ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**D Duke, University of York, UK and A S Evans, University of Bradford, UK
(Eds)**

BCS-FACS Northern Formal Methods Workshop

Proceedings of the BCS-FACS Northern Formal Methods Workshop,
Ilkley, UK. 23-24 September 1996

The Refinement of Embedded Software with the B-Method

J. Draper and H. Treharne

Springer

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

The Refinement of Embedded Software with the B-Method

Jonathan Draper

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

Helen Treharne

Royal Holloway University of London,
Egham, Surrey, TW20 0EX, UK

Abstract

This paper describes the use of formal refinement within the MIST project. MIST (Measurable Improvement in Specification Techniques) is ESSI application experiment 10228. It is an 18 month project involving three companies: GEC-Marconi Avionics, who are the prime user; Praxis, who are the main subcontractor, acting as an independent reviewer; and B-Core (UK), who provide the tools used and consultancy. The main aim of MIST is to develop practical procedures for applying formal methods in conjunction with current methods for safety critical avionics software development.

The paper describes a specification style developed by the project that models embedded software within a systems context. It also describes a style of refinement, known as structural refinement. The paper illustrates both with a small example and also reports on their application to a large case study within the MIST project. Initially, there were some problems in using the B-Toolkit with structural refinement, but most of these were overcome by a new B-Toolkit. The embedded specification style worked well and allowed the embedded software to be specified with abstract interfaces and refined with concrete interfaces. The structural refinement allowed the design to be partitioned fairly quickly. Overall, refinement was easier than expected, taking 65 days compared to 48 days needed to write the abstract specification. The proof of the refinement was only achieved because the design had been partitioned by the structural refinement.

1 Introduction

This paper describes the use of formal refinement within the MIST project [1, 2]. MIST (Measurable Improvement in Specification Techniques) is ESSI application experiment 10228. It is an 18 month project involving three companies: GEC-Marconi Avionics, who are the prime user; Praxis, who are the main subcontractor, acting as an independent reviewer; and B-Core (UK), who provide the tools and consultancy. The main aim of MIST is to develop practical procedures for applying formal methods in conjunction with current methods for safety critical avionics software development. It is intended that these procedures will be used to meet standards such as DEF-STAN 00-55 [3].

There are three main phases in the MIST project. First, to propose an initial set of procedures for using the B-Method on parts of embedded real-time systems. Second, to apply these procedures to an avionics case study allowing the procedures to be improved and data collected on the use of the procedures. Third, to use the data to compare the formal development with a parallel development of the same case study using structured methods.

The project developed a style for the application of the B-Method to embedded systems and a

style of refinement, known as structural refinement, where the B-Method is used for partitioning specifications.

The rest of this paper is laid out as follows. Section 2 gives an overview of the B-Method. Section 3 describes the MIST development lifecycle. Section 4 gives an overview of the refinement styles in the MIST project. Section 5 illustrates structural refinement with an example and the final sections contain the results and conclusions.

2 Overview of the B-Method

The formal method used in the MIST project is the B-Method [4, 5] which is supported by the B-Toolkit [6]. The extensive support provided by the toolkit was a major factor in the choice of the B-Method.

The B-Method is a collection of mathematically based techniques for the specification, design and implementation of software components. Systems are modelled as a collection of interdependent abstract machines, for which an object-based approach is employed at all stages of development.

An abstract **MACHINE** is described using the Abstract Machine Notation (AMN), a state-based formal specification language in the same school as VDM and Z. A uniform notation is used at all levels of description, from specification, through design, to implementation. Large **MACHINEs** can be constructed from other **MACHINEs** using the **INCLUDEs** and **SEEs** constructs. **MACHINEs** are refined using **IMPLEMENTATIONs**. **IMPLEMENTATIONs** are constructed using a number of **IMPORTED** lower level **MACHINEs**. The operations of an **IMPLEMENTATION** are described using a programming subset of AMN.

The B-Method prescribes how to check the specification for consistency (preservation of invariant) and how to check designs and implementations for correctness (correctness of data refinement and correctness of algorithmic refinement).

