical functionality of embedded safety-critical systems where inputs are received from the system and used by the embedded software.

5.1 Abstract Specification

In the abstract specification the states of a system are represented as **VARIABLES** in **MACHINE**s. These variables have a type and an initial state. For example, in figure 4 the variables *com* and *checkcom* which represent part of the state of the system have the type subset of *COMMAND* and are initialised to the empty set.

```
MACHINE
    Abstract
SETS
    COMMAND = { com1 , com2 , com3 }
CONSTANTS
    specialCommands
PROPERTIES
    specialCommands = { com1 , com2 }
VARIABLES
    com ,
    checkcom
INVARIANT
    com ⊆ COMMAND ∧
    checkcom ⊆ COMMAND
INITIALISATION
    com := ∅ ||
    checkcom := ∅

OPERATIONS
    LoadCom ( inputcoms )  ≙
        PRE
            inputcoms ⊆ COMMAND
        THEN
            com := inputcoms
        END ;
    UpdateCheckCom  ≙
        BEGIN
            checkcom := checkcom ∩ specialCommands ∪ com
        END ;
    outcom ⟵ GetCheckCom  ≙
        BEGIN
            outcom := checkcom
        END
END
```

Fig. 4. Example Abstract Machine

Operations which manipulate and query the state are also defined in a **MACHINE**. In the example there are three operations; the load operation which takes commands as inputs (via the parameter *inputcoms*) and assigns them to the state variable *com*, the get operation which provides a query to retrieve the validated state *checkcom* and the update operation which provides a validation function which updates the validated state.

The validation function makes use of the constant *specialCommands*. The constant is defined in the **CONSTANTS** clause and the explicit property of this constant is included in a **PROPERTIES** clause. In some instances properties of the declared constants may still be abstract. However, all constants must have explicit properties in order to write a prototype **IMPLEMENTATION**. The structuring mechanisms provided by the B-Toolkit support the adding of these explicit properties during the prototyping process. This separation of concerns provides a way of clearly identifying the extra detail added during prototyping.

5.2 Prototype Specification

The purpose of the prototype **MACHINE** is to capture the functionality of the system so that it can be exercised using a menu-driven interface and to remove non-determinism in the abstract specification. This example is deterministic and therefore concentrates on providing a suitable reference basis for test case generation.

The prototype **MACHINE** is built on top of the abstract specification using the **INCLUDES** clause. This means that the prototype **MACHINE** inherits all the state (*com* and *checkcom*) and can use all the **OPERATIONS** of the abstract specification.

OPERATIONS which do not have any input or output arguments can simply be promoted into this new specification **MACHINE** as shown by *UpdateCheckCom* in figure 5. However, **OPERATIONS** with abstract inputs and outputs need to be handled differently. These **OPERATIONS** are replaced by new **OPERATIONS**. The new **OPERATIONS** simply define a new interface but the information content will be the same. This is necessary so that inputting and outputting of parameter values can be implemented in the body of these new **OPERATIONS** using the support of the automatic coding features of the BToolkit.

For example, in the abstract specification there is an **OPERATION** which loads commands, *LoadCom*, and assigns the input argument *inputcoms* to the state variable *com*. In the prototype specification, *Proto*, the operation is replaced with *PLoadCom*. The body of the specification indicates how the state of the system changes; a set of the correct type is assigned to the state *com* via a call to *LoadCom*. This clearly shows that the effect on the system is the same whether *LoadCom* or *PLoadCom* is called. However, the interface to the *PLoadCom* **OPERATION** makes it possible to write an implementation which allows the user to input each element of the set interactively using the menu-driven interface.

Another **OPERATION** which illustrates this interface refinement is the query **OPERATION** *PGetCheckCom*. The purpose of this **OPERATION** is to output the value of the state *checkcom*. Notice that the specification for this new output **OPERATION** is different to *PLoadCom*. *PGetCheckCom* does not change the state *checkcom* and so the function skip was used in the body of the specification to indicate that the value of the variable does not change.

The prototype specification also provides new **OPERATIONS** so that all the state variables in the system can be set and retrieved. This is important during test case generation because the system must be put into a specific state before running the test scenario. Thus in this example the prototype specifies an additional access **OPERATIONS** for the state variable *com*, *PGetCom*.