3 MIST Development Lifecycle

The MIST development lifecycle, shown in figure 1, starts with a set of requirements written in an informal but structured notation. The B-Method is used to respecify these requirements and produce a formal abstract specification written in AMN. The abstract specification models the software within the system context. It describes system operations that interface with the software as well as the operations required to be performed by the software. Some of the lower level details are not included at this abstract level.

The abstract specification is animated, formally proved consistent and reviewed. The first level of refinement is carried out on types and constants by enumerating sets and giving values for constants.

There are two parallel development routes that produce executable code from the specification containing refined types and constants. The main development route gradually designs and implements the specification through a number of formal refinement steps. The entire model, including both the system and the software operations, is refined but only the software operations are translated, by hand, into Ada. The Ada derived from the formal specification is interfaced with informally developed Ada in order to produce executable code. The prototype development route uses the code generation features of the B-Toolkit to rapidly produce a C implementation. The C prototype is used to generate test cases to verify the Ada code against the abstract specification.

Figure 1: Software Development Life-Cycle

4 Overview of Refinement in MIST

In the MIST project there were several styles of refining abstract specifications:

1. Data Refinement — to replace abstract states with concrete states,
2. Operational Refinement — to replace abstract algorithms with concrete algorithms,
3. Structural Refinement — to partition the top level structure,
4. Detail Refinement — to expand the abstract state and functions,
5. Type Refinement — to fully define deferred types and constants.

Most of the initial refinements are structural refinements, used to partition the design. The smaller specifications, resulting from the structural refinement, are refined using detail, data or operational refinement. Data and operational refinements build their implementations from library machines or very low level specifications that can be translated into code. Detail refinements increase the complexity of the specification and may be followed by further structural refinement. Type refinement can be performed at any stage, but it is often the first refinement step.

4.1 Data Refinement

Data refinement is the most general form of refinement. It is used to implement abstract state variables using concrete variables. In the MIST project, data refinement was only applied to small component specifications in order to make verification practicable.

The concrete state and operations on that state are defined in a concrete machine, such as library machines. This machine is **IMPORTed** into the **IMPLEMENTATION** machine which implements the abstract operations using concrete operations. The **IMPLEMENTATION** links the abstract and concrete variables by a refinement relation. This relation is usually defined in a separate context machine.

For example, an abstract state could be an enumerated set of values representing hardware registers and their concrete representation could be an array of booleans.

4.2 Operational Refinement

Operational refinement is a specialisation of data refinement where the abstract and concrete types are the same. This makes it easier to verify than general data refinement.

Operational refinement rewrites the abstract operations with concrete operations. Abstract operations are specified using abstract statements such as parallel composition and unbounded choice, whereas concrete operations use concrete statements such as sequential composition and loops.

4.3 Structural Refinement

Structural refinement is a specialisation of operational refinement where the changes to the operation specifications are minimised. It allows the design to be partitioned and each part to be refined independently. Structural refinements are the largest refinements performed on the development, and hence it is important to keep the complexity of the refinement as simple as possible.

Generally, large abstract specifications are built up from a tree of **INCLUDED** machines. The principal step in structural refinement is to write an **IMPLEMENTATION** for the top level abstract machine and **IMPORT** lower level abstract machines. The structure of the lower level abstract machines will partition the development into specifications which can be refined independently. Machines with state variables of the same type will be grouped together.

In the lower level machines the interface to operations must use concrete values so that when the operations are used in the top level **IMPLEMENTATION** the parameter passing between operations is concrete, enabling the operations to be translated directly into code. The lower level machines will also have to provide enquiry operations for their state variables. State variables can be examined directly in abstract specifications but are only visible in **IMPLEMENTATIONs** through enquiry operations. New operations are also provided which perform the abstract functions from the top level abstract specification.

An example of structural refinement is given in section 5.

4.4 Detail Refinement

Detail refinement is a style of refinement that is used to remove under-specification. It expands abstract state variables and respecifies the abstract operations to show how they act on the expanded state. As detail refinement is usually performed on a small specification, after the overall design has been partitioned using structural refinement, it is usually relatively easy to verify.

An example of detail refinement is splitting an abstract message validation function, and its associated check data, into several concrete validation functions.

4.5 Type Refinement

Type refinement is a specialisation of detail refinement where only the types and constants are refined. A new context machine is written that **INCLUDES** the abstract context machine and expands the definitions of the types and constants. An example is where a deferred set is given explicit values. The verification demonstrates that the concrete values for the types and constants introduced by the type refinement, meet their abstract specifications.

4.6 Verification and Validation of Refinement

All the refinements are syntax and type checked by the B-Toolkit. The B-Toolkit also generates the proof obligations to show the correctness of the refinement. For type refinement there are only context proof obligations. All the refinements are reviewed to validate the design. The detail and type refinement reviews also validate the extra information, added in the specification, against the informal requirements.

5 Example of Structural Refinement

This section uses a small example to illustrate the process of structural refinement. The example is a simple, embedded software system. The software receives inputs commands from one hardware interface. It validates the commands and sets the appropriate hardware registers.

The starting point is an abstract specification, which is a formal description of the functionality of the system encapsulated within a collection of abstract **MACHINES**. Structural refinement is then performed to partition the top level structure into two parts that can be refined independently.

5.1 Abstract Specification

The abstract specification models the system and software operations to input, validate and process commands and read the registers. This functionality is described in three separate machines which are combined into one top level machine, as shown in figure 2. This structuring provides a clear view of the system boundary by separating the input commands from the software functions.

Figure 2: Example Abstract Machine Structure

The system machine *Input* in figure 3 contains the variable *com* which represents commands input to the software.

The state variable *com* is a subset of *COMMAND* and is initialised to the empty set. The definition of the type *COMMAND* is provided in a separate context machine shown in figure 3. This separation of concerns provides a clear way of identifying the context information in the abstract

MACHINE	MACHINE
<i>Input</i>	<i>Context</i>
SEES	SETS
<i>Context</i>	$COMMAND = \{ com1, com2, com3 \} ;$
VARIABLES	$REGISTER = \{ reg1, reg2 \}$
<i>com</i>	CONSTANTS
INVARIANT	<i>specialCom</i> ,
$com \subseteq COMMAND$	<i>commandsToRegisters</i>
INITIALISATION	PROPERTIES
$com := \{\}$	$specialCom \subseteq COMMAND \wedge$
OPERATIONS	$specialCom = \{ com1, com2 \} \wedge$
$LoadCommand(inputcoms) \hat{=}$	$commandsToRegisters \in$
PRE	$COMMAND \rightarrow REGISTER \wedge$
$inputcoms \subseteq COMMAND$	$commandsToRegisters =$
THEN	$\{ com1 \mapsto reg1, com2 \mapsto reg1,$
$com := inputcoms$	$com3 \mapsto reg2 \}$
END	END
END	

Figure 3: Example System Input and Context Machines

specification and allows several machines to make use of the same definitions. The *Input* machine provides a system operation, *LoadCommand*, to update the system input.

The input commands are validated in the *Validation* machine, shown in figure 4. The state variable *checkcom*, of the type subset of *COMMAND*, represents the software copy of the input commands after they have been validated. This state is set by the operation *RecordCheckedCommand* which has a precondition that all the inputs must be in *specialCom*. The constant *specialCom* is a subset of *COMMAND*, as defined in the *Context* machine.

All the processing of the commands and the resultant setting of hardware registers is performed in the *Processing* machine, shown in figure 4. The hardware state is modelled as the variable *registers*. The operation *SetRegisters* takes valid commands as input and sets the appropriate hardware register using the constant function *commandsToRegisters*, defined in the *Context* machine. The state of the hardware registers is retrieved by the system operation, *GetRegisters*.

The *TopLevel* abstract specification, shown in figure 5, combines the lower level machines using the **INCLUDES** structuring mechanism. This provides visibility to all the state and the ability to use all of the operations in the **INCLUDED** machines. The system operations to update the input commands (*LoadCommand*) and to retrieve the output state (*GetRegisters*) are **PROMOTED** directly which means that these become operations at the *TopLevel* and are visible to the external system. The top level processing operation (*ProcessCommand*) combines the validation and processing functionality into one operation by calling the two lower level operations from the

Figure 4: Example Validation and Processing Machines

INCLUDED machines. The **IF** statement in *ProcessCommand* models the precondition from the *RecordCheckedCommand* operation in the *Validation* machine. All preconditions, which are not type restrictions, must be captured in this way in the **INCLUDING** machine.