5.3 Base Generation

The state of the prototype system defined in the specification must be completely encapsulated in the prototype **IMPLEMENTATION** to allow a direct translation into C source code. The example needs to encapsulate the state variables, *com* and *checkcom*. The state is not of simple types such as naturals, strings or booleans and so the Base Generator is used to build appropriate encapsulating structures for these sets.

The Base Generator takes as its input a description of the structure of the state it captures. Figure 6 shows declarative statements for *cmd* and *checkcmd*. The identifiers are of the same set type, *COM*, with a maximum size of three elements. The identifiers must be different from the state variables but a link between the commands will be established in an **INVARIANT** when the prototype **IMPLEMENTATION** is written in section 5.4.

The Base Generator uses this description to build a system **MACHINE** containing a collection of **OPERATIONS** which form a basic instruction set that can be used to manipulate the *cmd* and

```
MACHINE
    Proto
INCLUDES
    Abstract
PROMOTES
    UpdateCheckCom

OPERATIONS
    PLoadCom  ≙
        ANY  comset  WHERE
            comset ⊆ COMMAND  THEN
            LoadCom ( comset )
        END ;
    PGetCheckCom  ≙
        BEGIN
            skip
        END ;
    PGetCom  ≙
        BEGIN
            skip
        END
END
```

Fig. 5. Example Prototype Specification Machine

checkcmd data structures. For example, an **OPERATION** *add_cmd* with one input argument will add a command to the set *cmd*. The **OPERATION** returns a boolean value indicating whether the command element has been added to the set *cmd* successfully. These instruction sets are used to implement **OPERATIONS** in the prototype **IMPLEMENTATION**.

Sets and sequences which are **CONSTANTS** also need to be encapsulated using base generation. Thus the declarative statement for the identifier *specialcmd*, in figure 6, is used to encapsulate the structure of the constant set *specialCommands* in the example. Again a link will be provided between the identifier and the constant in the **INVARIANT** of the prototype **IMPLEMENTATION**.

5.4 Prototype Implementation

The final manual stage in producing the prototype is to make use of the **MACHINE**s generated from the system building block in the prototype **IMPLEMENTATION**.

The **IMPLEMENTATION** is written in a restricted subset of AMN. It contains constructs which are familiar to programmers, such as **IF THEN ELSE** and **CASE** statements.

In order to produce code, all the state of the system must be encapsulated in instantiations of library **MACHINE**s or **MACHINE**s produced during base generation using the **IMPORTS** clause, as shown in figure 7. When importing the system **MACHINE** (*ProtoBase*) the abstract enumerated set *COMMAND* is passed as a parameter. This is matched with the formal parameter *COM* in the generated system **MACHINE** which is derived from the set name used in the system declarative statements in figure 6. A *basic_io* **MACHINE** is also **IMPORT**ed so that input and output facility provided by the system **MACHINE** can be used.

```
SYSTEM
    ProtoBase
SUPPORTS
    Proto_1
IS
    GLOBAL
        cmd ∈ SET ( COM ) [ 3 ] ;
        checkcmd ∈ SET ( COM ) [ 3 ] ;
        specialcmd ∈ SET ( COM ) [ 3 ]
    END
END
```

Fig. 6. Example Prototype Base Description

The **INVARIANT** clause in the **IMPLEMENTATION** allows the definition of relationships between the abstract and concrete state. For example, in figure 7 the set variable *cmd* in the system **MACHINE** is connected with the variable *com* in the abstract specification.

In a **MACHINE**, sets are declared in the **SETS** clause, constants are declared in a **CONSTANTS** clause and their properties are defined in the **PROPERTIES** clause. In an **IMPLEMENTATION**, constants and sets are captured in a Base description. When the generated system **MACHINE** is **IMPORT**ed these constants and sets are not initialised. The **INITIALISATION** clause contains **OPERATIONS** to fill the sets with their enumerated elements and provide constants with the appropriate values. For example, *cmd* is initialised to the empty set using the operation *clear_cmd*. Any temporary storage declared in the system **MACHINE** is not initialised until it is used.