The AMN is structured carefully to enforce the constraints on the embedded software. For example, the load operation in the *Input* machine is externally visible but should not be accessible by the software functions in the abstract specification. This is achieved by **INCLUDING** the *Input* machine at the *TopLevel* but only **SEEing** it from the software machines, *Validation* and *Processing*.

5.2 Structural Refinement

In this section the example abstract specification is structurally refined. The purpose of structural refinement is to allow the design to be partitioned so that each part can be refined independently.


```

MACHINE
    TopLevel
SEES
    Context
INCLUDES
    Input ,
    Validation ,
    Processing
PROMOTES
    GetRegisters ,
    LoadCommand
OPERATIONS
    ProcessCommand   $\hat{=}$ 
        BEGIN
            IF     $com \subseteq specialCom$     THEN
                RecordCheckedCommand ||
                SetRegisters ( com )
            END
        END
END

```

Figure 5: Example Top Level Machine

The structure of the example refinement is shown in figure 6.

Figure 6: Example Top Level Implementation Structure

An **IMPLEMENTATION** *TopLevelI*, as shown in figure 7, is written for the *TopLevel* abstract machine. *TopLevelI* forms part of the final low level design which will be directly translated into

IMPLEMENTATION

TopLevelI

REFINES

TopLevel

SEES

Bool_TYPE ,
Context2

IMPORTS

ValidateInput ,
Processing2

PROMOTES

LoadCommand ,
GetRegisters

OPERATIONS

```

ProcessCommand  ≡
  VAR  concretevalcom , result  IN
    result ← CheckCommand ;
  IF   result = TRUE   THEN
    RecordCheckedCommand ;
    concretevalcom ← GetCheckedCommand ;
    SetRegisters2 ( concretevalcom )
  END
END
END

```

END

Figure 7: Example Top Level Implementation

code.

TopLevelI **IMPORTS** abstract machines based on the machines **INCLUDED** in the *TopLevel* abstract specification. The *Input* and *Validation* machine both contain states of the same type. Hence they are **INCLUDED** into a new abstract machine, *ValidateInput*, which is **IMPORTED** into *TopLevelI*. *ValidateInput* is shown in figure 8.

The names of the operations in *TopLevelI* and their signature are the same as the operations in the *TopLevel* abstract specification. The system operations, *LoadCommand* and *GetRegisters*, have abstract input and output parameters and return values of abstract types. The operations are reused from the abstract specification by **PROMOTING** them.

The operation *ProcessCommand* is the top level software function in the embedded system. In the abstract specification of *ProcessCommand* an **IF** statement ensures that only valid input commands are processed. In the refined operation, the condition of the **IF** statement is captured as a

MACHINE

ValidateInput

SEES

Bool_TYPE ,
Context2

INCLUDES

Input ,
Validation

PROMOTES

LoadCommand ,
RecordCheckedCommand

OPERATIONS

```

result  $\leftarrow$  CheckCommand  $\hat{=}$ 
    IF    com  $\subseteq$  specialCom    THEN
        result := TRUE
    ELSE
        result := FALSE
    END ;

concretevalcom  $\leftarrow$  GetCheckedCommand  $\hat{=}$ 
    BEGIN
        concretevalcom := comToBits ( checkcom )
    END

```

END

Figure 8: Example Intermediate ValidateInput Machine

new operation, *CheckCommand*, which is specified as an operation in the new machine *ValidateInput*.

In the abstract specification, provided the input commands are valid two operations, *RecordCheckedCommand* and *SetRegisters*, are called in parallel. In the refinement this parallel composition is replaced by a sequence of operations to record the valid input commands and then set the appropriate hardware registers.

The *RecordCheckedCommand* is reused from the abstract specification because it does not have any input or output parameters. Operations with parameters are rewritten to use concrete types. The type of the parameters passed between internal operations must be concrete because they cannot be changed and need to be translated directly into code. For example, the *SetRegisters* operation is replaced by a new operation, *SetRegisters2*. The only difference between the two operations is their interface. *SetRegisters* receives commands of the abstract type whereas *SetRegisters2* receives commands of the concrete type. The result of applying the operations is the same - the state variable *registers* is updated.

SetRegisters2 is specified in a new machine, *Processing2*, as shown in figure 9. *Processing2* **INCLUDES** the *Processing* machine and has visibility to the *registers* state variable and the *SetRegisters* operation. This allows *SetRegister2* to accept concrete parameters as input, convert these into their abstract equivalent and call the original *SetRegisters* operation.

MACHINE	MACHINE
<i>Processing2</i>	<i>Context2</i>
SEES	INCLUDES
<i>Context2</i>	<i>Context</i>
INCLUDES	SETS
<i>Processing</i>	<i>COMBITS</i>
PROMOTES	CONSTANTS
<i>GetRegisters</i>	<i>comToBits</i>
OPERATIONS	PROPERTIES
<i>SetRegisters2</i> (<i>concheckcom</i>) $\hat{=}$	<i>comToBits</i> $\in$
PRE	$\mathbb{P} (\textit{COMMAND}) \mapsto \textit{COMBITS}$
<i>concheckcom</i> $\in$ <i>COMBITS</i>	END
THEN	
<i>SetRegisters</i>	
(<i>comToBits</i> ⁻¹ (<i>concheckcom</i>))	
END	
END	

Figure 9: Example Refined Processing and Context Machines

The concrete type is defined in a new context machine, *Context2*, as shown in figure 9, and a refinement relation is defined between the abstract and concrete types. This relation, *comToBits*, allows conversions from abstract to concrete representations and vice versa by inverting it.

The *Context2* machine is **SEEN** by all the new abstract **MACHINE**s which make use of the refinement relation to convert the interface of their operations. *TopLevelI* also needs visibility to *Context2* in order to allow proof of the top level operations involving the refinement relation.

In the abstract specification, *TopLevel* can directly examine state variables of **INCLUDED** machines. In *TopLevelI* the state is not directly visible because **IMPORT**ing machines only allows access to operations. Hence, the value of the parameter used in *SetRegisters2* must be retrieved using a new enquiry operation, *GetCheckedCommands*. This operation is specified in *ValidateInput*.

The **IMPLEMENTATION** *TopLevelI* along with its **IMPORT**ed abstract machines is a refinement of the abstract specification. The two new abstract machines, *ValidateInput* and *Processing2*, can be separately refined.

6 Experiences of Refining a Case Study

The Case Study used for the MIST project addresses part of the software controlling a Station Unit on a military aircraft. The Station Unit holds one store (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 is restricted to the main control and function of the Station Unit and covers about 20% of the total software. It is estimated to be equivalent to approximately 3,000 lines of Ada.

In total, 15 refinement steps were performed during the MIST project. The first refinement was a type refinement that gave explicit values, although still abstract, for many of the abstract types and constants. This first refinement was used as the starting point for both the prototype and the remaining formal refinements. Table 1 shows that 8 new context machines were used to capture these explicit values. There were 31 context proof obligations (shown in the Proofs from Mach column) generated from the types refinement.

The other proof figures for the abstract specification and other refinement stages do not include context proof obligations. These were found to be difficult existential proofs that were repeated as the context was built up. It was thought more appropriate to review these obligations.

Of the 793 proof obligations generated from the refinement steps, 790 were proved. The remaining 3 proof obligations were difficult existential proofs produced by a non-deterministic operation which reset state variables. It was thought more appropriate to review these proof obligations.

	Machines	Context	Pages of		Operations in		Proofs from	
			Imp	Mach	Imp	Mach	Imp	Mach
Abstract Spec	14	9		80		49		311
Type	0	8		9				31
Structural	23	2	14	43	19	68	135	57
Data and Operational	8	1	19	7	25	11	377	1
Detail	7	3	7	17	4	40	179	13
Total Refinement	40	14	41	82	48	138	691	102

Table 1: Size of Case Study Refinements

Following type refinement, five structural refinements were performed (Level 1 to 1.1.2). A more detailed breakdown of the figures for structural refinement is given in table 2.