The style adopted in a prototype **IMPLEMENTATION** is important. The structure of the clauses should remain as close to the original specification as possible to aid manual verification by review. For example, in a parallel composition of two **OPERATIONS** their sequential implementation should have the same order as that of the specification. This is illustrated by the **INITIALISATION** clauses of figure 4 and figure 7.

Prototype **OPERATIONS** which are specified using a parallel composition of **OPERATIONS** are implemented as one **OPERATION** containing the combined functionality. This tends to lead to a very large flat implementation. However, provided the code is commented clearly it is easy to identify the **OPERATION** boundaries and provide traceability back to the abstract specification. This aids the process of manually verifying the low level AMN against the specification.

The **OPERATIONS** used to load and retrieve the state make use of the *read_cmd*, *write_cmd* and *write_checkcmd* from the system **MACHINE** to handle the inputting and outputting of sets automatically. Notice *outbool* is simply a report boolean indicating success or failure of the operation which is called.

5.5 Interface Generation and Code

Once all the supporting constructs have been generated and the prototype **MACHINE** and **IMPLEMENTATION** have been analysed, code generation is an entirely automatic and straightforward process. An Interface description is introduced which contains all the **OPERATIONS** from the prototype. This list of **OPERATIONS** is reviewed and any that are not needed can be

```
IMPLEMENTATION    ProtoI
REFINES
    Proto
SEES
    Bool_TYPE , Scalar_TYPE , String_TYPE
IMPORTS
    ProtoBase ( COMMAND ) , basic_io
INVARIANT
    com = cmd ∧ checkcom = checkcmd ∧ specialCommands = specialcmd
INITIALISATION
    VAR    outbool    IN
        clear_cmd ; clear_checkcmd ; clear_specialcmd ;
        outbool ⟵ add_specialcmd ( com1 ) ;
        outbool ⟵ add_specialcmd ( com2 )
    END

OPERATIONS
    PLoadCom    ≙
        VAR    outbool    IN
            clear_cmd ;
            outbool ⟵ read_cmd
        END ;
    PGetCheckCom    ≙
        write_checkcmd ;
    PGetCom    ≙
        write_cmd ;
    UpdateCheckCom    ≙
        VAR    spec , cc , outbool    IN
            spec ⟵ val_specialcmd ;
            inter_checkcmd ( spec ) ;
            cc ⟵ val_cmd ;
            outbool ⟵ union_checkcmd ( cc )
        END
END
```

Fig. 7. Example Prototype Implementation Machine

removed. The Interface Generator is used to build the interface **MACHINE**s and automatically carry out all the code translation and linking to provide an executable prototype in C code.

6 Results

The main aim of the MIST project was to apply the B-Method to a Case Study and to compare this with a parallel development of the same system using a conventional software development process [6]. This allowed the B-Method to be evaluated and procedures to be developed which can be used on future avionics projects within GEC Marconi Avionics Limited.

The Case Study used for the MIST project addressed part of the software controlling a Station Unit on a military aircraft. The Station Unit holds one store (e.g. a fuel tank or missile). The Station Unit receives commands from a central armament control unit. These commands can order a

store to be armed or released, or the Station Unit to perform tests on itself. Before reacting to any commands, the Station Unit checks that the message containing the command is valid by performing a number of data encoding checks. The Case Study was restricted to the main control and function of the Station Unit. It did not include the tests performed continuously, on power-up or on demand, nor any of the minor functions performed by the software. The Case Study covers 36% of the total software for a Station Unit.

This section summarises a testing comparison for the main control of the Station Unit (about 9% of the total software). The effort required to code, unit and integration test these functions using the formal approach was 78% of the effort taken using the standard approach. This reduced effort later in the development lifecycle compensates for the greater effort needed in the requirements and specification phases early in a formal lifecycle.

The effort taken to write tests for these critical functions using the formal approach was 12 man weeks. This included 3 man weeks for writing the prototype which could be considered as part of the validation effort for the abstract specification. The effort expended on writing the tests using the conventional approach was only 9 man weeks. However the number of tests that were produced using the formal approach was significantly more than the number developed manually, in the order of 40 times more tests. It has already been stated that 100% coverage of statement, branch and LCSAJ (Linear Code Sequence and Jump) may not be achieved for all operations using tests based on the prototype due to the level of abstraction in the abstract specification. However, the conventional approach achieved lower coverage than the formal approach. It would not be impossible to write the tests needed to achieve the same level of coverage but it would increase the effort required. The highly automated nature of the formal test case generation means that no more effort is needed to produce large combinations of tests. It is also felt that the formally produced tests would be more robust in terms of coverage against minor changes to the Ada code.

The formal development lifecycle is based on an integrated verification approach [10] where each step in the specification and refinement process is verified. Given the high degree of verification it was expected that errors introduced during specification and refinement would also be found early in the lifecycle. This means that any errors found during testing would mostly be due to errors in coding. For example, figure 8 shows where the errors were introduced and detected during the formal development of the critical functions of the case study. As was expected most of the errors were detected by the verification and validation processes. The two errors detected during testing were introduced when translating the AMN design into Ada code.

A fault grid [4] was also produced for the same critical functions which were developed using conventional development methods, shown in figure 8. It shows that most of the errors were found during testing. For example, seven errors were found during unit test which were introduced during the design phase. It is clear that the conventional development lifecycle detects many more errors later in the lifecycle which means that the cost of re-work is relatively high.

7 Discussion

The example in section 5 showed that prototyping deterministic specifications is not difficult. However, there are further issues involved in prototyping non-deterministic specifications.

A prototype **IMPLEMENTATION** cannot contain any non-determinism which means that non-determinism present in an abstract specification must be resolved during its implementation. There are two main problems with resolving the non-determinism. The first problem occurs when the abstract specification contains an error. It is hoped that the review of the abstract specification against the requirements and its validation using animation would capture the error. However, if the error is undetected it gives rise to the following scenario. The prototype could resolve the non-determinism to exhibit the valid behaviour of the system whereas the main development is refined