The top level refinement, Level 1, was functionally equivalent to the abstract specification. It described exactly the same operations as the abstract specification, only in a more concrete style. Thus, the refinement was expected to be larger than the specification. It was larger but because of the reuse of the abstract specification, there were only 14 pages of new items in the refinement (shown in the Pages of Implementations and Machines). This is compared with 80 pages in the abstract specification. It can also be seen that the refinement operations were, on average, smaller than the abstract operations. This is because many of the refinement operations simply provided new interfaces to abstract operations and enquiry operations to retrieve the value of state variables.

A similar pattern is exhibited in the second and third structural refinements, Level 1.1 and 1.2 where there were a total of 24 pages of new items.

During the structural refinements of Levels 1, 1.1 and 1.2 a problem was encountered with the Proof Obligation Generator (POG) of the B-Toolkit. The algorithms used by the POG needed a lot of temporary storage and early versions of the B-Toolkit failed to generate all the proof

Level	Machines	Context	Pages of		Operations in		Proofs from	
			Imp	Mach	Imp	Mach	Imp	Mach
Abstract Spec	14	9		80		49		311
1	6	1	4	10	6	15	?	12
1.1	3	0	3	8	4	18	90	17
1.2	5	1	2	11	2	24	11	11
1.1.1	5	0	2	5	2	8	11	11
1.1.2	4	0	3	9	5	18	23	6
Total	23	2	14	43	19	68	135	57

Table 2: Size of Structural Refinements

obligations. For example, one operation in Level 1.2 had a seven-way **CASE** statement (in both the specification and the refinement) which led to 49 copies of each proof obligation (of which 42 were trivially true). Other refinements, in Level 1.1, were simply large and had large proof obligations. Later versions of the B-Toolkit, with more efficient POG algorithms, managed to generate all the proof obligations for the second level refinements (Levels 1.1 and 1.2). However, the top level refinement, Level 1, which was large and had a **CASE** statement, is still too complex for the current version of the B-Toolkit (3.1). However, B-Core are making improvements to the POG algorithm so that it can cope with refinements of the size and complexity of the top level. Once generated, the proof obligations for the structural refinement were easy to prove.

Level	Machines	Context	Pages of		Operations in		Proofs from	
			Imp	Mach	Imp	Mach	Imp	Mach
Last Structural Refinement	4	0	3	9	5	18	23	6
Higher Level Detail	3	1	3	9	2	17	93	6
Lower Level Detail	4	2	4	12	2	23	86	7
Total Detail Refinement	7	3	7	17	4	40	179	13

Table 3: Size of Detail Refinements

One of the abstract specifications produced by the last structural refinement was expanded with detail refinement. (The size of the last structural refinement is shown in the first row of table 3). It was expected that the detail refinements of this specification would result in a large number of new specifications required to capture the extra information. This is reflected in the figures in table 3. The higher level detail refinement is the same size as the specification of the last structural refinement even though it only forms a third of the actual refinement step. Similar conclusions can be drawn from the lower detail refinement. Again, it is only a partial refinement of the higher level and is of a comparable size.

The lowest level refinements were data or operational refinements. These use library machines and so generally introduced fewer new abstract specifications. The number of proof obligations generated by data refinement was dependent on the programming style of the AMN implementations. For example, one data refinement initially generated an excessive number of proof obligations (over 4000) as a result of a large number of paths through the code (over 1000). The refinement was re-written to reduce the number of paths and this reduced the number of proof obligations to 94, but increased their complexity.

Far fewer proof obligations were generated for the machines used to support the refinement than were generated for the abstract specification (102 vs. 311, as shown in table 1). This was expected, as the abstract specification had a number of complex invariants stating important properties of the system. The new specifications used by the refinements tended to have simpler invariants.

In total, 13 errors were found during refinement. Two of these were in the abstract specification and were discovered whilst writing the refinement. Writing a refinement forces an engineer to carefully review and understand the specification. The required corrections to the abstract specification did not have a large impact on the refinement. The remaining 11 errors were in the refinement and were discovered during proof.