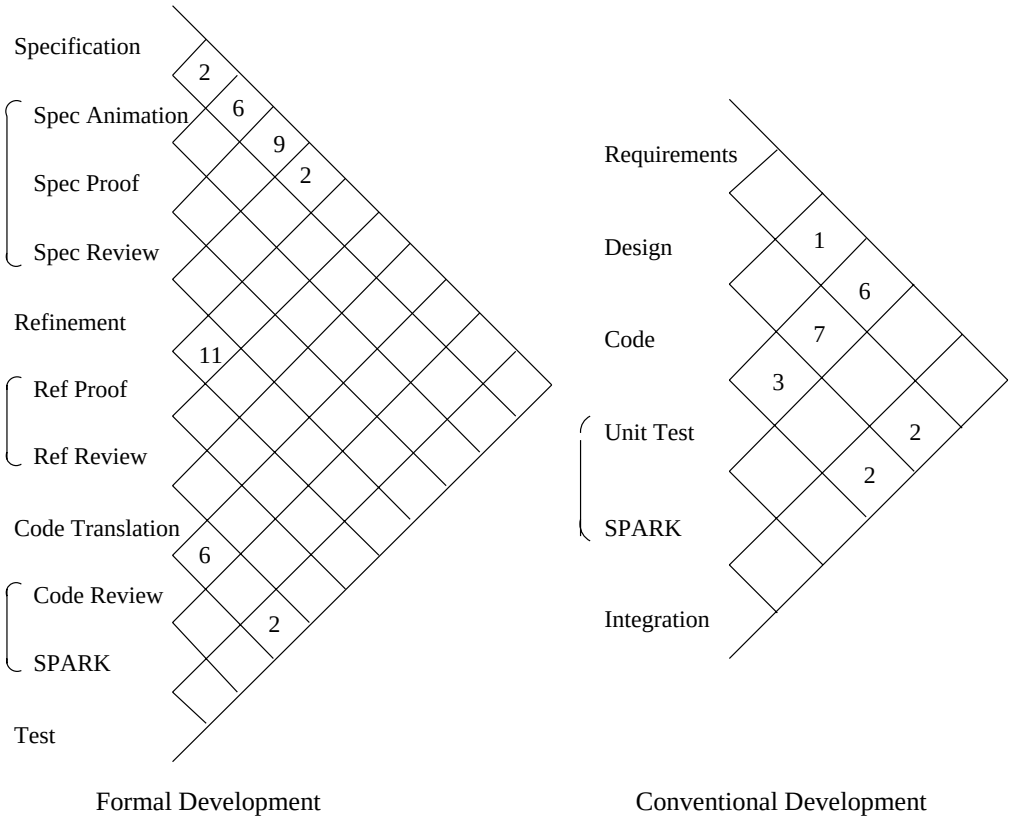

Fig. 8. Errors in Developments

to an implementation which contains the erroneous behaviour. This would be a real error in the final system. Conversely, if the implementation of the final system described the valid behaviour the prototype would be invalid and the test cases faulty.

The second problem arises when the abstract specification contains non-determinism where the prototype resolves the non-determinism one way and the implementation of the system resolves it the other way. This would provide spurious errors in testing.

The style of specification adopted within GMAv is to restrict the non-determinism in the specification and to resolve it as early as possible in order to control both the problems that could arise. In the abstract specifications the only place where non-determinism could occur is in the **INITIALISATION** of the specification or through under-specification in the definitions of a constant function.

In practice, a non-deterministic **INITIALISATION** is not a concern because of the way the system will be used. It is assumed that there is no control over the initial state of the system and so the style used to model the initial system state is with a non-deterministic **ANY** clause. It is always the case that in large systems there will be an **OPERATION** to reset all the state variables to a safe state. This **OPERATION** will always be invoked after powering up the system. Thus the system will never depend on the **INITIALISATION** of the **MACHINE**.

The following **CONSTANT** function, *checkfun*, illustrates how non-determinism is resolved for under-specified functions.

$$checkfun \in \mathbb{P}(\ COMMAND\ ) \times CHECKDATA \rightarrow BOOL$$

which takes two inputs of a power set of type COMMAND and CHECKDATA. In the abstract specification the set COMMAND is enumerated but CHECKDATA is under-specified and remains abstract. This is because it will be refined to several different inputs but the details are unimportant in the abstract specification.

Before the prototyping process can begin the non-determinism must be resolved. Thus the prototyping specification would include an enumeration for CHECKDATA by introducing another enumerated set of *good* and *bad* elements and stating in the **PROPERTIES** clause that this set is equivalent to CHECKDATA. It would also include a property resolving the non-deterministic function. For example,

$$\forall\ ss\ .\ (\ ss \in \mathbb{P}(\ COMMAND\ ) \Rightarrow checkfun\ (\ ss\ ,\ good\ ) = TRUE\ \ ) \wedge$$
$$\forall\ ss\ .\ (\ ss \in \mathbb{P}(\ COMMAND\ ) \Rightarrow checkfun\ (\ ss\ ,\ bad\ ) = FALSE\ \ )$$

It would have been possible to include this detail in the abstract specification but this would have introduced too much unnecessary detail. It would also have involved writing a more complex refinement relation in the main development which would increase the proof effort. However, by excluding it there is a danger of resolving the non-determinism in a different way in the prototype and in the main development. Thus the test cases produced from the prototype would not be appropriate for application to final Ada code.

Nevertheless, the way the non-determinism was resolved still retained the shape of the function. The barest minimum of detail was introduced to make it deterministic and implementable so that when the test cases are refined from abstract to concrete values they could still be refined to be compatible with the final code. There is no explicit refinement relation between the prototype and the refinement specification in the main development, the relationships are embedded into the ML which performs the conversion from abstract to concrete test cases.

One alternative solution would be to produce a set of possible outputs for each case of non-determinism in the specification. Even in this situtation the prototype would still have to define all the abstract sets in order to be able to build the prototype. Producing sets of outputs would add an extra level of complexity when building the prototype and is a topic for further research.

The prototyping aims to be a fast process as shown by the results of the Case Study. This was made possible by building the prototype as one development layer and using B-toolkit library **MACHINE**s. A layered design approach was not adopted. However, in very large systems the prototype **IMPLEMENTATION** would simply be too large to control as one entity. In such cases a layered development approach would be necessary but the principles of resolving the non-determinism early would still apply.

In this paper an executable prototype has been used in the testing process. It could also be used to provide preliminary validation of the formal specification which is another requirement of the Defence Standard 00-55. In the lifecycle described in section 2 preliminary validation of the specification, to identify errors in expected behaviour against the informal requirements, was carried out using the animation facility of the B-Toolkit.

The Ada code produced from a formal development may not be the code for the whole system. Other functions may have been developed using a conventional development lifecycle. Thus all the Ada code would be integrated together to provide the source code for the complete system. The test cases produced for the formally developed code would still be valid and would form almost all the test cases needed to test these formally developed software functions. The only additional tests, developed using conventional testing methods, would be ones needed to achieve the required structural coverage metrics.

8 Conclusions

This paper described an integrated approach to formal software development and focused on a testing process based on the use of a prototype. The approach to prototyping using the automatic code generation features of the B-Toolkit enabled a prototype to be produced rapidly from a formal specification. Tests were generated using this prototype taking no significantly greater effort than the equivalent tests produced using the conventional approach but provided a much higher number of tests and a higher test coverage.

The formal development lifecycle has been tested on a large Case Study which captured all the safety critical functions of a typical avionics system. When a formal approach is used within the development of systems less effort will be required during testing.

Acknowledgements

The authors would like to thank GEC Marconi Avionics Limited for permission to publish this paper. The views expressed in this paper are those of the authors and do not necessarily represent the views of GEC Marconi Avionics Limited.

References

1. Abrial J. R.: *The B Book: Assigning Programs to Meaning*, Cambridge University Press (1996).
2. B-Core(UK): *B-Toolkit Version 4 Beta Release*, Magdalen Centre, The Oxford Science Park, UK (1997).
3. B-Core(UK): *Case Study: Design Using Generated Base Objects*, Magdalen Centre, The Oxford Science Park, UK (1994).
4. Dick J. and Maisey D.: *Fault grids: Another Way of Presenting Fault Counts*, Proceedings of Software Quality Management 1996, Cambridge, UK (April 1996).
5. Draper J.: *Applying the B-Method to Avionics Software: an Initial Report on the MIST Project*, Encress 1995 Conference Proceedings.
6. Draper J., Treharne H. *et al.*: *Evaluating the B-Method on an Avionics Example*, Proceedings of DASIA, Rome (1996).
7. Draper J. and Treharne H.: *The Refinement of Embedded Software with the B-Method*, Proceedings of Northern Formal Methods Workshop, Bradford, UK (September 1996).
8. Hoare J. P.: *Application of the B-Method to CICS*. Chapter 6 in H. Bowen, editor, Application of Formal Methods, pp. 97-123, Prentice-Hall International (1995).
9. Hörcher H-M. and Peleska J.: *Using formal specifications to support software testing*, Software Quality Journal **4**, pp. 309-327 (1995).
10. Kemmerer R. A.: *Integrating Formal Methods into the Development Process*, IEEE Software (September 1990), pp. 37-50.
11. Lichter H., Schneider-Hufschmidt M. and Züllighoven: *Prototyping in Industrial Software Projects - Bridging the Gap Between Theory and Practice*, IEEE Transactions on Software Engineering, **20**(11) (1994), pp. 825-832.
12. Myers G. J.: *The Art of Software Testing*, Wiley (1979).
13. UK Ministry of Defence: *The Procurement of Safety Critical Software in Defence Equipment*, Defence Standard 00-55 (1997).
14. Ormsby B.: *An Approach to Testing during Formal Development with the B-Method*, Industrial Report, Proceedings of 22nd EUROMICRO conference, Prague (September 1996).
15. Waeselynck H. and Boulanger J-L.: *The Role of Testing in the B Formal Development Process*, Proceedings of 6th International Symposium on Software Reliability (ISSRE '95), Toulouse, France (October 1995), pp. 58-67.
16. Wikström Å.: *Functional Programming using Standard ML*, Prentice Hall, International Series in Computer Science (1987).