In the MIST project the effort taken to specify, prove and document all 15 refinement steps was 65 days, as shown in table 4.

	Spec	Proof	Doc	Total
Abstract Spec	31.0	7.0	10.0	48.0
Type Refinement	1.0	1.5	0.5	3.0
First Structural Ref	3.0	1 +	2.0	6.0
Other Structural Ref	7.0	2.0	2.5	11.5
Difficult Data Refinements	3.5	30.5	2.0	36.0
Other Data Refinements	0.5	1.0	0.5	2.0
Detail Refinement	2.0	4.0	0.5	6.5
Total Refinement	17.0	40.0	8.0	65.0

+ - Top level refinement proofs were not proved.

6 days of animation and 8 days of review on the abstract specification.

7 days of review on the refinements.

Table 4: Effort used during refinement (in days)

The table is dominated by the effort needed to prove the four difficult data refinements, over 75% of the proof time and almost half the total time spent on the refinement. This is not a general problem with data refinements, the other data refinements were proved in a single day, but reflects the complexity of these particular data refinements. A significant length of time was needed to prove the detail proof obligations, but the structural proof obligations were relatively easy.

Overall, if the difficult proofs are ignored, the refinement took less time than the abstract specification. This was a very surprising result — it was expected that the refinement would take much longer. Even including the difficult proofs the refinement took only slightly longer. As refinement was thought to be harder than specification, more experienced engineers performed the refinements. Even allowing for this, the refinements were written and verified with much less effort than expected.

The major gain in effort in the refinement process was due to the substantial reuse of the abstract specification during structural refinement. All 5 structural refinement steps were completed in less than a third of the time it took to write the abstract specification.

The final product of the refinement process was a collection of AMN **IMPLEMENTATIONS** and simple specifications of interfaces, mainly hardware. Ada source code was produced for part of the Case Study. The AMN **IMPLEMENTATIONS** were translated into 800 lines of Ada source code. A further 1400 lines were informally developed from the simple specifications. It was estimated that 600 more lines of Ada would be produced from the remainder of the Case Study.

7 Conclusions

The refinement of the MIST case study has been very successful, in particular the style of specification and refinement used to model embedded software and to partition the top level design has worked well.

The abstract model of embedded software, which includes system operations that interact with the software, has been preserved throughout the refinement. However, as abstract implementations can be used for these system operations, there was found to be little overhead in refining the complete model rather than just the software operations.

The structural refinement style has proved successful in breaking the design down into manageable parts. There were some problems with generating the proof obligations for these refinements but these are being addresses. Even though errors were introduced during the writing of the refinements these were all discovered during proof. No further refinement errors were found during extensive testing.

Generally, the refinement was performed with less effort than expected. A large proportion of the refinement effort was spent on a group of related data refinements. The features of this data refinement have been recognised and it is believed that similar hard data refinements can be identified in future designs.

The MIST procedures on refinement will be used on any future designs with the B-Method. The costs and benefits of data refinement will be weighed up for each project individually. Structural refinement will be used on all future B-Method designs.

References

- [1] Draper J. Applying the B-Method to Avionics Software: an Initial Report on the MIST Project. In: Shaw R C. (Ed) Safety and Reliability of Software Based Systems, Springer-Verlag, 1996.
- [2] Draper J, Treharne H, Boyce B, Ormsby O. Evaluating the B-Method on an Avionics Example. In: Data Systems in Aerospace (DASIA) Conference, Rome, 1996, pp 89 - 97 (European Space Agency Publications Division WPP-116).
- [3] Ministry of Defence: The Procurement of Safety Critical Software in Defence Equipment, Interim Defence Standard 00-55, 1991.
- [4] Abrial J R. The B Book: Assigning Programs to Meaning, Cambridge University Press, 1996.
- [5] Abrial J R. A refinement case study (Using the Abstract Machine Notation). In : Morris J M and Shaw R C. (Eds) Proceedings of 4th refinement workshop, (Workshops in computing) Springer-Verlag, 1991.
- [6] B-Core(UK): B-Toolkit User's Manual, Version 3.0, Magdalen Centre, The Oxford Science Park 1995.